

Язык программирования C# 7 и платформы .NET и .NET Core

8-е издание

Эндрю Троелсен
Филипп Джепикс

 **ДИАЛЕКТИКА**
www.williamspublishing.com

apress®

Язык программирования

C# 7

и платформы .NET и .NET Core

Pro C# 7

With .NET and .NET Core

Eighth Edition

Andrew Troelsen
Philip Japikse

Apress®

Язык программирования

C# 7

и платформы .NET и .NET Core

8-е издание

Эндрю Троелсен
Филипп Джепикс



Москва • Санкт-Петербург
2018

ББК 32.973.26-018.2.75

Т70

УДК 681.3.07

Компьютерное издательство "Диалектика"

Перевод с английского и редакция Ю.Н. Артеменко

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:

info@dialektika.com, <http://www.dialektika.com>

Троелсен, Эндрю, Джепикс, Филипп.

Т70 Язык программирования C# 7 и платформы .NET и .NET Core, 8-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2018 — 1328 с. : ил. — Парал. тит. англ.

ISBN 978-5-6040723-1-8 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства APress, Berkeley, CA.

Authorized translation from the English language edition published by APress, Inc., Copyright © 2017 by Andrew Troelsen and Philip Japikse.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher. Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Научно-популярное издание

Эндрю Троелсен, Филипп Джепикс

Язык программирования C# 7 и платформы .NET и .NET Core 8-е издание

Подписано в печать 16.07.2018. Формат 70×100/16.

Гарнитура Times.

Усл. печ. л. 112. Уч.-изд. л. 92.3.

Тираж 500 экз. Заказ № 6680.

Отпечатано в АО "Первая Образцовая типография"

Филиал "Чеховский Печатный Двор"

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-6040723-1-8 (рус.)

ISBN 978-1-4842-3017-6 (англ.)

© 2018, ООО "Диалектика"

© 2017 by Andrew Troelsen and Philip Japikse

Оглавление

Часть I. Введение в C# и платформу .NET	47
Глава 1. Философия .NET	48
Глава 2. Создание приложений на языке C#	81
Часть II. Основы программирования на C#	97
Глава 3. Главные конструкции программирования на C#: часть I	98
Глава 4. Главные конструкции программирования на C#: часть II	150
Часть III. Объектно-ориентированное программирование на C#	193
Глава 5. Инкапсуляция	194
Глава 6. Наследование и полиморфизм	242
Глава 7. Структурированная обработка исключений	280
Глава 8. Работа с интерфейсами	306
Часть IV. Дополнительные конструкции программирования на C#	341
Глава 9. Коллекции и обобщения	342
Глава 10. Делегаты, события и лямбда-выражения	379
Глава 11. Расширенные средства языка C#	417
Глава 12. LINQ to Objects	452
Глава 13. Время существования объектов	483
Часть V. Программирование с использованием сборок .NET	509
Глава 14. Построение и конфигурирование библиотек классов	510
Глава 15. Рефлексия типов, позднее связывание и программирование на основе атрибутов	557
Глава 16. Динамические типы и среда DLR	600
Глава 17. Процессы, домены приложений и объектные контексты	620
Глава 18. Язык CIL и роль динамических сборок	645
Часть VI. Введение в библиотеки базовых классов .NET	685
Глава 19. Многопоточное, параллельное и асинхронное программирование	686
Глава 20. Файловый ввод-вывод и сериализация объектов	738
Глава 21. Доступ к данным с помощью ADO.NET	782
Глава 22. Введение в Entity Framework 6	836
Глава 23. Введение в Windows Communication Foundation	884
Часть VII. Windows Presentation Foundation	937
Глава 24. Введение в Windows Presentation Foundation и XAML	938
Глава 25. Элементы управления, компоновки, события и привязка данных в WPF	974
Глава 26. Службы визуализации графики WPF	1031
Глава 27. Ресурсы, анимация, стили и шаблоны WPF	1066
Глава 28. Уведомления, проверка достоверности, команды и MVVM	1105
Часть VIII. ASP.NET	1145
Глава 29. Введение в ASP.NET MVC	1146
Глава 30. Введение в ASP.NET Web API	1189
Часть IX. .NET Core	1209
Глава 31. Философия .NET Core	1210
Глава 32. Введение в Entity Framework Core	1220
Глава 33. Введение в веб-приложения ASP.NET Core	1244
Глава 34. Введение в приложения служб ASP.NET Core	1295
Предметный указатель	1316

Содержание

Об авторах	34
Введение	36
Авторы и читатели — одна команда	36
Краткий обзор книги	36
Часть I. Введение в C# и платформу .NET	37
Часть II. Основы программирования на C#	37
Часть III. Объектно-ориентированное программирование на C#	38
Часть IV. Дополнительные конструкции программирования на C#	38
Часть V. Программирование с использованием сборок .NET	40
Часть VI. Введение в библиотеки базовых классов .NET	41
Часть VII. Windows Presentation Foundation	42
Часть VIII. ASP.NET	43
Часть IX. .NET Core	44
Загружаемые приложения	45
Исходный код примеров	45
Ждем ваших отзывов!	46
 Часть I. Введение в C# и платформу .NET	 47
Глава 1. Философия .NET	48
Начальное знакомство с платформой .NET	48
Некоторые основные преимущества платформы .NET	49
Введение в строительные блоки платформы .NET (CLR, CTS и CLS)	50
Роль библиотек базовых классов	50
Что привносит язык C#	51
Сравнение управляемого и неуправляемого кода	54
Другие языки программирования, ориентированные на .NET	54
Жизнь в многоязычном мире	55
Обзор сборок .NET	55
Роль языка CIL	57
Роль метаданных типов .NET	59
Роль манифеста сборки	60
Понятие общей системы типов (CTS)	61
Типы классов CTS	61
Типы интерфейсов CTS	62
Типы структур CTS	62
Типы перечислений CTS	63
Типы делегатов CTS	63
Члены типов CTS	64
Встроенные типы данных CTS	64
Понятие общезыковой спецификации (CLS)	65
Обеспечение совместимости с CLS	66
Понятие общезыковой исполняющей среды (CLR)	67
Различия между сборками, пространствами имен и типами	68
Роль корневого пространства имен Microsoft	71
Доступ к пространству имен программным образом	72
Ссылка на внешние сборки	73

Исследование сборки с помощью <code>ildasm.exe</code>	74
Просмотр кода CIL	75
Просмотр метаданных типов	75
Просмотр метаданных сборки (манифеста)	76
Независимая от платформы природа .NET	77
Проект Mono	78
Xamarin	79
Microsoft .NET Core	79
Резюме	80
Глава 2. Создание приложений на языке C#	81
Построение приложений .NET в среде Windows	81
Установка Visual Studio 2017	82
Испытание Visual Studio 2017	84
Visual Studio 2017 Professional	95
Visual Studio 2017 Enterprise	95
Система документации .NET Framework	95
Построение приложений .NET под управлением операционной системы, отличающейся от Windows	96
Резюме	96
Часть II. Основы программирования на C#	97
Глава 3. Главные конструкции программирования на C#: часть I	98
Структура простой программы C#	98
Вариации метода <code>Main()</code>	100
Указание кода ошибки приложения	101
Обработка аргументов командной строки	102
Указание аргументов командной строки в Visual Studio	103
Интересное отступление от темы: некоторые дополнительные члены класса <code>System.Environment</code>	104
Класс <code>System.Console</code>	106
Базовый ввод-вывод с помощью класса <code>Console</code>	106
Форматирование консольного вывода	107
Форматирование числовых данных	108
Форматирование числовых данных за рамками консольных приложений	109
Системные типы данных и соответствующие ключевые слова C#	110
Объявление и инициализация переменных	111
Внутренние типы данных и операция <code>new</code>	113
Иерархия классов для типов данных	114
Члены числовых типов данных	115
Члены <code>System.Boolean</code>	116
Члены <code>System.Char</code>	116
Разбор значений из строковых данных	116
Типы <code>System.DateTime</code> и <code>System.TimeSpan</code>	118
Сборка <code>System.Numerics.dll</code>	118
Разделители групп цифр (нововведение)	120
Двоичные литералы (нововведение)	120
Работа со строковыми данными	120
Базовые манипуляции строками	121
Конкатенация строк	122

Управляющие последовательности	123
Определение дословных строк	123
Строки и равенство	124
Модификация поведения сравнения строк	125
Строки являются неизменяемыми	126
Тип <code>System.Text.StringBuilder</code>	127
Интерполяция строк	128
Сужающие и расширяющие преобразования типов данных	130
Ключевое слово <code>checked</code>	132
Настройка проверки переполнения на уровне проекта	134
Ключевое слово <code>unchecked</code>	134
Понятие неявно типизированных локальных переменных	135
Ограничения неявно типизированных переменных	136
Неявно типизированные данные являются строго типизированными	137
Полезность неявно типизированных локальных переменных	138
Итерационные конструкции C#	139
Цикл <code>for</code>	139
Цикл <code>foreach</code>	140
Циклы <code>while</code> и <code>do/while</code>	141
Конструкции принятия решений и операции отношения/равенства	141
Оператор <code>if/else</code>	142
Операции отношения и равенства	142
Условная операция	143
Логические операции	143
Оператор <code>switch</code>	144
Резюме	149
Глава 4. Главные конструкции программирования на C#: часть II	150
Понятие массивов C#	150
Синтаксис инициализации массивов C#	151
Неявно типизированные локальные массивы	152
Определение массива объектов	153
Работа с многомерными массивами	153
Использование массивов в качестве аргументов и возвращаемых значений	154
Базовый класс <code>System.Array</code>	155
Методы и модификаторы параметров	157
Возвращаемые значения и члены, сжатые до выражений (обновление)	157
Модификаторы параметров для методов	157
Отбрасывание	158
Стандартное поведение передачи параметров по значению	158
Модификатор <code>out</code> (обновление)	159
Модификатор <code>ref</code>	161
Ссылочные локальные переменные и возвращаемые ссылочные значения (нововведения)	162
Модификатор <code>params</code>	163
Определение необязательных параметров	165
Вызов методов с использованием именованных параметров	166
Понятие перегрузки методов	168
Локальные функции (нововведение)	169
Тип <code>enum</code>	170
Управление хранилищем, лежащим в основе перечисления	171

Объявление переменных типа перечисления	172
Тип <code>System.Enum</code>	173
Динамическое выяснение пар "имя-значение" перечисления	174
Понятие структур (как типа значения)	175
Создание переменных типа структур	177
Типы значений и ссылочные типы	178
Типы значений, ссылочные типы и операция присваивания	179
Типы значений, содержащие ссылочные типы	180
Передача ссылочных типов по значению	182
Передача ссылочных типов по ссылке	183
Заключительные детали относительно типов значений и ссылочных типов	184
Понятие типов <code>C#</code> , допускающих <code>null</code>	185
Работа с типами, допускающими <code>null</code>	186
Операция объединения с <code>null</code>	187
<code>null</code> -условная операция	188
Кортежи (нововведение)	189
Начало работы с кортежами	189
Выведение имен свойств кортежей (<code>C# 7.1</code>)	190
Кортежи как возвращаемые значения методов	191
Использование отбрасывания с кортежами	191
Деконструирование кортежей	192
Резюме	192

Часть III. Объектно-ориентированное программирование на `C#` 193

Глава 5. Инкапсуляция 194

Знакомство с типом класса <code>C#</code>	194
Размещение объектов с помощью ключевого слова <code>new</code>	196
Понятие конструкторов	197
Роль стандартного конструктора	197
Определение специальных конструкторов	198
Еще раз о стандартном конструкторе	199
Роль ключевого слова <code>this</code>	201
Построение цепочки вызовов конструкторов с использованием <code>this</code>	202
Изучение потока управления конструкторов	205
Еще раз о необязательных аргументах	206
Понятие ключевого слова <code>static</code>	207
Определение статических полей данных	208
Определение статических методов	210
Определение статических конструкторов	211
Определение статических классов	213
Импортирование статических членов с применением ключевого слова <code>using</code> языка <code>C#</code>	214
Основные принципы объектно-ориентированного программирования	215
Роль инкапсуляции	215
Роль наследования	216
Роль полиморфизма	217
Модификаторы доступа <code>C#</code>	219
Стандартные модификаторы доступа	220
Модификаторы доступа и вложенные типы	220

Первый принцип ООП: службы инкапсуляции C#	221
Инкапсуляция с использованием традиционных методов доступа и изменения	222
Инкапсуляция с использованием свойств .NET	224
Использование свойств внутри определения класса	227
Свойства, допускающие только чтение и только запись	228
Еще раз о ключевом слове <code>static</code> : определение статических свойств	229
Понятие автоматических свойств	230
Взаимодействие с автоматическими свойствами	231
Автоматические свойства и стандартные значения	232
Инициализация автоматических свойств	233
Понятие синтаксиса инициализации объектов	234
Вызов специальных конструкторов с помощью синтаксиса инициализации	235
Инициализация данных с помощью синтаксиса инициализации	236
Работа с данными константных полей	237
Понятие полей, допускающих только чтение	239
Статические поля, допускающие только чтение	239
Понятие частичных классов	240
Сценарии использования для частичных классов?	241
Резюме	241
Глава 6. Наследование и полиморфизм	242
Базовый механизм наследования	242
Указание родительского класса для существующего класса	243
Замечание относительно множества базовых классов	245
Ключевое слово <code>sealed</code>	245
Корректировка диаграмм классов Visual Studio	246
Второй принцип ООП: детали наследования	247
Управление созданием объектов базового класса с помощью ключевого слова <code>base</code>	249
Хранение секретов семейства: ключевое слово <code>protected</code>	251
Добавление запечатанного класса	252
Реализация модели включения/делегации	253
Определения вложенных типов	254
Третий принцип ООП: поддержка полиморфизма в C#	256
Ключевые слова <code>virtual</code> и <code>override</code>	256
Переопределение виртуальных членов в IDE-среде Visual Studio	258
Запечатывание виртуальных членов	259
Абстрактные классы	260
Полиморфные интерфейсы	261
Соккрытие членов	266
Правила приведения для базовых и производных классов	268
Ключевое слово <code>as</code>	269
Ключевое слово <code>is</code> (обновление)	271
Еще раз о сопоставлении с образцом (нововведение)	272
Главный родительский класс <code>System.Object</code>	273
Переопределение метода <code>System.Object.ToString()</code>	276
Переопределение метода <code>System.Object.Equals()</code>	276
Переопределение метода <code>System.Object.GetHashCode()</code>	277
Тестирование модифицированного класса <code>Person</code>	278
Статические члены класса <code>System.Object</code>	279
Резюме	279

Глава 7. Структурированная обработка исключений	280
Ода ошибкам, дефектам и исключениям	280
Роль обработки исключений .NET	281
Строительные блоки обработки исключений в .NET	282
Базовый класс <code>System.Exception</code>	282
Простейший пример	284
Генерация общего исключения (обновление)	285
Перехват исключений	287
Конфигурирование состояния исключения	288
Свойство <code>TargetSite</code>	288
Свойство <code>StackTrace</code>	289
Свойство <code>HelpLink</code>	290
Свойство <code>Data</code>	291
Исключения уровня системы (<code>System.SystemException</code>)	292
Исключения уровня приложения (<code>System.ApplicationException</code>)	293
Построение специальных исключений, способ первый	293
Построение специальных исключений, способ второй	295
Построение специальных исключений, способ третий	296
Обработка множества исключений	297
Общие операторы <code>catch</code>	300
Повторная генерация исключений	300
Внутренние исключения	301
Блок <code>finally</code>	302
Фильтры исключений	303
Отладка необработанных исключений с использованием Visual Studio	304
Резюме	305
Глава 8. Работа с интерфейсами	306
Понятие интерфейсных типов	306
Сравнение интерфейсных типов и абстрактных базовых классов	307
Определение специальных интерфейсов	309
Реализация интерфейса	311
Обращение к членам интерфейса на уровне объектов	313
Получение ссылок на интерфейсы: ключевое слово <code>as</code>	314
Получение ссылок на интерфейсы: ключевое слово <code>is</code> (обновление)	314
Использование интерфейсов в качестве параметров	315
Использование интерфейсов в качестве возвращаемых значений	317
Массивы интерфейсных типов	318
Реализация интерфейсов с использованием Visual Studio	319
Явная реализация интерфейсов	320
Проектирование иерархий интерфейсов	322
Множественное наследование с помощью интерфейсных типов	324
Интерфейсы <code>IEnumerable</code> и <code>IEnumerator</code>	326
Построение итераторных методов с использованием ключевого слова <code>yield</code>	328
Построение именованного итератора	330
Интерфейс <code>ICloneable</code>	331
Более сложный пример клонирования	333
Интерфейс <code>IComparable</code>	335
Указание множества порядков сортировки с помощью <code>IComparer</code>	338
Специальные свойства и специальные типы сортировки	339
Резюме	340

Часть IV. Дополнительные конструкции программирования на C#	341
Глава 9. Коллекции и обобщения	342
Побудительные причины создания классов коллекций	342
Пространство имен <code>System.Collections</code>	344
Обзор пространства имен <code>System.Collections.Specialized</code>	346
Проблемы, присущие необобщенным коллекциям	347
Проблема производительности	347
Проблема безопасности к типам	350
Первый взгляд на обобщенные коллекции	353
Роль параметров обобщенных типов	354
Указание параметров типа для обобщенных классов и структур	355
Указание параметров типа для обобщенных членов	356
Указание параметров типов для обобщенных интерфейсов	357
Пространство имен <code>System.Collections.Generic</code>	358
Синтаксис инициализации коллекций	360
Работа с классом <code>List<T></code>	361
Работа с классом <code>Stack<T></code>	362
Работа с классом <code>Queue<T></code>	363
Работа с классом <code>SortedSet<T></code>	364
Работа с классом <code>Dictionary<TKey, TValue></code>	366
Пространство имен <code>System.Collections.ObjectModel</code>	367
Работа с классом <code>ObservableCollection<T></code>	367
Создание специальных обобщенных методов	369
Выведение параметров типа	371
Создание специальных обобщенных структур и классов	372
Ключевое слово <code>default</code> в обобщенном коде	373
Ограничение параметров типа	374
Примеры использования ключевого слова <code>where</code>	375
Отсутствие ограничений операций	377
Резюме	378
Глава 10. Делегаты, события и лямбда-выражения	379
Понятие типа делегата .NET	379
Определение типа делегата в C#	380
Базовые классы <code>System.MulticastDelegate</code> и <code>System.Delegate</code>	383
Пример простейшего делегата	384
Исследование объекта делегата	385
Отправка уведомлений о состоянии объекта с использованием делегатов	387
Включение группового вызова	389
Удаление целей из списка вызовов делегата	391
Синтаксис групповых преобразований методов	392
Понятие обобщенных делегатов	393
Обобщенные делегаты <code>Action<></code> и <code>Func<></code>	394
Понятие событий C#	396
Ключевое слово <code>event</code>	398
“За кулисами” событий	399
Прослушивание входящих событий	400
Упрощение регистрации событий с использованием Visual Studio	401
Приведение в порядок кода обращения к событиям с использованием <code>null</code> -условной операции C# 6.0	403
Создание специальных аргументов событий	403

Обобщенный делегат <code>EventHandler<T></code>	405
Понятие анонимных методов C#	406
Доступ к локальным переменным	407
Понятие лямбда-выражений	408
Анализ лямбда-выражения	411
Обработка аргументов внутри множества операторов	412
Лямбда-выражения с несколькими параметрами и без параметров	413
Модернизация примера <code>CarEvents</code> с использованием лямбда-выражений	414
Лямбда-выражения и члены, сжатые до выражений (обновление)	415
Резюме	416
Глава 11. Расширенные средства языка C#	417
Понятие индексаторных методов	417
Индексация данных с использованием строковых значений	419
Перегрузка методов индексаторов	420
Многомерные индексаторы	421
Определения индексаторов в интерфейсных типах	422
Понятие перегрузки операций	422
Перегрузка бинарных операций	423
А как насчет операций <code>+=</code> и <code>-=</code> ?	425
Перегрузка унарных операций	425
Перегрузка операций эквивалентности	426
Перегрузка операций сравнения	427
Финальные соображения относительно перегрузки операций	428
Понятие специальных преобразований типов	428
Повторение: числовые преобразования	428
Повторение: преобразования между связанными типами классов	428
Создание специальных процедур преобразования	429
Дополнительные явные преобразования для типа <code>Square</code>	432
Определение процедур неявного преобразования	432
Понятие расширяющих методов	434
Понятие анонимных типов	438
Определение анонимного типа	439
Анонимные типы, содержащие другие анонимные типы	443
Работа с типами указателей	444
Ключевое слово <code>unsafe</code>	445
Работа с операциями <code>*</code> и <code>&</code>	447
Небезопасная (и безопасная) функция обмена	447
Доступ к полям через указатели (операция <code>-></code>)	448
Ключевое слово <code>stackalloc</code>	449
Закрепление типа посредством ключевого слова <code>fixed</code>	449
Ключевое слово <code>sizeof</code>	450
Резюме	451
Глава 12. LINQ to Objects	452
Программные конструкции, специфичные для LINQ	452
Неявная типизация локальных переменных	453
Синтаксис инициализации объектов и коллекций	454
Лямбда-выражения	454
Расширяющие методы	455
Анонимные типы	456

Роль LINQ	456
Выражения LINQ строго типизированы	457
Основные сборки LINQ	458
Применение запросов LINQ к элементарным массивам	458
Решение с использованием расширяющих методов	460
Решение без использования LINQ	460
Выполнение рефлексии результирующего набора LINQ	461
LINQ и неявно типизированные локальные переменные	462
LINQ и расширяющие методы	463
Роль отложенного выполнения	464
Роль немедленного выполнения	465
Возвращение результатов запроса LINQ	466
Возвращение результатов LINQ посредством немедленного выполнения	467
Применение запросов LINQ к объектам коллекций	468
Доступ к содержащимся в контейнере подобъектам	469
Применение запросов LINQ к необобщенным коллекциям	470
Фильтрация данных с использованием метода <code>OfType<T>()</code>	470
Исследование операций запросов LINQ	471
Базовый синтаксис выборки	472
Получение подмножества данных	473
Проецирование новых типов данных	474
Подсчет количества с использованием класса <code>Enumerable</code>	475
Изменение порядка следования элементов в результирующих наборах на противоположный	475
Выражения сортировки	476
LINQ как лучшее средство построения диаграмм Венна	476
Устранение дубликатов	477
Операции агрегирования LINQ	478
Внутреннее представление операторов запросов LINQ	478
Построение выражений запросов с применением операций запросов	479
Построение выражений запросов с использованием типа <code>Enumerable</code> и лямбда-выражений	479
Построение выражений запросов с использованием типа <code>Enumerable</code> и анонимных методов	481
Построение выражений запросов с использованием типа <code>Enumerable</code> и низкоуровневых делегатов	481
Резюме	482
Глава 13. Время существования объектов	483
Классы, объекты и ссылки	483
Базовые сведения о времени жизни объектов	484
Код CIL для ключевого слова <code>new</code>	485
Установка объектных ссылок в <code>null</code>	486
Роль корневых элементов приложения	487
Понятие поколений объектов	489
Параллельная сборка мусора до версии .NET 4.0	490
Фоновая сборка мусора в .NET 4.0 и последующих версиях	490
Тип <code>System.GC</code>	491
Принудительный запуск сборщика мусора	492
Построение финализируемых объектов	494
Переопределение метода <code>System.Object.Finalize()</code>	495

Подробности процесса финализации	497
Построение освобождаемых объектов	498
Повторное использование ключевого слова <code>using</code> в C#	500
Создание финализируемых и освобождаемых типов	501
Формализованный шаблон освобождения	502
Ленивое создание объектов	504
Настройка процесса создания данных <code>Lazy<></code>	507
Резюме	508
Часть V. Программирование с использованием сборок .NET	509
Глава 14. Построение и конфигурирование библиотек классов	510
Определение специальных пространств имен	510
Разрешение конфликтов имен с помощью полностью заданных имен	512
Разрешение конфликтов имен с помощью псевдонимов	513
Создание вложенных пространств имен	515
Стандартное пространство имен Visual Studio	516
Роль сборки .NET	516
Сборки содействуют многократному использованию кода	517
Сборки устанавливают границы типов	517
Сборки являются единицами, поддерживающими версии	517
Сборки являются самоописательными	518
Сборки являются конфигурируемыми	518
Формат сборки .NET	518
Заголовок файла Windows	519
Заголовок файла CLR	520
Код CIL, метаданные типов и манифест сборки	521
Дополнительные ресурсы сборки	522
Построение и потребление специальной библиотеки классов	522
Исследование манифеста	525
Исследование кода CIL	527
Исследование метаданных типов	528
Построение клиентского приложения C#	529
Построение клиентского приложения Visual Basic	530
Межъязыковое наследование в действии	531
Понятие закрытых сборок	532
Удостоверение закрытой сборки	533
Понятие процесса зондирования	533
Конфигурирование закрытых сборок	534
Роль файла <code>App.Config</code>	535
Понятие разделяемых сборок	537
Глобальный кеш сборок	537
Понятие строгих имен	539
Генерация строгих имен в командной строке	540
Генерация строгих имен в Visual Studio	542
Установка строго именованных сборок в GAC	543
Потребление разделяемой сборки	544
Изучение манифеста <code>SharedCarLibClient</code>	546
Конфигурирование разделяемых сборок	546
Замораживание текущей версии разделяемой сборки	547
Построение разделяемой сборки версии 2.0.0.0	548
Динамическое перенаправление на специфичные версии разделяемой сборки	549

Понятие сборок политик издателя	551
Отключение политики издателя	552
Элемент <code><codeBase></code>	552
Пространство имен <code>System.Configuration</code>	554
Документация по схеме конфигурационного файла	555
Резюме	556
Глава 15. Рефлексия типов, позднее связывание и программирование на основе атрибутов	557
Потребность в метаданных типов	557
Просмотр (частичных) метаданных для перечисления <code>EngineState</code>	558
Просмотр (частичных) метаданных для типа <code>Car</code>	559
Исследование блока <code>TypeRef</code>	560
Документирование определяемой сборки	561
Документирование ссылаемых сборок	561
Документирование строковых литералов	562
Понятие рефлексии	562
Класс <code>System.Type</code>	563
Получение информации о типе с помощью <code>System.Object.GetType()</code>	564
Получение информации о типе с помощью <code>typeof()</code>	564
Получение информации о типе с помощью <code>System.Type.GetType()</code>	564
Построение специального средства для просмотра метаданных	565
Рефлексия методов	566
Рефлексия полей и свойств	566
Рефлексия реализованных интерфейсов	567
Отображение разнообразных дополнительных деталей	567
Реализация метода <code>Main()</code>	568
Рефлексия обобщенных типов	569
Рефлексия параметров и возвращаемых значений методов	569
Динамическая загрузка сборок	571
Рефлексия разделяемых сборок	573
Позднее связывание	575
Класс <code>System.Activator</code>	575
Вызов методов без параметров	577
Вызов методов с параметрами	577
Роль атрибутов <code>.NET</code>	578
Потребители атрибутов	579
Применение атрибутов в <code>C#</code>	580
Сокращенная система обозначения атрибутов <code>C#</code>	581
Указание параметров конструктора для атрибутов	581
Атрибут <code>[Obsolete]</code> в действии	582
Построение специальных атрибутов	583
Применение специальных атрибутов	583
Синтаксис именованных свойств	584
Ограничение использования атрибутов	584
Атрибуты уровня сборки	585
Файл <code>AssemblyInfo.cs</code> , генерируемый Visual Studio	586
Рефлексия атрибутов с использованием раннего связывания	587
Рефлексия атрибутов с использованием позднего связывания	588
Практическое использование рефлексии, позднего связывания и специальных атрибутов	589

Построение расширяемого приложения	590
Построение мультипроектного решения ExtendableApp	591
Построение сборки CommonSnappableTypes.dll	591
Добавление проектов в решение	592
Добавление ссылок на проекты	593
Построение оснастки на C#	594
Построение оснастки на Visual Basic	594
Установка запускаемого проекта	594
Установка порядка построения проектов	595
Построение расширяемого консольного приложения	597
Резюме	599
Глава 16. Динамические типы и среда DLR	600
Роль ключевого слова <code>dynamic</code> языка C#	600
Вызов членов на динамически объявленных данных	602
Роль сборки <code>Microsoft.CSharp.dll</code>	603
Область применения ключевого слова <code>dynamic</code>	604
Ограничения ключевого слова <code>dynamic</code>	605
Практическое использование ключевого слова <code>dynamic</code>	605
Роль исполняющей среды динамического языка	606
Роль деревьев выражений	607
Роль пространства имен <code>System.Dynamic</code>	607
Динамический поиск в деревьях выражений во время выполнения	608
Упрощение вызовов с поздним связыванием посредством динамических типов	608
Использование ключевого слова <code>dynamic</code> для передачи аргументов	609
Упрощение взаимодействия с COM посредством динамических данных	612
Роль основных сборок взаимодействия	613
Встраивание метаданных взаимодействия	614
Общие сложности взаимодействия с COM	615
Взаимодействие с COM с использованием динамических данных C#	616
Взаимодействие с COM без использования динамических данных C#	617
Резюме	619
Глава 17. Процессы, домены приложений и объектные контексты	620
Роль процесса Windows	620
Роль потоков	621
Взаимодействие с процессами на платформе .NET	623
Перечисление выполняющихся процессов	625
Исследование конкретного процесса	626
Исследование набора потоков процесса	626
Исследование набора модулей процесса	628
Запуск и останов процессов программным образом	629
Управление запуском процесса с использованием класса <code>ProcessStartInfo</code>	630
Домены приложений .NET	631
Класс <code>System.AppDomain</code>	632
Взаимодействие со стандартным доменом приложения	634
Перечисление загруженных сборок	635
Получение уведомлений о загрузке сборок	636
Создание новых доменов приложений	637
Загрузка сборок в специальные домены приложений	638
Выгрузка доменов приложений программным образом	639

Контекстные границы объектов	640
Контекстно-свободные и контекстно-связанные типы	641
Определение контекстно-связанного объекта	642
Исследование контекста объекта	642
Итоговые сведения о процессах, доменах приложений и контекстах	644
Резюме	644
Глава 18. Язык CIL и роль динамических сборок	645
Причины для изучения грамматики языка CIL	645
Директивы, атрибуты и коды операций CIL	646
Роль директив CIL	647
Роль атрибутов CIL	647
Роль кодов операций CIL	647
Разница между кодами операций и их мнемоническими эквивалентами в CIL	648
Заталкивание и выталкивание: основанная на стеке природа CIL	649
Возвратное проектирование	650
Роль меток в коде CIL	653
Взаимодействие с CIL: модификация файла *.il	654
Компиляция кода CIL с помощью ilasm.exe	655
Роль инструмента peverify.exe	656
Директивы и атрибуты CIL	656
Указание ссылок на внешние сборки в CIL	656
Определение текущей сборки в CIL	657
Определение пространств имен в CIL	658
Определение типов классов в CIL	658
Определение и реализация интерфейсов в CIL	659
Определение структур в CIL	660
Определение перечислений в CIL	660
Определение обобщений в CIL	661
Компиляция файла CILTypes.il	661
Соответствия между типами данных в библиотеке базовых классов .NET, C# и CIL	662
Определение членов типов в CIL	663
Определение полей данных в CIL	663
Определение конструкторов типа в CIL	663
Определение свойств в CIL	664
Определение параметров членов	665
Исследование кодов операций CIL	665
Директива .maxstack	667
Объявление локальных переменных в CIL	668
Отображение параметров на локальные переменные в CIL	668
Скрытая ссылка this	669
Представление итерационных конструкций в CIL	669
Построение сборки .NET на CIL	670
Построение сборки CILCars.dll	671
Построение сборки CILCarClient.exe	673
Динамические сборки	674
Исследование пространства имен System.Reflection.Emit	675
Роль типа System.Reflection.Emit.ILGenerator	676
Выпуск динамической сборки	677
Выпуск сборки и набора модулей	679
Роль типа ModuleBuilder	680

Выпуск типа <code>HelloClass</code> и строковой переменной-члена	681
Выпуск конструкторов	681
Выпуск метода <code>SayHello()</code>	682
Использование динамически сгенерированной сборки	683
Резюме	684

Часть VI. Введение в библиотеки базовых классов .NET 685

Глава 19. Многопоточное, параллельное и асинхронное программирование 686

Отношения между процессом, доменом приложения, контекстом и потоком	687
Проблема параллелизма	688
Роль синхронизации потоков	688
Краткий обзор делегатов .NET	689
Асинхронная природа делегатов	691
Методы <code>BeginInvoke()</code> и <code>EndInvoke()</code>	691
Интерфейс <code>System.IAsyncResult</code>	691
Асинхронный вызов метода	692
Синхронизация вызывающего потока	693
Роль делегата <code>AsyncCallback</code>	694
Роль класса <code>AsyncResult</code>	696
Передача и получение специальных данных состояния	697
Пространство имен <code>System.Threading</code>	698
Класс <code>System.Threading.Thread</code>	699
Получение статистики о текущем потоке выполнения	700
Свойство <code>Name</code>	700
Свойство <code>Priority</code>	701
Ручное создание вторичных потоков	702
Работа с делегатом <code>ThreadStart</code>	702
Работа с делегатом <code>ParametrizedThreadStart</code>	704
Класс <code>AutoResetEvent</code>	705
Потоки переднего плана и фоновые потоки	706
Проблемы параллелизма	707
Синхронизация с использованием ключевого слова <code>lock</code> языка C#	709
Синхронизация с использованием типа <code>System.Threading.Monitor</code>	711
Синхронизация с использованием типа <code>System.Threading.Interlocked</code>	712
Синхронизация с использованием атрибута <code>[Synchronization]</code>	713
Программирование с использованием обратных вызовов <code>Timer</code>	714
Использование автономного отбрасывания	715
Пул потоков CLR	716
Параллельное программирование с использованием TPL	717
Пространство имен <code>System.Threading.Tasks</code>	718
Роль класса <code>Parallel</code>	718
Обеспечение параллелизма данных с помощью класса <code>Parallel</code>	719
Доступ к элементам пользовательского интерфейса во вторичных потоках	722
Класс <code>Task</code>	723
Обработка запроса на отмену	723
Обеспечение параллелизма задач с помощью класса <code>Parallel</code>	725
Запросы <code>Parallel LINQ (PLINQ)</code>	728
Создание запроса <code>PLINQ</code>	729
Отмена запроса <code>PLINQ</code>	729

Асинхронные вызовы с помощью ключевого слова <code>async</code>	731
Знакомство с ключевыми словами <code>async</code> и <code>await</code> языка C#	731
Соглашения об именовании асинхронных методов	733
Асинхронные методы, возвращающие <code>void</code>	733
Асинхронные методы с множеством контекстов <code>await</code>	734
Вызов асинхронных методов из неасинхронных методов	734
Ожидание с помощью <code>await</code> в блоках <code>catch</code> и <code>finally</code>	734
Обобщенные возвращаемые типы в асинхронных методах (нововведение)	735
Локальные функции (нововведение)	735
Итоговые сведения о ключевых словах <code>async</code> и <code>await</code>	736
Резюме	737
Глава 20. Файловый ввод-вывод и сериализация объектов	738
Исследование пространства имен <code>System.IO</code>	738
Классы <code>Directory</code> (<code>DirectoryInfo</code>) и <code>File</code> (<code>FileInfo</code>)	740
Абстрактный базовый класс <code>FileSystemInfo</code>	740
Работа с типом <code>DirectoryInfo</code>	741
Перечисление файлов с помощью типа <code>DirectoryInfo</code>	742
Создание подкаталогов с помощью типа <code>DirectoryInfo</code>	743
Работа с типом <code>Directory</code>	744
Работа с типом <code>DriveInfo</code>	745
Работа с классом <code>FileInfo</code>	746
Метод <code>FileInfo.Create()</code>	747
Метод <code>FileInfo.Open()</code>	747
Методы <code>FileInfo.OpenRead()</code> и <code>FileInfo.OpenWrite()</code>	749
Метод <code>FileInfo.OpenText()</code>	749
Методы <code>FileInfo.CreateText()</code> и <code>FileInfo.AppendText()</code>	749
Работа с типом <code>File</code>	750
Дополнительные члены <code>File</code>	751
Абстрактный класс <code>Stream</code>	752
Работа с классом <code>FileStream</code>	753
Работа с классами <code>StreamWriter</code> и <code>StreamReader</code>	754
Запись в текстовый файл	755
Чтение из текстового файла	756
Прямое создание объектов типа <code>StreamWriter/StreamReader</code>	756
Работа с классами <code>StringWriter</code> и <code>StringReader</code>	757
Работа с классами <code>BinaryWriter</code> и <code>BinaryReader</code>	758
Программное слежение за файлами	760
Понятие сериализации объектов	762
Роль графов объектов	764
Конфигурирование объектов для сериализации	765
Определение сериализируемых типов	765
Открытые поля, закрытые поля и открытые свойства	766
Выбор формatera сериализации	767
Интерфейсы <code>IFormatter</code> и <code>IRemotingFormatter</code>	767
Точность воспроизведения типов среди форматов	768
Сериализация объектов с использованием <code>BinaryFormatter</code>	769
Десериализация объектов с использованием <code>BinaryFormatter</code>	771
Сериализация объектов с использованием <code>SoapFormatter</code>	771
Сериализация объектов с использованием <code>XmlSerializer</code>	772
Управление генерацией данных XML	773

Сериализация коллекций объектов	775
Настройка процесса сериализации SOAP и двоичной сериализации	776
Углубленный взгляд на сериализацию объектов	776
Настройка сериализации с использованием ISerializable	777
Настройка сериализации с использованием атрибутов	780
Резюме	781
Глава 21. Доступ к данным с помощью ADO.NET	782
Высокоуровневое определение ADO.NET	783
Три грани ADO.NET	783
Поставщики данных ADO.NET	784
Поставщики данных ADO.NET производства Microsoft	785
Получение сторонних поставщиков данных ADO.NET	787
Дополнительные пространства имен ADO.NET	787
Типы из пространства имен System.Data	788
Роль интерфейса IDbConnection	789
Роль интерфейса IDbTransaction	789
Роль интерфейса IDbCommand	790
Роль интерфейсов IDbDataParameter и IDataParameter	790
Роль интерфейсов IDbDataAdapter и IDataAdapter	791
Роль интерфейсов IDataReader и IDataRecord	791
Абстрагирование поставщиков данных с использованием интерфейсов	792
Повышение гибкости с использованием конфигурационных файлов приложения	794
Создание базы данных AutoLot	796
Установка SQL Server 2016 и SQL Server Management Studio	796
Создание таблицы Inventory	796
Добавление тестовых записей в таблицу Inventory	799
Создание хранимой процедуры GetPetName()	799
Создание таблиц Customers и Orders	800
Создание отношений между таблицами	801
Модель фабрики поставщиков данных ADO.NET	803
Полный пример фабрики поставщиков данных	804
Потенциальный недостаток модели фабрики поставщиков данных	807
Элемент <connectionStrings>	808
Понятие подключенного уровня ADO.NET	809
Работа с объектами подключений	809
Работа с объектами ConnectionStringBuilder	811
Работа с объектами команд	812
Работа с объектами чтения данных	814
Получение множества результирующих наборов с использованием объекта чтения данных	815
Работа с запросами создания, обновления и удаления	816
Добавление конструкторов	817
Открытие и закрытие подключения	817
Создание модели автомобиля	818
Добавление методов выборки	818
Вставка новой записи об автомобиле	819
Создание строго типизированного метода InsertCar()	820
Добавление логики удаления	820
Добавление логики обновления	821
Работа с параметризованными объектами команд	821

Указание параметров с использованием типа <code>DbParameter</code>	822
Выполнение хранимой процедуры	823
Создание консольного клиентского приложения	825
Понятие транзакций базы данных	826
Основные члены объекта транзакции ADO.NET	827
Добавление таблицы <code>CreditRisks</code> в базу данных <code>AutoLot</code>	828
Добавление метода транзакции в <code>InventoryDAL</code>	828
Тестирование транзакции базы данных	830
Выполнение массового копирования с помощью ADO.NET	831
Исследование класса <code>SqlBulkCopy</code>	831
Создание специального класса чтения данных	831
Выполнение массового копирования	833
Тестирование массового копирования	834
Резюме	835
Глава 22. Введение в Entity Framework 6	836
Роль Entity Framework	837
Роль сущностей	838
Строительные блоки Entity Framework	839
Роль класса <code>DbContext</code>	839
Аннотации данных Entity Framework	842
Парадигма Code First из существующей базы данных	843
Генерация модели	843
Что мы получили?	846
Изменение стандартных отображений	849
Добавление возможностей в сгенерированные классы моделей	850
Использование классов моделей в коде	851
Вставка данных	851
Выборка записей	852
Роль навигационных свойств	855
Удаление данных	858
Обновление записи	860
Обработка изменений в базе данных	860
Создание уровня доступа к данным <code>AutoLotDAL</code>	861
Добавление классов моделей	862
Обновление класса, производного от <code>DbContext</code>	862
Инициализация базы данных	863
Тестирование сборки <code>AutoLotDAL</code>	865
Миграции Entity Framework	865
Создание начальной миграции	866
Обновление модели	867
Создание финальной миграции	869
Начальное заполнение базы данных	870
Добавление хранилищ для многократного использования кода	871
Добавление интерфейса <code>IRepo</code>	871
Добавление класса <code>BaseRepo</code>	871
Продолжение тестирования <code>AutoLotDAL</code>	874
Вывод инвентаризационных записей	875
Добавление инвентаризационных записей	875
Редактирование записей	875
Удаление записей	875

Параллелизм	876
Перехват	877
Интерфейс IDbCommandInterceptor	877
Добавление перехвата в AutoLotDAL	878
Регистрация перехватчика	879
Добавление перехватчика DatabaseLogger	879
События ObjectMaterialized и SavingChanges	880
Доступ к объектному контексту	880
Событие ObjectMaterialized	880
Событие SavingChanges	880
Отделение модели от уровня доступа к данным	882
Развертывание в SQL Server Express	882
Развертывание в SQL Server Express с использованием миграций	882
Создание сценария миграции	883
Резюме	883
Глава 23. Введение в Windows Communication Foundation	884
Выбор API-интерфейса распределенных вычислений	884
Роль WCF	885
Обзор функциональных возможностей WCF	886
Обзор архитектуры, ориентированной на службы	886
Итоговые сведения об инфраструктуре WCF	888
Исследование основных сборок WCF	888
Шаблоны проектов WCF в Visual Studio	888
Шаблон проекта WCF Service для веб-сайта	890
Базовая структура приложения WCF	891
Адрес, привязка и контракт в WCF	892
Понятие контрактов WCF	892
Понятие привязок WCF	893
Понятие адресов WCF	896
Построение службы WCF	897
Атрибут [ServiceContract]	898
Атрибут [OperationContract]	899
Типы служб как контракты операций	900
Хостинг службы WCF	900
Установка ABC внутри файла App.config	901
Кодирование с использованием типа ServiceHost	902
Указание базовых адресов	903
Подробный анализ типа ServiceHost	904
Подробный анализ элемента <system.serviceModel>	906
Включение обмена метаданными	907
Построение клиентского приложения WCF	909
Генерация кода прокси с использованием svcutil.exe	909
Генерация кода прокси в Visual Studio	910
Конфигурирование привязки на основе TCP	912
Упрощение конфигурационных настроек	914
Использование стандартных конечных точек	914
Открытие доступа к одиночной службе WCF, использующей множество привязок	915
Изменение параметров привязки WCF	916
Использование конфигурации стандартного поведения MEX	918
Обновление клиентского прокси и выбор привязки	919

Использование шаблона проекта WCF Service Library	920
Построение простой математической службы	920
Тестирование службы WCF с помощью WcfTestClient.exe	921
Изменение конфигурационных файлов с использованием SvcConfigEditor.exe	922
Хостинг службы WCF внутри Windows-службы	923
Указание ABC в коде	924
Включение MEX	925
Создание программы установки для Windows-службы	926
Установка Windows-службы	927
Асинхронный вызов службы из клиента	928
Проектирование контрактов данных WCF	930
Использование веб-ориентированного шаблона проекта WCF Service	931
Обновление пакетов NuGet и установка AutoMapper и EntityFramework	933
Реализация контракта службы	934
Роль файла *.svc	934
Исследование файла Web.config	935
Тестирование службы	936
Резюме	936
Часть VII. Windows Presentation Foundation	937
Глава 24. Введение в Windows Presentation Foundation и XAML	938
Мотивация, лежащая в основе WPF	938
Унификация несходных API-интерфейсов	939
Обеспечение разделения обязанностей через XAML	940
Обеспечение оптимизированной модели визуализации	940
Упрощение программирования сложных пользовательских интерфейсов	941
Исследование сборок WPF	942
Роль класса Application	943
Роль класса Window	945
Роль класса System.Windows.Controls.ContentControl	946
Роль класса System.Windows.Controls.Control	947
Роль класса System.Windows.FrameworkElement	947
Роль класса System.Windows.UIElement	948
Роль класса System.Windows.Media.Visual	949
Роль класса System.Windows.DependencyObject	949
Роль класса System.Windows.Threading.DispatcherObject	949
Синтаксис XAML для WPF	949
Введение в Xaml	949
Пространства имен XML и "ключевые слова" XAML	950
Управление видимостью классов и переменных-членов	953
Элементы XAML, атрибуты XAML и преобразователи типов	954
Понятие синтаксиса "свойство-элемент" в XAML	955
Понятие присоединяемых свойств XAML	956
Понятие расширений разметки XAML	957
Построение приложений WPF с использованием Visual Studio	958
Шаблоны проектов WPF	959
Панель инструментов и визуальный конструктор/редактор XAML	959
Установка свойств с использованием окна Properties	960
Обработка событий с использованием окна Properties	962
Обработка событий в редакторе XAML	962
Окно Document Outline	963

Включение и отключение отладчика XAML	964
Исследование файла App.xaml	965
Отображение разметки XAML окна на код C#	966
Роль BAML	967
Разгадывание загадки Main()	968
Взаимодействие с данными уровня приложения	968
Обработка закрытия объекта Window	970
Перехват событий мыши	971
Перехват событий клавиатуры	972
Изучение документации WPF	972
Резюме	973
Глава 25. Элементы управления, компоновки, события и привязка данных в WPF	974
Обзор основных элементов управления WPF	974
Элементы управления Ink API	974
Элементы управления документов WPF	975
Общие диалоговые окна WPF	976
Подробные сведения находятся в документации	976
Краткий обзор визуального конструктора WPF в Visual Studio	976
Работа с элементами управления WPF в Visual Studio	977
Работа с окном Document Outline	978
Управление компоновкой содержимого с использованием панелей	978
Позиционирование содержимого внутри панелей Canvas	979
Позиционирование содержимого внутри панелей WrapPanel	981
Позиционирование содержимого внутри панелей StackPanel	983
Позиционирование содержимого внутри панелей Grid	984
Позиционирование содержимого внутри панелей DockPanel	986
Включение прокрутки в типах панелей	988
Конфигурирование панелей с использованием визуальных конструкторов Visual Studio	988
Построение окна с использованием вложенных панелей	991
Построение системы меню	992
Построение панели инструментов	994
Построение строки состояния	994
Завершение проектирования пользовательского интерфейса	995
Реализация обработчиков событий MouseEnter/MouseLeave	996
Реализация логики проверки правописания	996
Понятие команд WPF	997
Внутренние объекты команд	997
Подключение команд к свойству Command	998
Подключение команд к произвольным действиям	999
Работа с командами Open и Save	1000
Понятие маршрутизируемых событий	1002
Роль пузырьковых маршрутизируемых событий	1003
Продолжение или прекращение пузырькового распространения	1004
Роль туннельных маршрутизируемых событий	1004
Более глубокий взгляд на API-интерфейсы и элементы управления WPF	1006
Работа с элементом управления TabControl	1006
Построение вкладки Ink API	1007
Проектирование панели инструментов	1008
Элемент управления RadioButton	1008

Добавление кнопок сохранения, загрузки и удаления	1009
Добавление элемента управления InkCanvas	1009
Предварительный просмотр окна	1009
Обработка событий для вкладки Ink API	1010
Добавление элементов управления в панель инструментов	1010
Элемент управления InkCanvas	1010
Элемент управления ComboBox	1012
Сохранение, загрузка и очистка данных InkCanvas	1014
Введение в модель привязки данных WPF	1015
Построение вкладки Data Binding	1016
Установка привязки данных	1016
Свойство DataContext	1016
Форматирование привязанных данных	1018
Преобразование данных с использованием интерфейса IValueConverter	1018
Установление привязок данных в коде	1019
Построение вкладки DataGrid	1020
Роль свойств зависимости	1021
Исследование существующего свойства зависимости	1023
Важные замечания относительно оболочек свойств CLR	1025
Построение специального свойства зависимости	1026
Добавление процедуры проверки достоверности данных	1028
Реагирование на изменение свойства	1029
Резюме	1030
Глава 26. Службы визуализации графики WPF	1031
Службы графической визуализации WPF	1031
Варианты графической визуализации WPF	1032
Визуализация графических данных с использованием фигур	1033
Добавление прямоугольников, эллипсов и линий на поверхность Canvas	1035
Удаление прямоугольников, эллипсов и линий с поверхности Canvas	1037
Работа с элементами Polyline и Polygon	1038
Работа с элементом Path	1039
Кисти и перья WPF	1042
Конфигурирование кистей с использованием Visual Studio	1042
Конфигурирование кистей в коде	1045
Конфигурирование перьев	1046
Применение графических трансформаций	1046
Первый взгляд на трансформации	1047
Трансформация данных Canvas	1048
Работа с редактором трансформаций Visual Studio	1050
Построение начальной компоновки	1050
Применение трансформаций на этапе проектирования	1052
Трансформация холста в коде	1052
Визуализация графических данных с использованием рисунков и геометрических объектов	1053
Построение кисти DrawingBrush с использованием геометрических объектов	1054
Рисование с помощью DrawingBrush	1055
Включение типов Drawing в DrawingImage	1056
Работа с векторными изображениями	1056
Преобразование файла с векторной графикой в файл XAML	1057
Импортирование графических данных в проект WPF	1058

Взаимодействие с изображением	1058
Визуализация графических данных с использованием визуального уровня	1059
Базовый класс <code>Visual</code> и производные дочерние классы	1059
Первый взгляд на класс <code>DrawingVisual</code>	1060
Визуализация графических данных в специальном диспетчере компоновки	1062
Реагирование на операции проверки попадания	1064
Резюме	1065
Глава 27. Ресурсы, анимация, стили и шаблоны WPF	1066
Система ресурсов WPF	1066
Работа с двоичными ресурсами	1067
Программная загрузка изображения	1068
Работа с объектными (логическими) ресурсами	1070
Роль свойства <code>Resources</code>	1071
Определение ресурсов уровня окна	1071
Расширение разметки <code>{StaticResource}</code>	1073
Расширение разметки <code>{DynamicResource}</code>	1074
Ресурсы уровня приложения	1074
Определение объединенных словарей ресурсов	1075
Определение сборки, включающей только ресурсы	1076
Службы анимации WPF	1078
Роль классов анимации	1078
Свойства <code>To</code> , <code>From</code> и <code>By</code>	1079
Роль базового класса <code>Timeline</code>	1080
Реализация анимации в коде C#	1080
Управление темпом анимации	1082
Запуск в обратном порядке и циклическое выполнение анимации	1082
Реализация анимации в разметке XAML	1083
Роль раскадровок	1084
Роль триггеров событий	1084
Анимация с использованием дискретных ключевых кадров	1085
Роль стилей WPF	1086
Определение и применение стиля	1087
Переопределение настроек стиля	1087
Влияние атрибута <code>TargetType</code> на стили	1088
Создание подклассов существующих стилей	1089
Определение стилей с триггерами	1090
Определение стилей с множеством триггеров	1090
Анимированные стили	1091
Применение стилей в коде	1091
Логические деревья, визуальные деревья и стандартные шаблоны	1093
Программное инспектирование логического дерева	1093
Программное инспектирование визуального дерева	1095
Программное инспектирование стандартного шаблона элемента управления	1096
Построение шаблона элемента управления с помощью инфраструктуры триггеров	1099
Шаблоны как ресурсы	1100
Встраивание визуальных подсказок с использованием триггеров	1101
Роль расширения разметки <code>{TemplateBinding}</code>	1102
Роль класса <code>ContentPresenter</code>	1103
Встраивание шаблонов в стили	1103
Резюме	1104

Глава 28. Уведомления, проверка достоверности, команды и MVVM	1105
Введение в паттерн MVVM	1105
Модель	1106
Представление	1106
Модель представления	1106
Анемичные модели или анемичные модели представлений	1106
Система уведомлений привязки WPF	1107
Наблюдаемые модели и коллекции	1107
Добавление привязок и данных	1109
Изменение данных об автомобиле в коде	1110
Наблюдаемые модели	1111
Наблюдаемые коллекции	1112
Проверка достоверности WPF	1116
Модификация примера для демонстрации проверки достоверности	1117
Класс Validation	1117
Варианты проверки достоверности	1117
Использование аннотаций данных в WPF	1127
Настройка свойства ErrorTemplate	1129
Итоговые сведения о проверке достоверности	1132
Создание специальных команд	1132
Реализация интерфейса ICommand	1132
Добавление команды ChangeColorCommand	1132
Присоединение команды к CommandManager	1133
Изменение файла MainWindow.xaml.cs	1133
Изменение файла MainWindow.xaml	1134
Тестирование приложения	1134
Объекты RelayCommand	1136
Итоговые сведения о командах	1138
Перенос кода и данных в модель представления	1139
Перенос кода MainWindow.xaml.cs	1139
Обновление кода и разметки MainWindow	1140
Обновление разметки элементов управления	1140
Итоговые сведения о моделях представлений	1141
Изменение сборки AutoLotDAL для MVVM	1141
Обновление класса EntityBase	1141
Обновление частичного класса Inventory	1142
Добавление PropertyChanged.Fody в проект Models	1142
Добавление Entity Framework и строки подключения в проект приложения WPF	1142
Обновление разметки XAML для MainWindow	1142
Обновление модели представления	1142
Обновление команды AddCarCommand	1143
Использование события ObjectMaterialized с инфраструктурой Entity Framework	1143
Резюме	1143
Часть VIII. ASP.NET	1145
Глава 29. Введение в ASP.NET MVC	1146
Введение в паттерн MVC	1146
Модель	1146
Представление	1147
Контроллер	1147
Почему появилась инфраструктура MVC?	1147

Появление ASP.NET MVC	1148
Шаблон приложения ASP.NET MVC	1148
Мастер создания проекта	1148
Окно обзора проекта	1151
Файлы в корневой папке проекта	1151
Файл Global.asax.cs	1151
Папка Models	1152
Папка Controllers	1152
Папка Views	1153
Папки ASP.NET	1153
Папка App_Start	1154
Папка Content	1156
Папка Fonts	1156
Папка Scripts	1157
Обновление пакетов NuGet проекта	1157
Обновление настроек проекта	1157
Маршрутизация	1158
Шаблоны URL	1158
Создание маршрутов для страниц Contact и About	1159
Перенаправление пользователей с применением маршрутизации	1160
Добавление библиотеки AutoLotDAL	1161
Контроллеры и действия	1162
Результаты действий	1162
Добавление контроллера Inventory	1162
Исследование и обновление InventoryController	1165
Механизм представлений Razor	1172
Синтаксис Razor	1172
Операторы, блоки кода и разметка	1173
Встроенные вспомогательные методы HTML	1174
Специальные вспомогательные методы HTML	1175
Функции Razor	1176
Делегаты Razor	1176
Представления MVC	1177
Компоновки	1177
Частичные представления	1178
Отправка данных представлениям	1179
Аннотации данных Display	1180
Специальные файлы метаданных	1180
Шаблоны Razor	1181
Создание специального шаблона для отображения	1181
Создание специального шаблона для редактирования	1183
Работа с формами	1184
Вспомогательный метод HTML по имени BeginForm()	1184
Вспомогательный метод HTML по имени AntiForgeryToken()	1185
Обновление представления Delete	1185
Проверка достоверности	1185
Отображение ошибок	1185
Отключение проверки достоверности клиентской стороны	1186
Проверка достоверности клиентской стороны	1186
Итоговые сведения об инфраструктуре ASP.NET MVC	1187
Резюме	1188

Глава 30. Введение в ASP.NET Web API	1189
Инфраструктура ASP.NET Web API	1189
Создание проекта Web API	1190
Создание класса InventoryController	1191
Маршрутизация	1192
Формат JSON	1194
Результаты действий Web API	1195
Проблема сериализации, связанная с использованием Entity Framework	1196
Получение складских данных	1197
Добавление метода Dispose()	1197
Использование Fiddler	1197
Обновление складской записи (HttpPut)	1199
Добавление складских записей (HttpPost)	1200
Удаление складских записей (HttpDelete)	1201
Итоговые сведения о Web API	1202
Обновление проекта CarLotMVC для использования CarLotWebAPI	1202
Добавление приложения CarLotMVC	1202
Обновление класса InventoryController в проекте MVC	1203
Обновление метода действия Index()	1203
Обновление метода действия Details()	1204
Обновление метода действия Create()	1204
Обновление метода действия Edit()	1205
Обновление метода действия Delete()	1206
Резюме	1207
 Часть IX. .NET Core	 1209
Глава 31. Философия .NET Core	1210
От проекта Project K до .NET Core	1210
Будущее полной платформы .NET Framework	1211
Цели .NET Core	1211
Межплатформенная поддержка	1212
Производительность	1213
Переносимые библиотеки классов, соответствующие стандарту .NET Standard	1213
Переносимые или автономные модели развертывания	1214
Полная поддержка командной строки	1214
Открытый код	1214
Возможность взаимодействия с .NET Framework	1214
Состав .NET Core	1215
Исполняющая среда .NET Core (CoreCLR)	1215
Библиотеки инфраструктуры (CoreFX)	1215
Инструменты SDK и хост приложений dotnet	1215
Компиляторы языков	1216
Жизненный цикл поддержки .NET Core	1216
Установка .NET Core 2.0	1217
Сравнение с полной платформой .NET Framework	1218
Сниженное количество поддерживаемых моделей приложений	1218
Меньшее число реализованных API-интерфейсов и подсистем	1218
Резюме	1219

Глава 32. Введение в Entity Framework Core	1220
Сравнение наборов функциональных возможностей	1220
Средства, которые не дублировались	1221
Изменения по сравнению с EF 6	1221
Новые функциональные средства в EF Core	1221
Сценарии использования	1222
Создание AutoLotCoreDAL_Core2	1223
Создание проектов и решения	1223
Добавление пакетов NuGet	1223
Добавление классов моделей	1224
Создание класса AutoLotContext	1227
Создание базы данных с помощью миграций	1232
Инициализация базы данных начальной информацией	1233
Добавление хранилищ для многократного использования кода	1236
Добавление интерфейса IRepo	1237
Добавление класса BaseRepo	1237
Создание InventoryRepo	1240
Тестирование библиотеки AutoLotDAL_Core2	1242
Резюме	1243
Глава 33. Введение в веб-приложения ASP.NET Core	1244
Шаблон ASP.NET Core Web Application	1244
Мастер создания проекта	1244
Организация проекта ASP.NET Core	1246
Добавление библиотеки доступа к данным	1246
Обновление пакетов NuGet	1247
Запуск приложений ASP.NET Core	1247
Развертывание приложений ASP.NET Core	1249
Нововведения ASP.NET Core	1249
Унифицированный подход к построению веб-приложений и служб	1250
Встроенное внедрение зависимостей	1250
Система конфигурации, основанная на среде и готовая к взаимодействию с облачными технологиями	1251
Способность выполняться под управлением .NET Core или полной платформы .NET Framework	1253
Легковесный и модульный конвейер запросов HTTP	1253
Интеграция современных инфраструктур клиентской стороны	1253
Вспомогательные функции дескрипторов	1253
Компоненты представлений	1263
Изменения механизма представлений Razor	1264
Построение AutoLotMVC_Core2	1264
Файл Program.cs	1264
Файл Startup.cs	1266
Управление пакетами с помощью Bower	1269
Объединение в пакеты и минификация	1270
Содержимое клиентской стороны (папка wwwroot)	1274
Папки для моделей, контроллеров и представлений	1274
Контроллеры и действия	1275
Базовый класс Controller	1275
Действия	1275
Типы ViewResult	1277

Добавление контроллера Inventory	1277
Обновление класса InventoryController	1279
Итоговые сведения о контроллерах и действиях	1283
Представления	1283
Обновление файла импортирования представлений	1283
Представление компоновки	1284
Частичное представление со сценариями проверки достоверности	1285
Шаблон для отображения складских данных	1285
Шаблон для редактирования складских данных	1286
Представление Index	1287
Представление Details	1288
Представление Create	1288
Представление Edit	1289
Представление Delete	1290
Итоговые сведения о представлениях	1290
Компоненты представлений	1290
Написание кода серверной стороны	1291
Написание кода клиентской стороны	1292
Обращение к компонентам представлений	1293
Обращение к компонентам представлений как к специальным вспомогательным функциям дескрипторов	1293
Добавление компонента представления в AutoLotMVC_Core2	1293
Резюме	1294
Глава 34. Введение в приложения служб ASP.NET Core	1295
Шаблон ASP.NET Core Web API	1295
Мастер создания проекта	1295
Организация проекта приложения службы ASP.NET Core	1297
Добавление библиотеки доступа к данным	1297
Обновление и добавление пакетов NuGet	1297
Запуск и развертывание приложений служб	1297
Изменения в приложениях служб ASP.NET Core	1297
Формат возвращаемых данных JSON	1297
Явная маршрутизация для методов HTTP	1298
Управление привязкой моделей в ASP.NET Core	1298
Построение AutoLotAPI_Core2	1299
Добавление строки подключения	1299
Обновление файла Program.cs для инициализации данных	1299
Обновление файла Startup.cs	1300
Добавление обработки исключений на уровне приложений	1301
Добавление контроллера Inventory	1303
Итоговые сведения о приложениях служб ASP.NET Core	1308
Обновление AutoLotMVC_Core2 для использования AutoLotAPI_Core2	1309
Копирование и добавление приложения AutoLotMVC_Core2	1309
Удаление AutoLotDAL_Core2 из AutoLotMVC_Core2	1309
Создание нового контроллера Inventory	1310
Итоговые сведения о миграции AutoLotMVC_Core2	1315
Резюме	1315
Предметный указатель	1316

*Всему роду Троелсен: Мэри (маме), Уолтеру (папе), Мэнди (жене)
и Сорену (сыну). Нам не хватает тебя, Микко (кот).*

Эндрю

*Моей семье, Эми (жене), Коннеру (сыну), Логану (сыну) и Скайлер
(дочери). Спасибо за поддержку и терпение с вашей стороны.*

Филипп

Об авторах

Эндрю Троелсен обладает более чем 20-летним опытом работы в индустрии программного обеспечения (ПО). На протяжении этого времени он выступал в качестве разработчика, преподавателя, автора, публичного докладчика и руководителя команды, а теперь является менеджером платформы больших данных в компании Thomson Reuters. Он был автором многочисленных книг, посвященных миру Microsoft, в которых раскрывалась разработка для COM на языке C++ с помощью ATL, COM и взаимодействия с .NET, а также разработка на языках Visual Basic и C# с применением платформы .NET. Эндрю Троелсен получил степень магистра в области разработки ПО (MSSE) в Университете Святого Томаса и трудится над получением второй степени магистра по математической лингвистике (CLMS) в Вашингтонском университете.

Филипп Джепикс — международный докладчик, обладатель званий Microsoft MVP, ASPInsider, MCSO, CSM и CSP, а также активный участник сообщества разработчиков. Фил имел дело еще с самыми первыми бета-версиями платформы .NET, разрабатывая ПО свыше 30 лет, и с 2005 года интенсивно вовлечен в сообщество гибкой разработки. Он является ведущим руководителем группы пользователей .NET в Цинциннати (www.cinnug.org), основанной на конференции “День гибкой разработки Цинциннати” (www.dayofagile.org), и волонтером Национального лыжного патруля. Фил также выступает автором в LinkedIn Learning (<https://www.lynda.com/Phil-Japikse/7908546-1.html>). В течение дня он работает консультантом и наставником по гибкой методологии для крупных и средних фирм повсеместно в США. Фил любит изучать новые технологии и постоянно стремится совершенствовать свои навыки. Вы можете отслеживать Филя в Твиттере по адресу [www.twitter.com/skimedic](https://twitter.com/skimedic) и читать его блог на www.skimedic.com/blog.

О технических рецензентах

Эрик Поттер — архитектор ПО в Aptera Software и обладатель звания Microsoft MVP для Visual Studio и технологий разработки. Он имеет дело преимущественно с веб-платформой .NET, но ему нравится, когда появляются возможности опробовать другие технологические стеки. Эрик разрабатывал высококачественные специальные программные решения, начиная с 2001 года. В компании Aptera Software он успешно поставлял решения клиентам из широкого спектра отраслей. В свободное время Эрик возится с проектами Arduino. Он с любовью вспоминает, как ему нравилось разрабатывать ПО для Palm OS. У Эрика есть изумительная жена и пятеро замечательных детей. Его блог доступен по адресу <http://humbletoolsmith.com/>, и Эрика можно также отслеживать в Твиттере как @pottereric.

После почти двух десятилетий профессионального написания ПО (и несколько лет до этого непрофессионального) **Ли Брандт** продолжает ежедневно учиться. Он руководил командами в небольших и крупных компаниях и всегда умудрялся поддерживать

потребности бизнеса на переднем крае усилий по разработке ПО. Ли выступал на международных конференциях по разработке ПО, делая доклады как с технической, так и экономической точки зрения, и любит обучать других тому, что знает сам. Он создавал ПО главным образом на Objective-C, JavaScript и C#. Ли обладает званием Microsoft Most Valuable Professional для Visual Studio и технологий разработки и является одним из руководителей Конференции разработчиков Канзас-Сити (Kansas City Developer Conference — KCDC). Он также награжденный ветеран войны в Персидском заливе, супруг и опытный владелец домашних животных, в свободное от работы время увлекающийся игрой на барабанах.

Шон Уайтселл — разработчик ПО в городе Талса (штат Оклахома) с более чем 17-летним опытом разработки клиент-серверных, веб- и встраиваемых приложений, а также электроники. Он является президентом группы пользователей .NET в Талсе (Tulsa .NET User Group) и часто участвует в региональных группах пользователей и конференциях. Шон страстно увлекается программным решением задач, мастерски кодируя и обучая этому других. Кроме того, он капеллан и звукооператор в своей церкви, а также учитель на курсах самозащиты для детей.

Благодарности

Как всегда, я хочу искренне поблагодарить всю команду издательства Apress. Мне повезло работать с Apress при написании многочисленных книг, начиная с 2001 года. Помимо подготовки высококачественного технического материала персонал великолепен, и без него эта книга не увидела бы свет. Спасибо вам всем!

Я также выражаю благодарность своему соавтору Филиппу Джепиксу. Спасибо, Фил, за напряженную работу по поддержанию той же самой степени доступности текста книги наряду добавлением персонального опыта и мнения. Я уверен, что наша книга и ее читатели непременно выиграют от такого партнерства!

Последнее, но от того не менее важное: я благодарю свою жену Мэнди и сына Сорена за поддержку.

Эндрю Троелсен

Я также хочу поблагодарить издательство Apress и всю команду, вовлеченную в работу над данной книгой. Как я и ожидал в отношении всех книг, издаваемых в Apress, меня впечатлил тот уровень поддержки, который мы получали в процессе написания. Я благодарю вас, читатель, и надеюсь, что наша книга окажется полезной в вашей карьере, как было в моем случае. Наконец, я не сумел бы сделать эту работу без моей семьи и поддержки, которую я получил от них. Без вашего понимания, сколько времени занимает написание и вычитывание, мне никогда не удалось бы завершить работу! Люблю вас всех!

Филипп Джепикс

Введение

Авторы и читатели — одна команда

Авторам книг по технологиям приходится писать для очень требовательной группы людей (по вполне понятным причинам). Вам известно, что построение программных решений с применением любой платформы или языка исключительно сложно и специфично для отдела, компании, клиентской базы и поставленной задачи. Возможно, вы работаете в индустрии электронных публикаций, разрабатываете системы для правительства или местных органов власти либо сотрудничаете с NASA или какой-то военной отраслью. Вместе мы трудимся в нескольких отраслях, включая разработку обучающего ПО для детей (Oregon Trail/Amazon Trail), разнообразных производственных систем и проектов в медицинской и финансовой сферах. Написанный вами код на месте вашего трудоустройства почти на 100% будет иметь мало общего с кодом, который мы создавали на протяжении многих лет.

По указанной причине в книге мы намеренно решили избегать демонстрации примеров кода, свойственного какой-то конкретной отрасли или направлению программирования. Таким образом, мы объясняем язык C#, объектно-ориентированное программирование, среду CLR и библиотеки базовых классов .NET с использованием примеров, не привязанных к отрасли. Вместо того чтобы заставлять каждый пример наполнять сетку данными, подчитывать фонд заработной платы или делать еще что-нибудь подобное, мы придерживаемся темы, близкой каждому из нас: автомобили (с добавлением умеренного количества геометрических структур и систем расчета заработной платы для сотрудников). И вот тут наступает ваш черед.

Наша работа заключается в как можно лучшем объяснении языка программирования C# и основных аспектов платформы .NET. Мы также будем делать все возможное для того, чтобы снарядить вас инструментами и стратегиями, которые необходимы для продолжения обучения после завершения работы с данной книгой.

Ваша работа предусматривает усвоение этой информации и ее применение к решению своих задач программирования. Мы полностью отдаем себе отчет, что ваши проекты с высокой вероятностью не будут связаны с автомобилями и их дружественными именами, но именно в том и состоит суть прикладных знаний.

Мы уверены, что после освоения тем и концепций, представленных в настоящей книге, вы сможете успешно строить решения .NET, которые соответствуют вашей конкретной среде программирования.

Краткий обзор книги

Книга логически разделена на девять частей, каждая из которых содержит несколько связанных друг с другом глав. Ниже приведено краткое содержание частей и глав.

Часть I. Введение в C# и платформу .NET

Эта часть книги предназначена для ознакомления с природой платформы .NET и различными инструментами разработки, которые используются во время построения приложений .NET.

Глава 1. Философия .NET

Первая глава выступает в качестве основы для всего остального материала. Ее главная цель в том, чтобы представить вам набор строительных блоков .NET, таких как общезыковая исполняющая среда (Common Language Runtime), общая система типов (Common Type System), общезыковая спецификация (Common Language Specification) и библиотеки базовых классов. Здесь вы впервые взглянете на язык программирования C# и формат сборок .NET. Кроме того, будет исследована независимая от платформы природа .NET.

Глава 2. Создание приложений на языке C#

Целью этой главы является введение в процесс компиляции файлов исходного кода C#. Здесь вы узнаете о совершенно бесплатной (но полнофункциональной) редакции Visual Studio Community, которая главным образом задействована в книге, а также о редакциях Professional и Enterprise продукта Visual Studio 2017. Вы научитесь конфигурировать машину разработки с применением нового процесса установки Visual Studio, основанного на рабочих нагрузках, включать в своих проектах средства C# 7.1 и устанавливать все важные инфраструктуры .NET 4.7 и .NET Core 2.0.

Часть II. Основы программирования на C#

Темы, представленные в этой части книги, очень важны, потому что они связаны с разработкой программного обеспечения .NET любого типа (например, веб-приложений, настольных приложений с графическим пользовательским интерфейсом, библиотек кода или служб Windows). Здесь вы узнаете о фундаментальных типах данных .NET, освоите манипулирование текстом и ознакомитесь с ролью модификаторов параметров C# (включая необязательные и именованные аргументы).

Глава 3. Главные конструкции программирования на C#: часть I

В этой главе начинается формальное исследование языка программирования C#. Здесь вы узнаете о роли метода `Main()` и многочисленных деталях, касающихся внутренних типов данных платформы .NET и объявления переменных, а также научитесь работать и манипулировать текстовыми данными, используя типы `System.String` и `System.Text.StringBuilder`. Кроме того, вы исследуете итерационные конструкции и конструкции принятия решений, сужающие и расширяющие операции и ключевое слово `unchecked`.

Глава 4. Главные конструкции программирования на C#: часть II

В этой главе завершается исследование ключевых аспектов C#, начиная с создания и манипулирования массивами данных. Затем вы узнаете, как конструировать перегруженные методы типов и определять параметры с применением ключевых слов `out`, `ref` и `params`. Также вы изучите тип перечисления, структуры и типы, допускающие `null`, плюс уясните отличие между типами значений и ссылочными типами. Наконец, вы освоите кортежи — новое средство версии C# 7.

Часть III. Объектно-ориентированное программирование на C#

В этой части вы изучите основные конструкции языка C#, включая детали объектно-ориентированного программирования. Здесь вы также научитесь обрабатывать исключения времени выполнения и взаимодействовать со строго типизированными интерфейсами.

Глава 5. Инкапсуляция

В этой главе начинается рассмотрение концепций объектно-ориентированного программирования (ООП) на языке C#. После представления главных принципов ООП (инкапсуляции, наследования и полиморфизма) будет показано, как строить надежные типы классов с использованием конструкторов, свойств, статических членов, констант и полей только для чтения. Глава завершается исследованием частичных определений типов, синтаксиса инициализации объектов и автоматических свойств.

Глава 6. Наследование и полиморфизм

Здесь вы ознакомитесь с оставшимися главными принципами ООП (наследованием и полиморфизмом), которые позволяют создавать семейства связанных типов классов. Вы узнаете о роли виртуальных и абстрактных методов (и абстрактных базовых классов), а также о природе полиморфных интерфейсов. Затем вы исследуете сопоставление с образцом — нововведение C# 7. Наконец, в главе будет объясняться роль главного базового класса платформы .NET — `System.Object`.

Глава 7. Структурированная обработка исключений

В этой главе обсуждаются способы обработки в кодовой базе аномалий, возникающих во время выполнения, за счет использования структурированной обработки исключений. Вы узнаете не только о ключевых словах C#, которые дают возможность решать такие задачи (`try`, `catch`, `throw`, `when` и `finally`), но и о разнице между исключениями уровня приложения и уровня системы. Вдобавок в главе будут исследованы разнообразные инструменты внутри Visual Studio, которые позволяют отлаживать исключения, оставшиеся без внимания.

Глава 8. Работа с интерфейсами

Материал этой главы опирается на ваше понимание объектно-ориентированной разработки и посвящен программированию на основе интерфейсов. Здесь вы узнаете, каким образом определять классы и структуры, поддерживающие несколько линий поведения, обнаруживать такие линии поведения во время выполнения и выборочно скрывать какие-то из них с применением явной реализации интерфейсов. В дополнение к созданию специальных интерфейсов вы научитесь реализовывать стандартные интерфейсы, доступные внутри платформы .NET, и использовать их для построения объектов, которые могут сортироваться, копироваться, перечисляться и сравниваться.

Часть IV. Дополнительные конструкции программирования на C#

В этой части книги вы углубите знания языка C# за счет исследования нескольких более сложных (и важных) концепций. Здесь вы завершите ознакомление с системой типов .NET, изучив интерфейсы и делегаты. Вы также узнаете о роли обобщений, кратко взглянете на язык LINQ (Language Integrated Query) и освоите ряд более сложных средств C# (например, методы расширения, частичные методы, манипулирование указателями и жизненный цикл объектов).

Глава 9. Коллекции и обобщения

В этой главе исследуется тема обобщений. Вы увидите, что обобщенное программирование предлагает способ создания типов и членов типов, которые содержат заполнители, указываемые вызывающим кодом. По существу обобщения значительно улучшают производительность приложений и безопасность в отношении типов. Здесь не только описаны разнообразные обобщенные типы из пространства имен `System.Collections.Generic`, но также показано, каким образом строить собственные обобщенные методы и типы (с ограничениями и без).

Глава 10. Делегаты, события и лямбда-выражения

Целью этой главы является прояснение типа делегата. Выражаясь просто, делегат .NET представляет собой объект, который указывает на определенные методы в приложении. С помощью делегатов можно создавать системы, которые позволяют многочисленным объектам участвовать в двухстороннем взаимодействии. После исследования способов применения делегатов .NET вы ознакомитесь с ключевым словом `event` языка C#, которое упрощает манипулирование низкоуровневыми делегатами в коде. В завершение вы узнаете о роли лямбда-операции C# (`=>`), а также о связи между делегатами, анонимными методами и лямбда-выражениями.

Глава 11. Расширенные средства языка C#

В этой главе вы сможете углубить понимание языка C# за счет исследования нескольких расширенных приемов программирования. Здесь вы узнаете, как перегружать операции и создавать специальные процедуры преобразования (явного и неявного) для типов. Вы также научитесь строить и взаимодействовать с индексаторами типов и работать с расширяющими методами, анонимными типами, частичными методами и указателями C#, используя контекст небезопасного кода.

Глава 12. LINQ to Objects

В этой главе начинается исследование языка интегрированных запросов (LINQ). Язык LINQ дает возможность строить строго типизированные выражения запросов, которые могут применяться к многочисленным целевым объектам LINQ для манипулирования данными в самом широком смысле этого слова. Здесь вы изучите API-интерфейс LINQ to Objects, который позволяет применять выражения LINQ к контейнерам данных (например, массивам, коллекциям и специальным типам). Приведенная в главе информация будет полезна позже в книге при рассмотрении других API-интерфейсов.

Глава 13. Время существования объектов

В финальной главе этой части исследуется управление памятью средой CLR с использованием сборщика мусора .NET. Вы узнаете о роли корневых элементов приложения, поколений объектов и типа `System.GC`. После представления основ будут рассматриваться темы освобождаемых объектов (реализующих интерфейс `IDisposable`) и процесса финализации (с применением метода `System.Object.Finalize()`). В главе также описан класс `Lazy<T>`, позволяющий определять данные, которые не будут размещаться в памяти вплоть до поступления запроса со стороны вызывающего кода. Вы увидите, что такая возможность очень полезна, когда нежелательно загромождать кучу объектами, которые в действительности программе не нужны.

Часть V. Программирование с использованием сборок .NET

Эта часть книги посвящена деталям формата сборок .NET. Здесь вы узнаете не только о том, как развертывать и конфигурировать библиотеки кода .NET, но также о внутреннем устройстве двоичного образа .NET. Будет описана роль атрибутов .NET и распознавания информации о типе во время выполнения. Кроме того, объясняется роль исполняющей среды динамического языка (Dynamic Language Runtime — DLR) и ключевого слова `dynamic` языка C#. Наконец, рассматриваются более сложные темы, касающиеся сборок, такие как домены приложений, синтаксис языка CIL и построение сборки в памяти.

Глава 14. Построение и конфигурирование библиотек классов

На самом высоком уровне термин “сборка” применяется для описания двоичного файла *.dll или *.exe, созданного с помощью компилятора .NET. Однако в действительности понятие сборки намного шире. Здесь будет показано, чем отличаются однофайловые и многофайловые сборки, как создавать и развертывать сборки обеих разновидностей. Также объясняется, каким образом конфигурировать закрытые и разделяемые сборки с помощью XML-файлов *.config и специальных сборок политик издателя. По ходу дела раскрывается внутренняя структура глобального кеша сборки (Global Assembly Cache — GAC).

Глава 15. Рефлексия типов, позднее связывание и программирование на основе атрибутов

В этой главе продолжается исследование сборок .NET. Здесь будет показано, как обнаруживать типы во время выполнения с использованием пространства имен `System.Reflection`. Посредством типов из упомянутого пространства имен можно строить приложения, способные считывать метаданные сборки на лету. Вы также узнаете, как загружать и создавать типы динамически во время выполнения с применением позднего связывания. Напоследок в главе обсуждается роль атрибутов .NET (стандартных и специальных). Для закрепления материала в главе демонстрируется построение расширяемого консольного приложения.

Глава 16. Динамические типы и среда DLR

В версии .NET 4.0 появился новый аспект исполняющей среды .NET, который называется *исполняющей средой динамического языка* (DLR). Используя DLR и ключевое слово `dynamic` языка C#, можно определять данные, которые в действительности не будут распознаваться вплоть до времени выполнения. Такие средства существенно упрощают решение ряда сложных задач программирования для .NET. В этой главе вы ознакомитесь со сценариями применения динамических данных, включая использование API-интерфейсов рефлексии .NET и взаимодействие с унаследованными библиотеками COM с минимальными усилиями.

Глава 17. Процессы, домены приложений и объектные контексты

Базируясь на хорошем понимании вами сборки, в этой главе подробно раскрывается внутреннее устройство загруженной исполняемой сборки .NET. Целью главы является иллюстрация отношений между процессами, доменами приложений и контекстными границами. Упомянутые темы формируют основу для главы 19, где будет исследоваться конструирование многопоточных приложений.

Глава 18. Язык CIL и роль динамических сборок

Последняя глава в этой части преследует двойную цель. В первой половине главы рассматривается синтаксис и семантика языка CIL, а во второй — роль пространства имен `System.Reflection.Emit`. Типы из указанного пространства имен можно применять для построения программного обеспечения, которое способно генерировать сборки .NET в памяти во время выполнения. Формально сборки, которые определяются и выполняются в памяти, называются *динамическими сборками*.

Часть VI. Введение в библиотеки базовых классов .NET

К настоящему моменту вы уже должны хорошо ориентироваться в языке C# и в подробностях формата сборки .NET. В данной части книги ваши знания расширяются исследованием нескольких часто используемых служб, которые можно обнаружить внутри библиотек базовых классов .NET, включая создание многопоточных приложений, файловый ввод-вывод и доступ к базам данных посредством ADO.NET. Здесь также раскрывается процесс создания распределенных приложений с применением Windows Communication Foundation (WCF) и API-интерфейс Linq to XML.

Глава 19. Многопоточное, параллельное и асинхронное программирование

Эта глава посвящена построению многопоточных приложений. В ней демонстрируются приемы, которые можно использовать для написания кода, безопасного к потокам. Глава начинается с краткого напоминания о том, что собой представляет тип делегата .NET, и объяснения внутренней поддержки делегата для асинхронного вызова методов. Затем рассматриваются типы из пространства имен `System.Threading` и библиотека параллельных задач (Task Parallel Library — TPL). С применением TPL разработчики .NET могут строить приложения, которые распределяют рабочую нагрузку по всем доступным процессорам в исключительно простой манере. В главе также раскрыта роль API-интерфейса Parallel Linq, который предлагает способ создания запросов Linq, масштабируемых среди множества процессорных ядер. В завершение главы исследуется создание неблокирующих вызовов с использованием ключевых слов `async/await`, введенных в версии C# 5, а также локальных функций и обобщенных возвращаемых типов `async`, появившихся в версии C# 7.

Глава 20. Файловый ввод-вывод и сериализация объектов

Пространство имен `System.IO` позволяет взаимодействовать со структурой файлов и каталогов машины. В этой главе вы узнаете, как программно создавать (и удалять) систему каталогов. Вы также научитесь перемещать данные между различными потоками (например, файловыми, строковыми и находящимися в памяти). Кроме того, в главе рассматриваются службы сериализации объектов платформы .NET. Сериализация позволяет сохранять состояние объекта (или набора связанных объектов) в потоке для последующего использования. Десериализация представляет собой процесс извлечения объекта из потока в память с целью потребления внутри приложения. После описания основ вы освоите настройку процесса сериализации с применением интерфейса `ISerializable` и набора атрибутов .NET.

Глава 21. Доступ к данным с помощью ADO.NET

В этой первой из двух глав, посвященных работе с базами данных с использованием полной платформы .NET Framework, представлено введение в API-интерфейс доступа к базам данных платформы .NET, который называется ADO.NET. В частности, здесь рас-

сматривается роль поставщиков данных .NET и взаимодействие с реляционной базой данных с применением подключенного уровня ADO.NET, который представлен объектами подключений, объектами команд, объектами транзакций и объектами чтения данных.

Глава 22. Введение в Entity Framework 6

В этой главе изучение взаимодействия с базами данных завершается рассмотрением роли Entity Framework (EF) 6. Инфраструктура EF представляет собой систему объектно-реляционного отображения (object-relational mapping — ORM), которая предлагает способ написания кода доступа к данным с использованием строго типизированных классов, напрямую отображаемых на бизнес-модель. Здесь вы узнаете о роли класса `DbContext` из EF, применении аннотаций данных и интерфейса `Fluent API` для формирования базы данных, реализации хранилищ для инкапсуляции общего кода, транзакциях, миграциях, проверке параллелизма и перехвате команд. Вдобавок вы научитесь взаимодействовать с реляционными базами данных с помощью `LINQ to Entities`. В главе также создается специальная библиотека доступа к данным (`AutoLotDAL.dll`), которая будет использоваться в нескольких оставшихся главах книги.

Глава 23. Введение в Windows Communication Foundation

До этого места в книге все примеры приложений запускались на единственном компьютере. В настоящей главе вы ознакомитесь с API-интерфейсом `Windows Communication Foundation (WCF)`, который позволяет создавать распределенные приложения в симметричной манере независимо от лежащих в их основе низкоуровневых деталей. Здесь будет объясняться конструкция служб, хостов и клиентов WCF, также применение конфигурационных файлов на основе XML для декларативного указания адресов, привязок и контрактов.

Часть VII. Windows Presentation Foundation

Первоначальный API-интерфейс для построения графических пользовательских интерфейсов настольных приложений, поддерживаемый платформой .NET, назывался `Windows Forms`. Хотя он по-прежнему доступен в полной платформе .NET Framework, в версии .NET 3.0 программистам был предложен замечательный API-интерфейс под названием `Windows Presentation Foundation (WPF)`. В отличие от `Windows Forms` эта высокопроизводительная инфраструктура для построения пользовательских интерфейсов объединяет несколько основных служб, включая привязку данных, двумерную и трехмерную графику, анимацию и форматированные документы, в единую унифицированную модель. Все это достигается с использованием декларативной грамматики разметки, которая называется расширяемым языком разметки приложений (`Extendable Application Markup Language — XAML`). Более того, архитектура элементов управления WPF предлагает легкий способ радикального изменения внешнего вида и поведения типового элемента управления с применением всего лишь правильно оформленной разметки XAML.

Глава 24. Введение в Windows Presentation Foundation и XAML

Эта глава начинается с исследования мотивации создания WPF (с учетом того, что в .NET уже существовала инфраструктура для разработки графических пользовательских интерфейсов настольных приложений). Затем вы узнаете о синтаксисе XAML и ознакомитесь с поддержкой построения приложений WPF в `Visual Studio`.

Глава 25. Элементы управления, компоновки, события и привязка данных в WPF

В этой главе будет показано, как работать с элементами управления и диспетчерами компоновки, предлагаемыми WPF. Вы узнаете, каким образом создавать системы меню, окна с разделителями, панели инструментов и строки состояния. Также в главе рассматриваются API-интерфейсы (и связанные с ними элементы управления), входящие в состав WPF, в том числе Ink API, команды, маршрутизируемые события, модель привязки данных и свойства зависимости.

Глава 26. Службы визуализации графики WPF

Инфраструктура WPF является API-интерфейсом, интенсивно использующим графику, и с учетом этого WPF предоставляет три подхода к визуализации графических данных: фигуры, рисунки и геометрические объекты, а также визуальные объекты. В настоящей главе вы ознакомитесь с каждым подходом и попутно изучите несколько важных графических примитивов (например, кисти, перья и трансформации). Кроме того, вы узнаете, как встраивать векторные изображения в графику WPF, а также выполнять операции проверки попадания в отношении графических данных.

Глава 27. Ресурсы, анимация, стили и шаблоны WPF

В этой главе освещены важные (и взаимосвязанные) темы, которые позволят углубить знания API-интерфейса WPF. Первым делом вы изучите роль логических ресурсов. Система логических ресурсов (также называемых *объектными ресурсами*) предлагает способ именования и ссылки на часто используемые объекты внутри приложения WPF. Затем вы узнаете, каким образом определять, выполнять и управлять анимационной последовательностью. Вы увидите, что применение анимации WPF не ограничивается видеоиграми или мультимедиа-приложениями. В завершение главы вы ознакомитесь с ролью стилей WPF. Подобно веб-странице, использующей CSS или механизм тем ASP.NET, приложение WPF может определять общий вид и поведение для целого набора элементов управления.

Глава 28. Уведомления, проверка достоверности, команды и MVVM

Эта глава начинается с исследования трех основных возможностей инфраструктуры WPF: уведомлений, проверки достоверности и команд. В разделе, в котором рассматриваются уведомления, вы узнаете о наблюдаемых моделях и коллекциях, а также о том, как они поддерживают данные приложения и пользовательский интерфейс в синхронизированном состоянии. Затем вы научитесь создавать специальные команды для инкапсуляции кода. В разделе, посвященном проверке достоверности, вы ознакомитесь с несколькими механизмами проверки достоверности, которые доступны для применения в приложениях WPF. Глава завершается исследованием паттерна “модель-представление-модель представления” (Model View ViewModel — MVVM) и созданием приложения, демонстрирующего паттерн MVVM в действии.

Часть VIII. ASP.NET

Эта часть посвящена построению веб-приложений с использованием API-интерфейса ASP.NET. Легковесная инфраструктура ASP.NET MVC задействует паттерн “модель-представление-контроллер” (Model-View-Controller) и предназначена для создания веб-приложений. Версия ASP.NET Web API 2.2 основана (и похожа) на ASP.NET MVC и является инфраструктурой для разработки веб-служб REST.

Глава 29. Введение в ASP.NET MVC

В этой главе рассматривается ASP.NET MVC. Инфраструктура ASP.NET MVC основана на паттерне MVC, после освоения которого вы приступите к построению приложения MVC. Вы узнаете о шаблонах, маршрутизации, контроллерах, действиях и представлениях. Затем будет создано приложение ASP.NET MVC с применением уровня доступа к данным, разработанного в главе 22.

Глава 30. Введение в ASP.NET Web API

В этой главе вы постройте службу REST, используя ASP.NET Web API 2.2. Служба будет поддерживать операции создания, чтения, обновления и удаления (create, read, update, delete — CRUD) в отношении складских данных с применением уровня доступа к данным, созданного в главе 22. Наконец, вы обновите приложение ASP.NET MVC для использования службы REST в качестве уровня доступа к данным.

Часть IX. .NET Core

Эта часть посвящена .NET Core — межплатформенной версии .NET. После общего ознакомления с .NET Core, мотивацией, а также отличиями между .NET Core и полной платформой .NET Framework уровень доступа к данным `AutoLotDAL` будет создан заново уже с применением `Entity Framework Core`. В последних двух главах части рассматривается построение веб-приложений и служб REST с помощью инфраструктуры ASP.NET Core.

Глава 31. Философия .NET Core

В этой главе предлагается введение в .NET Core — кардинально новую межплатформенную версию .NET. Вы узнаете о целях .NET Core, различных ее частях (вроде `CoreCLR` и `CoreFX`) и поддержке жизненного цикла .NET Core. После установки (и проверки) .NET Core будет приведено сравнение .NET Core с полной платформой .NET Framework.

Глава 32. Введение в Entity Framework Core

В этой главе рассматривается версия инфраструктуры `Entity Framework` в .NET Core. Хотя многие концепции EF сохранили свою актуальность, между EF 6 и EF Core существует несколько значительных и важных отличий. Глава начинается со сравнения инфраструктур EF 6 и EF Core и продолжается созданием библиотеки `AutoLotDAL_Core2` — версии EF Core уровня доступа к данным, построенного в главе 22. Обновленный уровень доступа к данным будет использоваться в оставшихся главах книги.

Глава 33. Введение в веб-приложения ASP.NET Core

Это первая из двух глав, посвященных ASP.NET Core, в которой строятся веб-приложения в стиле MVC. В начале главы с помощью нового шаблона `ASP.NET Core Web Application` (Веб-приложение ASP.NET Core) будет создано приложение `AutoLotMVC_Core2`. Затем рассматриваются нововведения ASP.NET Core (в сравнении с ASP.NET MVC 5), включая поддержку внедрения зависимостей, новую систему конфигурации, осведомленность о среде, вспомогательные функции дескрипторов и компоненты представлений. В конце главы будет построена версия ASP.NET Core приложения `AutoLotMVC` (из главы 29) с применением `AutoLotDAL_Core2` в качестве уровня доступа к данным.

Глава 34. Введение в приложения служб ASP.NET Core

Данная глава завершает исследование ASP.NET Core построением службы REST с использованием ASP.NET Core. Вместо того чтобы быть отдельными (хотя похожими) инфраструктурами, службы и веб-приложения в ASP.NET Core применяют одну и ту же кодовую базу, в итоге совместно используя MVC и Web API. Подобно службе, созданной в главе 30, служба `AutoLotAPI_Core2` поддерживает все операции CRUD для складских данных, применяя уровень доступа к данным `AutoLotDAL_Core2` из главы 32. Наконец, веб-приложение ASP.NET Core будет обновлено с целью использования новой службы вместо прямого обращения к `AutoLotDAL_Core2`.

Загружаемые приложения

В дополнение к печатным материалам хранилище GitHub содержит исходный код примеров, рассмотренных в книге (доступный по адресу www.apress.com/9781484230176), и дополнительные приложения в формате PDF (версии на русском языке доступны для загрузки на веб-сайте издательства). В них раскрывается несколько дополнительных API-интерфейсов платформы .NET, которые вы можете считать удобными для своего рода занятий. В частности, вы найдете следующие материалы:

- Приложение А. Наборы данных, таблицы данных и адаптеры данных ADO.NET
- Приложение Б. Введение в LINQ to XML
- Приложение В. Введение в ASP.NET Web Forms
- Приложение Г. Веб-элементы управления, мастер-страницы и темы ASP.NET
- Приложение Д. Управление состоянием в ASP.NET

Исходный код примеров

Исходный код всех примеров, представленных в книге, доступен в открытом хранилище GitHub (<https://github.com/apress/pro-csharp-7>). Проекты организованы по главам.

Имейте в виду, что почти во всех главах книги присутствуют врезки, помеченные как “Исходный код”. Они служат визуальной подсказкой о том, что вы можете загрузить код обсуждаемого примера в IDE-среду Visual Studio для дальнейшего исследования и модификации.

Исходный код. Здесь дается ссылка на определенный каталог в хранилище GitHub.

Чтобы открыть решение в Visual Studio, выберите пункт меню `File⇒Open⇒Project/Solution` (Файл⇒Открыть⇒Проект/решение) и перейдите к интересующему файлу `*.sln` в соответствующем каталоге.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@dialektika.com
WWW: <http://www.dialektika.com>

часть I

Введение в C# и платформу .NET

В этой части

Глава 1. Философия .NET

Глава 2. Создание приложений на языке C#

глава 1

Философия .NET

Платформа Microsoft .NET (и связанный с ней язык программирования C#) впервые была представлена примерно в 2002 году и быстро стала главной опорой современной индустрии разработки программного обеспечения. Как отмечалось во введении, при ее написании преследовались две цели. Первая из них — предоставление читателям глубокого и подробного описания синтаксиса и семантики языка C#. Вторая (не менее важная) цель — иллюстрация применения многочисленных API-интерфейсов .NET, в числе которых доступ к базам данных с помощью ADO.NET и Entity Framework (EF), пользовательские интерфейсы, построенные посредством Windows Presentation Foundation (WPF), ориентированные на службы приложения, созданные с помощью Windows Communication Foundation (WCF), а также веб-службы и веб-сайты, реализованные посредством ASP.NET MVC. Последняя часть книги посвящена самому новому члену семейства .NET, .NET Core, который представляет собой межплатформенную версию .NET. Как говорят, пеший поход длиной тысячу километров начинается с первого шага, который и будет сделан в настоящей главе.

Задача этой первой главы заключается в построении концептуальной основы для успешного освоения всего остального материала книги. Здесь вы найдете высокоуровневое обсуждение нескольких связанных с .NET тем, таких как сборки, общий промежуточный язык (Common Intermediate Language — CIL) и оперативная (Just-In-Time — JIT) компиляция. В дополнение к предварительному обзору ряда ключевых слов C# вы узнаете о взаимоотношениях между разнообразными компонентами платформы .NET, среди которых общезыковая исполняющая среда (Common Language Runtime — CLR), общая система типов (Common Type System — CTS) и общезыковая спецификация (Common Language Specification — CLS).

Кроме того, в главе представлен обзор функциональности, поставляемой в библиотеках базовых классов .NET, для обозначения которых иногда применяется аббревиатура BCL (base class library — библиотека базовых классов). Здесь вы кратко ознакомитесь с независимой от языка и платформы природой .NET. Как несложно догадаться, многие затронутые здесь темы будут более детально исследоваться в оставшихся главах книги.

Начальное знакомство с платформой .NET

До выпуска компанией Microsoft языка C# и платформы .NET разработчики программного обеспечения, создававшие приложения для операционных систем (ОС) семейства Windows, часто использовали модель программирования COM. Технология COM (Component Object Model — модель компонентных объектов) позволяла строить библиотеки кода, которые можно было разделять между несходными языками программирования. Например, программист на C++ был в состоянии построить библиотеку COM, которой мог пользоваться разработчик на Visual Basic.

Безусловно, независимая от языка природа COM являлась удобной; тем не менее, технология COM досаждала своей усложненной инфраструктурой, хрупкой моделью развертывания и возможностью работы только в среде ОС Windows.

Несмотря на сложность и ограничения архитектуры COM, с ее применением было успешно создано бесчисленное количество приложений. Однако в наши дни большинство приложений, ориентированных на ОС семейства Windows, не создаются посредством COM. Взамен приложения для настольных компьютеров, веб-сайты, службы ОС и библиотеки многократно применяемой логики доступа к данным и бизнес-логики строятся с помощью платформы .NET.

Некоторые основные преимущества платформы .NET

Как уже упоминалось, язык C# и платформа .NET впервые были представлены миру в 2002 году и предназначались для обеспечения более мощной, гибкой и простой модели программирования по сравнению с COM. В оставшихся главах книги вы увидите, что .NET Framework представляет собой программную платформу для построения приложений, функционирующих под управлением ОС семейства Windows, а также многочисленных ОС производства не Microsoft, таких как macOS, iOS, Android и разнообразные дистрибутивы Unix/Linux. Ниже приведен краткий перечень ключевых средств, предлагаемых .NET.

- *Возможность взаимодействия с существующим кодом.* Несомненно, это очень полезно. Существующее программное обеспечение COM можно смешивать (т.е. взаимодействовать) с более новым программным обеспечением .NET и наоборот. В .NET 4.0 и последующих версиях возможность взаимодействия еще больше упростилась благодаря добавлению ключевого слова *dynamic* (рассматривается в главе 16).
- *Поддержка многочисленных языков программирования.* Приложения .NET могут создаваться с использованием любого числа языков программирования (C#, Visual Basic, F# и т.д.).
- *Общий исполняющий механизм, разделяемый всеми поддерживаемыми .NET языками.* Одним из аспектов такого механизма является наличие четко определенного набора типов, которые способен опознавать каждый поддерживающий .NET язык.
- *Языковая интеграция.* В .NET поддерживается межъязыковое наследование, межъязыковая обработка исключений и межъязыковая отладка кода. Например, можно определить базовый класс в C# и расширить этот тип в Visual Basic.
- *Обширная библиотека базовых классов.* Данная библиотека предоставляет тысячи предварительно определенных типов, которые позволяют строить библиотеки кода, простые терминальные приложения, графические настольные приложения и веб-сайты уровня предприятия.
- *Упрощенная модель развертывания.* В отличие от COM библиотеки .NET не регистрируются в системном реестре. Более того, платформа .NET позволяет нескольким версиям одной и той же сборки *.dll благополучно сосуществовать на одном компьютере.

Все перечисленные темы (и многие другие) будут подробно рассматриваться в последующих главах.

Введение в строительные блоки платформы .NET (CLR, CTS и CLS)

Теперь, когда вы узнали кое-что об основных преимуществах, присущих платформе .NET, давайте рассмотрим три ключевых (и взаимосвязанных) компонента, которые делают возможным все упомянутое выше — CLR, CTS и CLS. С точки зрения программиста платформа .NET может восприниматься как исполняющая среда и обширная библиотека базовых классов. Уровень исполняющей среды правильно называть *общезыковой исполняющей средой* (Common Language Runtime) или сокращенно *средой CLR*. Главной задачей CLR является автоматическое обнаружение, загрузка и управление объектами .NET. Вдобавок среда CLR заботится о ряде низкоуровневых аспектов, таких как управление памятью, обслуживание приложений, координирование потоков и выполнение базовых проверок, связанных с безопасностью (помимо прочих низкоуровневых деталей).

Еще одним строительным блоком платформы .NET является *общая система типов* (Common Type System) или сокращенно *система CTS*. Спецификация CTS полностью описывает все возможные типы данных и все программные конструкции, поддерживаемые исполняющей средой, указывает, каким образом эти сущности могут взаимодействовать друг с другом, и как они представлены в формате метаданных .NET (дополнительную информацию о метаданных .NET ищите далее в главе, а исчерпывающие сведения — в главе 15).

Важно понимать, что отдельно взятый язык, совместимый с .NET, может не поддерживать абсолютно все функциональные средства, определяемые спецификацией CTS. Существует родственная *общезыковая спецификация* (Common Language Specification) или сокращенно *спецификация CLS*, где описано подмножество общих типов и программных конструкций, которое должны поддерживать все языки программирования для .NET. Таким образом, если вы строите типы .NET, открывающие доступ только к совместимым с CLS средствам, то можете быть уверены в том, что их смогут потреблять все языки, поддерживающие .NET. И наоборот, если вы применяете тип данных или программную конструкцию, которая выходит за границы CLS, тогда не можете гарантировать, что каждый язык программирования для .NET окажется способным взаимодействовать с вашей библиотекой кода .NET. К счастью, как вы увидите далее в главе, компилятору C# довольно просто сообщить о необходимости проверки всего кода на предмет совместимости с CLS.

Роль библиотек базовых классов

В дополнение к спецификациям CLR, CTS и CLS платформа .NET предоставляет библиотеку базовых классов, которая доступна всем языкам программирования .NET. Эта библиотека базовых классов не только инкапсулирует разнообразные примитивы вроде потоков, файлового ввода-вывода, систем визуализации графики и механизмов взаимодействия с разнообразными внешними устройствами, но также обеспечивает поддержку для многочисленных служб, требуемых большинством реальных приложений.

Библиотеки базовых классов определяют типы, которые можно использовать для построения программного приложения любого вида. Например, инфраструктуру ASP.NET можно применять для создания веб-сайтов и служб REST, инфраструктуру WCF — для построения распределенных систем, инфраструктуру WPF — для написания настольных приложений с графическим пользовательским интерфейсом и т.д. Вдобавок библиотеки базовых классов предлагают типы для взаимодействия с каталогами и файловой системой отдельного компьютера, для коммуникаций с реляционными базами данных (через ADO.NET) и т.п. С высокоуровневой точки зрения отношения между CLR, CTS, CLS и библиотеками базовых классов выглядят так, как показано на рис. 1.1.



Рис. 1.1. Отношения между CLR, CTS, CLS и библиотеками базовых классов

Что привносит язык C#

Синтаксис языка программирования C# выглядит очень похожим на синтаксис языка Java. Однако называть C# клоном Java неправильно. В действительности и C#, и Java являются членами семейства языков программирования, основанных на C (например, C, Objective C, C++), поэтому они разделяют сходный синтаксис.

Правда заключается в том, что многие синтаксические конструкции C# смоделированы в соответствии с разнообразными аспектами языков Visual Basic (VB) и C++. Например, подобно VB язык C# поддерживает понятия свойств класса (как противоположность традиционным методам извлечения и установки) и необязательных параметров. Подобно C++ язык C# позволяет перегружать операции, а также создавать структуры, перечисления и функции обратного вызова (через делегатов).

Более того, по мере проработки материала книги вы очень скоро заметите, что C# поддерживает множество средств, которые традиционно встречаются в различных языках функционального программирования (например, LISP или Haskell), такие как лямбда-выражения и анонимные типы. Вдобавок с появлением технологии LINQ (*Language Integrated Query* — язык интегрированных запросов) язык C# стал поддерживать конструкции, которые делают его довольно-таки уникальным в мире программирования. Но, несмотря на все это, наибольшее влияние на него оказали именно языки, основанные на C.

Поскольку C# — гибрид из нескольких языков, он является таким же синтаксически чистым, как Java (если не чище), почти настолько же простым, как VB, и практически таким же мощным и гибким, как C++. Ниже приведен неполный перечень ключевых особенностей языка C#, которые характерны для всех его версий.

- **Указатели необязательны!** В программах на C# обычно не возникает потребность в прямых манипуляциях указателями (хотя в случае абсолютной необходимости можно опуститься и на уровень указателей, как объясняется в главе 11).
- **Автоматическое управление памятью посредством сборки мусора.** С учетом этого в C# не поддерживается ключевое слово вроде `delete`.

- Формальные синтаксические конструкции для классов, интерфейсов, структур, перечислений и делегатов.
- Аналогичная языку C++ возможность перегрузки операций для специальных типов без особой сложности (например, обеспечение возвращения `*this`, чтобы позволить связывать в цепочку — не ваша забота).
- Поддержка программирования на основе атрибутов. Разработка такого вида позволяет аннотировать типы и их члены для дополнительного уточнения их поведения. Например, если пометить метод атрибутом `[Obsolete]`, то при попытке его использования программисты увидят ваше специальное предупреждение.

С выходом версии .NET 2.0 (примерно в 2005 году) язык программирования C# был обновлен с целью поддержки многочисленных новых функциональных возможностей, наиболее значимые из которых перечислены далее.

- Возможность создания обобщенных типов и обобщенных членов. Применяя обобщения, можно писать очень эффективный и безопасный код, который определяет множество *заполнителей*, указываемых во время взаимодействия с обобщенными элементами.
- Поддержка анонимных методов, которые позволяют предоставлять встраиваемую функцию везде, где требуется тип делегата.
- Возможность определения одиночного типа в нескольких файлах кода (или при необходимости в виде представления в памяти) с использованием ключевого слова `partial`.

В версии .NET 3.5 (вышедшей приблизительно в 2008 году) в язык программирования C# была добавлена дополнительная функциональность, в том числе следующие средства.

- Поддержка строго типизированных запросов (например, LINQ), применяемых для взаимодействия с разнообразными формами данных. Вы впервые встретите запросы LINQ в главе 12.
- Поддержка анонимных типов, позволяющая моделировать на лету в коде *устройство* типа, а не его поведение.
- Возможность расширения функциональности существующего типа (не создавая его подклассы) с использованием расширяющих методов.
- Включение лямбда-операции (`=>`), которая еще больше упрощает работу с типами делегатов .NET.
- Новый синтаксис инициализации объектов, позволяющий устанавливать значения свойств во время создания объекта.

Версия .NET 4.0 (выпущенная в 2010 году) снова дополнила язык C# рядом средств, которые указаны ниже.

- Поддержка необязательных параметров и именованных аргументов в методах.
- Поддержка динамического поиска членов во время выполнения через ключевое слово `dynamic`. Как будет показано в главе 18, это обеспечивает универсальный подход к вызову членов на лету независимо от инфраструктуры, в которой они реализованы (COM, IronRuby, IronPython или службы рефлексии .NET).
- Работа с обобщенными типами стала намного понятнее, учитывая возможность легкого отображения обобщенных данных на универсальные коллекции `System.Object` через ковариантность и контравариантность.

В выпуске .NET 4.5 язык C# обрел пару новых ключевых слов (`async` и `await`), которые значительно упрощают многопоточное и асинхронное программирование. Если вы работали с предшествующими версиями C#, то можете вспомнить, что вызов методов через вторичные потоки требовал довольно большого объема малопонятного кода и применения разнообразных пространств имен .NET. Учитывая то, что теперь в C# поддерживаются языковые ключевые слова, которые автоматически устраняют эту сложность, процесс вызова методов асинхронным образом оказывается почти настолько же легким, как их вызов в синхронной манере. Данные темы детально раскрываются в главе 19.

Версия C# 6 появилась в составе .NET 4.6 и получила несколько мелких средств, которые помогают упростить кодовую базу. Ниже представлен краткий обзор возможностей, введенных в C# 6.

- Встраиваемая инициализация для автоматических свойств, а также поддержка автоматических свойств, предназначенных только для чтения.
- Реализация однострочных методов с использованием лямбда-операции C#.
- Поддержка статического импортирования для предоставления прямого доступа к статическим членам внутри пространства имен.
- `null`-условная операция, которая помогает проверять параметры на предмет `null` в реализации метода.
- Новый синтаксис форматирования строк, называемый *интерполяцией строк*.
- Возможность фильтрации исключений с применением нового ключевого слова `when`.
- Использование `await` в блоках `catch` и `finally`.
- Выражения `nameof` для возвращения строкового представления символов.
- Инициализаторы индексов.
- Улучшенное распознавание перегруженных версий.

Все упомянутое ранее подводит нас к текущей крупной версии C#, которая была выпущена вместе с .NET 4.7 в марте 2017 года. Подобно C# 6 в версии C# 7 введены дополнительные возможности, упрощающие кодовую базу, и добавлено несколько более значительных средств (вроде кортежей и ссылочных локальных переменных и возвращаемых ссылочных значений), которые разработчики просили включить довольно долгое время. Они будут подробно описаны в оставшихся главах книги; тем не менее, вот краткий обзор новых возможностей C# 7.

- Объявление переменных `out` как встраиваемых аргументов.
- Вложение функций внутри других функций для ограничения области действия и видимости.
- Дополнительные члены, сжатые до выражения.
- Обобщенные асинхронные возвращаемые типы.
- Новые маркеры для улучшения читабельности числовых констант.
- Легковесные неименованные типы (называемые *кортежами*), которые содержат множество полей.
- Обновление логического потока применением сопоставления с типом в дополнение с проверкой значений (сопоставление с шаблоном).
- Возвращение ссылки на значение вместо только самого значения (ссылочные локальные переменные и возвращаемые ссылочные значения).
- Введение одноразовых переменных (называется *отбрасыванием*).
- Выражения `throw`, позволяющие размещать конструкцию `throw` в большем числе мест — в условных выражениях, лямбда-выражениях и др.

Через некоторое время после выхода C# 7 в августе 2017 года была выпущена версия C# 7.1. В C# 7.1 появились следующие средства.

- Возможность иметь асинхронный метод `Main()` программы.
- Новый литерал `default`, который делает возможной инициализацию любого типа.
- Устранение проблемы, выявленной в сопоставлении с шаблоном, которая препятствовала использованию обобщений в этом новом средстве сопоставления с шаблоном.
- Подобно анонимным методам имена кортежей могут выводиться из проекции, которая их создает.

Сравнение управляемого и неуправляемого кода

Важно отметить, что язык C# может применяться только для построения программного обеспечения, которое функционирует под управлением исполняющей среды .NET (вы никогда не будете использовать C# для создания COM-сервера или неуправляемого приложения в стиле C/C++). Выражаясь официально, для обозначения кода, ориентированного на исполняющую среду .NET, используется термин *управляемый код*. Двоичный модуль, который содержит управляемый код, называется *сборкой* (сборки более подробно рассматриваются далее в главе). И наоборот, код, который не может напрямую обслуживаться исполняющей средой .NET, называется *неуправляемым кодом*.

Ранее уже упоминалось (и об этом пойдет речь позже в текущей и следующей главе), что платформа .NET способна функционировать в средах разнообразных ОС. Таким образом, вполне вероятно строить приложение C# на машине с применением Visual Studio и запускать его на машине macOS с использованием исполняющей среды .NET Core. Кроме того, приложение C# можно построить на машине Linux с помощью Xamarin Studio и запускать его под управлением Windows, macOS и т.д. Самый последний выпуск Visual Studio 2017 позволяет строить на компьютере Mac приложения .NET Core, предназначенные для выполнения под управлением Windows, macOS или Linux. Конечно, идея управляемой среды делает возможными построение, развертывание и запуск программ .NET на широком разнообразии целевых машин.

Другие языки программирования, ориентированные на .NET

Помните, что C# — не единственный язык, который может применяться для построения приложений .NET. Изначально среда Visual Studio предлагает пять управляемых языков — C#, Visual Basic, C++/CLI, JavaScript и F#.

На заметку! F# является языком .NET, основанным на синтаксисе функциональных языков. Наряду с тем, что F# можно использовать как чистый функциональный язык, он также поддерживает конструкции объектно-ориентированного программирования (ООП) и библиотеки базовых классов .NET. Дополнительные сведения об этом управляемом языке доступны на его официальной домашней странице по адресу <http://msdn.microsoft.com/fsharp>.

В дополнение к управляемым языкам, предоставляемым Microsoft, существуют компиляторы .NET для языков Smalltalk, Ruby, Python, COBOL и Pascal (перечень далеко не полон). Хотя в настоящей книге внимание сосредоточено почти полностью на C#, на следующей странице Википедии приведен большой список языков программирования, которые ориентированы на платформу .NET:

https://ru.wikipedia.org/wiki/Список_.NET-языков

Даже если вы интересуетесь главным образом построением программ .NET с применением синтаксиса C#, то все равно посетите указанную страницу, т.к. вы наверняка сочтете многие языки для .NET заслуживающими дополнительного внимания (скажем, LISP.NET).

Жизнь в многоязычном мире

Как только разработчики приходят к осознанию независимой от языка природы .NET, у них возникает множество вопросов. Самый распространенный вопрос формулируется так: если все языки .NET компилируются в управляемый код, то зачем нам больше одного языка/компилятора?

Ответить на такой вопрос можно по-разному. Прежде всего, программисты бывают крайне привередливы, когда дело касается выбора языка программирования. Одни предпочитают языки с многочисленными точками с запятой и фигурными скобками, но с минимальным количеством ключевых слов. Другим нравятся языки, предлагающие более читабельные синтаксические конструкции (такие как Visual Basic). Кто-то после перехода на платформу .NET желает задействовать имеющийся опыт работы на мэйнфреймах (выбрав компилятор COBOL .NET).

Теперь ответьте честно: если бы компания Microsoft предложила единственный "официальный" язык .NET, производный от семейства языков BASIC, то все ли программисты обрадовались бы такому выбору? А если бы единственный "официальный" язык .NET основывался на синтаксисе Fortran, то вообразите, сколько людей в мире просто бы проигнорировали платформу .NET. Из-за того, что исполняющая среда .NET демонстрирует меньшую зависимость от языка, используемого для построения блока управляемого кода, программисты .NET могут придерживаться своих синтаксических предпочтений и разделять скомпилированный код между коллегами, отделами и внешними организациями (безотносительно к тому, какой язык .NET выбрали для работы другие).

Еще одним превосходным побочным эффектом интеграции разнообразных языков .NET в единственное унифицированное программное решение является тот простой факт, что каждый язык программирования обладает своими сильными и слабыми сторонами. Например, некоторые языки программирования предлагают великолепную встроенную поддержку для реализации сложных математических вычислений. Другие языки лучше приспособлены для финансовых или логических вычислений, взаимодействия с мэйнфреймами и т.п. Когда преимуществами конкретного языка программирования объединяются с преимуществами платформы .NET, то выигрывают все.

Конечно, в реальности довольно велики шансы на то, что большую часть времени вы будете уделять построению программного обеспечения, применяя предпочитаемый язык .NET. Однако, освоив синтаксис одного языка .NET, легко изучить другой. Это также очень выгодно, особенно консультантам по разработке программного обеспечения. Например, если вашим предпочитаемым языком является C#, но вы находитесь на клиентской площадке, где все настроено на Visual Basic, то все равно сможете использовать функциональность .NET Framework и понимать общую структуру кодовой базы с минимальными усилиями.

Обзор сборок .NET

Независимо от того, какой язык .NET выбран для программирования, важно понимать, что хотя двоичные модули .NET имеют такое же файловое расширение, как и управляемые двоичные компоненты Windows (*.dll или *.exe), внутренне они устроены совершенно по-другому. В частности, двоичные модули .NET содержат не специфические, а независимые от платформы инструкции на *промежуточном языке* (Intermediate Language — IL) и метаданные типов.

На рис. 1.2 показано, как все выглядит схематически.

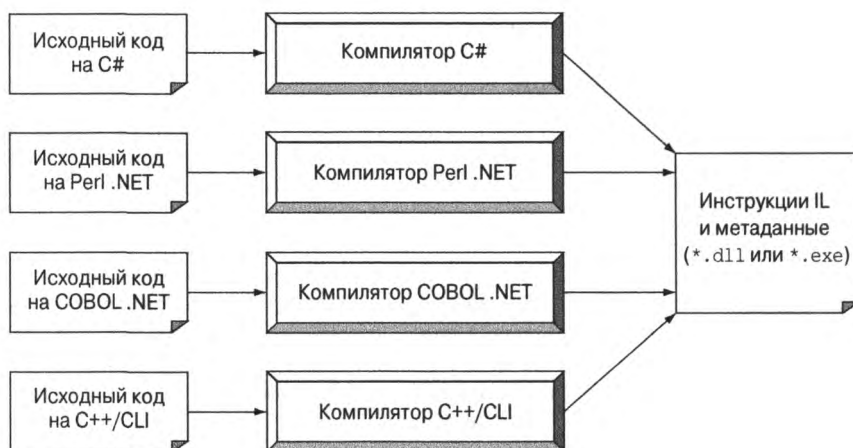


Рис. 1.2. Все компиляторы .NET выпускают инструкции IL и метаданные

На заметку! Язык IL также известен как промежуточный язык Microsoft (Microsoft Intermediate Language — MSIL) или общий промежуточный язык (Common Intermediate Language — CIL). Таким образом, при чтении литературы по .NET не забывайте о том, что IL, MSIL и CIL описывают в точности одну и ту же концепцию. В настоящей книге при ссылке на этот низкоуровневый набор инструкций будет применяться аббревиатура CIL.

Когда файл *.dll или *.exe был создан с использованием компилятора .NET, результирующий большой двоичный объект называется *сборкой*. Все многочисленные детали, касающиеся сборки .NET, подробно рассматриваются в главе 14. Тем не менее, для упрощения текущего обсуждения вы должны усвоить ряд основных свойств нового файлового формата.

Как упоминалось ранее, сборка содержит код CIL, который концептуально похож на байт-код Java тем, что не компилируется в специфичные для платформы инструкции до тех пор, пока это не станет абсолютно необходимым. Обычно “абсолютная необходимость” наступает тогда, когда на блок инструкций CIL (такой как реализация метода) производится ссылка с целью его применения исполняющей средой .NET.

В дополнение к инструкциям CIL сборки также содержат *метаданные*, которые детально описывают характеристики каждого “типа” внутри двоичного модуля. Например, если имеется класс по имени SportsCar, то метаданные типа представляют такие детали, как базовый класс SportsCar, указывают реализуемые SportsCar интерфейсы (если есть) и дают полные описания всех членов, поддерживаемых типом SportsCar. Метаданные .NET всегда присутствуют внутри сборки и автоматически генерируются компилятором языка .NET.

Наконец, помимо инструкций CIL и метаданных типов сами сборки также описываются с помощью метаданных, которые официально называются *манифестом*. Манифест содержит информацию о текущей версии сборки, сведения о культуре (используемые для локализации строковых и графических ресурсов) и список ссылок на все внешние сборки, которые требуются для надлежащего функционирования. Разнообразные инструменты, которые можно применять для исследования типов, метаданных и манифестов сборки, рассматриваются в нескольких последующих главах.

Роль языка CIL

Теперь давайте займемся детальными исследованиями кода CIL, метаданных типов и манифеста сборки. Язык CIL находится выше любого набора инструкций, специфичных для конкретной платформы. Например, приведенный далее код C# моделирует простой калькулятор. Не углубляясь пока в подробности синтаксиса, обратите внимание на формат метода `Add()` в классе `Calc`.

```
// Calc.cs
using System;
namespace CalculatorExample
{
    // Этот класс содержит точку входа приложения.
    class Program
    {
        static void Main()
        {
            Calc c = new Calc();
            int ans = c.Add(10, 84);
            Console.WriteLine("10 + 84 is {0}.", ans);
            // Ожидать нажатия пользователем клавиши <Enter> перед завершением работы.
            Console.ReadLine();
        }
    }

    // Калькулятор C#.
    class Calc
    {
        public int Add(int x, int y)
        { return x + y; }
    }
}
```

В результате компиляции такого файла кода с помощью компилятора C# (`csc.exe`) получается однофайловая сборка `*.exe`, которая содержит манифест, инструкции CIL и метаданные, описывающие каждый аспект классов `Calc` и `Program`.

На заметку! В главе 2 будет показано, как использовать для компиляции файлов кода графические среды интегрированной разработки (integrated development environment — IDE), такие как Visual Studio Community.

Например, открыв полученную сборку в утилите `ildasm.exe` (которая рассматривается далее в главе), вы обнаружите, что метод `Add()` был представлен в CIL следующим образом:

```
.method public hidebysig instance int32 Add(int32 x,
      int32 y) cil managed
{
    // Code size 9 (0x9)
    // Размер кода 9 (0x9)
    .maxstack 2
    .locals init (int32 V_0)
    IL_0000: nop
    IL_0001: ldarg.1
    IL_0002: ldarg.2
    IL_0003: add
```

```

IL_0004: stloc.0
IL_0005: br.s IL_0007
IL_0007: ldloc.0
IL_0008: ret
} // end of method Calc::Add
// конец метода Calc::Add

```

Не стоит беспокоиться, если итоговый код CIL метода Add() выглядит непонятным — в главе 18 будут описаны базовые аспекты языка программирования CIL. Важно понимать, что компилятор C# выпускает CIL-код, а не инструкции, специфичные для платформы.

Теперь вспомните, что сказанное справедливо для всех компиляторов .NET. В целях иллюстрации создадим то же самое приложение на языке Visual Basic вместо C#:

```

' Calc.vb
Imports System

Namespace CalculatorExample
    ' "Модуль" (Module) в VB - это класс, который
    ' содержит только статические члены.
    Module Program
        Sub Main()
            Dim c As New Calc
            Dim ans As Integer = c.Add(10, 84)
            Console.WriteLine("10 + 84 is {0}.", ans)
            Console.ReadLine()
        End Sub
    End Module

    Class Calc
        Public Function Add(ByVal x As Integer, ByVal y As Integer) As Integer
            Return x + y
        End Function
    End Class
End Namespace

```

Просмотрев код CIL такого метода Add(), можно найти похожие инструкции (слегка скорректированные компилятором Visual Basic, vbc.exe):

```

.method public instance int32 Add(int32 x,
    int32 y) cil managed
{
    // Code size 8 (0x8)
    // Размер кода 8 (0x8)
    .maxstack 2
    .locals init (int32 V_0)
    IL_0000: ldarg.1
    IL_0001: ldarg.2
    IL_0002: add.ovf
    IL_0003: stloc.0
    IL_0004: br.s IL_0006
    IL_0006: ldloc.0
    IL_0007: ret
} // end of method Calc::Add
// конец метода Calc::Add

```

Преимущества языка CIL

В этот момент вас может интересовать, какую выгоду приносит компиляция исходного кода в CIL, а не напрямую в специфичный набор инструкций. Одним из преимуществ является языковая интеграция. Как вы уже видели, все компиляторы .NET выпускают практически идентичные инструкции CIL. Следовательно, все языки способны взаимодействовать в рамках четко определенной "двоичной арены".

Более того, учитывая независимость от платформы языка CIL, сама инфраструктура .NET Framework не зависит от платформы и обеспечивает те же самые преимущества, к которым так привыкли разработчики на Java (например, единую кодовую базу, функционирующую в средах многочисленных ОС). В действительности для языка C# предусмотрен международный стандарт, и уже существует крупное подмножество платформы .NET и реализации для многих ОС, отличающихся от Windows (дополнительные детали приведены в конце главы).

Компиляция кода CIL в инструкции, специфичные для платформы

Поскольку сборки содержат инструкции CIL, а не инструкции, специфичные для платформы, перед применением код CIL должен компилироваться на лету. Компонентом, который транслирует код CIL в содержательные инструкции центрального процессора (ЦП), является оперативный (JIT) компилятор (иногда называемый *jitter*). Для каждого целевого ЦП в исполняющей среде .NET имеется JIT-компилятор, оптимизированный под лежащую в основе платформу.

Например, если строится приложение .NET, предназначенное для развертывания на карманном устройстве (таком как устройство Windows Phone), то соответствующий JIT-компилятор будет оснащен возможностями запуска в среде с ограниченным объемом памяти. С другой стороны, если сборка развертывается на внутреннем сервере компании (где память редко оказывается проблемой), тогда JIT-компилятор будет оптимизирован для функционирования в среде с большим объемом памяти. Таким образом, разработчики могут писать единственный блок кода, который способен эффективно транслироваться JIT-компилятором и выполняться на машинах с разной архитектурой.

Вдобавок при трансляции инструкций CIL в соответствующий машинный код JIT-компилятор будет кешировать результаты в памяти в манере, подходящей для целевой ОС. В таком случае, если производится вызов метода по имени `PrintDocument()`, то инструкции CIL компилируются в специфичные для платформы инструкции при первом вызове и остаются в памяти для более позднего использования. Благодаря этому при вызове метода `PrintDocument()` в следующий раз повторная компиляция инструкций CIL не понадобится.

На заметку! Можно также выполнять "предварительную JIT-компиляцию" сборки во время установки приложения с помощью инструмента командной строки `ngen.exe`, который поставляется вместе с .NET Framework SDK. Такой прием позволяет улучшить показатели времени запуска для графически насыщенных приложений.

Роль метаданных типов .NET

В дополнение к инструкциям CIL сборка .NET содержит полные и точные метаданные, которые описывают каждый определенный в двоичном модуле тип (например, класс, структуру, перечисление), а также члены каждого типа (скажем, свойства, методы, события). К счастью, за выпуск актуальных метаданных типов всегда отвечает компилятор, а не программист. Из-за того, что метаданные .NET настолько основательны, сборки являются целиком самоописательными сущностями.

Чтобы проиллюстрировать формат метаданных типов .NET, давайте взглянем на метаданные, которые были сгенерированы для исследуемого ранее метода `Add()` класса `Calc`, написанного на C# (метаданные для версии Visual Basic метода `Add()` аналогичны; дополнительные сведения о работе с утилитой `ildasm` будут приведены чуть позже):

```
TypeDef #2 (02000003)
-----
TypeDefName: CalculatorExample.Calc (02000003)
Flags       : [NotPublic] [AutoLayout] [Class]
[AnsiClass] [BeforeFieldInit] (00100001)
Extends     : 01000001 [TypeRef] System.Object
Method #1 (06000003)
-----
MethodName: Add (06000003)
Flags       : [Public] [HideBySig] [ReuseSlot] (00000086)
RVA         : 0x00002090
ImplFlags   : [IL] [Managed] (00000000)
CallConvntn: [DEFAULT]
hasThis
ReturnType: I4
  2 Arguments
    Argument #1: I4
    Argument #2: I4
  2 Parameters
    (1) ParamToken : (08000001) Name : x flags: [none] (00000000)
    (2) ParamToken : (08000002) Name : y flags: [none] (00000000)
```

Метаданные применяются многочисленными аспектами исполняющей среды .NET, а также разнообразными инструментами разработки. Например, средство `IntelliSense`, предоставляемое такими инструментами, как `Visual Studio`, стало возможным благодаря чтению метаданных сборки во время проектирования. Метаданные также используются различными утилитами просмотра объектов, инструментами отладки и самим компилятором C#. Бесспорно, метаданные являются принципиальной основой многочисленных технологий .NET, включая WCF, рефлексия, позднее связывание и сериализацию объектов. Роль метаданных .NET будет раскрыта в главе 15.

Роль манифеста сборки

Последний, но не менее важный момент: вспомните, что сборка .NET содержит и метаданные, которые описывают ее саму (формально называемые *манифестом*). Помимо прочего манифест документирует все внешние сборки, которые требуются текущей сборке для ее корректного функционирования, номер версии сборки, информацию об авторских правах и т.д. Подобно метаданным типов за генерацию манифеста сборки всегда отвечает компилятор. Ниже представлены некоторые существенные детали манифеста, сгенерированного при компиляции показанного ранее в главе файла `Calc.cs` (предполагается, что вы проинструктировали компилятор о назначении сборке имени `Calc.exe`):

```
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
```

```
.assembly Calc
{
    .hash algorithm 0x00008004
    .ver 0:0:0:0
}
.module Calc.exe
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 0x00000200
.corflags 0x00000001
```

Выражаясь кратко, показанный манифест документирует набор внешних сборок, требуемых для Calc.exe (в директиве `.assembly extern`), а также разнообразные характеристики самой сборки (скажем, номер версии и имя модуля). Полезность данных манифеста будет более подробно исследоваться в главе 14.

Понятие общей системы типов (CTS)

Сборка может содержать любое количество различающихся типов. В мире .NET *тип* — это просто общий термин, применяемый для ссылки на член из набора (класс, интерфейс, структура, перечисление, делегат). При построении решений на любом языке .NET почти наверняка придется взаимодействовать со многими такими типами. Например, в сборке может быть определен класс, реализующий некоторое количество интерфейсов. Возможно, метод одного из интерфейсов принимает перечисление в качестве входного параметра и возвращает вызывающему компоненту структуру.

Вспомните, что CTS является формальной спецификацией, которая документирует, каким образом типы должны быть определены, чтобы они могли обслуживаться средой CLR. Внутренние детали CTS обычно интересуют только тех, кто занимается построением инструментов и/или компиляторов, предназначенных для платформы .NET. Однако для всех программистов .NET важно знать о том, как работать с пятью типами, определенными в CTS, на выбранных ими языках. Ниже приведен краткий обзор.

Типы классов CTS

Каждый язык .NET поддерживает, по меньшей мере, понятие *типа класса*, которое является краеугольным камнем ООП. Класс может состоять из любого количества членов (таких как конструкторы, свойства, методы и события) и элементов данных (полей). В языке C# классы объявляются с использованием ключевого слова `class`, примерно так:

```
// Тип класса C# с одним методом.
class Calc
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

Формальное знакомство с построением типов классов в C# начнется в главе 5, а пока в табл. 1.1 приведен перечень характеристик, свойственных типам классов.

Таблица 1.1. Характеристики классов CTS

Характеристика класса	Практический смысл
Является ли класс запечатанным?	Запечатанные классы не могут выступать в качестве базовых для других классов
Реализует ли класс какие-то интерфейсы?	Интерфейс — это коллекция абстрактных членов, которая предоставляет контракт между объектом и пользователем объекта. Система CTS позволяет классу реализовывать любое количество интерфейсов
Является класс абстрактным или конкретным?	Абстрактные классы не допускают прямого создания экземпляров и предназначены для определения общего поведения производных типов. Экземпляры конкретных классов могут создаваться напрямую
Какова видимость класса?	Каждый класс должен конфигурироваться с ключевым словом видимости, таким как <code>public</code> или <code>internal</code> . По существу оно управляет тем, может ли класс использоваться во внешних сборках или же только внутри определяющей его сборки

Типы интерфейсов CTS

Интерфейсы представляют собой всего лишь именованные коллекции определений абстрактных членов, которые могут поддерживаться (т.е. быть реализованными) в заданном классе или структуре. В языке C# типы интерфейсов определяются с применением ключевого слова `interface`. По соглашению имена всех интерфейсов .NET начинаются с прописной буквы I, как показано в следующем примере:

```
// Тип интерфейса C# обычно объявляется как
// public, чтобы позволить типам из других
// сборок реализовывать его поведение.
public interface IDraw
{
    void Draw();
}
```

Сами по себе интерфейсы приносят не особо много пользы. Тем не менее, когда класс или структура реализует выбранный интерфейс уникальным образом, появляется возможность получать доступ к предоставленной функциональности, используя ссылку на этот интерфейс в полиморфной манере. Программирование на основе интерфейсов подробно рассматривается в главе 8.

Типы структур CTS

Концепция структуры также формализована в CTS. Если вы имели дело с языком C, то вас наверняка обрадует, что эти определяемые пользователем типы (*user-defined type* — UDT) сохранились в мире .NET (хотя их внутренне поведение несколько изменилось). Попросту говоря, *структуру* можно считать легковесным типом класса, который имеет семантику, основанную на значении. Тонкости структур более подробно исследуются в главе 4. Обычно структуры лучше всего подходят для моделирования геометрических и математических данных и создаются в языке C# с применением ключевого слова `struct`, например:

```
// Тип структуры C#.
struct Point
{
    // ...
}
```

```
// Структуры могут содержать поля.  
public int xPos, yPos;  
  
// Структуры могут содержать параметризованные конструкторы.  
public Point(int x, int y)  
{ xPos = x; yPos = y;}  
  
// Структуры могут определять методы.  
public void PrintPosition()  
{  
    Console.WriteLine("{0}, {1}", xPos, yPos);  
}  
}
```

Типы перечислений CTS

Перечисления — это удобная программная конструкция, которая позволяет группировать пары “имя-значение”. Например, предположим, что требуется создать игровое приложение, в котором игроку бы позволялось выбирать персонажа из трех категорий: Wizard (маг), Fighter (воин) или Thief (вор). Вместо отслеживания простых числовых значений, представляющих каждую категорию, можно было бы создать строго типизированное перечисление, используя ключевое слово `enum`:

```
// Тип перечисления C#.  
enum CharacterType  
{  
    Wizard = 100,  
    Fighter = 200,  
    Thief = 300  
}
```

По умолчанию для хранения каждого элемента выделяется блок памяти, соответствующий 32-битному целому, однако при необходимости (скажем, при программировании для устройств с малым объемом памяти наподобие мобильных устройств) область хранения можно изменить. Кроме того, спецификация CTS требует, чтобы перечислимые типы были производными от общего базового класса `System.Enum`. Как будет показано в главе 4, в этом базовом классе определено несколько интересных членов, которые позволяют извлекать, манипулировать и преобразовывать лежащие в основе пары “имя-значение” программным образом.

Типы делегатов CTS

Делегаты являются .NET-эквивалентом безопасных к типам указателей на функции в стиле C. Основная разница в том, что делегат .NET представляет собой класс, производный от `System.MulticastDelegate`, а не простой указатель на низкоуровневый адрес в памяти. В языке C# делегаты объявляются с помощью ключевого слова `delegate`:

```
// Этот тип делегата C# может "указывать" на любой метод,  
// возвращающий тип int и принимающий два значения int.  
delegate int BinaryOp(int x, int y);
```

Делегаты критически важны, когда необходимо обеспечить объект возможностью перенаправления вызова другому объекту, и они формируют основу архитектуры событий .NET. Как будет показано в главах 10 и 19, делегаты обладают внутренней поддержкой группового вызова (т.е. перенаправления запроса множеству получателей) и асинхронного вызова методов (т.е. вызова методов во вторичном потоке).

Члены типов CTS

Теперь, когда было представлено краткое описание каждого типа, формализованного в CTS, следует осознать тот факт, что большинство таких типов располагает любым количеством *членов*. Формально член типа ограничен набором (конструктор, финализатор, статический конструктор, вложенный тип, операция, метод, свойство, индексатор, поле, поле только для чтения, константа, событие).

В спецификации CTS определены разнообразные *характеристики*, которые могут быть ассоциированы с заданным членом. Например, каждый член может иметь характеристику видимости (открытый, закрытый или защищенный). Некоторые члены могут быть объявлены как абстрактные (чтобы обеспечить полиморфное поведение в производных типах) или как виртуальные (чтобы определить заготовленную, но допускающую переопределение реализацию). Вдобавок большинство членов могут быть сконфигурированы как статические (связанные с уровнем класса) или члены экземпляра (связанные с уровнем объекта). Создание членов типов будет описано в нескольких последующих главах.

На заметку! Как объясняется в главе 9, язык C# также поддерживает создание обобщенных типов и обобщенных членов.

Встроенные типы данных CTS

Финальный аспект спецификации CTS, который следует знать на текущий момент, заключается в том, что она устанавливает четко определенный набор фундаментальных типов данных. Хотя в каждом отдельном языке для объявления фундаментального типа данных обычно имеется уникальное ключевое слово, ключевые слова всех языков .NET в конечном итоге распознаются как один и тот же тип CTS, определенный в сборке по имени `mscorlib.dll`. В табл. 1.2 показано, каким образом основные типы данных CTS выражаются в различных языках .NET.

Таблица 1.2. Встроенные типы данных CTS

Тип данных CTS	Ключевое слово VB	Ключевое слово C#	Ключевое слово C++/CLI
System.Byte	Byte	byte	unsigned char
System.SByte	SByte	sbyte	signed char
System.Int16	Short	short	short
System.Int32	Integer	int	int или long
System.Int64	Long	long	__int64
System.UInt16	UShort	ushort	unsigned short
System.UInt32	UInteger	uint	unsigned int или unsigned long
System.UInt64	ULong	ulong	unsigned __int64
System.Single	Single	float	float
System.Double	Double	double	double
System.Object	Object	object	object^
System.Char	Char	char	wchar_t
System.String	String	string	String^
System.Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	bool	bool

Учитывая, что уникальные ключевые слова в управляемом языке являются просто сокращенными обозначениями для реальных типов в пространстве имен *System*, больше не нужно беспокоиться об условиях переполнения/потери значимости для числовых данных или о том, как строки и булевские значения внутренне представляются в разных языках. Взгляните на следующие фрагменты кода, в которых определяются 32-битные целочисленные переменные в C# и Visual Basic с применением ключевых слов языка, а также формального типа данных CTS:

```
// Определение целочисленных переменных в C#.
int i = 0;
System.Int32 j = 0;

' Определение целочисленных переменных в VB.
Dim i As Integer = 0
Dim j As System.Int32 = 0
```

Понятие общезыковой спецификации (CLS)

Как вы уже знаете, разные языки программирования выражают одни и те же программные конструкции с помощью уникальных и специфичных для конкретного языка терминов. Например, в C# конкатенация строк обозначается с использованием операции “плюс” (+), а в VB для этого обычно применяется амперсанд (&). Даже если два разных языка выражают одну и ту же программную идиому (скажем, функцию, не возвращающую значение), то высока вероятность того, что синтаксис на первый взгляд будет выглядеть не сильно отличающимся:

```
// Ничего не возвращающий метод C#.
public void MyMethod()
{
    // Некоторый код...
}

' Ничего не возвращающий метод VB.
Public Sub MyMethod()
    ' Некоторый код...
End Sub
```

Ранее вы уже видели, что такие небольшие синтаксические вариации для исполняющей среды .NET несущественны, учитывая, что соответствующие компиляторы (в данном случае *csc.exe* и *vbc.exe*) выпускают сходный набор инструкций CIL. Тем не менее, языки могут также отличаться в отношении общего уровня функциональности. Например, язык .NET может иметь или не иметь ключевое слово для представления данных без знака и поддерживать или не поддерживать типы указателей. При таких возможных вариациях было бы идеально располагать опорными требованиями, которым удовлетворяли бы все языки, ориентированные на .NET.

Спецификация CLS — это набор правил, подробно описывающих минимальное и полное множество характеристик, которые отдельный компилятор .NET должен поддерживать, чтобы генерировать код, обслуживаемый средой CLR и в то же время доступный в унифицированной манере всем ориентированным на платформу .NET языкам. Во многих отношениях CLS можно рассматривать как *подмножество* полной функциональности, определенной в CTS.

В конечном итоге CLS является набором правил, которых должны придерживаться создатели компиляторов, если они намерены обеспечить гладкое функционирование своих продуктов в мире .NET. Каждое правило имеет простое название (например, “Правило номер 6”), и каждое правило описывает воздействие на тех, кто строит ком-

пиляторы, и на тех, кто (каким-либо образом) взаимодействует с ними. Самым важным в CLS является правило номер 1.

- *Правило номер 1.* Правила CLS применяются только к тем частям типа, которые видны извне определяющей сборки.

Из данного правила можно сделать корректный вывод о том, что остальные правила CLS не применяются к логике, используемой для построения внутренних рабочих деталей типа .NET. Единственными аспектами типа, которые должны быть согласованы с CLS, являются сами определения членов (т.е. соглашения об именовании, параметры и возвращаемые типы). В рамках логики реализации члена может применяться любое количество приемов, не соответствующих CLS, т.к. для внешнего мира это не играет никакой роли.

В целях иллюстрации ниже представлен метод `Add()` в C#, который не совместим с CLS, поскольку его параметры и возвращаемое значение используют данные без знака (что не является требованием CLS):

```
class Calc
{
    // Открытые для доступа данные без знака не совместимы с CLS!
    public ulong Add(ulong x, ulong y)
    {
        return x + y;
    }
}
```

Тем не менее, если просто работать с данными без знака внутри метода, как в следующем примере:

```
class Calc
{
    public int Add(int x, int y)
    {
        // Поскольку эта переменная ulong используется только
        // внутренне, совместимость с CLS сохраняется.
        ulong temp = 0;
        ...
        return x + y;
    }
}
```

то правила CLS по-прежнему соблюдены и все языки .NET смогут обращаться к такому методу `Add()`.

Разумеется, помимо “Правила номер 1” в спецификации CLS определено множество других правил. Например, в CLS описано, каким образом заданный язык должен представлять текстовые строки, как внутренне представлять перечисления (базовый тип, применяемый для хранения их значений), каким образом определять статические члены и т.д. К счастью, для того, чтобы стать умелым разработчиком .NET, запоминать все правила вовсе не обязательно. В общем и целом глубоко разбираться в спецификациях CTS и CLS обычно должны только создатели инструментов и компиляторов.

Обеспечение совместимости с CLS

Как вы увидите при чтении книги, в языке C# определено несколько программных конструкций, несовместимых с CLS. Однако хорошая новость заключается в том, что

компилятор C# можно инструктировать о необходимости проверки кода на предмет совместимости с CLS, используя единственный атрибут .NET:

```
// Сообщить компилятору C# о том, что он должен осуществлять проверку
// на совместимость с CLS.
[assembly: CLSCompliant(true)]
```

Детали программирования на основе атрибутов подробно рассматриваются в главе 15. А пока следует просто запомнить, что атрибут [CLSCompliant] заставляет компилятор C# проверять каждую строку кода на соответствие правилам CLS. В случае обнаружения любых нарушений спецификации CLS компилятор сообщит об ошибке и выдаст описание проблемного кода.

Понятие общезыковой исполняющей среды (CLR)

Помимо спецификаций CTS и CLS для получения общей картины на данный момент осталось рассмотреть еще одну аббревиатуру — CLR. С точки зрения программирования термин *исполняющая среда* можно понимать как коллекцию служб, которые требуются для выполнения скомпилированной единицы кода. Например, когда разработчики на Java развертывают программное обеспечение на новом компьютере, им необходимо удостовериться в том, что на компьютере установлена виртуальная машина Java (Java Virtual Machine — JVM), которая обеспечит выполнение их программного обеспечения.

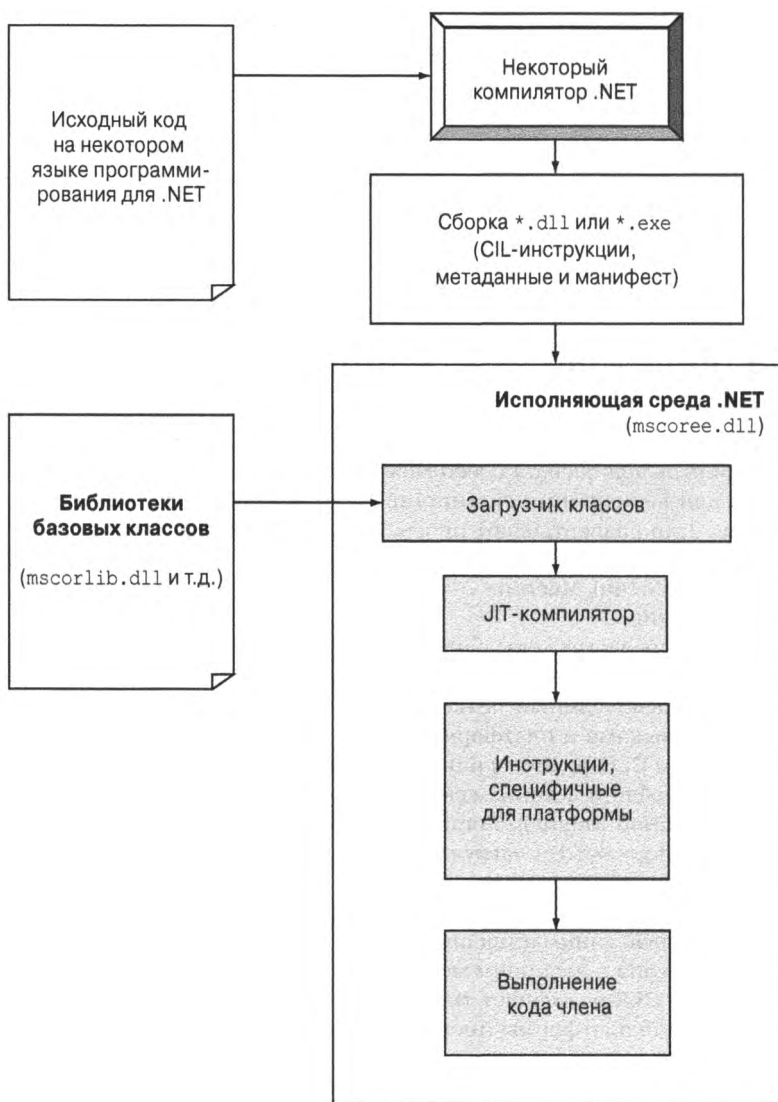
Платформа .NET предлагает еще одну исполняющую среду. Основное отличие исполняющей среды .NET от упомянутых выше сред заключается в том, что исполняющая среда .NET обеспечивает единый четко определенный уровень выполнения, который разделяется *всеми* языками и платформами, ориентированными на .NET.

Главный механизм CLR физически представлен библиотекой по имени `mscorlib.dll` (также известной как общий механизм выполнения исполняемого кода объектов (Common Object Runtime Execution Engine)). Когда на сборку производится ссылка для ее применения, библиотека `mscorlib.dll` загружается автоматически и в свою очередь загружает требуемую сборку в память.

Исполняющая среда отвечает за решение нескольких задач. Прежде всего, она является агентом, который занимается определением местоположения сборки и нахождением запрошенного типа в двоичном модуле за счет чтения содержащихся в нем метаданных. Затем среда CLR размещает тип в памяти, компилирует ассоциированный код CIL в специфичные для платформы инструкции, производит все необходимые проверки безопасности и, наконец, выполняет нужный код.

В дополнение к загрузке специальных сборок и созданию специальных типов среда CLR будет взаимодействовать с типами, содержащимися в библиотеках базовых классов .NET, когда это требуется. Хотя полная библиотека базовых классов разделена на ряд обособленных сборок, главной сборкой считается `mscorlib.dll`, которая содержит большое количество основных типов, инкапсулирующих широкий спектр распространенных задач программирования, а также основные типы данных, используемые во всех языках .NET. При построении решений .NET доступ к указанной сборке предоставляется автоматически.

На рис. 1.3 показан высокоуровневый рабочий поток, возникающий между исходным кодом (в котором применяются типы из библиотеки базовых классов), заданным компилятором .NET и исполняющей средой .NET.

Рис. 1.3. Сборка `mscorlib.dll` в действии

Различия между сборками, пространствами имен и типами

Любой из нас понимает важность библиотек кода. Главное назначение библиотек платформы — предоставлять разработчикам четко определенный набор готового кода, который можно задействовать в создаваемых приложениях. Однако C# не поставляется с какой-то специфичной для языка библиотекой кода. Взамен разработчики на C# используют нейтральные к языкам библиотеки .NET. Для поддержания всех типов внутри библиотек базовых классов в организованном виде в рамках платформы .NET широко применяется концепция *пространств имен*.

Пространство имен — это группа семантически родственных типов, которые содержатся в одной или нескольких связанных друг с другом сборках. Например, пространство имен `System.IO` содержит типы, относящиеся к файловому вводу-выводу, пространство имен `System.Data` — типы для работы с базами данных и т.д. Важно понимать, что в одной сборке (такой как `mscorlib.dll`) может содержаться любое количество пространств имен, каждое из которых может иметь любое число типов.

В целях прояснения на рис. 1.4 приведен экранный снимок окна браузера объектов Visual Studio (доступного через меню View (Вид)). Этот инструмент позволяет просматривать сборки, на которые имеются ссылки в текущем проекте, пространства имен внутри отдельной сборки, типы в конкретном пространстве имени и члены специфического типа. Обратите внимание, что сборка `mscorlib.dll` содержит много разных пространств имен (вроде `System.IO`), каждое с собственными семантически связанными типами (например, `BinaryReader`).

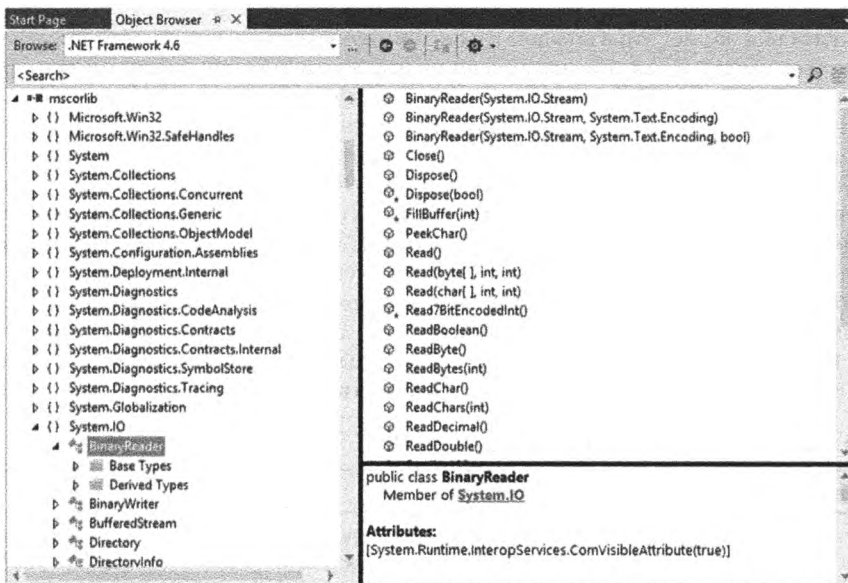


Рис. 1.4. Одиночная сборка может иметь любое количество пространств имен, а пространство имен может содержать любое число типов

Основное отличие между таким подходом и специфичной для языка библиотекой заключается в том, что любой язык, ориентированный на исполняющую среду .NET, использует *те же самые* пространства имен и *те же самые* типы. Например, ниже показан код вездесущего приложения "Hello World" на языках C#, VB и C++/CLI:

```
// Приложение "Hello World" на языке C#.
using System;

public class MyApp
{
    static void Main()
    {
        Console.WriteLine("Hi from C#");
    }
}
```



```
' Приложение "Hello World" на языке VB.
Imports System
Public Module MyApp
    Sub Main()
        Console.WriteLine("Hi from VB")
    End Sub
End Module

// Приложение "Hello World" на языке C++/CLI
#include "stdafx.h"
using namespace System;

int main(array<System::String ^> ^args)
{
    Console::WriteLine(L"Hi from C++/CLI");
    return 0;
}
```

Обратите внимание, что в каждом языке применяется класс `Console`, определенный в пространстве имен `System`. Помимо очевидных синтаксических различий три представленных приложения выглядят довольно похожими как физически, так и логически.

Понятно, что после освоения выбранного языка программирования для .NET вашей следующей целью как разработчика в .NET будет освоение изобилия типов, определенных в многочисленных пространствах имен .NET. Наиболее фундаментальное пространство имен, с которого нужно начать, называется `System`. Оно предлагает основной набор типов, которые вам как разработчику в .NET придется задействовать неоднократно. Фактически без добавления, по крайней мере, ссылки на пространство имен `System` построить сколько-нибудь функциональное приложение C# невозможно, т.к. в `System` определены основные типы данных (например, `System.Int32` и `System.String`). В табл. 1.3 приведены краткие описания некоторых (конечно же, не всех) пространств имен .NET, сгруппированные по функциональности.

Таблица 1.3. Избранные пространства имен .NET

Пространство имен .NET	Описание
System	Внутри пространства имен System содержится множество полезных типов, предназначенных для работы с внутренними данными, математическими вычислениями, генерацией случайных чисел, переменными среды и сборкой мусора, а также ряд распространенных исключений и атрибутов
System.Collections System.Collections.Generic	В этих пространствах имен определен набор контейнерных типов, а также базовые типы и интерфейсы, которые позволяют строить настраиваемые коллекции
System.Data System.Data.Common System.Data.EntityClient System.Data.SqlClient	Эти пространства имен используются для взаимодействия с базами данных через ADO.NET
System.IO System.IO.Compression System.IO.Ports	В этих пространствах имен определены многочисленные типы, предназначенные для работы с файловым вводом-выводом, сжатием данных и портами
System.Reflection System.Reflection.Emit	В этих пространствах имен определены типы, которые поддерживают обнаружение типов во время выполнения, а также динамическое создание типов

Пространство имен .NET	Описание
System.Runtime.InteropServices	Это пространство имен предоставляет средства, позволяющие типам .NET взаимодействовать с неуправляемым кодом (например, DLL-библиотеками на основе C и серверами COM) и наоборот
System.Drawing System.Windows.Forms	В этих пространствах имен определены типы, применяемые при построении настольных приложений с использованием первоначального инструментального набора .NET для создания пользовательских интерфейсов (Windows Forms)
System.Windows System.Windows.Controls System.Windows.Shapes	Пространство имен System.Windows является корневым для нескольких пространств имен, которые представляют инструментальный набор для построения пользовательских интерфейсов WPF
System.Linq System.Xml.Linq System.Data.DataSetExtensions	В этих пространствах имен определены типы, применяемые при программировании с использованием API-интерфейса LINQ
System.Web	Это одно из многих пространств имен, которые позволяют строить веб-приложения ASP.NET
System.Web.Http	Это одно из многих пространств имен, которые позволяют строить веб-службы REST
System.ServiceModel	Это одно из многих пространств имен, применяемых для построения распределенных приложений с использованием API-интерфейса Windows Communication Foundation (WCF)
System.Workflow.Runtime System.Workflow.Activities	Это два из множества пространств имен, определяющих типы, которые применяются при построении приложений, поддерживающих рабочие потоки, с использованием API-интерфейса Windows Workflow Foundation (WF)
System.Threading System.Threading.Tasks	В этих пространствах имен определены многочисленные типы для построения многопоточных приложений, которые могут распределять рабочую нагрузку по нескольким ЦП
System.Security	Безопасность является неотъемлемым аспектом мира .NET. В пространствах имен, связанных с безопасностью, содержится множество типов, которые позволяют работать с разрешениями, криптографией и т.д.
System.Xml	В пространствах имен, относящихся к XML, определены многочисленные типы, применяемые для взаимодействия с XML-данными

Роль корневого пространства имен Microsoft

Во время изучения списка, представленного в табл. 1.3, вероятно вы заметили, что `System` является корневым пространством имен для большинства вложенных пространств имен (таких как `System.IO`, `System.Data` и т.д.). Однако оказывается, что помимо `System` в библиотеке базовых классов определено несколько корневых пространств имен наивысшего уровня, наиболее полезное из которых называется `Microsoft`.

Любое пространство имен, вложенное внутрь `Microsoft` (скажем, `Microsoft.CSharp`, `Microsoft.ManagementConsole`, `Microsoft.Win32`), содержит типы, которые используются для взаимодействия со службами, уникальными для ОС Windows. С учетом этого вы не должны предполагать, что такие типы могли бы успешно применяться в

средах других ОС, поддерживающих .NET, таких как macOS. По большей части в настоящей книге мы не будем углубляться в детали пространств имен, для которых Microsoft является корневым, так что обращайтесь в документацию .NET Framework 4.7 SDK, если вам интересно.

На заметку! В главе 2 будет показано, как работать с документацией по .NET Framework 4.7 SDK, в которой содержатся подробные описания всех пространств имен, типов и членов внутри библиотек базовых классов.

Доступ к пространству имен программным образом

Полезно снова повторить, что пространство имен — всего лишь удобный способ логической организации связанных типов, содействующий их пониманию. Давайте еще раз обратимся к пространству имен `System`. С точки зрения разработчика можно предположить, что конструкция `System.Console` представляет класс по имени `Console`, который содержится внутри пространства имен под названием `System`. Однако с точки зрения исполняющей среды .NET это не так. Исполняющая среда видит только одиночный класс по имени `System.Console`.

В языке C# ключевое слово `using` упрощает процесс ссылки на типы, определенные в отдельном пространстве имен. Рассмотрим, каким образом оно работает. Пусть требуется построить графическое настольное приложение с использованием API-интерфейса WPF. Хотя освоение типов в каждом пространстве имен предполагает изучение и экспериментирование, ниже приведен ряд возможных кандидатов на ссылку из такого приложения:

```
// Некоторые возможные пространства имен,
// применяемые при построении приложения WPF.
using System;                // Общие типы из библиотек базовых классов.
using System.Windows.Shapes; // Типы для графической визуализации.
using System.Windows.Controls; // Типы виджетов графического пользователь-
                               // ского интерфейса Windows Forms.
using System.Data;           // Общие типы, связанные с данными.
using System.Data.SqlClient; // Типы доступа к данным MS SQL Server.
```

После указания нескольких необходимых пространств имен (и установки ссылки на сборки, где они определены) можно свободно создавать экземпляры типов, которые в них содержатся. Например, если нужно создать экземпляр класса `Button` (определенного в пространстве имен `System.Windows.Controls`), то можно написать следующий код:

```
// Явно перечислить пространства имен, используемые в этом файле.
using System;
using System.Windows.Controls;

class MyGUIBuilder
{
    public void BuildUI()
    {
        // Создать элемент управления типа кнопки.
        Button btnOK = new Button();
        ...
    }
}
```

Благодаря импортированию в файле кода пространства имен `System.Windows.Controls` компилятор имеет возможность распознать класс `Button` как член указанного пространства имен. Если не импортировать пространство имен `System.Windows.Controls`, тогда компилятор сообщит об ошибке. Тем не менее, можно также объявлять переменные с применением *полностью заданных имен*:

```
// Пространство имен System.Windows.Controls не указано!
using System;

class MyGUIBuilder
{
    public void BuildUI()
    {
        // Использование полностью заданного имени.
        System.Windows.Controls.Button btnOK =
            new System.Windows.Controls.Button();
        ...
    }
}
```

Хотя определение типа с использованием полностью заданного имени позволяет делать код более читабельным, трудно не согласиться с тем, что применение ключевого слова `using` в C# значительно сокращает объем набора на клавиатуре. В настоящей книге полностью заданные имена в основном использоваться не будут (разве что для устранения установленной неоднозначности), а предпочтение отдается упрощенному подходу с применением ключевого слова `using`.

Однако всегда помните о том, что ключевое слово `using` — просто сокращенный способ указать полностью заданное имя типа. Любой из подходов дает в результате тот же самый код CIL (учитывая, что в коде CIL всегда используются полностью заданные имена) и не влияет ни на производительность, ни на размер сборки.

Ссылка на внешние сборки

В дополнение к указанию пространства имен через ключевое слово `using` языка C# компилятору C# также необходимо сообщить имя сборки, содержащей действительную реализацию на CIL типа, на который производится ссылка. Как упоминалось ранее, многие основные пространства имен .NET определены внутри сборки `mscorlib.dll`. Однако, например, класс `System.Drawing.Bitmap` содержится в отдельной сборке по имени `System.Drawing.dll`. Подавляющее большинство сборок .NET Framework размещено в специальном каталоге, который называется *глобальным кешем сборок* (*global assembly cache* — GAC). На машине Windows GAC может по умолчанию находиться внутри каталога `C:\Windows\Assembly\GAC` (рис. 1.5).

В зависимости от инструмента разработки, применяемого для построения приложений .NET, вы будете иметь различные пути информирования компилятора о том, какие сборки необходимо включать в цикл компиляции. Подробности ищите в главе 2.

На заметку! Как будет показано в главе 14, в среде Windows есть несколько местоположений, где могут быть установлены библиотеки платформы; тем не менее, обычно это изолировано от разработчика. На машинах с ОС, отличающимися от Windows (такими как macOS или Linux), местоположение GAC зависит от дистрибутива .NET.

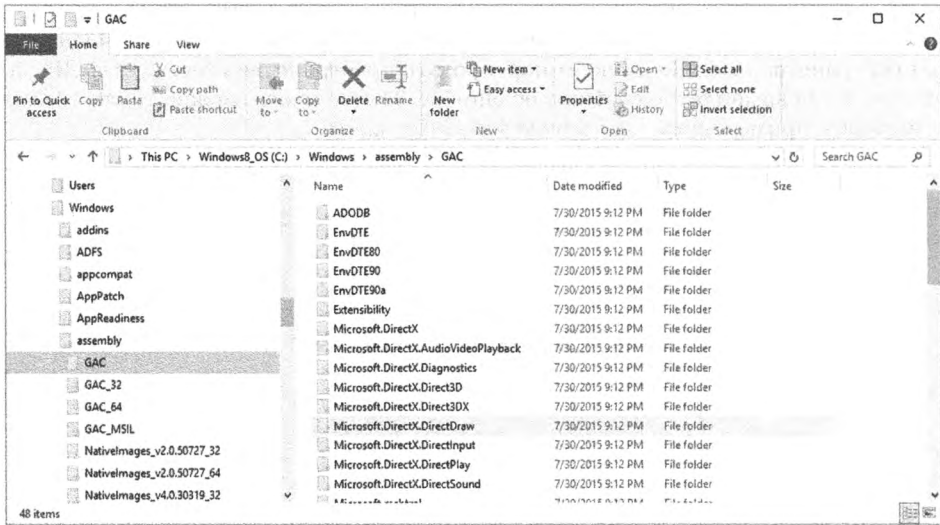


Рис. 1.5. Многие библиотеки .NET размещены в GAC

Исследование сборки с помощью `ildasm.exe`

Если вас начинает беспокоить мысль о необходимости освоения всех пространств имен .NET, то просто вспомните о том, что уникальность пространству имен придает факт наличия в нем типов, которые каким-то образом *семантически* связаны. Следовательно, если в качестве пользовательского интерфейса достаточно простого консольного режима, то можно вообще не думать о пространствах имен, предназначенных для построения интерфейсов настольных и веб-приложений. Если вы создаете приложение рисования, тогда пространства имен для работы с базами данных, скорее всего, не понадобятся. Как и в случае любого нового набора готового кода, изучение должно проходить постепенно.

Утилита `ildasm.exe` (Intermediate Language Disassembler — дизассемблер промежуточного языка), которая поставляется в составе .NET Framework, позволяет загрузить любую сборку .NET и изучить ее содержимое, включая ассоциированный с ней манифест, код CIL и метаданные типов. Инструмент `ildasm.exe` позволяет программистам более подробно разобраться, как их код C# отображается на код CIL, и в итоге помогает понять внутреннюю механику функционирования платформы .NET. Хотя для того, чтобы стать опытным программистом .NET, использовать `ildasm.exe` вовсе не обязательно, настоятельно рекомендуется время от времени применять данный инструмент, чтобы лучше понимать, каким образом написанный код C# укладывается в концепции исполняющей среды.

На заметку! Запустить утилиту `ildasm.exe` довольно легко, открыв окно командной строки Visual Studio, введя в нем `ildasm` и нажав клавишу <Enter>.

После запуска утилиты `ildasm.exe` выберите пункт меню `File⇨Open` (Файл⇨Открыть) и перейдите к сборке, которую желаете исследовать. В целях иллюстрации на рис. 1.6 показано окно утилиты `ildasm.exe` со сборкой `Calc.exe`, сгенерированной на основе файла `Calc.cs`, содержимое которого приводилось ранее в главе. Утилита `ildasm.exe` представляет структуру любой сборки в знакомом древовидном формате.

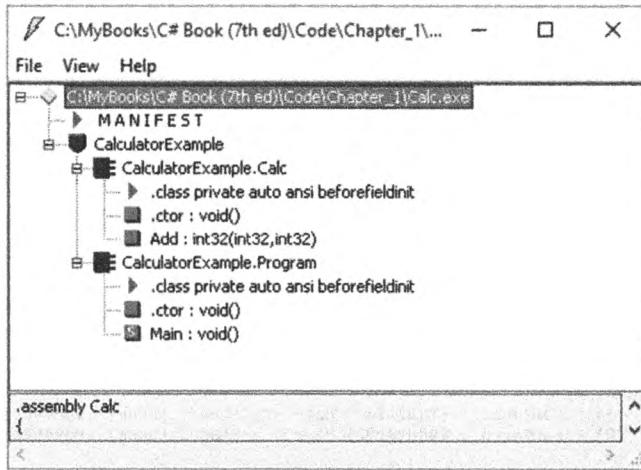


Рис. 1.6. Утилита ildasm.exe позволяет видеть содержащиеся внутри сборки .NET код CIL, манифест и метаданные типов

Просмотр кода CIL

В дополнение к отображению пространств имен, типов и членов, содержащихся в загруженной сборке, утилита ildasm.exe также позволяет просматривать инструкции CIL для заданного члена. Например, двойной щелчок на методе Main() класса Program приводит к открытию отдельного окна с кодом CIL для этого метода (рис. 1.7).

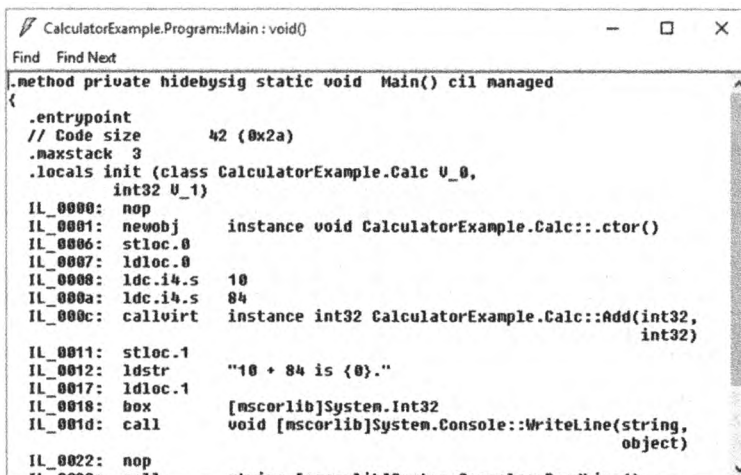


Рис. 1.7. Просмотр кода CIL для метода

Просмотр метаданных типов

Для просмотра метаданных типов, содержащихся в загруженной в текущий момент сборке, необходимо нажать комбинацию клавиш <Ctrl+M>. На рис. 1.8 показаны метаданные для метода Calc.Add().

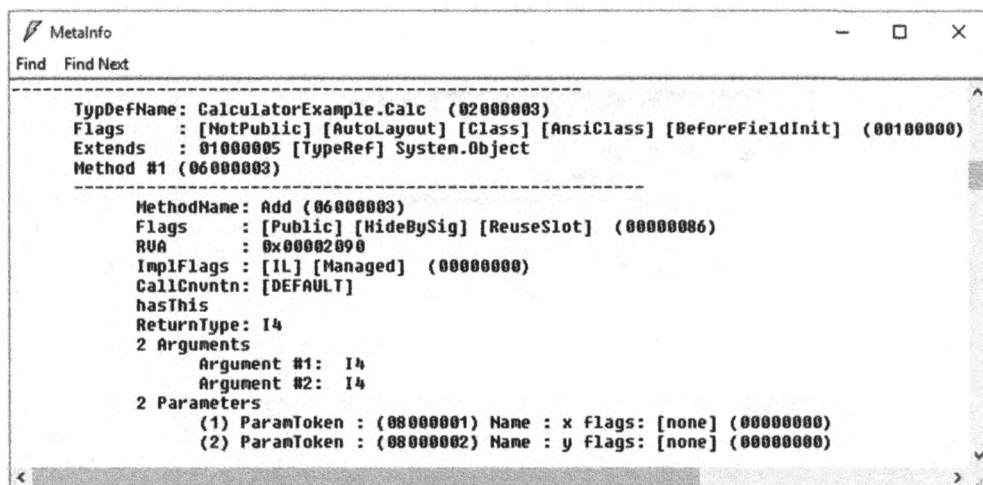


Рис. 1.8. Просмотр метаданных типов с помощью ildasm.exe

Просмотр метаданных сборки (манифеста)

Наконец, чтобы просмотреть содержимое манифеста сборки (рис. 1.9), нужно просто дважды щелкнуть на значке MANIFEST (Манифест) в главном окне утилиты ildasm.

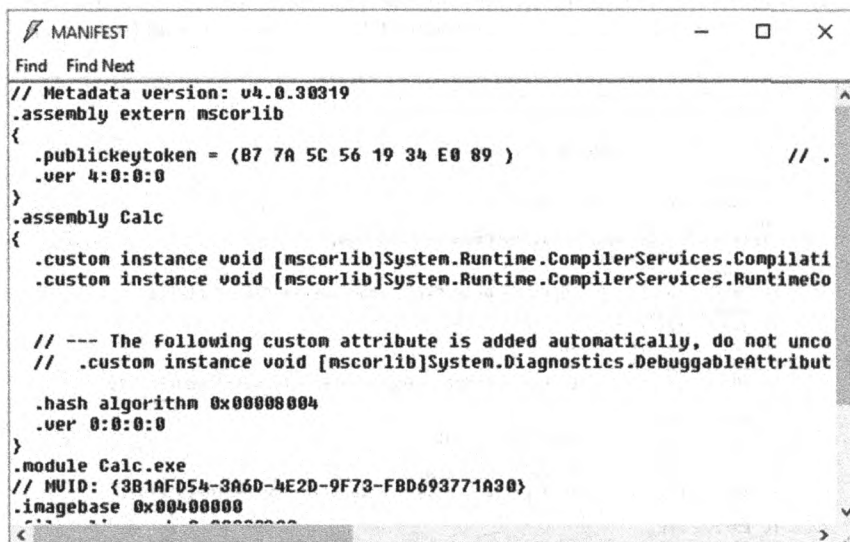


Рис. 1.9. Просмотр данных манифеста с помощью ildasm.exe

Разумеется, утилита ildasm.exe обладает более широким набором возможностей, чем было продемонстрировано здесь, и они еще будут рассматриваться в подходящих ситуациях.

Независимая от платформы природа .NET

Теперь хотелось бы сказать несколько слов о независимой от платформы природе .NET. В прошлых версиях .NET не было широко известно, что приложения .NET могут разрабатываться и выполняться в средах ОС, отличающихся от Microsoft, в числе которых macOS, различные дистрибутивы Linux, Solaris, а также iOS и Android на мобильных устройствах. С выходом платформы .NET Core и связанной с ней шумихой, вероятно, не представляет особого риска предположить, что большое количество разработчиков, по меньшей мере, *осведомлены* о такой возможности. Чтобы понять, почему это достижимо, необходимо принять во внимание еще одну аббревиатуру из мира .NET — CLI (Common Language Infrastructure — общезыковая инфраструктура).

Когда компания Microsoft выпустила язык программирования C# и платформу .NET, был также разработан набор официальных документов, которые описывали синтаксис и семантику языков C# и CIL, формат сборок .NET, основные пространства имен .NET и механику действия исполняющей среды .NET. Документы были поданы в организацию Ecma International (www.ecma-international.org) и утверждены ею в качестве официальных международных стандартов. Наибольший интерес среди них представляют следующие спецификации:

- ECMA-334: C# Language Specification (спецификация языка C#)
- ECMA-335: Common Language Infrastructure (CLI) (спецификация общезыковой инфраструктуры (CLI))

Важность указанных документов становится очевидной, как только вы осознаете тот факт, что они открывают третьим сторонам возможность создания дистрибутивов платформы .NET для любого числа ОС и/или процессоров. Из двух спецификаций ECMA-335 является более насыщенной, поэтому она разбита на разделы, которые описаны в табл. 1.4.

Таблица 1.4. Разделы спецификации CLI

Раздел ECMA-335	Практический смысл
Раздел I. Концепции и архитектура	Описывает общую архитектуру CLI, в том числе правила CTS и CLS и механику функционирования исполняющей среды .NET
Раздел II. Определение и семантика метаданных	Описывает детали метаданных и формат сборок .NET
Раздел III. Набор инструкций CIL	Описывает синтаксис и семантику кода CIL
Раздел IV. Профили и библиотеки	Предоставляет высокоуровневый обзор минимальных и полных библиотек классов, которые должны поддерживаться дистрибутивом .NET
Раздел V. Формат обмена информацией отладки	Описывает стандартный способ обмена информацией отладки между создателями и потребителями CLI
Раздел VI. Дополнения	Предоставляет набор разрозненных деталей, таких как руководящие указания по проектированию библиотек классов и детали реализации компилятора CIL

Имейте в виду, что в разделе IV (“Профили и библиотеки”) описан лишь минимальный набор пространств имен, которые представляют основные службы, ожидаемые от дистрибутива CLI (например, коллекции, консольный ввод-вывод, файловый ввод-вы-

вод, многопоточная обработка, рефлексия, доступ в сеть, основные средства защиты, манипулирование XML-данными). В CLI не определены пространства имен, которые упрощают разработку веб-приложений (ASP.NET), реализацию доступа к базам данных (ADO.NET) или создание настольных приложений с графическим пользовательским интерфейсом (WPF либо Windows Forms).

Однако хорошая новость состоит в том, что альтернативный дистрибутив .NET (названный Mono) расширяет библиотеки CLI совместимыми с Microsoft эквивалентами реализаций ASP.NET, реализаций ADO.NET и реализаций разнообразных инфраструктур для построения настольных приложений с графическим пользовательским интерфейсом, чтобы предложить полнофункциональные платформы разработки приложений производственного уровня. На сегодняшний день помимо специфичной для Windows платформы .NET от Microsoft существуют три крупных реализации, представленные в табл. 1.5.

Таблица 1.5. Дистрибутивы .NET с открытым кодом

Дистрибутив	Описание
Проект Mono	Проект Mono — это дистрибутив CLI с открытым кодом, который ориентирован на разнообразные версии Linux (например, SuSe, Fedora), Mac OS X, устройства iOS (iPad, iPhone), устройства Android и (сюрприз) Windows
Xamarin SDK	Проект Xamarin вырос из проекта Mono и делает возможной разработку межплатформенных приложений с графическим пользовательским интерфейсом для мобильных устройств. Комплект SDK доступен в виде открытого кода, тогда как полный продукт Xamarin — нет
.NET Core	В дополнение к ориентированной на Windows реализации .NET Framework в Microsoft также поддерживается межплатформенная версия .NET, которая сосредоточена на конструировании библиотек кода, приложений доступа к данным, веб-служб и веб-приложений

Проект Mono

Проект Mono будет великолепным вариантом, если вы планируете строить программное обеспечение .NET, которое способно выполняться под управлением разнообразных ОС. Вдобавок ко всем основным пространствам имен .NET проект Mono предоставляет дополнительные библиотеки, делая возможным создание настольного программного обеспечения с графическим пользовательским интерфейсом, веб-приложений ASP.NET и программного обеспечения для мобильных устройств (iPad, iPhone и Android). Загрузить дистрибутив Mono можно по следующему URL:

<http://www.mono-project.com/>

Изначально проект Mono состоит из нескольких инструментов командной строки и всех ассоциированных библиотек кода. Тем не менее, как вы увидите в главе 2, вместе с Mono обычно используется готовая графическая IDE-среда под названием Xamarin Studio. На самом деле проекты Microsoft Visual Studio могут быть загружены в проекты Xamarin и наоборот. Более подробную информацию вы можете почерпнуть в главе 2, но полезно также посетить указанный ниже веб-сайт Xamarin:

<http://xamarin.com/>

Xamarin

Проект Xamarin ведет свой род от проекта Mono и создавался многими инженерами, которые разрабатывали проект Mono. Хотя проект Mono по-прежнему живет и здравствует, с того времени, как проект Xamarin был приобретен компанией Microsoft и начал поставляться в составе Visual Studio 2017, он стал стандартной инфраструктурой для создания межплатформенных приложений с графическим пользовательским интерфейсом — особенно для мобильных устройств. Комплект Xamarin SDK поставляется со всеми версиями Visual Studio 2017, а владельцы лицензии на Visual Studio 2017 Enterprise получают также Xamarin Enterprise.

Microsoft .NET Core

Другой крупный межплатформенный дистрибутив .NET предлагается самой компанией Microsoft. В 2017 году компанией Microsoft была анонсирована версия с открытым кодом полномасштабной (специфичной для Windows) платформы .NET 4.7 Framework, получившая название .NET Core. Дистрибутив .NET Core не является полным дубликатом .NET 4.7 Framework. Взамен .NET Core фокусируется на строительстве веб-приложений ASP.NET, способных выполняться в средах Linux, macOS и Windows. Таким образом, по существу .NET Core можно считать подмножеством полной платформы .NET Framework. На веб-сайте .NET Blog в MSDN есть хорошая статья, в которой приводится сравнение и противопоставление полной платформы .NET Framework и .NET Core. Вот прямая ссылка (если она изменится, то просто выполните поиск в Интернете по ключевой фразе *.NET Core is Open Source*):

<https://blogs.msdn.microsoft.com/dotnet/2014/11/12/net-core-is-open-source/>

К счастью, в состав .NET Core включены все средства C# и несколько основных библиотек. По этой причине большая часть настоящей книги будет напрямую применима также к данному дистрибутиву. Однако вспомните, что дистрибутив .NET Core ориентирован на построение веб-служб REST и веб-приложений и не предоставляет реализаций API-интерфейсов создания графических пользовательских интерфейсов для настольных приложений (таких как WPF или Windows Forms). Если вам необходимо разрабатывать межплатформенные приложения с графическим пользовательским интерфейсом, тогда более удачным выбором будет Xamarin.

Платформа .NET Core подробно рассматривается в части IX книги, включая философию .NET Core, инфраструктуру Entity Framework Core, службы ASP.NET Core и веб-приложения ASP.NET Core.

Полезно отметить, что компания Microsoft выпустила бесплатный, легковесный и межплатформенный редактор кода для помощи в разработке с использованием .NET Core. Этот редактор назван просто — Visual Studio Code. Несмотря на то что он определенно не обладает полным набором средств, как у продукта Microsoft Visual Studio или Xamarin Studio, редактор Visual Studio Code представляет собой удобный инструмент для редактирования кода C# в межплатформенной манере. Хотя в книге он не рассматривается, дополнительные сведения можно получить на следующем веб-сайте:

<https://code.visualstudio.com/>

Резюме

Задачей настоящей главы было формирование концептуальной основы, требуемой для освоения остального материала книги. Сначала исследовались ограничения и сложности, присущие технологиям, которые предшествовали платформе .NET, после чего в общих чертах было показано, как .NET и C# пытаются упростить текущее положение дел.

По существу платформа .NET сводится к механизму исполняющей среды (`mscorlib.dll`) и библиотекам базовых классов (`mscorlib.dll` и связанным с ней библиотекам). Общеязыковая исполняющая среда (CLR) способна обслуживать любые двоичные модули .NET (называемые сборками), которые следуют правилам управляемого кода. Вы видели, что сборки содержат инструкции CIL (в дополнение к метаданным типов и манифестам сборок), которые с помощью JIT-компилятора транслируются в инструкции, специфичные для платформы. Кроме того, вы ознакомились с ролью общеязыковой спецификации (CLS) и общей системы типов (CTS). Вдобавок вы узнали об инструменте для просмотра объектов по имени `ildasm.exe`.

В следующей главе будет предложен обзор распространенных IDE-сред, которые можно применять при построении программных проектов на языке C#. Вас наверняка обрадует тот факт, что в книге будут использоваться полностью бесплатные (и богатые возможностями) IDE-среды, поэтому вы начнете изучение мира .NET без каких-либо финансовых затрат.

ГЛАВА 2

Создание приложений на языке C#

Как программист на языке C#, вы можете выбрать подходящий инструмент среди многочисленных средств для построения приложений .NET. Выбор инструмента (или инструментов) будет осуществляться главным образом на основе трех факторов: сопутствующие финансовые затраты, операционная система (ОС), используемая при разработке программного обеспечения, и вычислительные платформы, на которые оно должно ориентироваться. Цель настоящей главы — предоставить обзор наиболее распространенных интегрированных сред разработки (integrated development environment — IDE), которые поддерживают язык C#. Вы должны иметь в виду, что здесь не будут описаны мельчайшие подробности каждой IDE-среды. В главе лишь предоставляется достаточный объем информации для того, чтобы вы сумели успешно установить среду программирования, и формируется основа, позволяющая двигаться дальше.

В первой части главы будет исследоваться набор IDE-сред от Microsoft, которые дают возможность разрабатывать приложения .NET в среде ОС Windows (7, 8.x и 10). Вы увидите, что некоторые IDE-среды могут применяться для построения только приложений, ориентированных на Windows, в то время как другие поддерживают создание приложений C#, ориентированных на альтернативные ОС и устройства (наподобие macOS, Linux или Android). Во второй части главы будут рассматриваться IDE-среды, которые могут выполняться под управлением ОС, отличающейся от Windows. С их помощью разработчики имеют возможность строить программы на C#, используя компьютеры Apple, а также дистрибутивы Linux.

На заметку! В этой главе будет предложен обзор довольно большого числа IDE-сред. Тем не менее, во всей книге предполагается, что вы применяете (совершенно бесплатную) IDE-среду Visual Studio 2017 Community. Если вы хотите строить свои приложения в среде другой ОС (macOS или Linux), то глава укажет правильное направление, но окна выбранной вами IDE-среды будут отличаться от тех, которые изображены на экранных снимках, приводимых в тексте.

Построение приложений .NET в среде Windows

Изучая материал данной главы, вы увидите, что для построения приложений C# можно выбирать подходящий вариант из множества IDE-сред; одни IDE-среды предлагаются компанией Microsoft, а другие поступают от независимых поставщиков (многие среды поставляются с открытым кодом). Вопреки тому, что вы могли подумать, в настоящее время многочисленные IDE-среды производства Microsoft являются бесплатными.

Таким образом, если ваш основной интерес заключается в построении программного обеспечения .NET для ОС Windows (7, 8.x или 10), то вы обнаружите следующие главные варианты:

- Visual Studio Community
- Visual Studio Professional
- Visual Studio Enterprise

Редакции Express были удалены, оставляя три версии Visual Studio 2017. Редакции Community и Professional *по существу* одинаковы; главное техническое отличие в том, что редакция Professional имеет средство CodeLens, а редакция Community — нет. Более значительная разница связана с моделью лицензирования. Редакция Community лицензируется для использования с проектами с открытым кодом, в учебных учреждениях и на малых предприятиях. Редакция Professional лицензируется для крупномасштабной разработки. Редакция Enterprise по сравнению с Professional вполне ожидаемо предлагает множество дополнительных средств.

На заметку! Детали лицензирования доступны на веб-сайте <https://www.visualstudio.com>. Лицензирование продуктов Microsoft может оказаться сложным и в книге его подробности не раскрываются. Для написания (и проработки) настоящей книги законно применять редакцию Community.

Каждая IDE-среда поставляется с развитыми редакторами кода, основными визуальными конструкторами баз данных, встроенными визуальными отладчиками, визуальными конструкторами графических пользовательских интерфейсов для настольных и веб-приложений и т.д. Поскольку все они разделяют общий набор основных средств, между ними легко перемещаться и чувствовать себя вполне комфортно в отношении их стандартной эксплуатации.

Установка Visual Studio 2017

Прежде чем среду Visual Studio 2017 можно будет использовать для разработки, запуска и отладки приложений C#, ее понадобится установить. Процесс установки значительно отличается от предшествующих версий, а потому полезно обсудить его более подробно.

На заметку! Загрузить Visual Studio 2017 Community можно по адресу <https://www.visualstudio.com/downloads>.

Процесс установки Visual Studio 2017 теперь разбит на рабочие нагрузки по типам приложений. В результате появляется возможность устанавливать только те компоненты, которые нужны для выполнения планируемых работ. Например, если вы собираетесь строить веб-приложения, тогда должны установить рабочую нагрузку "ASP.NET and web development" ("Разработка приложений ASP.NET и веб-приложений").

Еще одно (крайне) важное изменение связано с тем, что Visual Studio 2017 поддерживает подлинную установку бок о бок. Обратите внимание, что речь не о параллельной установке с предшествующими версиями, а о самой среде Visual Studio 2017! На главной рабочей станции может быть установлена редакция Visual Studio 2017 Enterprise. Для работы с этой книгой будет применяться редакция Visual Studio Community. В случае версии Visual Studio 2015 (и предыдущего издания данной книги) в подобной ситуации приходилось использовать разные машины. Теперь все можно делать на одной машине. При наличии редакции Professional или Enterprise, предоставленной вашим

работодателем, вы по-прежнему можете установить редакцию Community для работы над проектами с открытым кодом (или с кодом данной книги).

После запуска программы установки Visual Studio 2017 Community появляется экран, показанный на рис. 2.1. На нем предлагаются все доступные рабочие нагрузки, возможность выбора отдельных компонентов и сводка (в правой части), которая отображает, что было выбрано. Обратите внимание на предупреждение в нижней части экрана: "A nickname must be provided to disambiguate this install" ("Для разрешения неоднозначности данной установки должен быть предоставлен псевдоним"). Причина в том, что на текущей машине присутствуют другие установленные копии Visual Studio 2017. Если установка производится впервые, то такое предупреждение не отображается.

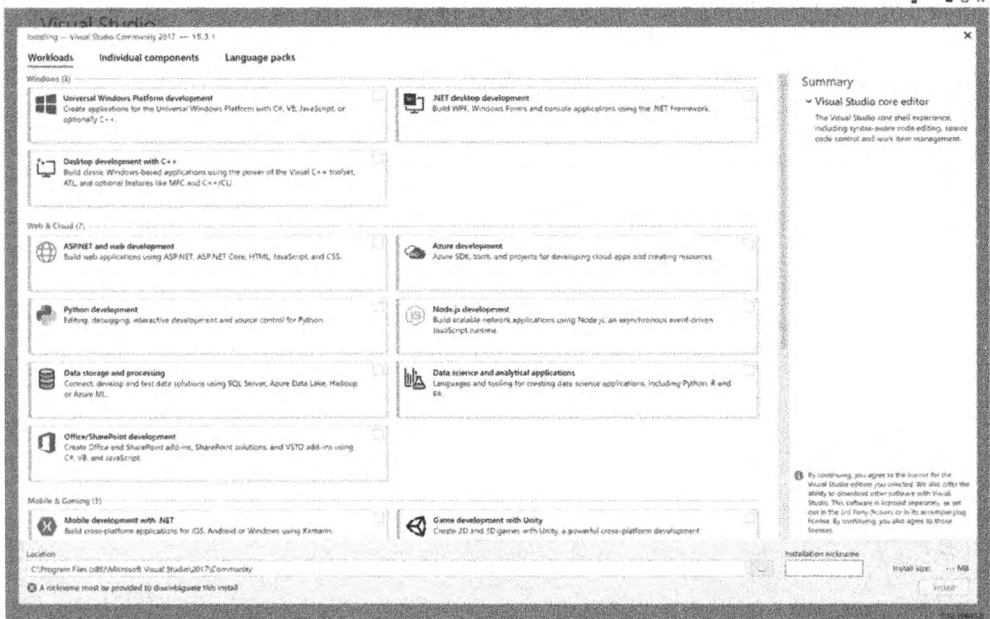


Рис. 2.1. Новая программа установки Visual Studio

Для настоящей книги понадобится установить следующие рабочие нагрузки:

- .NET desktop development (Разработка настольных приложений .NET)
- ASP.NET and web development (Разработка приложений ASP.NET и веб-приложений)
- Data storage and processing (Хранилище и обработка данных)
- .NET Core cross-platform development (Разработка межплатформенных приложений .NET Core)

Также придется добавить псевдоним установки — ProC#. Результирующий экран представлен на рис. 2.2.

Перейдите на вкладку Individual Components (Отдельные компоненты) в верхней части экрана и выберите перечисленные ниже дополнительные элементы:

- .NET Core runtime (Исполняющая среда .NET Core)
- .NET Framework 4.6.2 SDK (Комплект .NET Framework 4.6.2 SDK)

- .NET Framework 4.6.2 targeting pack (Целевой пакет .NET Framework 4.6.2)
- .NET Framework 4.7 SDK (Комплект .NET Framework 4.7 SDK)
- .NET Framework 4.7 targeting pack (Целевой пакет .NET Framework 4.7)
- Class Designer (Визуальный конструктор классов)
- Testing tools core features (Основные функциональные возможности инструментов тестирования)
- Visual Studio Tools for Office (VSTO) (Инструменты Visual Studio для офиса)

После выбора всех указанных элементов щелкните на кнопке Install (Установить). В итоге вам будет предоставлено все, что необходимо для проработки примеров в настоящей книге, включая новую часть по .NET Core.

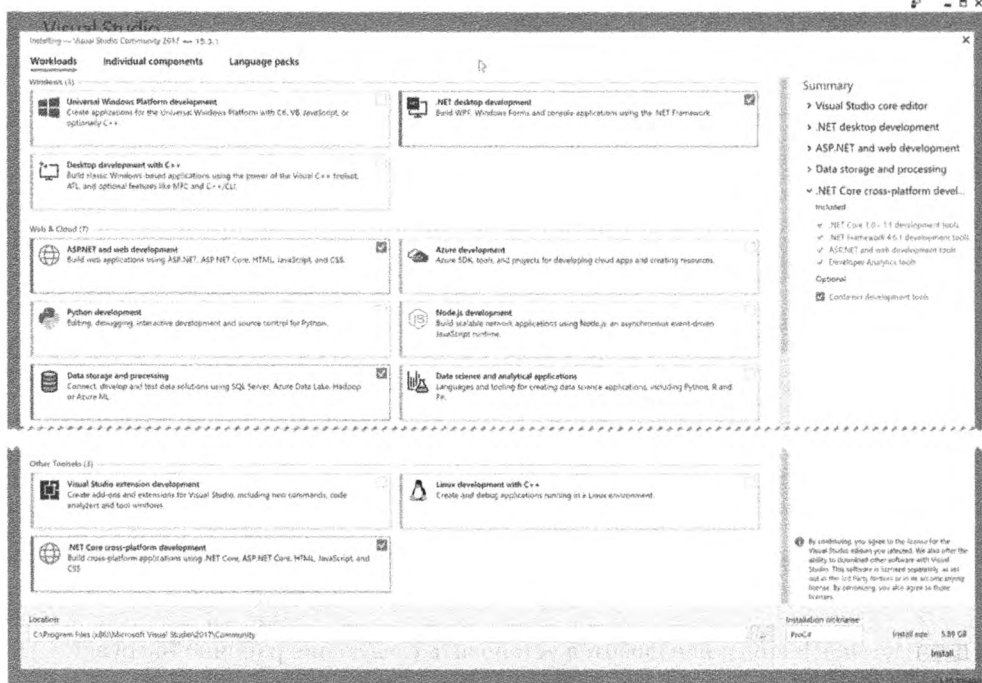


Рис. 2.2. Выбранные рабочие нагрузки

Испытание Visual Studio 2017

Среда Visual Studio 2017 — это универсальный инструмент для разработки программного обеспечения с помощью платформы .NET и языка C#. Давайте бегло посмотрим на работу Visual Studio, построив простое консольное приложение Windows.

Построение приложений .NET

В качестве первого дела мы посвятим какое-то время построению простого приложения C#, памятуя о том, что проиллюстрированные здесь аспекты относятся ко всем редакциям Visual Studio.

Диалоговое окно New Project и редактор кода C#

Теперь, имея установленную копию Visual Studio, выберите пункт меню **File**⇒**New Project** (Файл⇒Создать проект). Как видно в открывшемся диалоговом окне **New Project** (Новый проект), показанном на рис. 2.3, данная IDE-среда предлагает поддержку для консольных приложений, приложений WPF/Windows Forms, служб Windows и многого другого. Для начала создайте новый проект **Console App** (Консольное приложение) на C# по имени **SimpleCSharpConsoleApp**, не забыв изменить целевую версию платформы на **.NET Framework 4.7**.

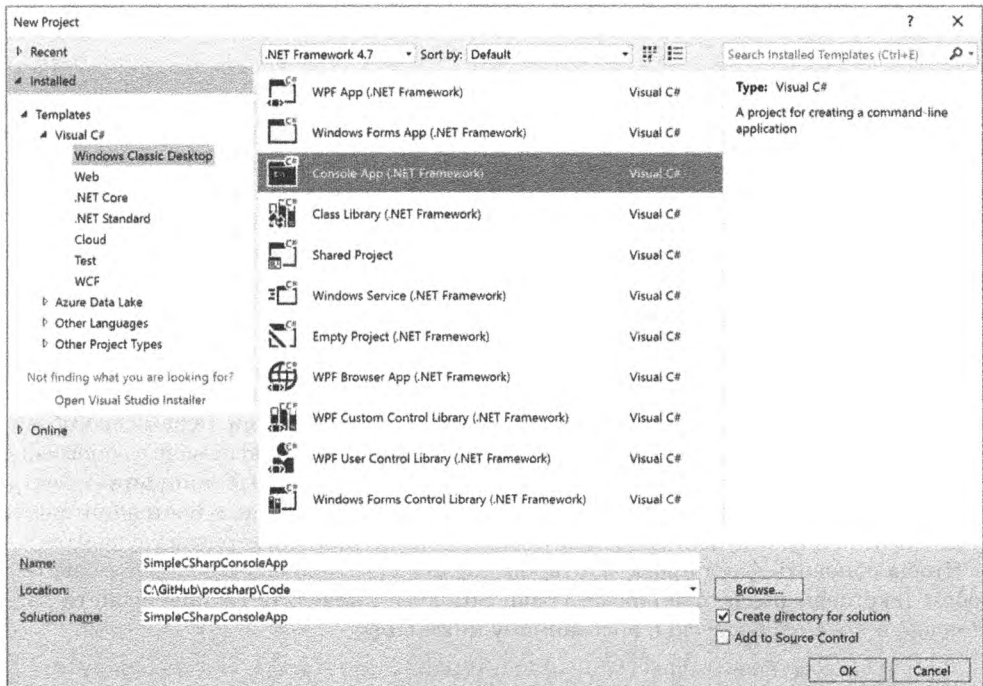


Рис. 2.3. Диалоговое окно New Project

На рис. 2.3 легко заметить, что среда Visual Studio способна создавать множество типов приложений, в том числе настольные приложения Windows, веб-приложения, приложения .NET Core и т.д. Они будут рассматриваться повсеместно в этой книге.

На заметку! Если в раскрывающемся списке вариант для .NET Framework 4.7 отсутствует, тогда вам придется установить пакет Microsoft .NET Framework 4.7 Developer Pack, доступный по адресу <https://www.microsoft.com/ru-ru/download/details.aspx?id=55168>. Чтобы получить упомянутый пакет, можно также выбрать в раскрывающемся списке вариант **Install other frameworks...** (Установить другие инфраструктуры...).

После создания проекта вы увидите содержимое начального файла кода C# (по имени **Program.cs**), который открывается в редакторе кода. Добавьте в метод **Main()** следующий код C#. Набирая код, вы заметите, что во время применения операции точки активизируется средство **IntelliSense**.


```
static void Main(string[] args)
{
    // Настройка консольного пользовательского интерфейса.
    Console.Title = "My Rocking App";
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.BackgroundColor = ConsoleColor.Blue;
    Console.WriteLine("*****");
    Console.WriteLine("***** Welcome to My Rocking App *****");
    Console.WriteLine("*****");
    Console.BackgroundColor = ConsoleColor.Black;

    // Ожидание нажатия клавиши <Enter>.
    Console.ReadLine();
}
```

Здесь используется класс `Console`, определенный в пространстве имен `System`. Поскольку пространство имен `System` было автоматически включено посредством оператора `using` в начале файла, указывать `System` перед именем класса не обязательно (например, `System.Console.WriteLine()`). Данная программа не делает ничего особо интересного; тем не менее, обратите внимание на последний вызов `Console.ReadLine()`. Он просто обеспечивает поведение, при котором пользователь должен нажать клавишу `<Enter>`, чтобы завершить приложение. Если этого не сделать, то программа исчезнет почти мгновенно при проведении ее отладки!

Использование средств C# 7.1

На время написания книги среда `Visual Studio` не поддерживала создание проектов C# 7.1. Задействовать новые средства языка можно двумя способами. Первый способ — обновить файл проекта вручную, а второй — позволить `Visual Studio` помочь с обновлением файла проекта (в форме значка с изображением лампочки, который инициирует быстрое исправление). Несмотря на то что второй способ выглядит проще, в настоящий момент он ненадежен, но в будущих выпусках `Visual Studio` ситуация наверняка улучшится.

Чтобы обновить файл проекта, откройте файл `SimpleCSharpConsoleApp.csproj` в любом текстовом редакторе (кроме `Visual Studio`) и приведите группы свойств `Debug` (отладка) и `Release` (выпуск) к показанному ниже виду:

```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
  <PlatformTarget>AnyCPU</PlatformTarget>
  <DebugSymbols>>true</DebugSymbols>
  <DebugType>full</DebugType>
  <Optimize>>false</Optimize>
  <OutputPath>bin\Debug\</OutputPath>
  <DefineConstants>DEBUG;TRACE</DefineConstants>
  <ErrorReport>prompt</ErrorReport>
  <WarningLevel>4</WarningLevel>
  <LangVersion>7.1</LangVersion>
</PropertyGroup>
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
  <PlatformTarget>AnyCPU</PlatformTarget>
  <DebugType>pdbonly</DebugType>
  <Optimize>true</Optimize>
  <OutputPath>bin\Release\</OutputPath>
  <DefineConstants>TRACE</DefineConstants>
  <ErrorReport>prompt</ErrorReport>
  <WarningLevel>4</WarningLevel>
  <LangVersion>7.1</LangVersion>
</PropertyGroup>
```

Запуск и отладка проекта

Теперь для запуска программы и просмотра вывода можете просто нажать комбинацию клавиш <Ctrl+F5> (или выбрать пункт меню Debug⇒Start Without Debugging (Отладка⇒Запустить без отладки)). На экране появится окно консоли Windows с вашим специальным (раскрашенным) сообщением. Вы должны понимать, что когда “запускаете” свою программу, то обходите интегрированный отладчик.

Если написанный код нужно отладить (что определенно станет важным при построении более крупных программ), тогда первым делом понадобится поместить точки останова на операторы кода, которые необходимо исследовать. Хотя в рассматриваемом примере не особенно много кода, установите точку останова, щелкнув на крайней слева полосе серого цвета в окне редактора кода (обратите внимание, что точки останова помечаются значком в виде красного кружка (рис. 2.4)).

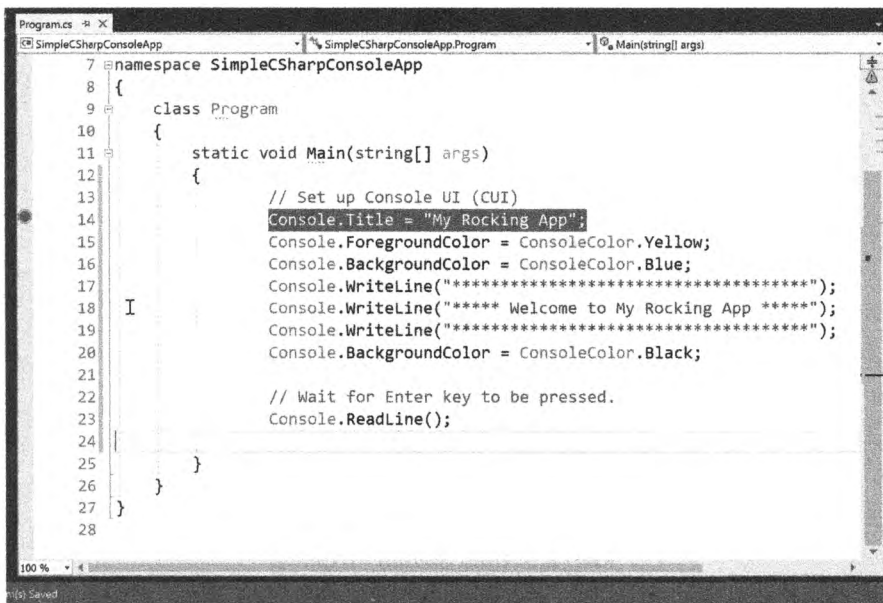


Рис. 2.4. Установка точки останова

Если теперь нажать клавишу <F5> (или выбрать пункт меню Debug⇒Start Debugging (Отладка⇒Запустить с отладкой) либо щелкнуть на кнопке с зеленой стрелкой и надписью Start (Пуск) в панели инструментов), то программа будет прекращать работу на каждой точке останова. Как и можно было ожидать, у вас есть возможность взаимодействовать с отладчиком с помощью разнообразных кнопок панели инструментов и пунктов меню IDE-среды. После прохождения всех точек останова приложение в конечном итоге завершится, когда закончится метод Main().

На заметку! Предлагаемые Microsoft среды IDE снабжены современными отладчиками, и в последующих главах вы изучите разнообразные приемы работы с ними. Пока нужно лишь знать, что при нахождении в сеансе отладки в меню Debug появляется большое количество полезных пунктов. Выделите время на ознакомление с ними.

Окно Solution Explorer

Взглянув на правую часть IDE-среды, вы заметите окно под названием Solution Explorer (Проводник решений), в котором отображено несколько важных элементов. Первым делом обратите внимание, что IDE-среда создала решение с единственным проектом (рис. 2.5). Поначалу это может сбивать с толку, т.к. решение и проект имеют одно и то же имя (SimpleCSharpConsoleApp). Идея в том, что “решение” может содержать множество проектов, работающих совместно. Скажем, в состав решения могут входить три библиотеки классов, одно приложение WPF и одна веб-служба WCF. В начальных главах книги будет всегда применяться одиночный проект; однако, когда мы займемся построением более сложных примеров, будет показано, каким образом добавлять новые проекты в первоначальное пространство решения.

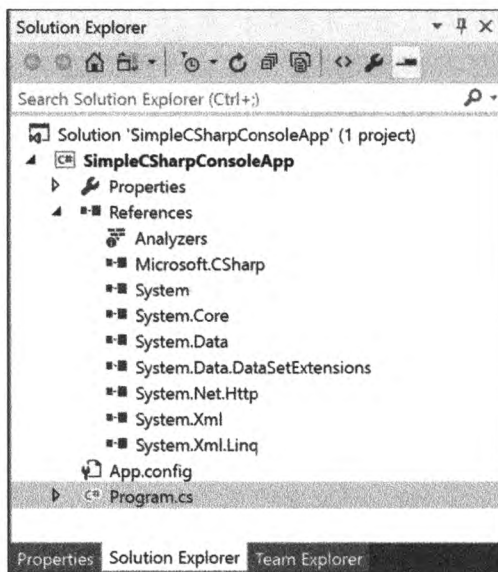


Рис. 2.5. Окно Solution Explorer

На заметку! Учтите, что в случае выбора решения в самом верху окна Solution Explorer система меню IDE-среды будет отображать набор пунктов, который отличается от ситуации, когда выбран проект. Если вы когда-нибудь обнаружите, что определенный пункт меню исчез, то проверьте, не выбран ли случайно неправильный узел.

Вы также заметите узел References (Ссылки). Его можно использовать, когда приложение нуждается в ссылке на дополнительные библиотеки .NET помимо тех, которые по умолчанию включаются для данного типа проекта. Поскольку был создан проект Console Application на языке C#, вы увидите несколько автоматически добавленных библиотек, таких как System.dll, System.Core.dll, System.Data.dll и т.д. (обратите внимание, что элементы, перечисленные в узле References, не отображают файловое расширение .dll). Вскоре будет показано, как добавлять библиотеки к проекту.

На заметку! Вспомните из главы 1, что все проекты .NET имеют доступ к фундаментальной библиотеке по имени mscorlib.dll. Она настолько необходима, что даже явно не отображается в окне Solution Explorer.

Окно Object Browser

Если щелкнуть правой кнопкой мыши на любой библиотеке в узле References и выбрать в контекстном меню пункт View in Object Browser (Просмотреть в браузере объектов), то откроется интегрированное окно Object Browser (Браузер объектов); его также можно открыть с помощью меню View (Вид). С применением данного инструмента можно просматривать разнообразные пространства имен в сборке, типы в пространстве имен и члены каждого типа. На рис. 2.6 показано окно Object Browser, которое отображает ряд пространств имен в постоянно присутствующей сборке mscorlib.dll.

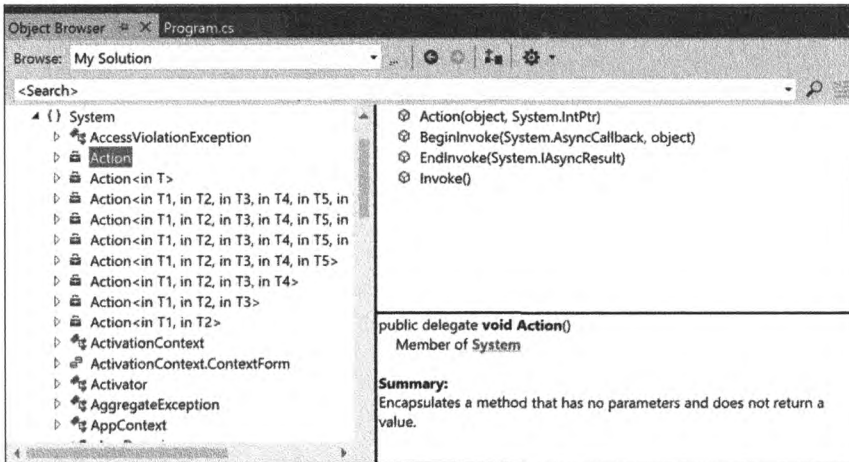


Рис. 2.6. Окно Object Browser

Окно Object Browser может оказаться полезным, когда необходимо увидеть внутреннюю организацию какой-то библиотеки .NET, а также получить краткое описание заданного элемента. Кроме того, обратите внимание на поле <Search> (<Поиск>) в верхней части окна. Оно может быть удобным, если вы знаете имя типа, который нужно использовать, но не имеете понятия, где он находится. В качестве связанного замечания имейте в виду, что по умолчанию средство поиска будет искать только в библиотеках, задействованных в решении (чтобы производить поиск в рамках всей инфраструктуры .NET Framework, понадобится изменить выбранный элемент в раскрывающемся списке Browse (Просмотр)).

Ссылка на дополнительные сборки

Продолжая исследование, давайте добавим сборку (также известную как библиотека кода), которая не включается в проект Console Application автоматически. Для этого щелкните правой кнопкой мыши на узле References в окне Solution Explorer и выберите в контекстном меню пункт Add Reference (Добавить ссылку) или же выберите пункт меню Project⇒Add Reference (Проект⇒Добавить ссылку). В открывшемся диалоговом окне Add Reference (Добавление ссылки) отыщите библиотеку по имени System.Windows.Forms.dll (файловое расширение снова не показано) и отметьте флажок рядом с ней (рис. 2.7).

После щелчка на кнопке ОК новая библиотека добавляется к нашему набору ссылок (вы увидите ее в списке под узлом References). Тем не менее, как объяснялось в главе 1, добавление ссылки на библиотеку — лишь первый шаг. Чтобы применять типы в определенном файле кода C#, понадобится указать оператор using.

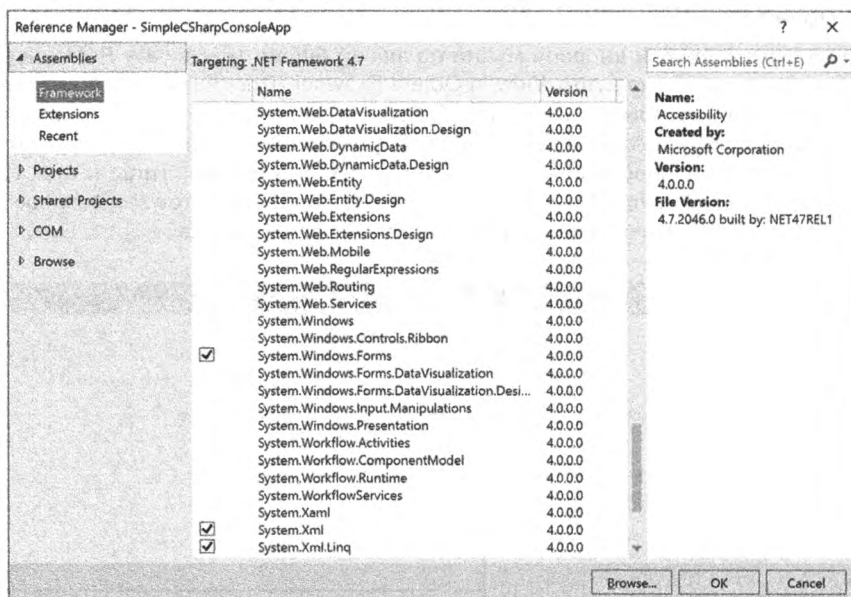


Рис. 2.7. Диалоговое окно Add Reference

Добавьте следующую строку к директивам `using` в своем файле кода:

```
using System.Windows.Forms;
```

Затем поместите приведенную ниже строку кода сразу после вызова `Console.ReadLine()` в методе `Main()`:

```
MessageBox.Show("All done!");
```

Запустив или начав отладку программы еще раз, вы обнаружите, что перед завершением работы программы появляется простое диалоговое окно сообщения.

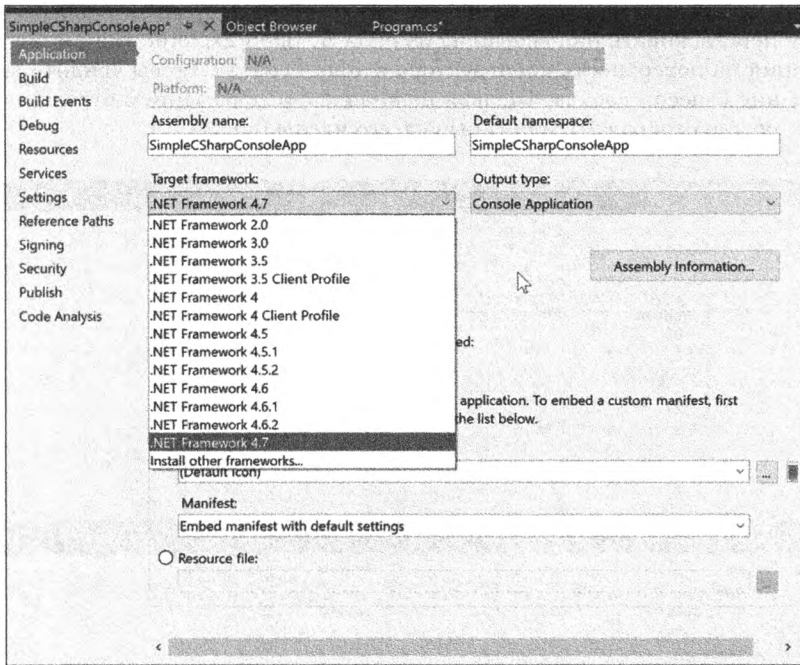
Просмотр свойств проекта

Далее обратите внимание на узел `Properties` (Свойства) в окне `Solution Explorer`. Двойной щелчок на нем приводит к открытию сложно устроенного редактора конфигурации проекта. Например, на рис. 2.8 видно, что можно изменять версию платформы .NET Framework, на которую нацелено решение.

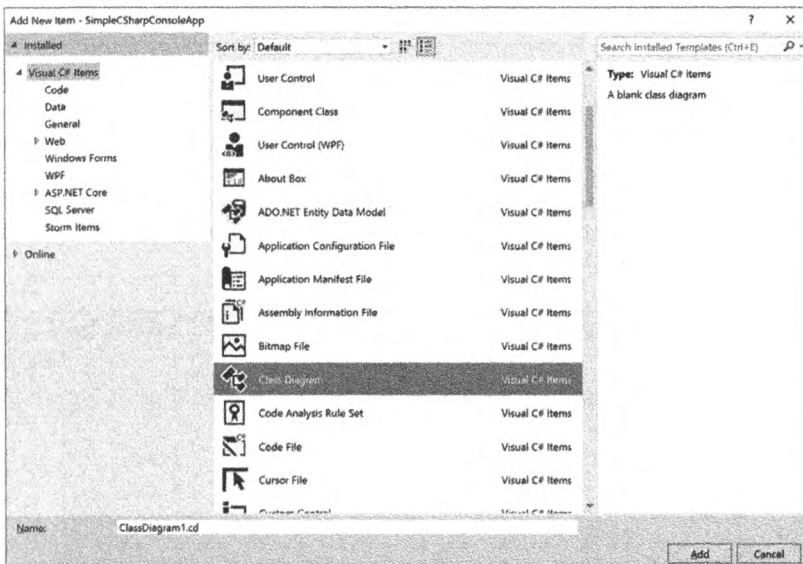
Различные аспекты окна свойств проекта будут обсуждаться в оставшихся главах книги. Здесь можно устанавливать разнообразные настройки безопасности, назначать сборке строгое имя (глава 14), развертывать приложение, вставлять ресурсы приложения и конфигурировать события, происходящие до и после построения.

Визуальный конструктор классов

Среда Visual Studio также снабжает вас возможностью конструирования классов и других типов (вроде интерфейсов или делегатов) в визуальной манере. Утилита `Class Designer` (Визуальный конструктор классов) позволяет просматривать и модифицировать отношения между типами (классами, интерфейсами, структурами, перечислениями и делегатами) в проекте. С помощью данного средства можно визуально добавлять (или удалять) члены типа с отражением этих изменений в соответствующем файле кода C#. Кроме того, по мере модификации отдельного файла кода C# изменения отражаются в диаграмме классов.

**Рис. 2.8.** Окно свойств проекта

Для доступа к инструментам визуального конструктора классов сначала понадобится вставить новый файл диаграммы классов. Выберите пункт меню **Project**⇒**Add New Item** (Проект⇒Добавить новый элемент) и в открывшемся окне найдите элемент **Class Diagram** (Диаграмма классов), как показано на рис. 2.9.

**Рис. 2.9.** Вставка файла диаграммы классов в текущий проект

Первоначально поверхность визуального конструктора будет пустой; тем не менее, вы можете перетаскивать на нее файлы из окна Solution Explorer. Например, после перетаскивания на поверхность конструктора файла Program.cs вы увидите визуальное представление класса Program. Щелкая на значке с изображением стрелки для заданного типа, можно отображать или скрывать его члены (рис. 2.10).

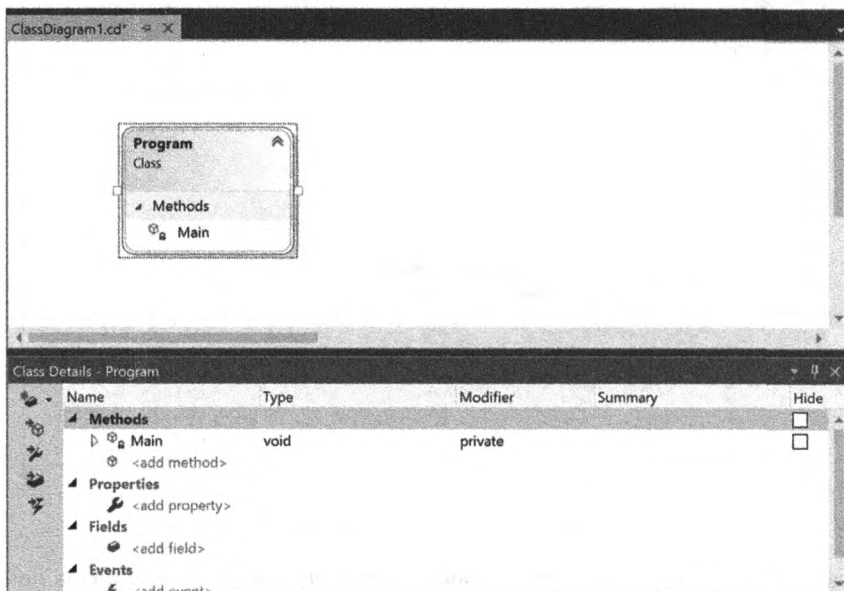


Рис. 2.10. Просмотр диаграммы классов

На заметку! Используя панель инструментов утилиты Class Designer, можно настраивать параметры отображения поверхности визуального конструктора.

Утилита Class Designer работает в сочетании с двумя другими средствами Visual Studio — окном Class Details (Детали класса), которое открывается через меню View ⇒ Other Windows (Вид ⇒ Другие окна), и панелью инструментов Class Designer, отображаемой выбором пункта меню View ⇒ Toolbox (Вид ⇒ Панель инструментов). Окно Class Details не только показывает подробные сведения о текущем выбранном элементе диаграммы, но также позволяет модифицировать существующие члены и вставлять новые члены на лету (рис. 2.11).

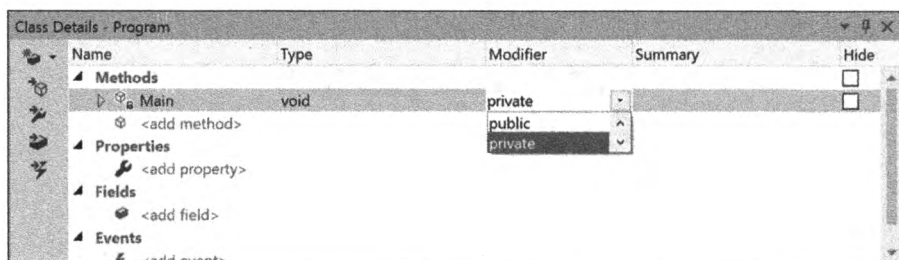


Рис. 2.11. Окно Class Details

Панель инструментов Class Designer, которая также может быть активизирована с применением меню View, позволяет вставлять в проект новые типы (и создавать между ними отношения) визуальным образом (рис. 2.12). (Чтобы видеть эту панель инструментов, должно быть активным окно диаграммы классов.) По мере выполнения таких действий IDE-среда создает на заднем плане новые определения типов на C#.

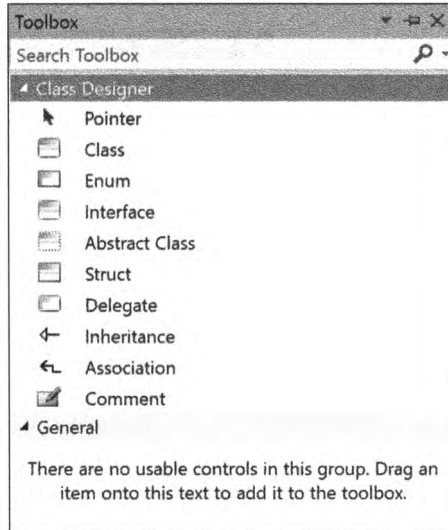


Рис. 2.12. Панель инструментов Class Designer

В качестве примера перетащите новый элемент Class (Класс) из панели инструментов Class Designer в окно Class Designer. В открывшемся диалоговом окне назначьте ему имя Car. В результате создается новый файл C# по имени Car.cs и автоматически добавляется к проекту. Теперь, используя окно Class Details, добавьте открытое поле типа string с именем PetName (рис. 2.13).

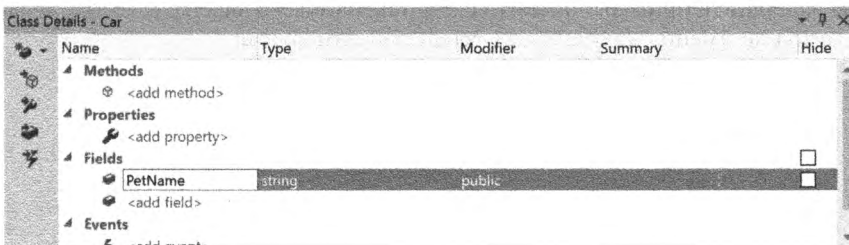


Рис. 2.13. Добавление поля с помощью окна Class Details

Затем вы увидите, что определение класса Car на C# было соответствующим образом обновлено (за исключением приведенного ниже комментария):

```
public class Car
{
    // Использовать открытые данные обычно не рекомендуется,
    // но здесь это упрощает пример.
    public string petName;
}
```


Снова активизируйте утилиту Class Designer, перетащите на поверхность визуального конструктора еще один новый элемент Class и назначьте ему имя SportsCar. Далее выберите значок Inheritance (Наследование) в панели инструментов Class Designer и щелкните в верхней части значка SportsCar. Щелкните в верхней части значка класса Car. Если все было сделано правильно, тогда класс SportsCar станет производным от класса Car (рис. 2.14).

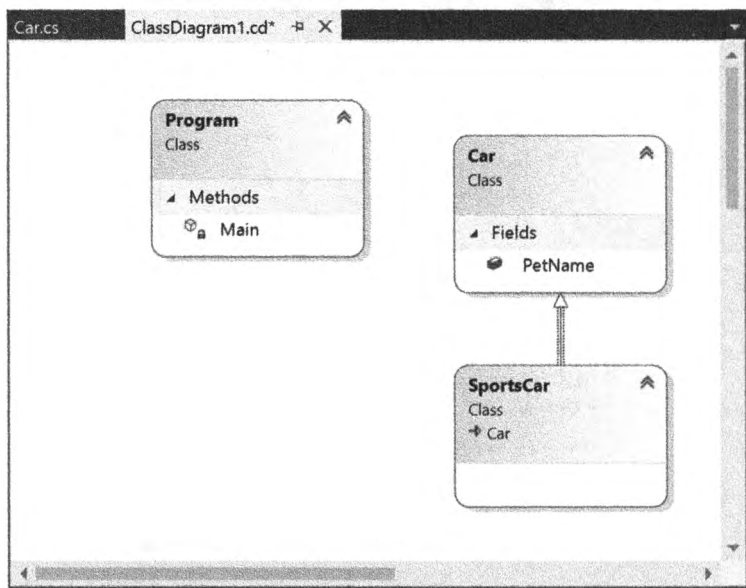


Рис. 2.14. Визуальное наследование от существующего класса

На заметку! Концепция наследования исчерпывающе объясняется в главе 6.

Чтобы завершить пример, обновите сгенерированный класс SportsCar, добавив открытый метод по имени GetPetName() со следующим кодом:

```
public class SportsCar : Car
{
    public string GetPetName()
    {
        petName = "Fred";
        return petName;
    }
}
```

Как и можно было ожидать, визуальный конструктор типов является одним из многочисленных средств Visual Studio Community. Ранее уже упоминалось, что в настоящем издании книги предполагается применение Visual Studio Community в качестве избранной IDE-среды. В последующих главах вы изучите намного больше возможностей данного инструмента.

Исходный код. Проект SimpleCSharpConsoleApp находится в подкаталоге Chapter_2.

Visual Studio 2017 Professional

Как упоминалось в начале главы, главное отличие между редакциями Community и Professional связано с разрешенными сценариями использования. Если в настоящее время вы занимаете должность инженера по разработке программного обеспечения, то велики шансы того, что компания приобрела копию этой редакции, которая и является вашим избранным инструментом.

Visual Studio 2017 Enterprise

В завершение исследования редакций Visual Studio, выполняющихся под управлением Windows, давайте кратко рассмотрим Visual Studio 2017 Enterprise. Редакция Visual Studio 2017 Enterprise обладает всеми теми же средствами, что и Visual Studio Professional, а также дополнительными возможностями, направленными на организацию совместной корпоративной разработки и поддержку межплатформенной разработки мобильных приложений с помощью Xamarin.

Мы больше не будем говорить о редакции Visual Studio 2017 Enterprise. Для целей данной книги подойдет любая из трех версий (Community, Professional и Enterprise).

На заметку! По адресу <https://www.visualstudio.com/ru/vs/compare/> можно ознакомиться с результатами сравнения версий Community, Professional и Enterprise.

Система документации .NET Framework

Последним аспектом Visual Studio, с которым вы должны уметь работать с самого начала, является полностью интегрированная справочная система. Документация .NET Framework представляет собой исключительно хороший, понятный и насыщенный полезной информацией источник. Из-за огромного количества предопределенных типов .NET (их тысячи) необходимо выделить время на исследование предлагаемой документации, иначе вряд ли вас ожидает особый успех на поприще разработки приложений .NET.

Вы можете просматривать документацию .NET Framework по следующему адресу:

<https://docs.microsoft.com/ru-ru/dotnet/>

На заметку! Не будет удивительным, если в Microsoft когда-нибудь изменят местоположение онлайн-документации по библиотеке классов .NET Framework. В таком случае поиск в Интернете по ключевой фразе *.NET Framework Class Library documentation* поможет выяснить ее текущее местоположение.

Оказавшись на главной странице, щелкните на элементе Switch to the Library TOC view (Переключиться на представление содержания библиотеки). Представление изменится на более легкое для навигации. Найдите узел .NET Development (Разработка .NET) и щелкните на стрелке, чтобы развернуть его. Затем щелкните на стрелке рядом с узлом .NET Framework 4.7, 4.6, and 4.5 (.NET Framework 4.7, 4.6 и 4.5). Наконец, щелкните на элементе .NET Framework class library (Библиотека классов .NET Framework). Далее вы можете применять окно с древовидным представлением для просмотра каждого пространства имен, типа и члена платформы. На рис. 2.15 показан пример просмотра типов пространства имен System.

На заметку! Хотя это может выглядеть надоедливым повторением, нельзя еще раз не напомнить о том, насколько важно научиться пользоваться документацией .NET Framework. Ни одна книга, какой бы объемной она ни была, не способна охватить все аспекты платформы .NET. Непременнo выделить время на освоение работы со справочной системой — впоследствии это окупится сторицей.

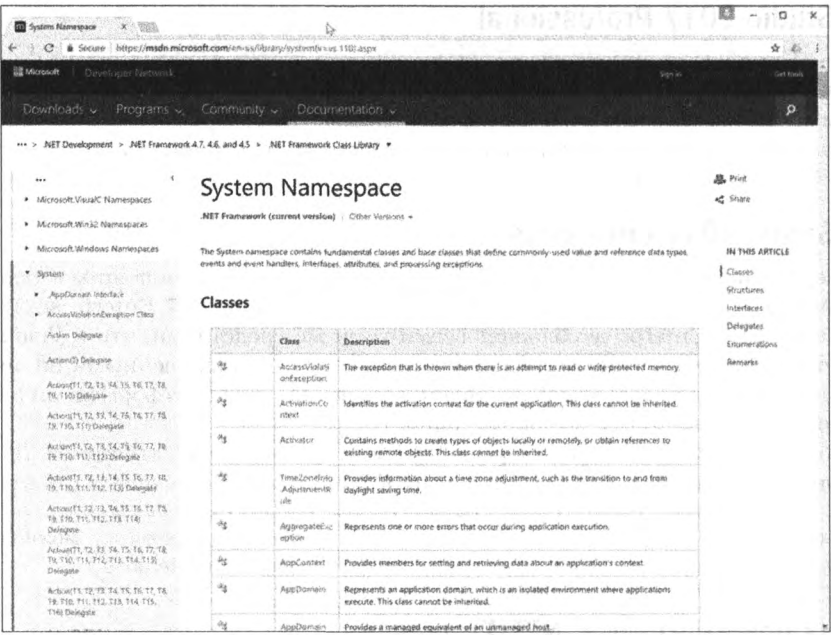


Рис. 2.15. Просмотр документации .NET Framework в онлайн-режиме

Построение приложений .NET под управлением операционной системы, отличающейся от Windows

Существует несколько вариантов построения приложений .NET под управлением ОС, отличающихся от Windows. Помимо Xamarin Studio есть также Visual Studio для Mac и Visual Studio Code (функционирует и в среде Linux). Типы приложений, которые могут быть построены с помощью упомянутых сред разработки, ограничиваются приложениями, ориентированными на использование под управлением .NET Core (Visual Studio Code и Visual Studio для Mac) либо на мобильных устройствах (Visual Studio для Mac, Xamarin Studio).

Вот и все, что мы планировали сообщить в настоящей книге об инструментах разработки в средах, отличающихся от Windows. Но будьте уверены, что в Microsoft охватывают всех разработчиков, а не только тех, кто располагает компьютерами на основе Windows.

Резюме

Как вы могли заметить, в вашем распоряжении появилось много новых игрушек! Целью главы было проведение краткого экскурса в основные средства, которые программист на языке C# может задействовать во время разработки. В главе указывалось, что если вас интересует только построение приложений .NET на машине разработки с ОС Windows, то лучшим выбором следует считать загрузку и установку среды Visual Studio Community. Вдобавок упоминалось о том, что в данном издании книги будет применяться эта конкретная IDE-среда, стремительно набирающая обороты. Соответственно, предстоящие экранные снимки, пункты меню и визуальные конструкторы рассчитаны на то, что вы используете Visual Studio Community.

Если вы хотите строить приложения .NET Core или межплатформенные мобильные приложения под управлением ОС, отличающейся от Windows, тогда лучшим выбором будет Visual Studio для Mac, Visual Studio Code или Xamarin Studio.

часть II

Основы программирования на C#

В этой части

Глава 3. Главные конструкции программирования на C#: часть I

Глава 4. Главные конструкции программирования на C#: часть II

ГЛАВА 3

Главные конструкции программирования на C#: часть I

В настоящей главе начинается формальное изучение языка программирования C# за счет представления набора отдельных тем, которые необходимо знать для освоения инфраструктуры .NET Framework. В первую очередь мы разберемся, каким образом строить *объект приложения*, и выясним структуру точки входа исполняемой программы — метода `Main()`. Затем мы исследуем фундаментальные типы данных C# (и их эквиваленты в пространстве имен `System`), в том числе классы `System.String` и `System.Text.StringBuilder`.

После ознакомления с деталями фундаментальных типов данных .NET мы рассмотрим несколько приемов преобразования типов данных, включая сужающие и расширяющие операции, а также использование ключевых слов `checked` и `unchecked`.

Кроме того, в главе будет описана роль ключевого слова `var` языка C#, которое позволяет *неявно* определять локальную переменную. Как будет показано далее в книге, неявная типизация чрезвычайно удобна (а порой и обязательна) при работе с набором технологий LINQ. Глава завершается кратким обзором ключевых слов и операций C#, которые дают возможность управлять последовательностью выполняемых в приложении действий с применением разнообразных конструкций циклов и принятия решений.

Структура простой программы C#

Язык C# требует, чтобы вся логика программы содержалась внутри определения типа (вспомните из главы 1, что *тип* — это общий термин, относящийся к любому члену из множества {класс, интерфейс, структура, перечисление, делегат}). В отличие от многих других языков создавать глобальные функции или глобальные элементы данных в C# невозможно. Взамен все данные-члены и все методы должны находиться внутри определения типа. Первым делом создадим новый проект консольного приложения по имени `SimpleCSharpApp`. Код в первоначальном файле `Program.cs` не особо примечателен:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Теперь модифицируем метод `Main()` класса `Program` следующим образом:

```
class Program
{
    static void Main(string[] args)
    {
        // Вывести пользователю простое сообщение.
        Console.WriteLine("***** My First C# App *****");
        Console.WriteLine("Hello World!");
        Console.WriteLine();

        // Ожидать нажатия клавиши <Enter>, прежде чем завершить работу.
        Console.ReadLine();
    }
}
```

На заметку! Язык программирования С# чувствителен к регистру. Следовательно, `Main` — не то же самое, что `main`, а `Readline` — не то же самое, что `ReadLine`. Запомните, что все ключевые слова С# вводятся в нижнем регистре (например, `public`, `lock`, `class`, `dynamic`), в то время как названия пространств имен, типов и членов (по соглашению) начинаются с заглавной буквы и имеют заглавные буквы в любых содержащихся внутри словах (скажем, `Console.WriteLine`, `System.Windows.MessageBox` и `System.Data.SqlClient`). Как правило, каждый раз, когда вы получаете от компилятора сообщение об ошибке, касающееся неопределенных символов, то в первую очередь должны проверить правильность написания и регистр.

В предыдущем коде имеется определение типа класса, который поддерживает единственный метод по имени `Main()`. По умолчанию среда Visual Studio назначает классу, определяющему метод `Main()`, имя `Program`; однако при желании его можно изменить. Каждое исполняемое приложение С# (консольная программа, настольная программа Windows или Windows-служба) должно содержать класс, определяющий метод `Main()`, который используется для обозначения точки входа в приложение.

Выражаясь формально, класс, в котором определен метод `Main()`, называется *объектом приложения*. Хотя в одном исполняемом приложении допускается иметь несколько объектов приложений (что может быть удобно при модульном тестировании), вы должны проинформировать компилятор о том, какой из методов `Main()` должен применяться в качестве точки входа. Это делается с помощью опции `/main` компилятора командной строки или посредством раскрывающегося списка `Startup Object` (Объект запуска) на вкладке `Application` (Приложение) окна свойств проекта Visual Studio (см. главу 2).

Обратите внимание, что сигнатура метода `Main()` снабжена ключевым словом `static`, которое подробно объясняется в главе 5. Пока достаточно знать, что статические члены имеют область видимости уровня класса (а не уровня объекта) и потому могут вызываться без предварительного создания нового экземпляра класса.

Помимо наличия ключевого слова `static` метод `Main()` принимает единственный параметр, который представляет собой массив строк (`string[] args`). Несмотря на то что в текущий момент данный массив никак не обрабатывается, параметр `args` может

содержать любое количество входных аргументов командной строки (доступ к ним будет вскоре описан). Наконец, метод `Main()` в примере был определен с возвращаемым значением `void`, т.е. перед выходом из области видимости метода мы не устанавливаем явным образом возвращаемое значение с использованием ключевого слова `return`.

Внутри метода `Main()` содержится логика класса `Program`. Здесь мы работаем с классом `Console`, который определен в пространстве имен `System`. В состав его членов входит статический метод `WriteLine()`, который отправляет текстовую строку и символ возврата каретки в стандартный вывод. Кроме того, мы производим вызов метода `Console.ReadLine()`, чтобы окно командной строки, открываемое IDE-средой `Visual Studio`, оставалось видимым на протяжении сеанса отладки до тех пор, пока не будет нажата клавиша `<Enter>`. (Если вы не добавите такую строку кода, то приложение завершится прямо во время сеанса отладки, и вы не сможете просмотреть результирующий вывод!) Класс `System.Console` более подробно рассматривается далее в главе.

Вариации метода `Main()`

По умолчанию `Visual Studio` будет генерировать метод `Main()` с возвращаемым значением `void` и одним входным параметром в виде массива строк. Тем не менее, это не единственно возможная форма метода `Main()`. Точку входа в приложение разрешено создавать с использованием любой из приведенных ниже сигнатур (предполагая, что они содержатся внутри определения класса или структуры C#):

```
// Возвращаемый тип int, массив строк в качестве параметра.
static int Main(string[] args)
{
    // Перед выходом должен возвращать значение!
    return 0;
}

// Нет возвращаемого типа, нет параметров.
static void Main()
{
}

// Возвращаемый тип int, нет параметров.
static int Main()
{
    // Перед выходом должен возвращать значение!
    return 0;
}
```

На заметку! Метод `Main()` может быть также определен как открытый в противоположность закрытому, что подразумевается, если не указан конкретный модификатор доступа. Среда `Visual Studio` определяет метод `Main()` как неявно закрытый.

Очевидно, что выбор способа создания метода `Main()` зависит от ответов на два вопроса. Первый вопрос: нужно ли возвращать значение системе, когда метод `Main()` заканчивается и работа программы завершается? Если да, тогда необходимо возвращать тип данных `int`, а не `void`. Второй вопрос: требуется ли обрабатывать любые предоставляемые пользователем параметры командной строки? Если да, то они будут сохранены в массиве строк. Ниже мы обсудим все варианты более подробно.

Асинхронные методы Main() (нововведение)

С выходом версии C# 7.1 метод Main() может быть асинхронным. Асинхронное программирование рассматривается в главе 19, а пока следует знать, что появились четыре дополнительных сигнатуры:

```
static Task Main()
static Task<int> Main()
static Task Main(string[])
static Task<int> Main(string[])
```

Они будут раскрыты в главе 19.

Указание кода ошибки приложения

Хотя в подавляющем большинстве случаев методы Main() будут иметь void в качестве возвращаемого значения, возможность возвращения int из Main() сохраняет согласованность C# с другими языками, основанными на C. По соглашению возврат значения 0 указывает на то, что программа завершилась успешно, тогда как любое другое значение (вроде -1) представляет условие ошибки (имейте в виду, что значение 0 автоматически возвращается даже в случае, если метод Main() прототипирован для возвращения void).

В ОС Windows возвращаемое приложением значение сохраняется в переменной среды по имени %ERRORLEVEL%. Если создается приложение, которое программно запускает другой исполняемый файл (тема, рассматриваемая в главе 18), тогда получить значение %ERRORLEVEL% можно с применением статического свойства System.Diagnostics.Process.ExitCode.

Поскольку возвращаемое значение передается системе в момент завершения работы приложения, вполне очевидно, что получить и отобразить финальный код ошибки во время выполнения приложения невозможно. Однако мы покажем, как просмотреть код ошибки по завершении программы, изменив метод Main() следующим образом:

```
// Обратите внимание, что теперь возвращается int, а не void.
static int Main(string[] args)
{
    // Вывести сообщение и ожидать нажатия клавиши <Enter>.
    Console.WriteLine("***** My First C# App *****");
    Console.WriteLine("Hello World!");
    Console.WriteLine();
    Console.ReadLine();

    // Возвратить произвольный код ошибки.
    return -1;
}
```

Теперь давайте захватим возвращаемое значение метода Main() с помощью пакетного файла. Используя проводник Windows, перейдите в папку, где находится файл решения (например, C:\SimpleCSharpApp\). Добавьте в нее новый текстовый файл (по имени SimpleCSharpApp.bat), содержащий приведенные далее инструкции (если раньше вам не приходилось создавать файлы .bat, то можете не беспокоиться о внутренних нюансах — это всего лишь тест):

```
@echo off

rem Пакетный файл для приложения SimpleCSharpApp.exe,
rem в котором захватывается возвращаемое им значение.

.\SimpleCSharpApp\bin\debug\SimpleCSharpApp
@if "%ERRORLEVEL%" == "0" goto success
```



```

:fail
    rem Приложение потерпело неудачу.
    echo This application has failed!
    echo return value = %ERRORLEVEL%
    goto end
:success
    rem Приложение успешно завершено.
    echo This application has succeeded!
    echo return value = %ERRORLEVEL%
    goto end
:end
rem Работа сделана.
echo All Done.

```

Откройте окно командной строки и перейдите в папку с исполняемым файлом и новым файлом *.bat. Запустите пакетный файл, набрав его имя и нажав <Enter>. Вы должны увидеть показанный ниже вывод, учитывая, что метод Main() возвращает -1. Если бы он возвращал 0, то вы увидели бы в окне консоли сообщение This application has succeeded!.

```

***** My First C# App *****
Hello World!

This application has failed!
return value = -1
All Done.

```

В подавляющем большинстве приложений C# (если только не во всех) в качестве возвращаемого значения метода Main() будет применяться void, что подразумевает неявное возвращение нулевого кода ошибки. Поэтому все методы Main() в книге (кроме текущего примера) на самом деле будут возвращать void (и оставшиеся проекты определенно не нуждаются в использовании пакетных файлов для перехвата кодов возврата).

Обработка аргументов командной строки

Теперь, когда вы лучше понимаете, что собой представляет возвращаемое значение метода Main(), давайте посмотрим на входной массив строковых данных. Предположим, что нам необходимо модифицировать приложение для обработки любых возможных параметров командной строки. Один из способов предусматривает применение цикла for языка C#. (Все итерационные конструкции языка C# более подробно рассматриваются в конце главы.)

```

static int Main(string[] args)
{
    ...
    // Обработать любые входные аргументы.
    for(int i = 0; i < args.Length; i++)
        Console.WriteLine("Arg: {0}", args[i]);

    Console.ReadLine();
    return -1;
}

```

Здесь с использованием свойства Length класса System.Array производится проверка, есть ли элементы в массиве строк. Как будет показано в главе 4, все массивы C# фактически являются псевдонимом класса System.Array и потому разделяют общий набор членов. По мере прохода в цикле по элементам массива их значения выводятся на консоль. Предоставить аргументы в командной строке в равной степени просто:

```
C:\SimpleCSharpApp\bin\Debug>SimpleCSharpApp.exe /arg1 -arg2
***** My First C# App *****
Hello World!
Arg: /arg1
Arg: -arg2
```

В качестве альтернативы стандартному циклу `for` для реализации прохода по входному массиву данных `string` можно также применять ключевое слово `foreach`. Вот пример использования `foreach` (особенности конструкций циклов обсуждаются далее в главе):

```
// Обратите внимание, что в случае применения foreach
// отпадает необходимость в проверке размера массива.
static int Main(string[] args)
{
    ...
    // Обработать любые входные аргументы, используя foreach.
    foreach(string arg in args)
        Console.WriteLine("Arg: {0}", arg);

    Console.ReadLine();
    return -1;
}
```

Наконец, доступ к аргументам командной строки можно также получать с помощью статического метода `GetCommandLineArgs()` типа `System.Environment`. Данный метод возвращает массив элементов `string`. Первый элемент содержит имя самого приложения, а остальные — индивидуальные аргументы командной строки. Обратите внимание, что при таком подходе больше не обязательно определять метод `Main()` как принимающий массив `string` во входном параметре, хотя никакого вреда от этого не будет.

```
static int Main(string[] args)
{
    ...
    // Получить аргументы с использованием System.Environment.
    string[] theArgs = Environment.GetCommandLineArgs();
    foreach(string arg in theArgs)
        Console.WriteLine("Arg: {0}", arg);

    Console.ReadLine();
    return -1;
}
```

Разумеется, именно на вас возлагается решение о том, на какие аргументы командной строки должна реагировать программа (если они вообще будут предусмотрены), и как они должны быть сформатированы (например, с префиксом `-` или `/`). В показанном выше коде мы просто передаем последовательность аргументов, которые выводятся прямо в окно командной строки. Однако предположим, что создается новое игровое приложение, запрограммированное на обработку параметра вида `-godmode`. Когда пользователь запускает приложение с таким флагом, в отношении него можно было бы предпринять соответствующие действия.

Указание аргументов командной строки в Visual Studio

В реальности конечный пользователь при запуске программы имеет возможность предоставлять аргументы командной строки. Тем не менее, указывать допустимые флаги командной строки также может требоваться во время разработки в целях тестирования программы. Чтобы сделать это в Visual Studio, дважды щелкните на значке

Properties (Свойства) в проводнике решений. Затем в открывшемся окне свойств перейдите на вкладку Debug (Отладка) и в текстовом поле Command line arguments (Аргументы командной строки) введите желаемые аргументы (рис. 3.1).

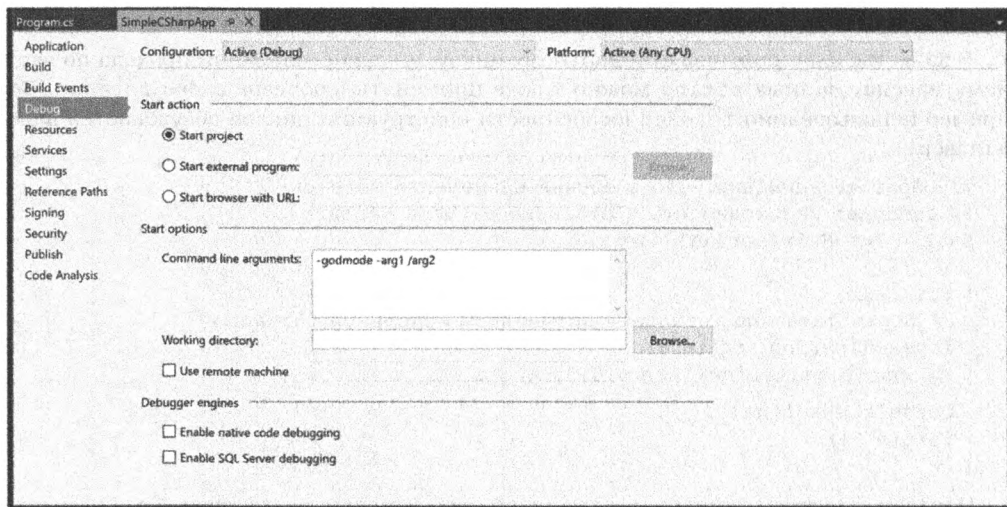


Рис. 3.1. Установка аргументов командной строки в Visual Studio

Указанные аргументы командной строки будут автоматически передаваться методу `Main()` во время отладки или запуска приложения внутри IDE-среды Visual Studio.

Интересное отступление от темы: некоторые дополнительные члены класса `System.Environment`

Помимо метода `GetCommandLineArgs()` класс `Environment` открывает доступ к ряду других чрезвычайно полезных методов. В частности, с помощью разнообразных статических членов этот класс позволяет получать детальные сведения, касающиеся операционной системы, под управлением которой в текущий момент функционирует приложение .NET. Чтобы проиллюстрировать полезность класса `System.Environment`, изменим код метода `Main()`, добавив вызов вспомогательного метода по имени `ShowEnvironmentDetails()`:

```
static int Main(string[] args)
{
    ...
    // Вспомогательный метод внутри класса Program.
    ShowEnvironmentDetails();
    Console.ReadLine();
    return -1;
}
```

Теперь реализуем метод `ShowEnvironmentDetails()` внутри класса `Program` для обращения в нем к разным членам типа `Environment`:

```

static void ShowEnvironmentDetails()
{
    // Вывести информацию о дисковых устройствах
    // данной машины и другие интересные детали.
    foreach (string drive in Environment.GetLogicalDrives())
        Console.WriteLine("Drive: {0}", drive);        // Логические устройства

    Console.WriteLine("OS: {0}", Environment.OSVersion); // Версия
                                                         // операционной системы

    Console.WriteLine("Number of processors: {0}",
        Environment.ProcessorCount);                    // Количество процессоров

    Console.WriteLine(".NET Version: {0}",
        Environment.Version);                            // Версия платформы .NET
}

```

Ниже показан возможный вывод, полученный в результате тестового запуска данного метода. Конечно, если на вкладке **Debug** в **Visual Studio** не были указаны аргументы командной строки, то они и не отобразятся в окне консоли.

```

***** My First C# App *****

Hello World!
Arg: -godmode
Arg: -arg1
Arg: /arg2

Drive: C:\
Drive: D:\
OS: Microsoft Windows NT 6.2.9200.0
Number of processors: 8
.NET Version: 4.0.30319.42000

```

В типе `Environment` определены и другие члены кроме тех, что задействованы в предыдущем примере. В табл. 3.1 описаны некоторые интересные дополнительные свойства; полные сведения о них можно найти в документации `.NET Framework 4.7 SDK`.

Таблица 3.1. Избранные свойства типа `System.Environment`

Свойство	Описание
<code>ExitCode</code>	Получает или устанавливает код возврата для приложения
<code>Is64BitOperatingSystem</code>	Возвращает булевское значение, которое представляет признак наличия на текущей машине 64-разрядной операционной системы
<code>MachineName</code>	Возвращает имя текущей машины
<code>NewLine</code>	Возвращает символ новой строки для текущей среды
<code>SystemDirectory</code>	Возвращает полный путь к каталогу системы
<code>UserName</code>	Возвращает имя пользователя, запустившего данное приложение
<code>Version</code>	Возвращает объект <code>Version</code> , который представляет версию платформы <code>.NET</code>

Класс System.Console

Почти во всех примерах приложений, создаваемых в начальных главах книги, будет интенсивно применяться класс `System.Console`. Справедливо заметить, что консольный пользовательский интерфейс может выглядеть не настолько привлекательно, как графический пользовательский интерфейс либо интерфейс веб-приложения. Однако ограничение первоначальных примеров консольными программами позволяет сосредоточиться на синтаксисе C# и ключевых аспектах платформы .NET, не отвлекаясь на сложности, которыми сопровождается построение настольных графических пользовательских интерфейсов или веб-сайтов.

Класс `Console` инкапсулирует средства манипулирования потоками ввода, вывода и ошибок для консольных приложений. В табл. 3.2 перечислены некоторые (но определенно не все) интересные его члены. Как видите, в классе `Console` имеется ряд членов, которые оживляют простые приложения командной строки, позволяя, например, изменять цвета фона и переднего плана и выдавать звуковые сигналы (еще и различной частоты).

Таблица 3.2. Избранные члены класса `System.Console`

Член	Описание
<code>Beep()</code>	Этот метод заставляет консоль выдать звуковой сигнал указанной частоты и длительности
<code>BackgroundColor</code> <code>ForegroundColor</code>	Эти свойства устанавливают цвета фона и переднего плана для текущего вывода. Им можно присваивать любой член перечисления <code>ConsoleColor</code>
<code>BufferHeight</code> <code>BufferWidth</code>	Эти свойства управляют высотой и шириной буферной области консоли
<code>Title</code>	Это свойство получает или устанавливает заголовок текущей консоли
<code>WindowHeight</code> <code>WindowWidth</code> <code>WindowTop</code> <code>WindowLeft</code>	Эти свойства управляют размерами консоли по отношению к установленному буферу
<code>Clear()</code>	Этот метод очищает установленный буфер и область отображения консоли

Базовый ввод-вывод с помощью класса Console

Дополнительно к членам, описанным в табл. 3.2, класс `Console` определяет набор методов для захвата ввода и вывода; все они являются статическими и потому вызываются с префиксом в виде имени класса (`Console`). Как вы уже видели, метод `WriteLine()` помещает в поток вывода строку текста (включая символ возврата каретки). Метод `Write()` помещает в поток вывода текст без символа возврата каретки. Метод `ReadLine()` позволяет получить информацию из потока ввода вплоть до нажатия клавиши <Enter>. Метод `Read()` используется для захвата одиночного символа из потока ввода.

Чтобы продемонстрировать реализацию базового ввода-вывода с применением класса `Console`, создадим новый проект консольного приложения по имени `BasicConsoleIO` и модифицируем метод `Main()` для вызова вспомогательного метода `GetUserData()`:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Basic Console I/O *****");
        GetUserData();
        Console.ReadLine();
    }
    private static void GetUserData()
    {
    }
}

```

На заметку! Среда Visual Studio поддерживает несколько “фрагментов кода”, которые после своей активизации вставляют код. Фрагмент кода `sw` очень полезен в начальных главах книги, т.к. он автоматически разворачивается в вызов метода `Console.WriteLine()`. Чтобы удостовериться в этом, введите `sw` где-нибудь внутри метода `Main()` и два раза нажмите клавишу `<Tab>` (к сожалению, фрагмент кода для метода `Console.ReadLine()` отсутствует). Чтобы просмотреть все фрагменты кода, щелкните правой кнопкой мыши в файле кода C# и в открывшемся контекстном меню выберите пункт `Insert Snippet` (Вставить фрагмент кода).

Теперь реализуем метод `GetUserData()` внутри класса `Program`, поместив в него логику, которая приглашает пользователя ввести некоторые сведения и затем дублирует их в стандартный поток вывода. Скажем, мы могли бы запросить у пользователя его имя и возраст (который для простоты будет трактоваться как текстовое значение, а не привычное числовое):

```

static void GetUserData()
{
    // Получить информацию об имени и возрасте.
    Console.Write("Please enter your name: "); // Предложить ввести имя
    string userName = Console.ReadLine();
    Console.Write("Please enter your age: "); // Предложить ввести возраст
    string userAge = Console.ReadLine();

    // Просто ради забавы изменить цвет переднего плана.
    ConsoleColor prevColor = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Yellow;

    // Вывести полученную информацию на консоль.
    Console.WriteLine("Hello {0}! You are {1} years old.",
        userName, userAge);

    // Восстановить предыдущий цвет переднего плана.
    Console.ForegroundColor = prevColor;
}

```

После запуска приложения входные данные вполне предсказуемо будут выводиться в окно консоли (с использованием указанного специального цвета).

Форматирование консольного вывода

В ходе изучения первых нескольких глав вы могли заметить, что внутри различных строковых литералов часто встречались такие конструкции, как `{0}` и `{1}`. Платформа .NET поддерживает стиль форматирования строк, который немного напоминает стиль, применяемый в операторе `printf()` языка C. Попросту говоря, когда вы определите строковый литерал, содержащий сегменты данных, значения которых остаются неизвестными до этапа выполнения, то имеете возможность указывать заполнитель, ис-

пользуя синтаксис с фигурными скобками. Во время выполнения все заполнители замещаются значениями, передаваемыми методом `Console.WriteLine()`.

Первый параметр метода `WriteLine()` представляет строковый литерал, который содержит заполнители, определяемые с помощью `{0}`, `{1}`, `{2}` и т.д. Запомните, что порядковые числа заполнителей в фигурных скобках всегда начинаются с 0. Остальные параметры `WriteLine()` — это просто значения, подлежащие вставке вместо соответствующих заполнителей.

На заметку! Если уникально нумерованных заполнителей больше, чем заполняющих аргументов, тогда во время выполнения будет сгенерировано исключение, связанное с форматом. Однако если количество заполняющих аргументов превышает число заполнителей, то лишние аргументы просто игнорируются.

Отдельный заполнитель допускается повторять внутри заданной строки. Например, если вы битломан и хотите построить строку "9, Number 9, Number 9", тогда могли бы написать такой код:

```
// Джон говорит...
Console.WriteLine("{0}, Number {0}, Number {0}", 9);
```

Также вы должны знать о возможности помещения каждого заполнителя в любую позицию внутри строкового литерала. К тому же вовсе не обязательно, чтобы заполнители следовали в возрастающем порядке своих номеров, например:

```
// Выводит: 20, 10, 30
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30);
```

Форматирование числовых данных

Если для числовых данных требуется более сложное форматирование, то каждый заполнитель может дополнительно содержать разнообразные символы форматирования, наиболее распространенные из которых описаны в табл. 3.3.

Таблица 3.3. Символы для форматирования числовых данных в .NET

Символ форматирования	Описание
C или c	Используется для форматирования денежных значений. По умолчанию значение предваряется символом локальной валюты (например, знаком доллара (\$)) для культуры US English)
D или d	Используется для форматирования десятичных чисел. В этом флаге можно также указывать минимальное количество цифр для представления значения
E или e	Используется для экспоненциального представления. Регистр этого флага указывает, в каком регистре должна представляться экспоненциальная константа — в верхнем (E) или в нижнем (e)
F или f	Используется для форматирования с фиксированной точкой. В этом флаге можно также указывать минимальное количество цифр для представления значения
G или g	Обозначает общий (<i>general</i>) формат. Этот флаг может использоваться для представления чисел в формате с фиксированной точкой или экспоненциальном формате
N или n	Используется для базового числового форматирования (с запятыми)
X или x	Используется для шестнадцатеричного форматирования. В случае символа X в верхнем регистре шестнадцатеричное представление будет содержать символы верхнего регистра

Символы форматирования добавляются к заполнителям в виде суффиксов после двоеточия (например, {0:C}, {1:d}, {2:X}). В целях иллюстрации изменим метод `Main()` для вызова нового вспомогательного метода по имени `FormatNumericalData()`, реализация которого в классе `Program` форматирует фиксированное числовое значение несколькими способами.

```
// Демонстрация применения некоторых дескрипторов формата.
static void FormatNumericalData()
{
    Console.WriteLine("The value 99999 in various formats:");
    Console.WriteLine("c format: {0:c}", 99999);
    Console.WriteLine("d9 format: {0:d9}", 99999);
    Console.WriteLine("f3 format: {0:f3}", 99999);
    Console.WriteLine("n format: {0:n}", 99999);

    // Обратите внимание, что использование для символа шестнадцатеричного формата
    // верхнего или нижнего регистра определяет регистр отображаемых символов.
    Console.WriteLine("E format: {0:E}", 99999);
    Console.WriteLine("e format: {0:e}", 99999);
    Console.WriteLine("X format: {0:X}", 99999);
    Console.WriteLine("x format: {0:x}", 99999);
}
```

Ниже показан вывод, получаемый в результате вызова метода `FormatNumericalData()`:

```
The value 99999 in various formats:
c format: $99,999.00
d9 format: 000099999
f3 format: 99999.000
n format: 99,999.00
E format: 9.9999000E+004
e format: 9.9999000e+004
X format: 1869F
x format: 1869f
```

В дальнейшем будут встречаться и другие примеры форматирования; если вас интересует дополнительные сведения о форматировании строк в .NET, тогда обратитесь в документацию .NET Framework 4.7 SDK.

Исходный код. Проект `BasicConsoleIO` доступен в подкаталоге `Chapter_3`.

Форматирование числовых данных за рамками консольных приложений

Напоследок следует отметить, что применение символов форматирования строк .NET не ограничено консольными приложениями. Тот же самый синтаксис форматирования может быть использован при вызове статического метода `string.Format()`. Прием удобен, когда необходимо формировать выходные текстовые данные во время выполнения в приложении любого типа (например, в настольном приложении с графическим пользовательским интерфейсом, веб-приложении ASP.NET и т.д.).

Метод `string.Format()` возвращает новый объект `string`, который форматируется согласно предоставляемым флагам. Затем текстовые данные могут применяться любым желаемым образом. Предположим, что требуется создать графическое настольное приложение WPF и сформатировать строку, подлежащую отображению в окне сообщения. В приведенном ниже коде показано, как это сделать, но имейте в виду, что

код не скомпилируется до тех пор, пока в проект не будет добавлена ссылка на сборку `PresentationFramework.dll` (добавление ссылок на библиотеки в Visual Studio рассматривалось в главе 2).

```
static void DisplayMessage()  
{  
    // Использование string.Format() для форматирования строкового литерала.  
    string userMessage = string.Format("100000 in hex is {0:x}", 100000);  
  
    // Для успешной компиляции этой строки кода требуется  
    // ссылка на библиотеку PresentationFramework.dll!  
    System.Windows.MessageBox.Show(userMessage);  
}
```

На заметку! Для представления заполнителей в фигурных скобках в .NET 4.6 и C# 6 появился альтернативный синтаксис, который называется *интерполяцией строк*. Мы исследуем такой подход позже в главе.

Системные типы данных и соответствующие ключевые слова C#

Подобно любому языку программирования в C# для фундаментальных типов данных определены ключевые слова, которые используются при представлении локальных переменных, переменных-членов данных в классах, возвращаемых значений и параметров методов. Тем не менее, в отличие от других языков программирования такие ключевые слова в C# являются чем-то большим, нежели просто лексемами, распознаваемыми компилятором. В действительности они представляют собой сокращенные обозначения полноценных типов из пространства имен `System`. В табл. 3.4 перечислены системные типы данных вместе с их диапазонами значений, соответствующими ключевыми словами C# и сведениями о совместимости с общезыковой спецификацией (CLS).

Таблица 3.4. Внутренние типы данных C#

Сокращенное обозначение в C#	Совместимость с CLS	Системный тип	Диапазон	Описание
<code>bool</code>	Да	<code>System.Boolean</code>	<code>true</code> или <code>false</code>	Признак истинности или ложности
<code>sbyte</code>	Нет	<code>System.SByte</code>	от -128 до 127	8-битное число со знаком
<code>byte</code>	Да	<code>System.Byte</code>	от 0 до 255	8-битное число без знака
<code>short</code>	Да	<code>System.Int16</code>	от -32 768 до 32 767	16-битное число со знаком
<code>ushort</code>	Нет	<code>System.UInt16</code>	от 0 до 65 535	16-битное число без знака
<code>int</code>	Да	<code>System.Int32</code>	от -2 147 483 648 до 2 147 483 647	32-битное число со знаком
<code>uint</code>	Нет	<code>System.UInt32</code>	от 0 до 4 294 967 295	32-битное число без знака

Окончание табл. 3.4

Сокращенное обозначение в C#	Совместимость с CLS	Системный тип	Диапазон	Описание
long	Да	System.Int64	от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	64-битное число со знаком
ulong	Нет	System.UInt64	от 0 до 18 446 744 073 709 551 615	64-битное число без знака
char	Да	System.Char	от U+0000 до U+ffff	Одиночный 16-битный символ Unicode
float	Да	System.Single	от -3.4×10^{38} до $+3.4 \times 10^{38}$	32-битное число с плавающей точкой
double	Да	System.Double	от $\pm 5.0 \times 10^{-324}$ до $\pm 1.7 \times 10^{308}$	64-битное число с плавающей точкой
decimal	Да	System.Decimal	(от -7.9×10^{28} до 7.9×10^{28}) / (10 ^{от 0 до 28})	128-битное число со знаком
string	Да	System.String	Ограничен объемом системной памяти	Представляет набор символов Unicode
object	Да	System.Object	В переменной object может храниться любой тип данных	Базовый класс для всех типов в мире .NET

На заметку! В главе 1 говорилось о том, что совместимый с CLS код .NET может быть задействован в любом управляемом языке программирования. Если в программах открыт доступ к данным, не совместимым с CLS, тогда другие языки могут быть не в состоянии их использовать. По умолчанию число с плавающей точкой трактуется как double. Чтобы объявить значение типа float, применяйте суффикс f или F к неформатированному числовому значению (5.3F), а чтобы объявить значение типа decimal, применяйте суффикс m или M к числу с плавающей точкой (300.5M). И, наконец, неформатированные целые числа по умолчанию относятся к типу int. Чтобы получить значение типа long, понадобится снабдить его суффиксом l или L (4L).

Объявление и инициализация переменных

Для объявления локальной переменной (например, переменной внутри области видимости члена) необходимо указать тип данных, за которым следует имя переменной. Давайте создадим новый проект консольного приложения по имени BasicDataTypes и модифицируем класс Program так, чтобы в его методе Main() вызывался следующий вспомогательный метод:

```
static void LocalVarDeclarations()
{
    Console.WriteLine("=> Data Declarations:");
    // Локальные переменные объявляются так:
    // типДанных имяПеременной;
    int myInt;
    string myString;
    Console.WriteLine();
}
```

Имейте в виду, что использование локальной переменной до присваивания ей начального значения приведет к *ошибке на этапе компиляции*. Таким образом, рекомендуется присваивать начальные значения локальным переменным непосредственно при их объявлении, что можно делать в одной строке или разносить объявление и присваивание на два отдельных оператора кода.

```
static void LocalVarDeclarations()
{
    Console.WriteLine("> Data Declarations:");
    // Локальные переменные объявляются и инициализируются так:
    // типДанных имяПеременной = начальноеЗначение;
    int myInt = 0;

    // Объявлять и присваивать можно также в двух отдельных строках.
    string myString;
    myString = "This is my character data";

    Console.WriteLine();
}
```

Кроме того, разрешено объявлять несколько переменных того же самого типа в одной строке кода, как в случае следующих трех переменных `bool`:

```
static void LocalVarDeclarations()
{
    Console.WriteLine("> Data Declarations:");
    int myInt = 0;
    string myString;
    myString = "This is my character data";

    // Объявить три переменных типа bool в одной строке.
    bool b1 = true, b2 = false, b3 = b1;
    Console.WriteLine();
}
```

Поскольку ключевое слово `bool` в C# — просто сокращенное обозначение структуры `System.Boolean`, то любой тип данных можно указывать с применением его полного имени (естественно, то же самое касается всех остальных ключевых слов C#, представляющих типы данных). Ниже приведена окончательная реализация метода `LocalVarDeclarations()`, в которой демонстрируются разнообразные способы объявления локальных переменных:

```
static void LocalVarDeclarations()
{
    Console.WriteLine("> Data Declarations:");
    // Локальные переменные объявляются и инициализируются так:
    // типДанных имяПеременной = начальноеЗначение;
    int myInt = 0;

    string myString;
    myString = "This is my character data";

    // Объявить три переменных типа bool в одной строке.
    bool b1 = true, b2 = false, b3 = b1;

    // Использовать тип данных System.Boolean для объявления булевской переменной.
    System.Boolean b4 = false;

    Console.WriteLine("Your data: {0}, {1}, {2}, {3}, {4}, {5}",
        myInt, myString, b1, b2, b3, b4);
    Console.WriteLine();
}
```

Литерал *default* (нововведение)

Литерал *default* является новым средством в версии C# 7.1, которое позволяет присваивать переменной стандартное значение ее типа данных. Литерал *default* работает для стандартных типов данных, а также для специальных классов (глава 5) и обобщенных типов (глава 9). Создадим новый метод по имени `DefaultDeclarations()`, поместив в него следующий код:

```
static void DefaultDeclarations()
{
    Console.WriteLine("=> Default Declarations:");
    int myInt = default;
}
```

Проект не скомпилируется, если только вы вручную не сконфигурировали его для использования версии C# 7.1. В результате наведения курсора на ключевое слово *default* среда Visual Studio отобразит значок с лампочкой, который позволит модернизировать проект до C# 7.1 (рис. 3.2).

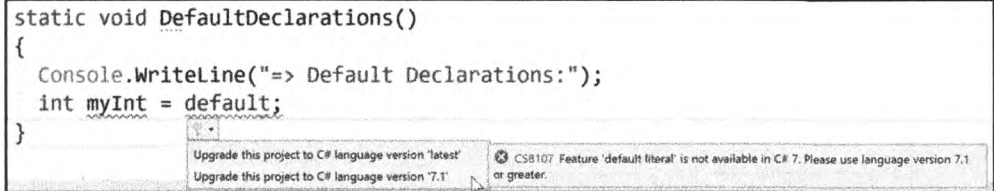


Рис. 3.2. Модернизация проекта до C# 7.1

Внутренние типы данных и операция *new*

Все внутренние типы данных поддерживают так называемый *стандартный конструктор* (см. главу 5). Это средство позволяет создавать переменную, используя ключевое слово *new*, что автоматически устанавливает переменную в ее стандартное значение:

- переменные типа `bool` устанавливаются в `false`;
- переменные числовых типов устанавливаются в 0 (или в 0.0 для типов с плавающей точкой);
- переменные типа `char` устанавливаются в пустой символ;
- переменные типа `BigInteger` устанавливаются в 0;
- переменные типа `DateTime` устанавливаются в 1/1/0001 12:00:00 AM;
- объектные ссылки (включая переменные типа `string`) устанавливаются в `null`.

На заметку! Тип данных `BigInteger`, упомянутый в приведенном выше списке, будет описан чуть позже.

Применение ключевого слова *new* при создании переменных базовых типов дает более громоздкий, но синтаксически корректный код C#:

```
static void NewingDataTypes()
{
    Console.WriteLine("=> Using new to create variables:");
    bool b = new bool(); // Устанавливается в false.
}
```

```
int i = new int();           // Устанавливается в 0.
double d = new double();    // Устанавливается в 0.
DateTime dt = new DateTime(); // Устанавливается в 1/1/0001 12:00:00 AM
Console.WriteLine("{0}, {1}, {2}, {3}", b, i, d, dt);
Console.WriteLine();
}
```

Иерархия классов для типов данных

Интересно отметить, что даже элементарные типы данных в .NET организованы в *иерархию классов*. Если вы не знакомы с концепцией наследования, тогда найдите все необходимые сведения в главе 6. А до тех пор просто знайте, что типы, находящиеся в верхней части иерархии классов, предоставляют определенное стандартное поведение, которое передается производным типам. На рис. 3.3 показаны отношения между основными системными типами.

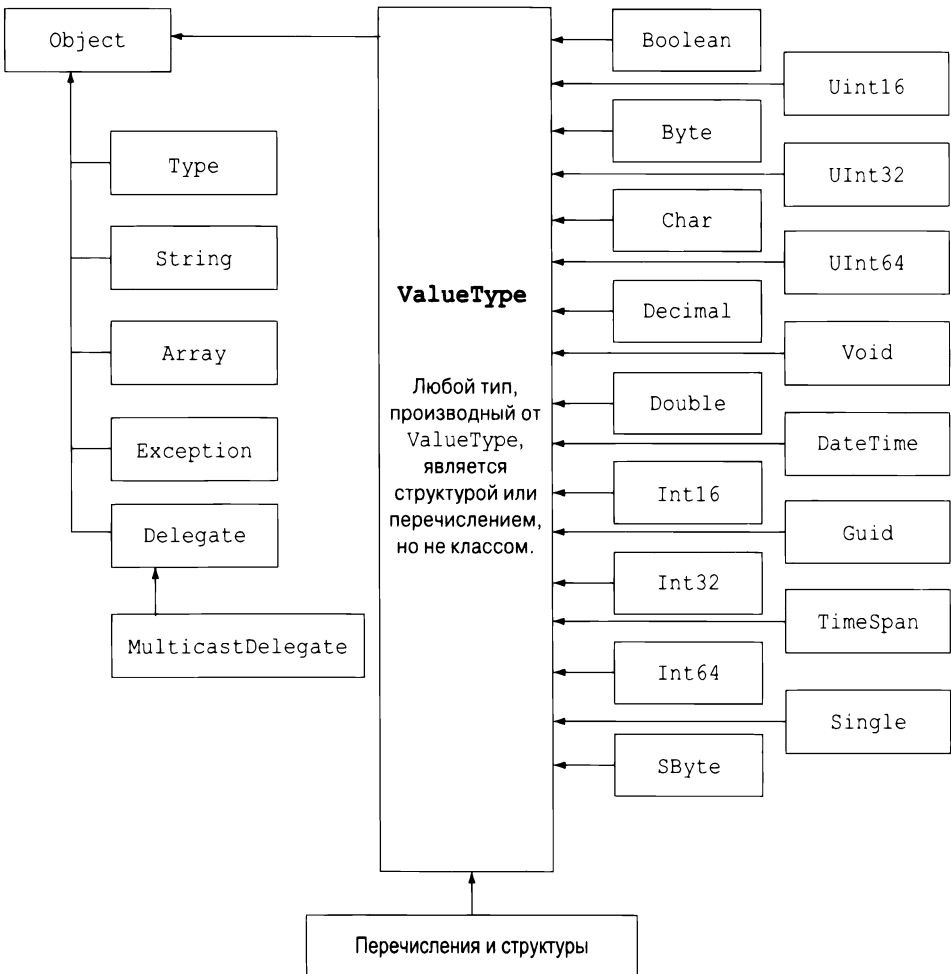


Рис. 3.3. Иерархия классов для системных типов

Обратите внимание, что каждый тип в конечном итоге оказывается производным от класса `System.Object`, в котором определен набор методов (например, `ToString()`, `Equals()`, `GetHashCode()`), общих для всех типов из библиотек базовых классов .NET (упомянутые методы подробно рассматриваются в главе 6).

Также важно отметить, что многие числовые типы данных являются производными от класса `System.ValueType`. Потомки `ValueType` автоматически размещаются в стеке и по этой причине имеют предсказуемое время жизни и довольно эффективны. С другой стороны, типы, в цепочке наследования которых класс `System.ValueType` отсутствует (такие как `System.Type`, `System.String`, `System.Array`, `System.Exception` и `System.Delegate`), размещаются не в стеке, а в куче с автоматической сборкой мусора. (Более подробно такое различие обсуждается в главе 4.)

Не вдаваясь глубоко в детали классов `System.Object` и `System.ValueType`, важно уяснить, что поскольку любое ключевое слово C# (скажем, `int`) представляет собой просто сокращенное обозначение соответствующего системного типа (в данном случае `System.Int32`), то приведенный ниже синтаксис совершенно законен. Дело в том, что тип `System.Int32` (`int` в C#) в конечном итоге является производным от класса `System.Object` и, следовательно, может обращаться к любому из его открытых членов, как продемонстрировано в еще одной вспомогательной функции:

```
static void ObjectFunctionality()
{
    Console.WriteLine("=> System.Object Functionality:");

    // Ключевое слово int языка C# - это в действительности сокращение для
    // типа System.Int32, который наследует от System.Object следующие члены:
    Console.WriteLine("12.GetHashCode() = {0}", 12.GetHashCode());
    Console.WriteLine("12.Equals(23) = {0}", 12.Equals(23));
    Console.WriteLine("12.ToString() = {0}", 12.ToString());
    Console.WriteLine("12.GetType() = {0}", 12.GetType());
    Console.WriteLine();
}
```

Вызов метода `ObjectFunctionality()` внутри `Main()` дает такой вывод:

```
=> System.Object Functionality:
12.GetHashCode() = 12
12.Equals(23) = False
12.ToString() = 12
12.GetType() = System.Int32
```

Члены числовых типов данных

Продолжая эксперименты со встроенными типами данных C#, следует отметить, что числовые типы .NET поддерживают свойства `MaxValue` и `MinValue`, предоставляющие информацию о диапазоне значений, которые способен хранить конкретный тип. В дополнение к свойствам `MinValue` и `MaxValue` каждый числовой тип может определять собственные полезные члены. Например, тип `System.Double` позволяет получать значения для бесконечно малой (эпсилон) и бесконечно большой величин (которые интересны тем, кто занимается решением математических задач). В целях иллюстрации рассмотрим следующую вспомогательную функцию:

```
static void DataTypeFunctionality()
{
    Console.WriteLine("=> Data type Functionality:");
    Console.WriteLine("Max of int: {0}", int.MaxValue);
    Console.WriteLine("Min of int: {0}", int.MinValue);
}
```

```

Console.WriteLine("Max of double: {0}", double.MaxValue);
Console.WriteLine("Min of double: {0}", double.MinValue);
Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
Console.WriteLine("double.PositiveInfinity: {0}",
    double.PositiveInfinity);
Console.WriteLine("double.NegativeInfinity: {0}",
    double.NegativeInfinity);
Console.WriteLine();
}

```

Члены System.Boolean

Рассмотрим тип данных `System.Boolean`. К допустимым значениям, которые могут присваиваться типу `bool` в C#, относятся только `true` и `false`. С учетом этого должно быть понятно, что `System.Boolean` не поддерживает свойства `MinValue` и `MaxValue`, но вместо них определяет свойства `TrueString` и `FalseString` (которые выдают, соответственно, строки `"True"` и `"False"`). Вот пример:

```

Console.WriteLine("bool.FalseString: {0}", bool.FalseString);
Console.WriteLine("bool.TrueString: {0}", bool.TrueString);

```

Члены System.Char

Текстовые данные в C# представляются посредством ключевых слов `string` и `char`, которые являются сокращенными обозначениями для типов `System.String` и `System.Char` (оба основаны на Unicode). Как вам уже может быть известно, `string` представляет непрерывное множество символов (например, `"Hello"`), а `char` — одиночную ячейку в `string` (например, `'H'`).

Помимо возможности хранения одиночного элемента символьных данных тип `System.Char` предлагает немало другой функциональности. Используя статические методы `System.Char`, можно выяснять, является данный символ цифрой, буквой, знаком пунктуации или чем-то еще. Рассмотрим следующий метод:

```

static void CharFunctionality()
{
    Console.WriteLine("=> char type Functionality:");
    char myChar = 'a';
    Console.WriteLine("char.IsDigit('a'): {0}", char.IsDigit(myChar));
    Console.WriteLine("char.IsLetter('a'): {0}", char.IsLetter(myChar));
    Console.WriteLine("char.IsWhiteSpace('Hello There', 5): {0}",
        char.IsWhiteSpace("Hello There", 5));
    Console.WriteLine("char.IsWhiteSpace('Hello There', 6): {0}",
        char.IsWhiteSpace("Hello There", 6));
    Console.WriteLine("char.IsPunctuation('?'): {0}",
        char.IsPunctuation('?'));
    Console.WriteLine();
}

```

В методе `CharFunctionality()` было показано, что для многих членов `System.Char` предусмотрены два соглашения о вызове: одиночный символ или строка с числовым индексом, указывающим позицию проверяемого символа.

Разбор значений из строковых данных

Типы данных .NET предоставляют возможность генерировать переменную лежащего в основе типа, имея текстовый эквивалент (например, путем выполнения разбора). Такой прием может оказаться исключительно удобным, когда вы хотите преобразовывать в чис-

ловые значения некоторые вводимые пользователем данные (вроде элемента, выбранного в раскрывающемся списке внутри графического пользовательского интерфейса). Ниже приведен пример метода `ParseFromStrings()`, содержащий логику разбора:

```
static void ParseFromStrings()
{
    Console.WriteLine("=> Data type parsing:");
    bool b = bool.Parse("True");
    Console.WriteLine("Value of b: {0}", b); // Вывод значения b
    double d = double.Parse("99.884");
    Console.WriteLine("Value of d: {0}", d); // Вывод значения d
    int i = int.Parse("8");
    Console.WriteLine("Value of i: {0}", i); // Вывод значения i
    char c = Char.Parse("w");
    Console.WriteLine("Value of c: {0}", c); // Вывод значения c
    Console.WriteLine();
}
```

Использование метода `TryParse()` для разбора значений из строковых данных

Проблема с предыдущим кодом связана с тем, что если строка не может быть аккуратно преобразована в корректный тип данных, то сгенерируется исключение. Например, следующий код потерпит неудачу во время выполнения:

```
bool b = bool.Parse("Hello");
```

Решение предусматривает помещение каждого вызова `Parse()` в блок `try-catch` (обработка исключений подробно раскрывается в главе 7), что добавит много кода, или применение метода `TryParse()`. Метод `TryParse()` принимает параметр `out` (модификатор `out` рассматривается в главе 4) и возвращает значение `bool`, которое указывает, успешно ли прошел разбор. Создадим новый метод по имени `ParseFromStringWithTryParse()` и поместим в него такой код:

```
static void ParseFromStringWithTryParse()
{
    Console.WriteLine("=> Data type parsing with TryParse:");
    if (bool.TryParse("True", out bool b))
    {
        Console.WriteLine("Value of b: {0}", b); // Вывод значения b
    }
    string value = "Hello";
    if (double.TryParse(value, out double d))
    {
        Console.WriteLine("Value of d: {0}", d); // Вывод значения d
    }
    else
    {
        // Преобразование потерпело неудачу
        Console.WriteLine("Failed to convert the input ({0}) to a double", value);
    }
    Console.WriteLine();
}
```

Если вы только начали осваивать программирование и не знаете, как работают операторы `if-else`, то они подробно рассматриваются позже в главе. В приведенном выше примере важно отметить, что когда строка может быть преобразована в запрошенный тип данных, метод `TryParse()` возвращает `true` и присваивает разобранное значение

переменной, переданной методу. В случае невозможности разбора значения переменной присваивается стандартное значение, а метод `TryParse()` возвращает `false`.

Типы `System.DateTime` и `System.TimeSpan`

В пространстве имен `System` определено несколько полезных типов данных, для которых отсутствуют ключевые слова языка C#, в том числе структуры `DateTime` и `TimeSpan`. (При желании можете самостоятельно ознакомиться с типами `System.Guid` и `System.Void`, показанными на рис. 3.2, но имейте в виду, что в большинстве приложений эти два типа данных редко оказываются практичными.)

Тип `DateTime` содержит данные, представляющие специфичное значение даты (месяц, день, год) и времени, которые могут форматироваться разнообразными способами с применением членов этого типа. Структура `TimeSpan` позволяет легко определять и трансформировать единицы времени, используя различные ее члены.

```
static void UseDatesAndTimes()
{
    Console.WriteLine("> Dates and Times:");
    // Этот конструктор принимает год, месяц и день.
    DateTime dt = new DateTime(2015, 10, 17);
    // Какой это день месяца?
    Console.WriteLine("The day of {0} is {1}", dt.Date, dt.DayOfWeek);
    // Сейчас месяц декабрь.
    dt = dt.AddMonths(2);
    Console.WriteLine("Daylight savings: {0}", dt.IsDaylightSavingTime());
    // Этот конструктор принимает часы, минуты и секунды.
    TimeSpan ts = new TimeSpan(4, 30, 0);
    Console.WriteLine(ts);
    // Вычесть 15 минут из текущего значения TimeSpan и вывести результат.
    Console.WriteLine(ts.Subtract(new TimeSpan(0, 15, 0)));
}
```

Сборка `System.Numerics.dll`

В пространстве имен `System.Numerics` определена структура по имени `BigInteger`. Тип данных `BigInteger` может применяться для представления огромных числовых значений, которые не ограничены фиксированным верхним или нижним пределом.

На заметку! В пространстве имен `System.Numerics` также определена вторая структура по имени `Complex`, которая позволяет моделировать математически сложные числовые данные (например, мнимые единицы, вещественные данные, гиперболические тангенсы). Дополнительные сведения о структуре `Complex` можно найти в документации .NET Framework 4.7 SDK.

Несмотря на то что во многих приложениях .NET потребность в структуре `BigInteger` может никогда не возникать, если все-таки необходимо определить большое числовое значение, то в первую очередь понадобится добавить в проект ссылку на сборку `System.Numerics.dll`. Выполните следующие действия.

1. Выберите в Visual Studio пункт меню `Project`⇒`Add Reference` (Проект⇒Добавить ссылку).
2. Найдите и выберите сборку `System.Numerics.dll` в списке имеющихся библиотек, который отображается на вкладке `Framework` (Инфраструктура) слева.
3. Щелкните на кнопке `OK`.

Затем добавьте показанную ниже директиву `using` в файл, в котором будет использоваться тип данных `BigInteger`:

```
// Здесь определен тип BigInteger:
using System.Numerics;
```

Теперь с применением операции `new` можно создать переменную `BigInteger`. Внутри конструктора можно указать числовое значение, включая данные с плавающей точкой. Однако вспомните, что когда определяется целочисленный литерал (вроде `500`), исполняющая среда по умолчанию трактует его как относящийся к типу `int`. Аналогично литерал с плавающей точкой (такой как `55.333`) по умолчанию будет отнесен к типу `double`. Как же тогда установить для `BigInteger` большое значение, не переполнив стандартные типы данных, которые задействуются для неформатированных числовых значений?

Простейший подход предусматривает определение большого числового значения в виде текстового литерала, который затем может быть преобразован в переменную `BigInteger` посредством статического метода `Parse()`. При желании можно также передавать байтовый массив непосредственно конструктору класса `BigInteger`.

На заметку! После того как переменной `BigInteger` присвоено значение, модифицировать ее больше нельзя, т.к. это неизменяемые данные. Тем не менее, в классе `BigInteger` определено несколько членов, которые возвращают новые объекты `BigInteger` на основе модификаций данных (такие как статический метод `Multiply()`, используемый в следующем примере кода).

В любом случае после определения переменной `BigInteger` вы обнаружите, что в этом классе определены члены, похожие на члены в других внутренних типах данных C# (например, `float` либо `int`). Вдобавок в классе `BigInteger` определен ряд статических членов, которые позволяют применять к переменным `BigInteger` базовые математические операции (наподобие сложения и умножения). Взгляните на пример работы с классом `BigInteger`:

[illegible]

Важно отметить, что тип данных `BigInteger` реагирует на внутренние математические операции `+`, `-` и `*`. Следовательно, вместо вызова метода `BigInteger.Multiply()` для перемножения двух больших чисел можно использовать такой код:

```
BigInteger reallyBig2 = biggy * reallyBig;
```

К настоящему моменту вы должны понимать, что ключевые слова C#, представляющие базовые типы данных, имеют соответствующие типы в библиотеках базовых классов .NET, каждый из которых предлагает фиксированную функциональность. Хотя абсолютно все члены этих типов данных в книге подробно не рассматриваются, имеет смысл изучить их самостоятельно. Подробные описания разнообразных типов данных .NET можно найти в документации .NET Framework 4.7 SDK — скорее всего, вы будете удивлены объемом их встроенной функциональности.

Исходный код. Проект BasicDataTypes доступен в подкаталоге Chapter_3.

Разделители групп цифр (нововведение)

Временами при присваивании числовой переменной крупных чисел цифр оказывается больше, чем способен отслеживать глаз. В версии C# 7 введен разделитель групп цифр в виде символа подчеркивания (`_`) для типов данных `integer`, `long`, `decimal` или `double`. Ниже представлен пример применения нового разделителя групп цифр:

```
static void DigitSeparators()
{
    Console.WriteLine("=> Use Digit Separators:");
    Console.WriteLine("Integer:");
    Console.WriteLine(123_456);
    Console.WriteLine("Long:");
    Console.WriteLine(123_456_789L);
    Console.WriteLine("Float:");
    Console.WriteLine(123_456.1234F);
    Console.WriteLine("Double:");
    Console.WriteLine(123_456.12);
    Console.WriteLine("Decimal:");
    Console.WriteLine(123_456.12M);
}
```

Двоичные литералы (нововведение)

В версии C# 7 появился новый литерал для двоичных значений, представляющих, например, битовые маски. Теперь двоичные числа можно записывать ожидаемым образом:

```
0b0001_0000
```

Новый разделитель групп цифр работает и с двоичными литералами. Вот метод, в котором иллюстрируется использование новых литералов с разделителем групп цифр:

```
private static void BinaryLiterals()
{
    Console.WriteLine("=> Use Binary Literals:");
    Console.WriteLine("Sixteen: {0}", 0b0001_0000);
    Console.WriteLine("Thirty Two: {0}", 0b0010_0000);
    Console.WriteLine("Sixty Four: {0}", 0b0100_0000);
}
```

Работа со строковыми данными

Класс `System.String` предоставляет набор членов, вполне ожидаемый от служебного класса такого рода, например, члены для возвращения длины символьных данных, поиска подстрок в текущей строке и преобразования символов между верхним и нижним регистрами. В табл. 3.5 перечислены некоторые интересные члены этого класса.

Таблица 3.5. Избранные члены класса `System.String`

Член <code>System.String</code>	Описание
<code>Length</code>	Свойство, которое возвращает длину текущей строки
<code>Compare()</code>	Статический метод, который позволяет сравнить две строки
<code>Contains()</code>	Метод, который позволяет определить, содержится ли в строке указанная подстрока
<code>Equals()</code>	Метод, который позволяет проверить, содержатся ли в двух строковых объектах идентичные символьные данные
<code>Format()</code>	Статический метод, позволяющий сформатировать строку с использованием других элементарных типов данных (например, числовых данных или других строк) и системы обозначений <code>{0}</code> , которая рассматривалась ранее в главе
<code>Insert()</code>	Метод, который позволяет вставить строку внутрь заданной строки
<code>PadLeft()</code> <code>PadRight()</code>	Методы, которые позволяют дополнить строку определенными символами
<code>Remove()</code> <code>Replace()</code>	Методы, которые позволяют получить копию строки с произведенными изменениями (удалением или заменой символов)
<code>Split()</code>	Метод, возвращающий массив <code>string</code> , который содержит подстроки в этом экземпляре, разделенные элементами из указанного массива <code>char</code> или <code>string</code>
<code>Trim()</code>	Метод, который удаляет все вхождения набора указанных символов с начала и конца текущей строки
<code>ToUpper()</code> <code>ToLower()</code>	Методы, которые создают копию текущей строки в верхнем или нижнем регистре

Базовые манипуляции строками

Работа с членами `System.String` выглядит предсказуемым образом. Нужно просто объявить переменную `string` и задействовать предлагаемую типом функциональность через операцию точки. Не следует забывать, что несколько членов `System.String` являются статическими и потому должны вызываться на уровне класса (а не объекта). В целях иллюстрации создадим новый проект консольного приложения по имени `FunWithStrings` и добавим в него показанный далее метод, который будет вызываться внутри `Main()`:

```
static void BasicStringFunctionality()
{
    Console.WriteLine("=> Basic String functionality:");
    string firstName = "Freddy";
    Console.WriteLine("Value of firstName: {0}", firstName);
    // Значение firstName.
    Console.WriteLine("firstName has {0} characters.", firstName.Length);
    // Длина firstName.
    Console.WriteLine("firstName in uppercase: {0}", firstName.ToUpper());
    // firstName в верхнем регистре.
    Console.WriteLine("firstName in lowercase: {0}", firstName.ToLower());
    // firstName в нижнем регистре.
}
```

```

Console.WriteLine("firstName contains the letter y?:{0}", firstName.Contains("y"));
// Содержит ли firstName букву y?
Console.WriteLine("firstName after replace: {0}", firstName.Replace("dy", ""));
// firstName после замены.
Console.WriteLine();
}

```

Здесь объяснять особо нечего: метод просто вызывает разнообразные члены, такие как `ToUpper()` и `Contains()`, на локальной переменной `string`, чтобы получить разные форматы и трансформации. Ниже приведен вывод:

```

***** Fun with Strings *****
=> Basic String functionality:
Value of firstName: Freddy
firstName has 6 characters.
firstName in uppercase: FREDDY
firstName in lowercase: freddy
firstName contains the letter y?: True
firstName after replace: Fred

```

Несмотря на то что вывод не выглядит особо неожиданным, вывод, полученный в результате вызова метода `Replace()`, может вводить в заблуждение. В действительности переменная `firstName` вообще не изменяется; взамен получается новая переменная `string` в модифицированном формате. Чуть позже мы еще вернемся к обсуждению неизменяемой природы строк.

Конкатенация строк

Переменные `string` могут соединяться вместе для построения строк большего размера с помощью операции `+` языка C#. Как вам должно быть известно, такой прием формально называется *конкатенацией строк*. Рассмотрим следующую вспомогательную функцию:

```

static void StringConcatenation()
{
    Console.WriteLine("=> String concatenation:");
    string s1 = "Programming the ";
    string s2 = "PsychoDrill (PTP)";
    string s3 = s1 + s2;
    Console.WriteLine(s3);
    Console.WriteLine();
}

```

Интересно отметить, что при обработке символа `+` компилятор C# выпускает вызов статического метода `String.Concat()`. В результате конкатенацию строк можно также выполнять, вызывая метод `String.Concat()` напрямую (хотя фактически это не дает никаких преимуществ, а лишь увеличивает объем набираемого кода):

```

static void StringConcatenation()
{
    Console.WriteLine("=> String concatenation:");
    string s1 = "Programming the ";
    string s2 = "PsychoDrill (PTP)";
    string s3 = String.Concat(s1, s2);
    Console.WriteLine(s3);
    Console.WriteLine();
}

```

Управляющие последовательности

Подобно другим языкам, основанным на C, строковые литералы C# могут содержать разнообразные *управляющие последовательности*, которые позволяют уточнять то, как символьные данные должны быть представлены в потоке вывода. Каждая управляющая последовательность начинается с символа обратной косой черты, за которым следует специфический знак. В табл. 3.6 перечислены наиболее распространенные управляющие последовательности.

Таблица 3.6. Управляющие последовательности в строковых литералах

Управляющая последовательность	Описание
\'	Вставляет в строковый литерал символ одинарной кавычки
\"	Вставляет в строковый литерал символ двойной кавычки
\\	Вставляет в строковый литерал символ обратной косой черты. Это особенно полезно при определении путей к файлам или сетевым ресурсам
\a	Заставляет систему выдать звуковой сигнал, который в консольных приложениях может служить аудио-подсказкой пользователю
\n	Вставляет символ новой строки (на платформах Windows)
\r	Вставляет символ возврата каретки
\t	Вставляет в строковый литерал символ горизонтальной табуляции

Например, чтобы вывести строку, которая содержит символ табуляции после каждого слова, можно задействовать управляющую последовательность `\t`. Или предположим, что нужно создать один строковый литерал с символами кавычек внутри, второй — с определением пути к каталогу и третий — со вставкой трех пустых строк после вывода символьных данных. Для этого можно применять управляющие последовательности `\`, `\\` и `\n`. Кроме того, ниже приведен еще один пример, в котором для привлечения внимания каждый строковый литерал сопровождается звуковым сигналом:

```
static void EscapeChars()
{
    Console.WriteLine("=> Escape characters:\a");
    string strWithTabs = "Model\tColor\tSpeed\tPet Name\a ";
    Console.WriteLine(strWithTabs);

    Console.WriteLine("Everyone loves \"Hello World\"\a ");
    Console.WriteLine("C:\\MyApp\\bin\\Debug\a ");

    // Добавить четыре пустых строки и снова выдать звуковой сигнал.
    Console.WriteLine("All finished.\n\n\n\a ");
    Console.WriteLine();
}
```

Определение дословных строк

За счет добавления к строковому литералу префикса `@` можно создавать так называемые *дословные строки*. Используя дословные строки, вы отключаете обработку управляющих последовательностей в литералах и заставляете выводить значения `string` в том виде, как есть. Такая возможность наиболее полезна при работе со строками, представляющими пути к каталогам и сетевым ресурсам.

Таким образом, вместо применения управляющей последовательности `\\` можно поступить следующим образом:

```
// Следующая строка воспроизводится дословно,  
// так что отображаются все управляющие символы.  
Console.WriteLine(@"C:\MyApp\bin\Debug");
```

Также обратите внимание, что дословные строки могут использоваться для предохранения пробельных символов в строках, разнесенных по нескольким строкам вывода:

```
// При использовании дословных строк пробельные символы предохраняются.  
string myLongString = @"This is a very  
    very  
        long string";  
Console.WriteLine(myLongString);
```

Применяя дословные строки, в литеральную строку можно также напрямую вставлять символы двойной кавычки, просто дублируя знак `"`:

```
Console.WriteLine(@"Cerebus said ""Darrrr! Pret-ty sun-sets""");
```

Строки и равенство

Как будет подробно объясняться в главе 4, *ссылочный тип* — это объект, размещаемый в управляемой куче со сборкой мусора. По умолчанию при выполнении проверки на предмет равенства ссылочных типов (с помощью операций `==` и `!=` языка C#) значение `true` будет возвращаться в случае, если обе ссылки указывают на один и тот же объект в памяти. Однако, несмотря на то, что тип `string` в действительности является ссылочным, операции равенства для него были переопределены так, чтобы можно было сравнивать значения объектов `string`, а не ссылки на объекты в памяти.

```
static void StringEquality()  
{  
    Console.WriteLine("=> String equality:");  
    string s1 = "Hello!";  
    string s2 = "Yo!";  
    Console.WriteLine("s1 = {0}", s1);  
    Console.WriteLine("s2 = {0}", s2);  
    Console.WriteLine();  
    // Проверить строки на равенство.  
    Console.WriteLine("s1 == s2: {0}", s1 == s2);  
    Console.WriteLine("s1 == Hello!: {0}", s1 == "Hello!");  
    Console.WriteLine("s1 == HELLO!: {0}", s1 == "HELLO!");  
    Console.WriteLine("s1 == hello!: {0}", s1 == "hello!");  
    Console.WriteLine("s1.Equals(s2): {0}", s1.Equals(s2));  
    Console.WriteLine("Yo.Equals(s2): {0}", "Yo!".Equals(s2));  
    Console.WriteLine();  
}
```

Операции равенства C# выполняют в отношении объектов `string` посимвольную проверку равенства с учетом регистра и нечувствительную к культуре. Следовательно, строка `"Hello!"` не равна строке `"HELLO!"` и также отличается от строки `"hello!"`. Кроме того, памятуя о связи между `string` и `System.String`, проверку на предмет равенства можно осуществлять с использованием метода `Equals()` класса `String` и других поддерживаемых им операций равенства. Наконец, поскольку каждый строковый литерал (такой как `"Yo"`) является допустимым экземпляром `System.String`, доступ к функциональности, ориентированной на работу со строками, можно получать для фиксированной последовательности символов.

Модификация поведения сравнения строк

Как уже упоминалось, операции равенства строк (`Compare()`, `Equals()` и `==`), а также функция `IndexOf()` по умолчанию являются чувствительными к регистру символов и нечувствительными к культуре. Если ваша программа не заботится о регистре символов, тогда может возникнуть проблема. Один из способов ее преодоления предполагает преобразование строк в верхний или нижний регистр и затем их сравнение:

```
if (firstString.ToUpper() == secondString.ToUpper())
{
    // Делать что-то
}
```

Здесь создается копия каждой строки со всеми символами верхнего регистра. В большинстве ситуаций это не проблема, но в случае очень крупных строк может пострадать производительность. И дело даже не производительности — написание каждый раз такого кода преобразования становится утомительным. А что, если вы забудете вызвать `ToUpper()`? Результатом будет трудная в обнаружении ошибка.

Гораздо лучший прием предусматривает применение перегруженных версий перечисленных ранее методов, которые принимают значение перечисления `StringComparison`, управляющего выполнением сравнения. Значения `StringComparison` описаны в табл. 3.7.

Таблица 3.7. Значения перечисления `StringComparison`

Операция равенства/отношения C#	Описание
<code>CurrentCulture</code>	Сравнивает строки с использованием правил сортировки, чувствительной к культуре, и текущей культуры
<code>CurrentCultureIgnoreCase</code>	Сравнивает строки с применением правил сортировки, чувствительной к культуре, и текущей культуры, игнорируя регистр символов сравниваемых строк
<code>InvariantCulture</code>	Сравнивает строки с использованием правил сортировки, чувствительной к культуре, и инвариантной культуры
<code>InvariantCultureIgnoreCase</code>	Сравнивает строки с применением правил сортировки, чувствительной к культуре, и инвариантной культуры, игнорируя регистр символов сравниваемых строк
<code>Ordinal</code>	Сравнивает строки с использованием правил ordinalной (двоичной) сортировки
<code>OrdinalIgnoreCase</code>	Сравнивает строки с использованием правил ordinalной (двоичной) сортировки, игнорируя регистр символов сравниваемых строк

Чтобы взглянуть на результаты применения `StringComparison`, создадим новый метод по имени `StringEqualitySpecifyingCompareRules()` со следующим кодом:

```
static void StringEqualitySpecifyingCompareRules()
{
    Console.WriteLine("> String equality (Case Insensitive:");
    string s1 = "Hello!";
    string s2 = "HELLO!";
    Console.WriteLine("s1 = {0}", s1);
    Console.WriteLine("s2 = {0}", s2);
    Console.WriteLine();
}
```



```
// Проверить результаты изменения стандартных правил сравнения.
Console.WriteLine("Default rules: s1={0},s2={1}s1.Equals(s2): {2}", s1, s2,
s1.Equals(s2));
Console.WriteLine("Ignore case: s1.Equals(s2,
                    StringComparison.OrdinalIgnoreCase): {0}",
                    s1.Equals(s2, StringComparison.OrdinalIgnoreCase));
Console.WriteLine("Ignore case, Invariant Culture:
                    s1.Equals(s2, StringComparison.InvariantCultureIgnoreCase): {0}",
                    s1.Equals(s2, StringComparison.InvariantCultureIgnoreCase));
Console.WriteLine();
Console.WriteLine("Default rules: s1={0},s2={1} s1.IndexOf(\"E\"): {2}",
                    s1, s2, s1.IndexOf("E"));
Console.WriteLine("Ignore case: s1.IndexOf(\"E\",
                    StringComparison.OrdinalIgnoreCase): {0}",
                    s1.IndexOf("E", StringComparison.OrdinalIgnoreCase));
Console.WriteLine("Ignore case, Invariant Culture: s1.IndexOf(\"E\",
                    StringComparison.InvariantCultureIgnoreCase): {0}",
                    s1.IndexOf("E", StringComparison.InvariantCultureIgnoreCase));
Console.WriteLine();
}
```

Во время как приведенные здесь примеры просты и используют те же самые буквы в большинстве культур, если ваше приложение должно принимать во внимание разные наборы культур, тогда применение перечисления `StringComparison` становится обязательным.

Строки являются неизменяемыми

Один из интересных аспектов класса `System.String` связан с тем, что после присваивания объекту `string` начального значения символьные данные *не могут быть изменены*. На первый взгляд это может показаться противоречащим действительности, ведь строкам постоянно присваиваются новые значения, а в классе `System.String` доступен набор методов, которые, похоже, только то и делают, что изменяют символьные данные тем или иным образом (скажем, преобразуя их в верхний или нижний регистр). Тем не менее, присмотревшись внимательнее к тому, что происходит "за кулисами", вы заметите, что методы типа `string` на самом деле возвращают новый объект `string` в модифицированном виде:

```
static void StringsAreImmutable()
{
    // Установить начальное значение для строки.
    string s1 = "This is my string.";
    Console.WriteLine("s1 = {0}", s1);

    // Преобразована ли строка s1 в верхний регистр?
    string upperString = s1.ToUpper();
    Console.WriteLine("upperString = {0}", upperString);

    // Нет! Строка s1 осталась в том же виде!
    Console.WriteLine("s1 = {0}", s1);
}
```

Просмотрев показанный далее вывод, можно убедиться, что в результате вызова метода `ToUpper()` исходный объект `string` (`s1`) не преобразовывался в верхний регистр. Взамен была возвращена копия переменной типа `string` в измененном формате.

```
s1 = This is my string.
upperString = THIS IS MY STRING.
s1 = This is my string.
```

Тот же самый закон неизменяемости строк действует и в случае применения операции присваивания C#. Чтобы проиллюстрировать, реализуем следующий метод `StringsAreImmutable2()`:

```
static void StringsAreImmutable2()
{
    string s2 = "My other string";
    s2 = "New string value";
}
```

Скомпилируем приложение и загрузим сборку в `ildasm.exe` (см. главу 1). Ниже приведен код CIL, который будет сгенерирован для метода `StringsAreImmutable2()`:

```
.method private hidebysig static void StringsAreImmutable2() cil managed
{
    // Code size 14 (0xe)
    .maxstack 1
    .locals init ([0] string s2)
    IL_0000: nop
    IL_0001: ldstr      "My other string"
    IL_0006: stloc.0
    IL_0007: ldstr      "New string value"
    IL_000c: stloc.0
    IL_000d: ret
} // end of method Program::StringAreImmutable2
```

Хотя низкоуровневые детали языка CIL пока подробно не рассматривались, обратите внимание на многочисленные вызовы кода операции `ldstr` ("load string" — "загрузить строку"). Попросту говоря, код операции `ldstr` языка CIL загружает новый объект `string` в управляемую кучу. Предыдущий объект `string`, который содержал значение "My other string", будет со временем удален сборщиком мусора.

Так что же в точности из всего этого следует? Выражаясь кратко, класс `string` может стать неэффективным и при неправильном употреблении приводить к "разбуханию" кода, особенно при выполнении конкатенации строк или при работе с большими объемами текстовых данных. Но если необходимо представлять элементарные символьные данные, такие как номер карточки социального страхования, имя и фамилия или простые фрагменты текста, используемые внутри приложения, тогда тип `string` будет идеальным вариантом.

Однако когда строится приложение, в котором текстовые данные будут часто изменяться (подобное текстовому процессору), то представление обрабатываемых текстовых данных с применением объектов `string` будет неудачным решением, т.к. оно практически наверняка (и часто косвенно) приведет к созданию излишних копий строковых данных. Тогда каким образом должен поступить программист? Ответ на этот вопрос вы найдете ниже.

Тип `System.Text.StringBuilder`

С учетом того, что тип `string` может оказаться неэффективным при безответственном использовании, библиотеки базовых классов .NET предоставляют пространство имен `System.Text`. Внутри этого (относительно небольшого) пространства имен находится класс `StringBuilder`. Как и `System.String`, класс `StringBuilder` определяет методы, которые позволяют, например, заменять или форматировать сегменты. Для применения класса `StringBuilder` в файлах кода C# первым делом понадобится импортировать следующее пространство имен в файл кода (что в случае нового проекта Visual Studio уже должно быть сделано):

```
// Здесь определен класс StringBuilder:
using System.Text;
```

Уникальность класса `StringBuilder` в том, что при вызове его членов производится прямое изменение внутренних символьных данных объекта (делая его более эффективным) без получения копии данных в модифицированном формате. При создании экземпляра `StringBuilder` начальные значения объекта могут быть заданы через один из множества *конструкторов*. Если вы не знакомы с понятием конструктора, тогда пока достаточно знать только то, что конструкторы позволяют создавать объект с начальным состоянием при использовании ключевого слова `new`. Взгляните на следующий пример применения `StringBuilder`:

```
static void FunWithStringBuilder()
{
    Console.WriteLine("=> Using the StringBuilder:");
    StringBuilder sb = new StringBuilder("**** Fantastic Games ****");
    sb.Append("\n");
    sb.AppendLine("Half Life");
    sb.AppendLine("Morrowind");
    sb.AppendLine("Deus Ex" + "2");
    sb.AppendLine("System Shock");
    Console.WriteLine(sb.ToString());
    sb.Replace("2", " Invisible War");
    Console.WriteLine(sb.ToString());
    Console.WriteLine("sb has {0} chars.", sb.Length);
    Console.WriteLine();
}
```

Здесь создается объект `StringBuilder` с начальным значением `"**** Fantastic Games ****"`. Как видите, можно добавлять строки в конец внутреннего буфера, а также заменять или удалять любые символы. По умолчанию `StringBuilder` способен хранить строку только длиной 16 символов или меньше (но при необходимости будет автоматически расширяться); однако стандартное начальное значение длины можно изменить посредством дополнительного аргумента конструктора:

```
// Создать экземпляр StringBuilder с исходным размером в 256 символов.
StringBuilder sb = new StringBuilder("**** Fantastic Games ****", 256);
```

При добавлении большего количества символов, чем в указанном лимите, объект `StringBuilder` скопирует свои данные в новый экземпляр и увеличит размер буфера на заданный лимит.

Интерполяция строк

Синтаксис с фигурными скобками, продемонстрированный ранее в главе `{0}`, `{1}` и т.д.), существовал внутри платформы .NET еще со времен версии 1.0. Начиная с выпуска C# 6, при построении строковых литералов, содержащих заполнители для переменных, программисты C# могут использовать альтернативный синтаксис. Формально он называется *интерполяцией строк*. Несмотря на то что вывод операции идентичен выводу, получаемому с помощью традиционного синтаксиса форматирования строк, новый подход позволяет напрямую внедрять сами переменные, а не помещать их в список с разделителями-запятыеми.

Взгляните на показанный ниже дополнительный метод в нашем классе `Program` (`StringInterpolation()`), который строит переменную типа `string` с применением обоих подходов:

```
static void StringInterpolation()
{
    // Некоторые локальные переменные будут включены в крупную строку.
    int age = 4;
    string name = "Soren";

    // Использование синтаксиса с фигурными скобками.
    string greeting = string.Format("Hello {0} you are {1} years old.", name, age);

    // Использование интерполяции строк.
    string greeting2 = $"Hello {name} you are {age} years old.";
}
```

В переменной `greeting2` обратите внимание на то, что конструируемая строка начинается с префикса `$`. Кроме того, фигурные скобки по-прежнему используются для пометки заполнителя под переменную; тем не менее, вместо применения числовой метки имеется возможность указывать непосредственно переменную. Предполагаемое преимущество заключается в том, что новый синтаксис несколько легче читать в линейной манере (слева направо) с учетом того, что не требуется “перескакивать в конец” для просмотра списка значений, подлежащих вставке во время выполнения.

С новым синтаксисом связан еще один интересный аспект: фигурные скобки, используемые в интерполяции строк, обозначают допустимую область видимости. Таким образом, с переменными можно применять операцию точки, чтобы изменять их состояние. Модифицируем код присваивания переменных `greeting` и `greeting2`:

```
string greeting = string.Format("Hello {0} you are {1} years old.",
                               name.ToUpper(), age);
string greeting2 = $"Hello {name.ToUpper()} you are {age} years old.";
```

Здесь посредством вызова `ToUpper()` производится преобразование `name` в верхний регистр. Обратите внимание, что при подходе с интерполяцией строк завершающая пара круглых скобок к вызову данного метода *не* добавляется. Учитывая это, использовать область видимости, определяемую фигурными скобками, как полноценную область видимости метода, которая содержит многочисленные строки исполняемого кода, невозможно. Взамен допускается только вызывать одиночный метод на объекте с применением операции точки, а также определять простое общее выражение наподобие `{age} += 1`).

Полезно также отметить, что в рамках нового синтаксиса по-прежнему можно использовать управляющие последовательности внутри строкового литерала. Таким образом, для вставки символа табуляции необходимо применять последовательность `\t`:

```
string greeting = string.Format("\tHello {0} you are {1} years old.",
                               name.ToUpper(), age);
string greeting2 = $" \tHello {name.ToUpper()} you are {age} years old.";
```

Как и следовало ожидать, при построении переменных типа `string` на лету вы вправе использовать любой из двух подходов. Однако имейте в виду, что в случае работы с более ранней версией платформы .NET применение синтаксиса интерполяции строк приведет к ошибке на этапе компиляции. Следовательно, если вам необходимо обеспечить успешную компиляцию кода C# с помощью множества версий компилятора, то безопаснее придерживаться традиционного подхода с нумерованными заполнителями.

Сужающие и расширяющие преобразования типов данных

Теперь, когда вы понимаете, как работать с внутренними типами данных C#, давайте рассмотрим связанную тему *преобразования типов данных*. Создадим новый проект консольного приложения по имени `TypeConversions` и определим в нем следующий класс:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with type conversions *****");
        // Добавить две переменные типа short и вывести результат.
        short numbl = 9, numb2 = 10;
        Console.WriteLine("{0} + {1} = {2}",
            numbl, numb2, Add(numbl, numb2));
        Console.ReadLine();
    }

    static int Add(int x, int y)
    {
        return x + y;
    }
}
```

Легко заметить, что метод `Add()` ожидает передачи двух параметров `int`. Тем не менее, в методе `Main()` ему на самом деле передаются две переменные типа `short`. Хотя это может выглядеть похожим на несоответствие типов данных, программа компилируется и выполняется без ошибок, возвращая ожидаемый результат 19.

Причина, по которой компилятор считает такой код синтаксически корректным, связана с тем, что потеря данных в нем невозможна. Из-за того, что максимальное значение для типа `short` (32 767) гораздо меньше максимального значения для типа `int` (2 147 483 647), компилятор неявно *расширяет* каждое значение `short` до типа `int`. Формально термин *расширение* используется для определения неявного *восходящего приведения*, которое не вызывает потерю данных.

На заметку! Разрешенные расширяющие и сужающие (обсуждаются далее) преобразования, поддерживаемые для каждого типа данных C#, описаны в разделе "Type Conversion Tables" ("Таблицы преобразования типов") документации .NET Framework 4.7 SDK.

Несмотря на то что неявное расширение типов благоприятствовало в предыдущем примере, в других ситуациях оно может стать источником ошибок на этапе компиляции. Например, пусть для переменных `numbl` и `numb2` установлены значения, которые (при их сложении) превышают максимальное значение типа `short`. Кроме того, предположим, что возвращаемое значение метода `Add()` сохраняется в новой локальной переменной `short`, а не напрямую выводится на консоль.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with type conversions *****");
    // Следующий код вызовет ошибку на этапе компиляции!
    short numbl = 30000, numb2 = 30000;
    short answer = Add(numbl, numb2);
}
```

```

Console.WriteLine("{0} + {1} = {2}",
    numb1, numb2, answer);
Console.ReadLine();
}

```

В данном случае компилятор сообщит об ошибке:

Cannot implicitly convert type 'int' to 'short'. An explicit conversion exists (are you missing a cast?)

Не удастся неявно преобразовать тип int в short. Существует явное преобразование (возможно, пропущено приведение)

Проблема в том, что хотя метод Add() способен возвратить значение int, равное 60 000 (т.к. оно уместается в допустимый диапазон для System.Int32), это значение не может быть сохранено в переменной short, потому что выходит за пределы диапазона допустимых значений для типа short. Выражаясь формально, среде CLR не удалось применить *сужающую операцию*. Нетрудно догадаться, что сужающая операция является логической противоположностью расширяющей операции, поскольку предусматривает сохранение большего значения внутри переменной типа данных с меньшим диапазоном допустимых значений.

Важно отметить, что все сужающие преобразования приводят к ошибкам на этапе компиляции, даже когда есть основание полагать, что такое преобразование должно пройти успешно. Например, следующий код также вызовет ошибку при компиляции:

```

// Снова ошибка на этапе компиляции!
static void NarrowingAttempt()
{
    byte myByte = 0;
    int myInt = 200;
    myByte = myInt;

    Console.WriteLine("Value of myByte: {0}", myByte);
}

```

Здесь значение, содержащееся в переменной типа int (myInt), благополучно уместается в диапазон допустимых значений для типа byte; следовательно, можно было бы ожидать, что сужающая операция не должна привести к ошибке во время выполнения. Однако из-за того, что язык C# создавался с расчетом на безопасность в отношении типов, все-таки будет получена ошибка на этапе компиляции.

Если нужно проинформировать компилятор о том, что вы готовы мириться с возможной потерей данных из-за сужающей операции, тогда потребуется применить *явное приведение*, используя операцию приведения () языка C#. Взгляните на показанную далее модификацию класса Program:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with type conversions *****");
        short numb1 = 30000, numb2 = 30000;

        // Явно привести int к short (и разрешить потерю данных).
        short answer = (short)Add(numbl, numb2);

        Console.WriteLine("{0} + {1} = {2}",
            numb1, numb2, answer);
        NarrowingAttempt();
        Console.ReadLine();
    }
}

```

```

static int Add(int x, int y)
{
    return x + y;
}

static void NarrowingAttempt()
{
    byte myByte = 0;
    int myInt = 200;

    // Явно привести int к byte (без потери данных).
    myByte = (byte)myInt;
    Console.WriteLine("Value of myByte: {0}", myByte);
}
}

```

Теперь компиляция кода проходит успешно, но результат сложения оказывается совершенно неправильным:

```

***** Fun with type conversions *****
30000 + 30000 = -5536
Value of myByte: 200

```

Как вы только что удостоверились, явное приведение заставляет компилятор применить сужающее преобразование, даже когда оно может вызвать потерю данных. В случае метода `NarrowingAttempt()` это не было проблемой, т.к. значение 200 умещалось в диапазон допустимых значений для типа `byte`. Тем не менее, в ситуации со сложением двух значений типа `short` внутри `Main()` конечный результат получился полностью неприемлемым ($30\,000 + 30\,000 = -5536$).

Для построения приложений, в которых потеря данных не допускается, язык C# предлагает ключевые слова `checked` и `unchecked`, которые позволяют гарантировать, что потеря данных не останется необнаруженной.

Ключевое слово `checked`

Давайте начнем с выяснения роли ключевого слова `checked`. Предположим, что в класс `Program` добавлен новый метод, который пытается просуммировать две переменные типа `byte`, причем каждой из них было присвоено значение, не превышающее допустимый максимум (255). По идее после сложения значений этих двух переменных (сприведением результата `int` к типу `byte`) должна быть получена точная сумма.

```

static void ProcessBytes()
{
    byte b1 = 100;
    byte b2 = 250;
    byte sum = (byte)Add(b1, b2);

    // В sum должно содержаться значение 350. Однако там оказывается значение 94!
    Console.WriteLine("sum = {0}", sum);
}

```

Удивительно, но при просмотре вывода приложения обнаруживается, что в переменной `sum` содержится значение 94 (а не 350, как ожидалось). Причина проста. Учитывая, что `System.Byte` может хранить только значение в диапазоне от 0 до 255 включительно, в `sum` будет помещено значение переполнения ($350 - 256 = 94$). По умолчанию, если не предпринимаются никакие корректирующие действия, то условия переполнения и потери значимости происходят без выдачи сообщений об ошибках.

Для обработки условий переполнения и потери значимости в приложении доступны два способа. Это можно делать вручную, полагаясь на свои знания и навыки в области

программирования. Недостаток такого подхода произрастает из того факта, что мы все-го лишь люди, и даже приложив максимум усилий, все равно можем попросту упустить из виду какие-то ошибки.

К счастью, язык C# предоставляет ключевое слово `checked`. Когда оператор (или блок операторов) помещен в контекст `checked`, компилятор C# выпускает дополнительные инструкции CIL, обеспечивающие проверку условий переполнения, которые могут возникать при сложении, умножении, вычитании или делении двух значений числовых типов.

Если происходит переполнение, тогда во время выполнения генерируется исключение `System.OverflowException`. В главе 7 будут предложены подробные сведения о структурированной обработке исключений, а также об использовании ключевых слов `try` и `catch`. Не вдаваясь пока в детали, взгляните на следующий модифицированный код:

```
static void ProcessBytes()
{
    byte b1 = 100;
    byte b2 = 250;

    // На этот раз сообщить компилятору о необходимости добавления
    // кода CIL, необходимого для генерации исключения, если возникает
    // переполнение или потеря значимости.
    try
    {
        byte sum = checked((byte)Add(b1, b2));
        Console.WriteLine("sum = {0}", sum);
    }
    catch (OverflowException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Обратите внимание, что возвращаемое значение метода `Add()` помещено в контекст ключевого слова `checked`. Поскольку значение `sum` выходит за пределы допустимого диапазона для типа `byte`, генерируется исключение времени выполнения. Сообщение об ошибке выводится посредством свойства `Message`:

```
Arithmetic operation resulted in an overflow.
Арифметическая операция привела к переполнению.
```

Чтобы обеспечить принудительную проверку переполнения для целого блока операторов, контекст `checked` можно определить так:

```
try
{
    checked
    {
        byte sum = (byte)Add(b1, b2);
        Console.WriteLine("sum = {0}", sum);
    }
}
catch (OverflowException ex)
{
    Console.WriteLine(ex.Message);
}
```

В любом случае интересующий код будет автоматически оцениваться на предмет возможных условий переполнения, и если они обнаружатся, то сгенерируется исключение, связанное с переполнением.

Настройка проверки переполнения на уровне проекта

Если создается приложение, в котором никогда не должно возникать молчаливое переполнение, то может обнаружиться, что в контекст ключевого слова `checked` приходится помещать слишком много строк кода. В качестве альтернативы компилятор С# поддерживает флаг `/checked`. Когда он указан, все присутствующие в коде арифметические операции будут оцениваться на предмет переполнения, не требуя применения ключевого слова `checked`. Если переполнение было обнаружено, тогда сгенерируется исключение времени выполнения.

Для активизации флага `/checked` в Visual Studio откройте окно свойств проекта, перейдите на вкладку **Build** (Сборка) и щелкните на кнопке **Advanced** (Дополнительно). В открывшемся диалоговом окне отметьте флажок **Check for arithmetic overflow/underflow** (Проверять арифметическое переполнение и потерю значимости), как показано на рис. 3.4.

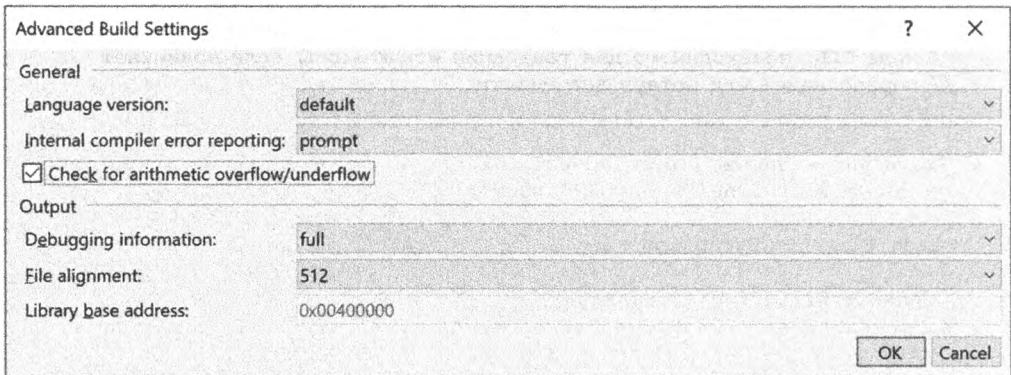


Рис. 3.4. Включение проверки переполнения и потери значимости в масштабах проекта

Включить эту настройку может быть удобно при создании отладочной версии сборки. После устранения всех условий переполнения из кодовой базы флаг `/checked` можно отключить для последующих построений сборки (что приведет к увеличению производительности приложения).

Ключевое слово `unchecked`

А теперь предположим, что проверка переполнения и потери значимости включена в масштабах проекта, но есть блок кода, в котором потеря данных приемлема. Как с ним быть? Учитывая, что действие флага `/checked` распространяется на всю арифметическую логику, в языке С# имеется ключевое слово `unchecked`, которое предназначено для отмены генерации исключений, связанных с переполнением, в отдельных случаях. Ключевое слово `unchecked` используется аналогично `checked`, т.е. его можно применять как к единственному оператору, так и к блоку операторов.

```
// Предполагая, что флаг /checked активизирован, этот блок
// не будет генерировать исключение времени выполнения.
unchecked
{
    byte sum = (byte) (b1 + b2);
    Console.WriteLine("sum = {0} ", sum);
}
```

Подводя итоги по ключевым словам `checked` и `unchecked` в C#, отметим, что стандартное поведение исполняющей среды .NET предусматривает игнорирование арифметического переполнения и потери значимости. Когда необходимо обрабатывать избранные операторы, должно использоваться ключевое слово `checked`. Если нужно перехватывать ошибки переполнения по всему приложению, то придется активизировать флаг `/checked`. Наконец, ключевое слово `unchecked` может применяться при наличии блока кода, в котором переполнение приемлемо (и, следовательно, не должно приводить к генерации исключения времени выполнения).

Исходный код. Проект `TypeConversions` доступен в подкаталоге `Chapter_3`.

Понятие неявно типизированных локальных переменных

Вплоть до этого места в главе при объявлении каждой локальной переменной *явно* указывался ее тип данных:

```
static void DeclareExplicitVars()
{
    // Явно типизированные локальные переменные
    // объявляются следующим образом:
    // типДанных имяПеременной = начальноеЗначение;
    int myInt = 0;
    bool myBool = true;
    string myString = "Time, marches on...";
}
```

В то время как многие (включая и авторов) согласятся с тем, что явное указание типа данных для каждой переменной является рекомендуемой практикой, язык C# поддерживает возможность *неявной типизации* локальных переменных с использованием ключевого слова `var`. Ключевое слово `var` может применяться вместо указания конкретного типа данных (такого как `int`, `bool` или `string`). Когда вы поступаете подобным образом, компилятор будет автоматически выводить лежащий в основе тип данных на основе начального значения, используемого для инициализации локального элемента данных.

Чтобы продемонстрировать роль неявной типизации, создадим новый проект консольного приложения по имени `ImplicitlyTypedLocalVars`. Обратите внимание, что локальные переменные, которые присутствовали в показанной выше версии метода, теперь объявлены следующим образом:

```
static void DeclareImplicitVars()
{
    // Неявно типизированные локальные переменные
    // объявляются следующим образом:
    // var имяПеременной = начальноеЗначение;
    var myInt = 0;
    var myBool = true;
    var myString = "Time, marches on...";
}
```

На заметку! Строго говоря, `var` не является ключевым словом языка C#. Вполне допустимо объявлять переменные, параметры и поля по имени `var`, не получая ошибок на этапе компиляции. Однако когда лексема `var` применяется в качестве типа данных, то в таком контексте она трактуется компилятором как ключевое слово.

В таком случае, основываясь на первоначально присвоенных значениях, компилятор способен вывести для переменной `myInt` тип `System.Int32`, для переменной `myBool` — тип `System.Boolean`, а для переменной `myString` — тип `System.String`. В сказанном легко убедиться за счет вывода на консоль имен типов с помощью *рефлексии*. Как будет показано в главе 15, рефлексия представляет собой действие по определению состава типа во время выполнения. Например, с помощью рефлексии можно определить тип данных неявно типизированной локальной переменной. Модифицируем метод `DeclareImplicitVars()`:

```
static void DeclareImplicitVars()
{
    // Неявно типизированные локальные переменные.
    var myInt = 0;
    var myBool = true;
    var myString = "Time, marches on...";

    // Вывести имена лежащих в основе типов.
    Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
    // Вывод типа myInt
    Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
    // Вывод типа myBool
    Console.WriteLine("myString is a: {0}", myString.GetType().Name);
    // Вывод типа myString
}
```

На заметку! Имейте в виду, что такую неявную типизацию можно использовать для любых типов, включая массивы, обобщенные типы (глава 9) и собственные специальные типы. В дальнейшем вы увидите и другие примеры неявной типизации.

Вызвав метод `DeclareImplicitVars()` внутри `Main()`, вы получите следующий вывод:

```
***** Fun with Implicit Typing *****
myInt is a: Int32
myBool is a: Boolean
myString is a: String
```

Ограничения неявно типизированных переменных

С использованием ключевого слова `var` связаны разнообразные ограничения. Прежде всего, неявная типизация применима *только* к локальным переменным внутри области видимости метода или свойства. Использовать ключевое слово `var` для определения возвращаемых значений, параметров или данных полей в специальном типе не допускается. Например, показанное ниже определение класса приведет к выдаче различных сообщений об ошибках на этапе компиляции:

```
class ThisWillNeverCompile
{
    // Ошибка! Ключевое слово var не может применяться к полям!
    private var myInt = 10;

    // Ошибка! Ключевое слово var не может применяться
    // к возвращаемому значению или типу параметра!
    public var MyMethod(var x, var y){}
}
```

Кроме того, локальным переменным, которые объявлены с ключевым словом `var`, **обязано** присваиваться начальное значение в самом объявлении, причем присваивать `null` в качестве начального значения **невозможно**. Последнее ограничение должно быть рациональным, потому что на основании только `null` компилятору не удастся вывести тип, на который бы указывала переменная.

```
// Ошибка! Должно быть присвоено значение!
var myData;

// Ошибка! Значение должно присваиваться в самом объявлении!
var myInt;
myInt = 0;

// Ошибка! Нельзя присваивать null в качестве начального значения!
var myObj = null;
```

Тем не менее, присваивать `null` локальной переменной, тип которой выведен в результате начального присваивания, разрешено (при условии, что это ссылочный тип):

```
// Допустимо, если SportsCar имеет ссылочный тип!
var myCar = new SportsCar();
myCar = null;
```

Вдобавок значение неявно типизированной локальной переменной допускается присваивать другим переменным, которые типизированы как неявно, так и явно:

```
// Также нормально!
var myInt = 0;
var anotherInt = myInt;

string myString = "Wake up!";
var myData = myString;
```

Наконец, неявно типизированную локальную переменную разрешено возвращать вызывающему компоненту при условии, что возвращаемый тип метода и выведенный тип переменной, определенной посредством `var`, совпадают:

```
static int GetAnInt()
{
    var retVal = 9;
    return retVal;
}
```

Неявно типизированные данные являются строго типизированными

Имейте в виду, что неявная типизация локальных переменных дает в результате *строго типизированные данные*. Таким образом, применение ключевого слова `var` в языке C# — не тот же самый прием, который используется в сценарных языках (вроде JavaScript или Perl). Кроме того, ключевое слово `var` — это не тип данных `Variant` в COM, когда переменная на протяжении своего времени жизни может хранить значения разных типов (что часто называют *динамической типизацией*).

На заметку! В C# поддерживается возможность динамической типизации с применением ключевого слова `dynamic`. Вы узнаете о таком аспекте языка в главе 16.

Взамен средство вывода типов сохраняет аспект строгой типизации языка C# и воздействует только на объявление переменных при компиляции. Затем данные трактуются, как если бы они были объявлены с выведенным типом; присваивание такой переменной значения другого типа будет приводить к ошибке на этапе компиляции.

```
static void ImplicitTypingIsStrongTyping()
{
    // Компилятору известно, что s имеет тип System.String.
    var s = "This variable can only hold string data!";
    s = "This is fine...";

    // Можно обращаться к любому члену лежащего в основе типа.
    string upper = s.ToUpper();

    // Ошибка! Присваивание числовых данных строке не допускается!
    s = 44;
}
```

Полезность неявно типизированных локальных переменных

Теперь, когда вы видели синтаксис, используемый для объявления неявно типизируемых локальных переменных, вас наверняка интересует, в каких ситуациях его следует применять. Прежде всего, использование `var` для объявления локальных переменных просто ради интереса особой пользы не принесет. Такой подход может вызвать путаницу у тех, кто будет изучать код, поскольку лишает возможности быстро определить лежащий в основе тип данных и, следовательно, затрудняет понимание общего назначения переменной. Поэтому если вы знаете, что переменная должна относиться к типу `int`, то сразу и объявляйте ее с типом `int`!

Однако, как будет показано в начале главы 12, в наборе технологий LINQ применяются выражения запросов, которые могут выдавать динамически создаваемые результирующие наборы, основанные на формате самого запроса. В таких случаях неявная типизация исключительно удобна, потому что вам не придется явно определять тип, который запрос может возвращать, а в ряде ситуаций это вообще невозможно. Посмотрите, сможете ли вы определить лежащий в основе тип данных `subset` в следующем примере кода LINQ?

```
static void LinqQueryOverInts()
{
    int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

    // Запрос LINQ!
    var subset = from i in numbers where i < 10 select i;
    Console.WriteLine("Values in subset: ");
    foreach (var i in subset)
    {
        Console.WriteLine("{0} ", i);
    }
    Console.WriteLine();

    // К какому же типу относится subset?
    Console.WriteLine("subset is a: {0}", subset.GetType().Name);
    Console.WriteLine("subset is defined in: {0}", subset.GetType().Namespace);
}
```

Вы можете предположить, что типом данных `subset` будет массив целочисленных значений. Но на самом деле он представляет собой низкоуровневый тип данных LINQ, о котором вы вряд ли что-то знаете, если только не работаете с LINQ длительное время или не откроете скомпилированный образ в утилите `ildasm.exe`. Хорошая новость в том, что при использовании LINQ вы редко (если вообще когда-либо) беспокоитесь о типе возвращаемого значения запроса; вы просто присваиваете значение неявно типизированной локальной переменной.

Фактически можно было бы даже утверждать, что *единственным случаем*, когда применение ключевого слова `var` полностью оправдано, является определение данных, возвращаемых из запроса LINQ. Запомните, если вы знаете, что нужна переменная `int`, то просто объявляйте ее с типом `int`! Злоупотребление неявной типизацией в производственном коде (через ключевое слово `var`) большинство разработчиков расценивают как плохой стиль кодирования.

Исходный код. Проект `ImplicitlyTypedLocalVars` доступен в подкаталоге `Chapter_3`.

Итерационные конструкции C#

Все языки программирования предлагают средства для повторения блоков кода до тех пор, пока не будет удовлетворено условие завершения. С каким бы языком вы не имели дело в прошлом, итерационные операторы C# не должны вызывать особого удивления или требовать лишь небольшого объяснения. В C# предоставляются четыре итерационные конструкции:

- цикл `for`;
- цикл `foreach/in`;
- цикл `while`;
- цикл `do/while`.

Давайте рассмотрим каждую конструкцию заикливания по очереди, создав новый проект консольного приложения по имени `IterationsAndDecisions`.

На заметку! Материал данного раздела главы будет кратким и по существу, т.к. здесь предполагается наличие у вас опыта работы с аналогичными ключевыми словами (`if`, `for`, `switch` и т.д.) в другом языке программирования. Если нужна дополнительная информация, просмотрите темы “Iteration Statements (C# Reference)” (“Операторы итераций (справочник по C#)”), “Jump Statements (C# Reference)” (“Операторы перехода (Справочник по C#)”) и “Selection Statements (C# Reference)” (“Операторы выбора (Справочник по C#)”) в документации .NET Framework 4.7 SDK.

Цикл `for`

Когда требуется повторять блок кода фиксированное количество раз, хороший уровень гибкости предлагает оператор `for`. В действительности вы имеете возможность указывать, сколько раз должен повторяться блок кода, а также задавать условие завершения. Не вдаваясь в излишние подробности, ниже представлен пример синтаксиса:

```
// Базовый цикл for.
static void ForLoopExample()
{
    // Обратите внимание, что переменная i видима только в контексте цикла for.
    for(int i = 0; i < 4; i++)
    {
        Console.WriteLine("Number is: {0} ", i);
    }
    // Здесь переменная i больше видимой не будет.
}
```

Все трюки, которые вы научились делать в языках C, C++ и Java, по-прежнему могут использоваться при формировании операторов `for` в C#. Допускается создавать слож-

ные условия завершения, строить бесконечные циклы и циклы в обратном направлении (посредством операции `--`), а также применять ключевые слова `goto`, `continue` и `break`.

Цикл `foreach`

Ключевое слово `foreach` языка C# позволяет проходить в цикле по всем элементам внутри контейнера без необходимости в проверке верхнего предела. Тем не менее, в отличие от цикла `for` цикл `foreach` будет выполнять проход по контейнеру только линейным ($n+1$) образом (т.е. не получится проходить по контейнеру в обратном направлении, пропускать каждый третий элемент и т.п.).

Однако если нужно просто выполнить проход по коллекции элемент за элементом, то цикл `foreach` будет великолепным выбором. Ниже приведены два примера использования цикла `foreach` — один для обхода массива строк и еще один для обхода массива целых чисел. Обратите внимание, что тип, указанный перед ключевым словом `in`, представляет тип данных контейнера.

```
// Проход по элементам массива посредством foreach.
static void ForEachLoopExample()
{
    string[] carTypes = { "Ford", "BMW", "Yugo", "Honda" };
    foreach (string c in carTypes)
        Console.WriteLine(c);

    int[] myInts = { 10, 20, 30, 40 };
    foreach (int i in myInts)
        Console.WriteLine(i);
}
```

За ключевым словом `in` может быть указан простой массив (как в приведенном примере) или, точнее говоря, любой класс, реализующий интерфейс `IEnumerable`. Как вы увидите в главе 9, библиотеки базовых классов .NET поставляются с несколькими коллекциями, которые содержат реализации распространенных абстрактных типов данных. Любой из них (скажем, обобщенный тип `List<T>`) может применяться внутри цикла `foreach`.

Использование неявной типизации в конструкциях `foreach`

В итерационных конструкциях `foreach` также допускается использование неявной типизации. Как и можно было ожидать, компилятор будет выводить корректный “вид типа”. Вспомните пример метода `Linq`, представленный ранее в главе. Даже не зная точного типа данных переменной `subset`, с применением неявной типизации все-таки можно выполнять итерацию по результирующему набору:

```
static void LinqQueryOverInts()
{
    int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
    // Запрос LINQ!
    var subset = from i in numbers where i < 10 select i;
    Console.WriteLine("Values in subset: ");

    foreach (var i in subset)
    {
        Console.WriteLine("{0} ", i);
    }
}
```

Циклы `while` и `do/while`

Итерационная конструкция `while` удобна, когда блок операторов должен выполняться до тех пор, пока не будет удовлетворено некоторое условие завершения. Внутри области видимости цикла `while` необходимо позаботиться о том, чтобы это условие действительно удовлетворялось, иначе получится бесконечный цикл. В следующем примере сообщение "In while loop" будет постоянно выводиться на консоль, пока пользователь не завершит цикл вводом `yes` в командной строке:

```
static void WhileLoopExample()
{
    string userIsDone = "";
    // Проверить копию строки в нижнем регистре.
    while(userIsDone.ToLower() != "yes")
    {
        Console.WriteLine("In while loop");
        Console.Write("Are you done? [yes] [no]: "); // Запрос продолжения
        userIsDone = Console.ReadLine();
    }
}
```

С циклом `while` тесно связан оператор `do/while`. Подобно простому циклу `while` цикл `do/while` используется, когда какое-то действие должно выполняться неопределенное количество раз. Разница в том, что цикл `do/while` гарантирует, по крайней мере, однократное выполнение своего внутреннего блока кода. С другой стороны, вполне возможно, что цикл `while` вообще не выполнит блок кода, если условие оказывается ложным с самого начала.

```
static void DoWhileLoopExample()
{
    string userIsDone = "";
    do
    {
        Console.WriteLine("In do/while loop");
        Console.Write("Are you done? [yes] [no]: ");
        userIsDone = Console.ReadLine();
    }while(userIsDone.ToLower() != "yes"); //Обратите внимание на точку с запятой!
}
```

Конструкции принятия решений и операции отношения/равенства

Теперь, когда вы умеете многократно выполнять блок операторов, давайте рассмотрим следующую связанную концепцию — управление потоком выполнения программы. Для изменения потока выполнения программы на основе разнообразных обстоятельств в C# определены две простые конструкции:

- оператор `if/else`;
- оператор `switch`.

На заметку! В версии C# 7 выражение `is` и операторы `switch` расширяются посредством приема, называемого сопоставлением с образцом. Оба расширения и влияние таких изменений на операторы `if/else` и `switch` будут обсуждаться в главе 6 после рассмотрения правил, касающихся базовых/производных классов, приведения и стандартной операции `is`.

Оператор if/else

Первым мы рассмотрим оператор if/else. В отличие от C и C++ оператор if/else в языке C# может работать только с булевскими выражениями, но не с произвольными значениями вроде -1 и 0.

Операции отношения и равенства

Обычно для получения литерального булевского значения в операторах if/else применяются операции, описанные в табл. 3.8.

Таблица 3.8. Операции отношения и равенства в C#

Операция отношения/равенства	Пример использования	Описание
==	if(age == 30)	Возвращает true, если выражения являются одинаковыми
!=	if("Foo" != myStr)	Возвращает true, если выражения являются разными
<	if(bonus < 2000)	Возвращает true, если выражение слева (bonus) меньше, больше, меньше или равно либо больше или равно выражению справа (2000)
>	if(bonus > 2000)	
<=	if(bonus <= 2000)	
>=	if(bonus >= 2000)	

И снова программисты на C и C++ должны помнить о том, что старые трюки с проверкой условия, которое включает значение, не равное нулю, в языке C# работать не будут. Пусть необходимо проверить, содержит ли текущая строка более нуля символов. У вас может возникнуть соблазн написать такой код:

```
static void IfElseExample()
{
    // Недопустимо, т.к. свойство Length возвращает int, а не bool.
    string stringData = "My textual data";
    if(stringData.Length)
    {
        // Строка длиннее 0 символов
        Console.WriteLine("string is greater than 0 characters");
    }
    else
    {
        // Строка не длиннее 0 символов
        Console.WriteLine("string is not greater than 0 characters");
    }
    Console.WriteLine();
}
```

Если вы хотите использовать свойство String.Length для определения истинности или ложности, тогда выражение в условии понадобится изменить так, чтобы оно давало в результате булевское значение:

```
// Допустимо, т.к. условие возвращает true или false.
if(stringData.Length > 0)
{
    Console.WriteLine("string is greater than 0 characters");
}
```

Условная операция

Условная операция (?:) является сокращенным способом написания простого оператора if-else. Вот ее синтаксис:

```
условие ? первое_выражение : второе_выражение;
```

Условие представляет собой условную проверку (часть if оператора if-else). Если проверка проходит успешно, тогда выполняется код, следующий сразу за знаком вопроса (?). Если результат проверки отличается от true, то выполняется код, находящийся после двоеточия (часть else оператора if-else). Приведенный ранее пример кода можно было бы переписать с применением условной операции:

```
private static void ExecuteIfElseUsingConditionalOperator()
{
    string stringData = "My textual data";
    Console.WriteLine(stringData.Length > 0
        ? "string is greater than 0 characters"
        : "string is not greater than 0 characters");
    Console.WriteLine();
}
```

С условной операцией связаны некоторые ограничения. Во-первых, типы конструкций первое_выражение и второе_выражение должны быть одинаковыми. Во-вторых, условная операция может использоваться только в операторах присваивания. Следующий код приведет к выдаче на этапе компиляции сообщения об ошибке “Only assignment, call, increment, decrement, and new object expressions can be used as a statement” (“В качестве оператора могут применяться только выражения присваивания, вызова, инкремента, декремента и создания объекта”):

```
stringData.Length > 0
    ? Console.WriteLine("string is greater than 0 characters")
    : Console.WriteLine("string is not greater than 0 characters");
```

Логические операции

Для выполнения более сложных проверок оператор if может также включать сложные выражения и содержать операторы else. Синтаксис идентичен своим аналогам в C (C++) и Java. Язык C# предлагает вполне ожидаемый набор логических операций, предназначенных для построения сложных выражений (табл. 3.9).

Таблица 3.9. Условные операции C#

Операция	Пример	Описание
&&	if (age == 30 && name == "Fred")	Операция "И". Возвращает true, если все выражения дают true
	if (age == 30 name == "Fred")	Операция "ИЛИ". Возвращает true, если хотя бы одно из выражений дает true
!	if (!myBool)	Операция "НЕ". Возвращает true, если выражение дает false, или false, если выражение дает true

На заметку! Операции && и || при необходимости поддерживают сокращенный путь выполнения. Другими словами, после того, как было определено, что сложное выражение должно дать в результате false, оставшиеся подвыражения вычисляться не будут. Если требуется, чтобы все выражения вычислялись безотносительно к чему-либо, тогда можно использовать операции & и |.

Оператор switch

Еще одной простой конструкцией C# для реализации выбора является оператор `switch`. Как и в остальных основанных на C языках, оператор `switch` позволяет организовать выполнение программы на основе заранее определенного набора вариантов. Например, в следующем методе `Main()` для каждого из двух возможных вариантов выводится специфичное сообщение (блок `default` обрабатывает недопустимый выбор):

```
// Переход на основе числового значения.
static void SwitchExample()
{
    Console.WriteLine("1 [C#], 2 [VB]");
    Console.Write("Please pick your language preference: ");
    // Выберите предпочитаемый язык:
    string langChoice = Console.ReadLine();
    int n = int.Parse(langChoice);
    switch (n)
    {
        case 1:
            Console.WriteLine("Good choice, C# is a fine language.");
            // Хороший выбор. C# - замечательный язык.
            break;
        case 2:
            Console.WriteLine("VB: OOP, multithreading, and more!");
            // VB: ООП, многопоточность и многое другое!
            break;
        default:
            Console.WriteLine("Well...good luck with that!");
            // Хорошо, удачи с этим!
            break;
    }
}
```

На заметку! Язык C# требует, чтобы каждый блок `case` (включая `default`), который содержит исполняемые операторы, завершался оператором `return`, `break` или `goto` во избежание сквозного прохода по блокам.

Одна из замечательных особенностей оператора `switch` в C# связана с тем, что вдобавок к числовым значениям он позволяет оценивать данные `string`. На самом деле все версии C# способны оценивать типы данных `char`, `string`, `bool`, `int`, `long` и `enum`. В следующем разделе вы увидите, что в версии C# 7 появились дополнительные возможности. Вот модифицированная версия оператора `switch`, которая оценивает переменную типа `string`:

```
static void SwitchOnStringExample()
{
    Console.WriteLine("C# or VB");
    Console.Write("Please pick your language preference: ");
    string langChoice = Console.ReadLine();
    switch (langChoice)
    {
        case "C#":
            Console.WriteLine("Good choice, C# is a fine language.");
            break;
        case "VB":
            Console.WriteLine("VB: OOP, multithreading and more!");
            break;
    }
}
```

```

    default:
        Console.WriteLine("Well...good luck with that!");
        break;
    }
}

```

Оператор `switch` также может применяться с перечислимым типом данных. Как будет показано в главе 4, ключевое слово `enum` языка C# позволяет определять специальный набор пар “имя-значение”. В качестве иллюстрации рассмотрим вспомогательный метод `SwitchOnEnumExample()`, который выполняет проверку `switch` для перечисления `System.DayOfWeek`. Пример содержит ряд синтаксических конструкций, которые пока еще не рассматривались, но сосредоточьте внимание на самом использовании `switch` с типом `enum`; недостающие фрагменты будут прояснены в последующих главах.

```

static void SwitchOnEnumExample()
{
    Console.Write("Enter your favorite day of the week: ");
    // Введите любимый день недели:
    DayOfWeek favDay;
    try
    {
        favDay = (DayOfWeek) Enum.Parse(typeof(DayOfWeek), Console.ReadLine());
    }
    catch (Exception)
    {
        Console.WriteLine("Bad input!");
        // Недопустимое входное значение!
        return;
    }
    switch (favDay)
    {
        case DayOfWeek.Sunday:
            Console.WriteLine("Football!!");
            // Футбол!!
            break;
        case DayOfWeek.Monday:
            Console.WriteLine("Another day, another dollar.");
            // Еще один день, еще один доллар.
            break;
        case DayOfWeek.Tuesday:
            Console.WriteLine("At least it is not Monday.");
            // Во всяком случае, не понедельник.
            break;
        case DayOfWeek.Wednesday:
            Console.WriteLine("A fine day.");
            // Хороший денек.
            break;
        case DayOfWeek.Thursday:
            Console.WriteLine("Almost Friday...");
            // Почти пятница...
            break;
        case DayOfWeek.Friday:
            Console.WriteLine("Yes, Friday rules!");
            // Да, пятница рулит!
            break;
    }
}

```

```

    case DayOfWeek.Saturday:
        Console.WriteLine("Great day indeed.");
        // Действительно великолепный день.
        break;
    }
    Console.WriteLine();
}

```

Сквозной проход от одного оператора `case` к другому оператору `case` не разрешен, но что, если множество операторов `case` должны вырабатывать тот же самый результат? К счастью, их можно комбинировать, как демонстрируется ниже:

```

case DayOfWeek.Saturday:
case DayOfWeek.Sunday:
    Console.WriteLine("It's the weekend!");
    // Выходные!
    break;

```

Помещение любого кода между операторами `case` приведет к тому, что компилятор сообщит об ошибке. До тех пор, пока операторы `case` следуют друг за другом, как показано выше, их можно комбинировать для разделения общего кода.

В дополнение к операторам `return` и `break`, показанным в предшествующих примерах кода, оператор `switch` также поддерживает применение `goto` для выхода из условия `case` и выполнения другого оператора `case`. Несмотря на наличие поддержки, данный прием обычно считается антипаттерном и в общем случае не рекомендуется. Ниже приведен пример использования оператора `goto` в блоке `switch`:

```

public static void SwitchWithGoto()
{
    var foo = 5;
    switch (foo)
    {
        case 1:
            // Делать что-то
            goto case 2;
        case 2:
            // Делать что-то другое
            break;
        case 3:
            // Еще одно действие
            goto default;
        default:
            // Стандартное действие
            break;
    }
}

```

Использование сопоставления с образцом в операторах `switch` (нововведение)

До появления версии C# 7 сопоставляющие выражения в операторах `switch` ограничивались сравнением переменной с константными значениями, что иногда называют *образцом с константами*. В C# 7 операторы `switch` способны также задействовать *образец с типами*, при котором операторы `case` могут оценивать *тип* проверяемой переменной, и выражения `case` больше не ограничиваются константными значениями. Правило относительно того, что каждый оператор `case` должен завершаться с помощью `return` или `break`, по-прежнему остается в силе; тем не менее, операторы `goto` не поддерживают применение образца с типами.

На заметку! Если вы новичок в объектно-ориентированном программировании, тогда материал этого раздела может слегка сбивать с толку. Все прояснится в главе 6, когда мы вернемся к новым средствам сопоставления с образцом C# 7 в контексте базовых и производных классов. Пока вполне достаточно понимать, что появился мощный новый способ написания операторов `switch`.

Добавим еще один метод по имени `ExecutePatternMatchingSwitch()` со следующим кодом:

```
static void ExecutePatternMatchingSwitch()
{
    Console.WriteLine("1 [Integer (5)], 2 [String (\"Hi\")], 3 [Decimal (2.5)]");
    Console.Write("Please choose an option: ");
    string userChoice = Console.ReadLine();
    object choice;

    // Стандартный оператор switch, в котором применяется
    // сопоставление с образцом с константами
    switch (userChoice)
    {
        case "1":
            choice = 5;
            break;
        case "2":
            choice = "Hi";
            break;
        case "3":
            choice = 2.5;
            break;
        default:
            choice = 5;
            break;
    }

    // Новый оператор switch, в котором применяется
    // сопоставление с образцом с типами
    switch (choice)
    {
        case int i:
            Console.WriteLine("Your choice is an integer.");
            // Выбрано целое число
            break;
        case string s:
            Console.WriteLine("Your choice is a string.");
            // Выбрана строка
            break;
        case decimal d:
            Console.WriteLine("Your choice is a decimal.");
            // Выбрано десятичное число
            break;
        default:
            Console.WriteLine("Your choice is something else");
            // Выбрано что-то другое
            break;
    }
    Console.WriteLine();
}
```

В первом операторе `switch` используется стандартный образец с константами. Во втором операторе `switch` переменная типизируется как `object` и на основе пользовательского ввода может быть разобрана в тип данных `int`, `string` или `decimal`. В зависимости от типа переменной совпадения дают разные операторы `case`. Вдобавок к проверке типа данных в каждом операторе `case` выполняется присваивание переменной (кроме случая `default`). Модифицируем код, чтобы задействовать значения таких переменных:

```
// Новый оператор switch, в котором применяется
// сопоставление с образцом с типами
switch (choice)
{
    case int i:
        Console.WriteLine("Your choice is an integer {0}.", i);
        break;
    case string s:
        Console.WriteLine("Your choice is a string {0}.", s);
        break;
    case decimal d:
        Console.WriteLine("Your choice is a decimal {0}.", d);
        break;
    default:
        Console.WriteLine("Your choice is something else.");
        break;
}
```

Кроме оценки типа сопоставляющего выражения к операторам `case` могут быть добавлены конструкции `when` для оценки условий на переменной. В представленном ниже примере в дополнение к проверке типа производится проверка на совпадение преобразованного типа:

```
static void ExecutePatternMatchingSwitchWithWhen()
{
    Console.WriteLine("1 [C#], 2 [VB]");
    Console.Write("Please pick your language preference: ");
    object langChoice = Console.ReadLine();
    var choice = int.TryParse(langChoice.ToString(), out int c) ? c : langChoice;
    switch (choice)
    {
        case int i when i == 2:
        case string s when s.Equals("VB", StringComparison.OrdinalIgnoreCase):
            Console.WriteLine("VB: OOP, multithreading, and more!");
            // VB: ООП, многопоточность и многое другое!

            break;
        case int i when i == 1:
        case string s when s.Equals("C#", StringComparison.OrdinalIgnoreCase):
            Console.WriteLine("Good choice, C# is a fine language.");
            // Хороший выбор. C# - замечательный язык.

            break;
        default:
            Console.WriteLine("Well...good luck with that!");
            // Хорошо, удачи с этим!

            break;
    }
    Console.WriteLine();
}
```

Здесь к оператору `switch` добавляется новое измерение, поскольку порядок следования операторов `case` теперь важен. При использовании образца с константами каждый оператор `case` обязан быть уникальным. В случае применения образца с типами это больше не так. Например, следующий код будет давать совпадение для каждого целого числа в первом операторе `case`, а второй и третий оператор `case` никогда не выполнятся (на самом деле такой код даже не скомпилируется):

```
switch (choice)
{
    case int 1:
        // Делать что-то
        break;
    case int i when i == 0:
        // Делать что-то
        break;
    case int i when i == -1:
        // Делать что-то
        break;
}
```

В первоначальном выпуске C# 7 возникало небольшое затруднение при сопоставлении с образцом, когда в нем использовались обобщенные типы. В версии C# 7.1 проблема была устранена.

Исходный код. Проект `IterationsAndDecisions` доступен в подкаталоге `Chapter_3`.

Резюме

Цель настоящей главы заключалась в демонстрации многочисленных ключевых аспектов языка программирования C#. Мы исследовали привычные конструкции, которые могут быть задействованы при построении любого приложения. После ознакомления с ролью объекта приложения вы узнали о том, что каждая исполняемая программа на C# должна иметь тип, определяющий метод `Main()`, который служит точкой входа в программу. Внутри метода `Main()` обычно создается любое число объектов, благодаря совместной работе которых приложение приводится в действие.

Затем были подробно описаны встроенные типы данных C# и разъяснено, что применяемые для их представления ключевые слова (например, `int`) на самом деле являются сокращенными обозначениями полноценных типов из пространства имен `System` (`System.Int32` в данном случае). С учетом этого каждый тип данных C# имеет набор встроенных членов. Кроме того, обсуждалась роль расширения и сужения, а также ключевых слов `checked` и `unchecked`.

В завершение главы рассматривалась роль неявной типизации с использованием ключевого слова `var`. Как было отмечено, неявная типизация наиболее полезна при работе с моделью программирования LINQ. Наконец, мы кратко взглянули на различные конструкции C#, предназначенные для организации циклов и принятия решений.

Теперь, когда вы понимаете некоторые базовые механизмы, в главе 4 будет завершено исследование основных средств языка. После этого вы будете хорошо подготовлены к изучению объектно-ориентированных возможностей C#, которое начнется в главе 5.

ГЛАВА 4

Главные конструкции программирования на С#: часть II

В настоящей главе завершается обзор основных аспектов языка программирования С#, который был начат в главе 3. Первым делом мы рассмотрим детали манипулирования массивами с использованием синтаксиса С# и продемонстрируем функциональность, содержащуюся внутри связанного класса `System.Array`.

Далее мы выясним различные подробности, касающиеся построения методов, за счет исследования ключевых слов `out`, `ref` и `params`. В ходе дела мы объясним роль необязательных и именованных параметров. Обсуждение темы методов завершится *перегрузкой методов*.

Затем будет показано, как создавать типы перечислений и структур, включая детальное исследование отличий между *типами значений* и *ссылочными типами*. В конце главы объясняется роль типов данных, допускающих `null`, и связанных с ними операций.

После освоения материалов главы вы можете смело переходить к изучению объектно-ориентированных возможностей языка С#, рассмотрение которых начнется в главе 5.

Понятие массивов С#

Как вам уже наверняка известно, *массив* — это набор элементов данных, для доступа к которым применяется числовой индекс. Выражаясь более конкретно, массив представляет собой набор расположенных рядом элементов данных одного и того же типа (массив элементов `int`, массив элементов `string`, массив элементов `SportCar` и т.д.). Объявлять, заполнять и получать доступ к массиву в языке С# довольно просто. В целях иллюстрации создадим новый проект консольного приложения (по имени `FunWithArrays`), который содержит вспомогательный метод `SimpleArrays()`, вызываемый из `Main()`:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Arrays *****");
        SimpleArrays();
        Console.ReadLine();
    }
}
```

```
static void SimpleArrays()
{
    Console.WriteLine("> Simple Array Creation.");
    // Создать массив int, содержащий 3 элемента с индексами 0, 1, 2.
    int[] myInts = new int[3];
    // Создать массив string, содержащий 100 элементов с индексами 0 - 99.
    string[] booksOnDotNet = new string[100];
    Console.WriteLine();
}
}
```

Внимательно взгляните на комментарии в коде. При объявлении массива C# с использованием подобного синтаксиса число, указанное в объявлении, обозначает общее количество элементов, а не верхнюю границу. Кроме того, нижняя граница в массиве всегда начинается с 0. Таким образом, в результате записи `int[] myInts = new int[3]` получается массив, который содержит три элемента, проиндексированные по позициям 0, 1, 2.

После определения переменной массива можно переходить к заполнению элементов от индекса к индексу, как показано ниже в модифицированном методе `SimpleArrays()`:

```
static void SimpleArrays()
{
    Console.WriteLine("> Simple Array Creation.");
    // Создать и заполнить массив из трех целочисленных значений.
    int[] myInts = new int[3];
    myInts[0] = 100;
    myInts[1] = 200;
    myInts[2] = 300;
    // Вывести все значения.
    foreach(int i in myInts)
        Console.WriteLine(i);
    Console.WriteLine();
}
```

На заметку! Имейте в виду, что если массив объявлен, но его элементы явно не заполнены по каждому индексу, то они получают стандартное значение для соответствующего типа данных (например, элементы массива `bool` будут установлены в `false`, а элементы массива `int` — в 0).

Синтаксис инициализации массивов C#

В дополнение к заполнению массива элемент за элементом есть также возможность заполнять его с применением *синтаксиса инициализации массива*. Для этого понадобится указать значения всех элементов массива в фигурных скобках `{ }`. Такой синтаксис удобен при создании массива известного размера, когда нужно быстро задать его начальные значения. Например, вот как выглядят альтернативные версии объявления массива:

```
static void ArrayInitialization()
{
    Console.WriteLine("> Array Initialization.");
    // Синтаксис инициализации массива с использованием ключевого слова new.
    string[] stringArray = new string[]
    { "one", "two", "three" };
    Console.WriteLine("stringArray has {0} elements", stringArray.Length);
}
```

```
// Синтаксис инициализации массива без использования ключевого слова new.
bool[] boolArray = { false, false, true };
Console.WriteLine("boolArray has {0} elements", boolArray.Length);
// Инициализация массива с применением ключевого слова new и указанием размера.
int[] intArray = new int[4] { 20, 22, 23, 0 };
Console.WriteLine("intArray has {0} elements", intArray.Length);
Console.WriteLine();
}
```

Обратите внимание, что в случае использования синтаксиса с фигурными скобками нет необходимости указывать размер массива (как видно на примере создания переменной `stringArray`), поскольку размер автоматически вычисляется на основе количества элементов внутри фигурных скобок. Кроме того, применять ключевое слово `new` не обязательно (как при создании массива `boolArray`).

В случае объявления `intArray` снова вспомните, что указанное числовое значение представляет количество элементов в массиве, а не верхнюю границу. Если объявленный размер и количество инициализаторов не совпадают (инициализаторов слишком много или не хватает), тогда на этапе компиляции возникнет ошибка. Пример представлен ниже:

```
// Несоответствие размера и количества элементов!
int[] intArray = new int[2] { 20, 22, 23, 0 };
```

Неявно типизированные локальные массивы

В главе 3 рассматривалась тема неявно типизированных локальных переменных. Как вы помните, ключевое слово `var` позволяет определять переменную, тип которой выводится компилятором. Аналогичным образом ключевое слово `var` можно использовать для определения *неявно типизированных локальных массивов*. Такой подход позволяет выделять память под новую переменную массива, не указывая тип элементов внутри массива (обратите внимание, что применение этого подхода предусматривает обязательное использование ключевого слова `new`):

```
static void DeclareImplicitArrays()
{
    Console.WriteLine("=> Implicit Array Initialization.");
    // Переменная a на самом деле имеет тип int[].
    var a = new[] { 1, 10, 100, 1000 };
    Console.WriteLine("a is a: {0}", a.ToString());
    // Переменная b на самом деле имеет тип double[].
    var b = new[] { 1, 1.5, 2, 2.5 };
    Console.WriteLine("b is a: {0}", b.ToString());
    // Переменная c на самом деле имеет тип string[].
    var c = new[] { "hello", null, "world" };
    Console.WriteLine("c is a: {0}", c.ToString());
    Console.WriteLine();
}
```

Разумеется, как и при создании массива с применением явного синтаксиса C#, элементы в списке инициализации массива должны принадлежать одному и тому же типу (например, должны быть все `int`, все `string` или все `SportsCar`). В отличие от возможных ожиданий, неявно типизированный локальный массив не получает по умолчанию тип `System.Object`, так что следующий код приведет к ошибке на этапе компиляции:

```
// Ошибка! Смешанные типы!
var d = new[] { 1, "one", 2, "two", false };
```

Определение массива объектов

В большинстве случаев массив определяется путем указания явного типа элементов, которые могут в нем содержаться. Хотя это выглядит довольно прямолинейным, существует одна важная особенность. Как будет показано в главе 6, изначальным базовым классом для каждого типа (включая фундаментальные типы данных) в системе типов .NET является `System.Object`. С учетом такого факта, если определить массив типа данных `System.Object`, то его элементы могут представлять все что угодно. Взгляните на следующий метод `ArrayOfObjects()`, который в целях тестирования может быть вызван внутри `Main()`:

```
static void ArrayOfObjects()
{
    Console.WriteLine("=> Array of Objects.");
    // Массив объектов может содержать все что угодно.
    object[] myObjects = new object[4];
    myObjects[0] = 10;
    myObjects[1] = false;
    myObjects[2] = new DateTime(1969, 3, 24);
    myObjects[3] = "Form & Void";
    foreach (object obj in myObjects)
    {
        // Вывести тип и значение каждого элемента в массиве.
        Console.WriteLine("Type: {0}, Value: {1}", obj.GetType(), obj);
    }
    Console.WriteLine();
}
```

Здесь во время прохода по содержимому массива `myObjects` для каждого элемента выводится лежащий в основе тип, получаемый с помощью метода `GetType()` класса `System.Object`, и его значение. Не вдаваясь пока в детали метода `System.Object.GetType()`, просто отметим, что он может использоваться для получения полностью заданного имени элемента (службы извлечения информации о типах и рефлексии исследуются в главе 15). Приведенный далее вывод является результатом вызова метода `ArrayOfObjects()`:

```
=> Array of Objects.
Type: System.Int32, Value: 10
Type: System.Boolean, Value: False
Type: System.DateTime, Value: 3/24/1969 12:00:00 AM
Type: System.String, Value: Form & Void
```

Работа с многомерными массивами

В дополнение к одномерным массивам, которые вы видели до сих пор, язык C# также поддерживает два вида многомерных массивов. Первый вид называется *прямоугольным массивом*, который имеет несколько измерений, а содержащиеся в нем строки обладают одной и той же длиной. Прямоугольный многомерный массив объявляется и заполняется следующим образом:

```
static void RectMultidimensionalArray()
{
    Console.WriteLine("=> Rectangular multidimensional array.");
    // Прямоугольный многомерный массив.
    int[,] myMatrix;
    myMatrix = new int[3,4];
}
```

```
// Заполнить массив (3 * 4).
for(int i = 0; i < 3; i++)
    for(int j = 0; j < 4; j++)
        myMatrix[i, j] = 1 * j;
// Вывести содержимое массива (3 * 4).
for(int i = 0; i < 3; i++)
{
    for(int j = 0; j < 4; j++)
        Console.Write(myMatrix[i, j] + "\t");
    Console.WriteLine();
}
Console.WriteLine();
}
```

Второй вид многомерных массивов носит название *зубчатого* (или *ступенчатого*) массива. Такой массив содержит какое-то число внутренних массивов, каждый из которых может иметь отличающийся верхний предел. Вот пример:

```
static void JaggedMultidimensionalArray()
{
    Console.WriteLine("=> Jagged multidimensional array.");
    // Зубчатый многомерный массив (т.е. массив массивов).
    // Здесь мы имеем массив из 5 разных массивов.
    int[][] myJagArray = new int[5][];
    // Создать зубчатый массив.
    for (int i = 0; i < myJagArray.Length; i++)
        myJagArray[i] = new int[i + 7];
    // Вывести все строки (помните, что каждый элемент имеет стандартное значение 0).
    for(int i = 0; i < 5; i++)
    {
        for(int j = 0; j < myJagArray[i].Length; j++)
            Console.Write(myJagArray[i][j] + " ");
        Console.WriteLine();
    }
    Console.WriteLine();
}
```

Ниже показан вывод, полученный в результате вызова внутри Main() методов RectMultidimensionalArray() и JaggedMultidimensionalArray():

=> Rectangular multidimensional array:

```
0      0      0      0
0      1      2      3
0      2      4      6
```

=> Jagged multidimensional array:

```
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Использование массивов в качестве аргументов и возвращаемых значений

После создания массив можно передавать как аргумент или получать его в виде возвращаемого значения. Например, приведенный ниже метод PrintArray() принимает входной

массив значений `int` и выводит все его элементы на консоль, а метод `GetStringArray()` заполняет массив значений `string` и возвращает его вызывающему коду:

```
static void PrintArray(int[] myInts)
{
    for(int i = 0; i < myInts.Length; i++)
        Console.WriteLine("Item {0} is {1}", i, myInts[i]);
}
static string[] GetStringArray()
{
    string[] theStrings = {"Hello", "from", "GetStringArray"};
    return theStrings;
}
```

Указанные методы могут вызываться вполне ожидаемым образом:

```
static void PassAndReceiveArrays()
{
    Console.WriteLine("> Arrays as params and return values.");
    // Передать массив в качестве параметра.
    int[] ages = {20, 22, 23, 0};
    PrintArray(ages);
    // Получить массив как возвращаемое значение.
    string[] strs = GetStringArray();
    foreach(string s in strs)
        Console.WriteLine(s);
    Console.WriteLine();
}
```

К настоящему моменту вы должны освоить процесс определения, заполнения и исследования содержимого переменной типа массива C#. Для полноты картины давайте проанализируем роль класса `System.Array`.

Базовый класс `System.Array`

Каждый создаваемый массив получает значительную часть своей функциональности от класса `System.Array`. Общие члены этого класса позволяют работать с массивом, применяя согласованную объектную модель. В табл. 4.1 приведено краткое описание наиболее интересных членов класса `System.Array` (полное описание всех его членов можно найти в документации .NET Framework 4.7 SDK).

Таблица 4.1. Избранные члены класса `System.Array`

Член класса <code>System.Array</code>	Описание
<code>Clear()</code>	Этот статический метод устанавливает для заданного диапазона элементов в массиве пустые значения (0 для чисел, <code>null</code> для объектных ссылок и <code>false</code> для булевских значений)
<code>CopyTo()</code>	Этот метод используется для копирования элементов из исходного массива в целевой массив
<code>Length</code>	Это свойство возвращает количество элементов в массиве
<code>Rank</code>	Это свойство возвращает количество измерений в массиве
<code>Reverse()</code>	Этот статический метод обращает содержимое одномерного массива
<code>Sort()</code>	Этот статический метод сортирует одномерный массив внутренних типов. Если элементы в массиве реализуют интерфейс <code>IComparer</code> , то можно сортировать также и специальные типы (глава 9)

Давайте посмотрим на некоторые из членов в действии. Показанный далее вспомогательный метод использует статические методы `Reverse()` и `Clear()` для вывода на консоль информации о массиве строковых типов:

```
static void SystemArrayFunctionality()
{
    Console.WriteLine("=> Working with System.Array.");
    // Инициализировать элементы при запуске.
    string[] gothicBands = {"Tones on Tail", "Bauhaus", "Sisters of Mercy"};
    // Вывести имена в порядке их объявления.
    Console.WriteLine("-> Here is the array:");
    for (int i = 0; i < gothicBands.Length; i++)
    {
        // Вывести имя.
        Console.Write(gothicBands[i] + ", ");
    }
    Console.WriteLine("\n");
    // Обратить порядок следования элементов...
    Array.Reverse(gothicBands);
    Console.WriteLine("-> The reversed array");
    // ... и вывести их.
    for (int i = 0; i < gothicBands.Length; i++)
    {
        // Вывести имя.
        Console.Write(gothicBands[i] + ", ");
    }
    Console.WriteLine("\n");
    // Удалить все элементы кроме первого.
    Console.WriteLine("-> Cleared out all but one...");
    Array.Clear(gothicBands, 1, 2);
    for (int i = 0; i < gothicBands.Length; i++)
    {
        // Вывести имя.
        Console.Write(gothicBands[i] + ", ");
    }
    Console.WriteLine();
}
```

Вызов метода `SystemArrayFunctionality()` в `Main()` дает в результате следующий вывод:

```
=> Working with System.Array.
-> Here is the array:
Tones on Tail, Bauhaus, Sisters of Mercy,
-> The reversed array
Sisters of Mercy, Bauhaus, Tones on Tail,
-> Cleared out all but one...
Sisters of Mercy, , ,
```

Обратите внимание, что многие члены класса `System.Array` определены как статические и потому вызываются на уровне класса (примерами могут служить методы `Array.Sort()` и `Array.Reverse()`). Методам подобного рода передается массив, подлежащий обработке. Другие члены `System.Array` (такие как свойство `Length`) действуют на уровне объекта, поэтому могут вызываться прямо на типе массива.

Методы и модификаторы параметров

Для начала мы займемся исследованием деталей определения методов. Подобно методу `Main()` (см. главу 3) ваши специальные методы могут принимать или не принимать параметры и возвращать или не возвращать значения вызывающему коду. В последующих нескольких главах вы увидите, что методы могут быть реализованы внутри области видимости классов или структур (и заодно прототипироваться внутри интерфейсных типов), а также декорированы разнообразными ключевыми словами (например, `static`, `virtual`, `public`, `new`) с целью уточнения их поведения. До настоящего момента в книге каждый из рассматриваемых методов следовал такому базовому формату:

```
// Вспомните, что статические методы могут вызываться
// напрямую без создания экземпляра класса.
class Program
{
    // static возвращаемыйТип ИмяМетода(список параметров) { /* Реализация */
}
    static int Add(int x, int y){ return x + y; }
}
```

Возвращаемые значения и члены, сжатые до выражений (обновление)

Вы уже знаете о простых методах, возвращающих значения. вроде метода `Add()`. В версии C# 6 появились члены, сжатые до выражений, которые сокращают синтаксис написания однострочных методов. Например, метод `Add()` можно переписать следующим образом:

```
static int Add(int x, int y) => x + y;
```

Обычно такой прием называют “синтаксическим сахаром”, имея в виду, что генерируемый код IL не изменяется по сравнению с первоначальной версией метода. Это просто другой способ написания метода. Одни находят его более легким для восприятия, другие — нет, так что выбор стиля зависит от ваших персональных предпочтений (или предпочтений команды).

Данный синтаксис также применим к свойствам, предназначенным только для чтения (классы и свойства рассматриваются в главе 5).

В версии C# 7 возможность сжатия до выражений была расширена с целью охвата однострочных конструкторов, финализаторов, а также средств доступа `get` и `set` для свойств и индексаторов (все они будут подробно раскрыты в книге, начиная с главы 5). Повсюду в книге вы будете встречать смесь использования членов, сжатых до выражений, и традиционного подхода.

На заметку! Не пугайтесь операции `=>`. Это лямбда-операция, которая подробно рассматривается в главе 10, где также объясняется, *каким образом* работают члены, сжатые до выражений. Пока просто считайте их сокращением при написании однострочных операторов.

Модификаторы параметров для методов

Стандартный способ передачи параметра в функцию — *по значению*. Попросту говоря, если вы не помечаете аргумент каким-то модификатором параметра, тогда в функцию передается копия данных. Как объясняется далее в главе, то, что в точности копируется, будет зависеть от того, относится параметр к типу значения или к ссылочному типу.

Хотя определение метода в C# выглядит достаточно понятно, с помощью модификаторов, описанных в табл. 4.2, можно управлять способом передачи аргументов интересующему методу.

Таблица 4.2. Модификаторы параметров в C#

Модификатор параметра	Практический смысл
(отсутствует)	Если параметр не помечен модификатором, то предполагается, что он должен передаваться по значению, т.е. вызываемый метод получает копию исходных данных
out	Выходным параметрам должны присваиваться значения внутри вызываемого метода, следовательно, они передаются по ссылке. Если в вызываемом методе выходным параметрам не были присвоены значения, тогда компилятор сообщит об ошибке
ref	Значение первоначально присваивается в вызывающем коде и может быть необязательно изменено в вызываемом методе (поскольку данные также передаются по ссылке). Если в вызываемом методе параметру <code>ref</code> значение не присваивалось, то никакой ошибки компилятор не генерирует
params	Этот модификатор позволяет передавать переменное количество аргументов как единственный логический параметр. Метод может иметь только один модификатор <code>params</code> , которым должен быть помечен последний параметр метода. В реальности потребность в использовании модификатора <code>params</code> возникает не особенно часто, однако имейте в виду, что он применяется многочисленными методами внутри библиотек базовых классов

Чтобы проиллюстрировать использование перечисленных модификаторов, мы создадим новый проект консольного приложения по имени `FunWithMethods`. А теперь давайте рассмотрим их роль.

Отбрасывание

Отбрасывание касается временных, фиктивных переменных, которые намеренно не используются. Значения им не присваивались, и возможно для них даже не выделялась память. Отбрасывание способно обеспечить выигрыш в производительности, но самое меньшее оно может улучшить читабельность кода. Отбрасывание может применяться с параметрами `out`, кортежами, сопоставлением с образцом (главы 6 и 8) и даже автономными переменными.

Вероятно, вас интересует, для чего может понадобиться присваивание значения отбрасываемой переменной. Как будет показано в главе 19, это становится удобным при асинхронном программировании.

Стандартное поведение передачи параметров по значению

По умолчанию параметр передается функции *по значению*. Другими словами, если аргумент не помечен модификатором параметра, то в функцию передается копия данных. Как объясняется в конце главы, что именно копируется, зависит от того, относится ли параметр к типу значения или же к ссылочному типу. В настоящий момент предположим, что внутри класса `Program` есть представленный далее метод, который оперирует с двумя параметрами числового типа, передаваемыми по значению:

```
// По умолчанию аргументы передаются по значению.
static int Add(int x, int y)
{
    int ans = x + y;
    // Вызывающий код не увидит эти изменения,
    // т.к. модифицируется копия исходных данных.
    x = 10000;
    y = 88888;
    return ans;
}
```

Числовые данные относятся к категории *типов значений*. Следовательно, в случае изменения значений параметров внутри контекста члена вызывающий код будет оставаться в полном неведении об этом, поскольку изменения вносятся только в копию первоначальных данных из вызывающего кода:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****\n");
    // Передать две переменные по значению.
    int x = 9, y = 10;
    Console.WriteLine("Before call: X: {0}, Y: {1}", x, y);
    Console.WriteLine("Answer is: {0}", Add(x, y));
    Console.WriteLine("After call: X: {0}, Y: {1}", x, y);
    Console.ReadLine();
}
```

Как и следовало ожидать, к тому же данный факт подтверждается показанным далее выводом, значения *x* и *y* остаются идентичными до и после вызова метода *Add()*, потому что элементы данных передавались по значению. Таким образом, любые изменения параметров, производимые внутри метода *Add()*, вызывающему коду не видны, т.к. метод *Add()* оперирует на копии данных.

```
***** Fun with Methods *****
Before call: X: 9, Y: 10
Answer is: 19
After call: X: 9, Y: 10
```

Модификатор *out* (обновление)

Теперь мы рассмотрим *выходные параметры*. Метод, который был определен для приема выходных параметров (посредством ключевого слова *out*), перед выходом обязан присваивать им соответствующие значения (иначе компилятор сообщит об ошибке).

В целях демонстрации ниже приведена альтернативная версия метода *Add()*, которая возвращает сумму двух целых чисел с применением модификатора *out* (обратите внимание, что возвращаемым значением метода теперь является *void*):

```
// Значения выходных параметров должны быть установлены
// внутри вызываемого метода.
static void Add(int x, int y, out int ans)
{
    ans = x + y;
}
```

Вызов метода с выходными параметрами также требует использования модификатора *out*. Однако предварительно устанавливать значения локальных переменных, которые передаются в качестве выходных параметров, вовсе не обязательно (после вы-

зова эти значения все равно будут утеряны). Причина, по которой компилятор позволяет передавать на первый взгляд неинициализированные данные, связана с тем, что вызываемый метод обязан выполнить присваивание. Начиная с версии C# 7, больше нет нужды объявлять параметры `out` до их применения. Другими словами, они могут объявляться внутри вызова метода:

```
Add(90, 90, out int ans);
```

В следующем коде представлен пример вызова метода с встраиваемым объявлением параметра `out`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
    // Присваивать начальные значения локальным переменным, используемым
    // как выходные параметры, не обязательно при условии, что они
    // впервые используются впервые в таком качестве.
    // Версия C# 7 позволяет объявлять параметры out в вызове метода.
    Add(90, 90, out int ans);
    int ans;
    Add(90, 90, out ans);
    Console.WriteLine("90 + 90 = {0}", ans);
    Console.ReadLine();
}
```

Предыдущий пример по своей природе предназначен только для иллюстрации; на самом деле нет никаких причин возвращать значение суммы через выходной параметр. Тем не менее, модификатор `out` в C# служит действительно практичной цели: он позволяет вызывающему коду получать несколько выходных значений из единственного вызова метода:

```
// Возвращение множества выходных параметров.
static void FillTheseValues(out int a, out string b, out bool c)
{
    a = 9;
    b = "Enjoy your string.";
    c = true;
}
```

Теперь вызывающий код имеет возможность обращаться к методу `FillTheseValues()`. Не забывайте, что модификатор `out` должен применяться как при вызове, так и при реализации метода:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
    int i; string str; bool b;
    FillTheseValues(out i, out str, out b);

    Console.WriteLine("Int is: {0}", i);
    Console.WriteLine("String is: {0}", str);
    Console.WriteLine("Boolean is: {0}", b);
    Console.ReadLine();
}
```

На заметку! В версии C# 7 появились кортежи, представляющие собой еще один способ возвращения множества значений из вызова метода. Они будут описаны далее в главе.

Всегда помните о том, что перед выходом из области видимости метода, определяющего выходные параметры, этим параметрам *должны* быть присвоены допустимые значения. Таким образом, следующий код вызовет ошибку на этапе компиляции, потому что внутри метода отсутствует присваивание значения выходному параметру:

```
static void ThisWontCompile(out int a)
{
    Console.WriteLine("Error! Forgot to assign output arg!");
    // Ошибка! Забыли присвоить значение выходному параметру!
}
```

Наконец, если значение параметра `out` не интересует, тогда в качестве заполнителя можно использовать отбрасывание. Например, когда нужно выяснить, имеет ли строка допустимый формат даты, но сама разобранный дата не требуется, можно было бы написать такой код:

```
if (DateTime.TryParse(dateString, out _))
{
    // Делать что-то
}
```

Модификатор `ref`

А теперь посмотрим, как в C# используется модификатор `ref`. Ссылочные параметры необходимы, когда вы хотите разрешить методу манипулировать различными элементами данных (и обычно изменять их значения), которые объявлены в вызывающем коде, таком как процедура сортировки или обмена. Обратите внимание на отличия между ссылочными и выходными параметрами.

- Выходные параметры не нуждаются в инициализации перед передачей методу. Причина в том, что метод до своего завершения обязан самостоятельно присваивать значения выходным параметрам.
- Ссылочные параметры должны быть инициализированы перед передачей методу. Причина связана с передачей ссылок на существующие переменные. Если начальные значения им не присвоены, то это будет равнозначно работе с неинициализированными локальными переменными.

Давайте рассмотрим применение ключевого слова `ref` на примере метода, меняющего местами значения двух переменных типа `string` (естественно, здесь мог бы использоваться любой тип данных, включая `int`, `bool`, `float` и т.д.):

```
// Ссылочные параметры.
public static void SwapStrings(ref string s1, ref string s2)
{
    string tempStr = s1;
    s1 = s2;
    s2 = tempStr;
}
```

Метод `SwapStrings()` можно вызвать следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
    string str1 = "Flip";
    string str2 = "Flop";
    Console.WriteLine("Before: {0}, {1} ", str1, str2);
```

```
SwapStrings(ref str1, ref str2);
Console.WriteLine("After: {0}, {1} ", str1, str2);
Console.ReadLine();
}
```

Здесь вызывающий код присваивает начальные значения локальным строковым данным (str1 и str2). После вызова метода SwapStrings() строка str1 будет содержать значение "Flop", а строка str2 — значение "Flip":

```
Before: Flip, Flop
After: Flop, Flip
```

На заметку! Мы еще возвратимся к ключевому слову модификатора параметра `ref` языка C# в разделе "Типы значений и ссылочные типы" далее в главе. Вы увидите, что поведение ключевого слова `ref` немного изменяется в зависимости от того, является аргумент типом значения или ссылочным типом.

Ссылочные локальные переменные и возвращаемые ссылочные значения (нововведения)

В дополнение к модифицированию параметров с помощью ключевого слова `ref` в версии C# 7 появилась возможность применения и возвращения ссылок на переменные, определенные где угодно. Прежде чем мы покажем, как это работает, рассмотрим приведенный ниже метод:

```
// Возвращает значение по позиции в массиве.
public static string SimpleReturn(string[] strArray, int position)
{
    return strArray[position];
}
```

Методу SimpleReturn() передается массив строк (по значению) и позиция. Затем возвращается значение из указанной позиции в массиве. Если возвращаемая из метода строка модифицируется за пределами метода, то вполне нормально ожидать, что массив по-прежнему содержит исходные значения. Как демонстрируется в следующем коде, именно это и происходит:

```
#region Ref locals and params
string[] stringArray = { "one", "two", "three" };
int pos = 1;
Console.WriteLine("=> Use Simple Return");
Console.WriteLine("Before: {0}, {1}, {2} ", stringArray[0],
                stringArray[1], stringArray[2]);
var output = SimpleReturn(stringArray, pos);
output = "new";
Console.WriteLine("After: {0}, {1}, {2} ", stringArray[0],
                stringArray[1], stringArray[2]);
#endregion
```

Вот результирующий вывод:

```
=> Use Simple Return
Before: one, two, three
After: one, two, three
```

А что, если вместо значения в позиции массива интересует ссылка на позицию в массиве? Задачу определенно можно было решить и до выхода версии C# 7, но новые возможности использования ключевого слова `ref` делают решение гораздо проще.

В простой метод понадобится внести два изменения. Во-первых, взамен `return` [возвращаемое значение] метод должен делать `return ref` [возвращаемая ссылка]. Во-вторых, объявление метода также обязано включать ключевое слово `ref`. Создадим новый метод по имени `SampleRefReturn()`:

```
// Возвращение ссылки.
public static ref string SampleRefReturn(string[] strArray, int position)
{
    return ref strArray[position];
}
```

По существу это тот же самый метод, что и ранее, с добавлением в двух местах ключевого слова `ref`. Теперь он возвращает ссылку на позицию в массиве, а не хранящееся в ней значение. Вызов метода также требует применения ключевого слова `ref` — для возвращаемой переменной и для самого вызова метода:

```
ref var refOutput = ref SampleRefReturn(stringArray, pos);
```

Любые изменения в отношении возвращенной ссылки приведут также к обновлению массива, как демонстрируется в следующем коде:

```
#region Ссылочные локальные переменные и возвращаемые ссылочные значения
Console.WriteLine("> Use Ref Return");
Console.WriteLine("Before: {0}, {1}, {2} ", stringArray[0],
    stringArray[1], stringArray[2]);
ref var refOutput = ref SampleRefReturn(stringArray, pos);
refOutput = "new";
Console.WriteLine("After: {0}, {1}, {2} ", stringArray[0],
    stringArray[1], stringArray[2]);
```

Вывод значений массива на консоль отражает результат изменения значения ссылочной переменной, которая возвращается из нового метода:

```
=> Use Ref Return
Before: one, two, three
After: one, new, three
```

С новым средством связано несколько правил, которые полезно здесь отметить.

- Результаты стандартного метода не могут присваиваться локальной переменной `ref`. Метод должен быть создан как возвращающий ссылочное значение.
- Локальную переменную внутри метода `ref` нельзя возвращать как локальную переменную `ref`. Следующий код работать не будет:

```
ThisWillNotWork(string[] array)
{
    int foo = 5;
    return ref foo;
}
```

- Новое средство не работает с асинхронными методами (см. главу 19).

Модификатор `params`

В языке C# поддерживаются массивы параметров с использованием ключевого слова `params`, которое позволяет передавать методу переменное количество идентично типизированных параметров (или классов, связанных отношением наследования) в виде единственного логического параметра. Вдобавок аргументы, помеченные ключевым словом `params`, могут обрабатываться, когда вызывающий код передает строго типизированный массив или список элементов, разделенных запятыми. Да, это может сбивать

с толку! В целях прояснения предположим, что вы хотите создать функцию, которая позволяет вызывающему коду передавать любое количество аргументов и возвращает их среднее значение.

Если вы прототипируете данный метод так, чтобы он принимал массив значений `double`, тогда в вызывающем коде придется сначала определить массив, затем заполнить его значениями и, наконец, передать его методу. Однако если вы определите метод `CalculateAverage()` как принимающий параметр `params` типа `double[]`, то вызывающий код может просто передавать список значений `double`, разделенных запятыми. "За кулисами" исполняющая среда .NET автоматически упакует набор значений `double` в массив типа `double`.

```
// Возвращение среднего из некоторого количества значений double.
static double CalculateAverage(params double[] values)
{
    Console.WriteLine("You sent me {0} doubles.", values.Length);
    double sum = 0;
    if(values.Length == 0)
        return sum;
    for (int i = 0; i < values.Length; i++)
        sum += values[i];
    return (sum / values.Length);
}
```

Метод `CalculateAverage()` был определен для приема массива параметров типа `double`. Фактически он ожидает передачи любого количества (включая ноль) значений `double` и вычисляет их среднее. Метод может вызываться любым из показанных далее способов:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
    // Передать список значений double, разделенных запятыми...
    double average;
    average = CalculateAverage(4.0, 3.2, 5.7, 64.22, 87.2);
    Console.WriteLine("Average of data is: {0}", average);

    // ... или передать массив значений double.
    double[] data = { 4.0, 3.2, 5.7 };
    average = CalculateAverage(data);
    Console.WriteLine("Average of data is: {0}", average);

    // Среднее из 0 равно 0!
    Console.WriteLine("Average of data is: {0}", CalculateAverage());
    Console.ReadLine();
}
```

Если модификатор `params` в определении метода `CalculateAverage()` не задействован, тогда его первый вызов приведет к ошибке на этапе компиляции, т.к. компилятору не удастся найти версию `CalculateAverage()`, принимающую пять аргументов типа `double`.

На заметку! Во избежание любой неоднозначности язык C# требует, чтобы метод поддерживал только один параметр `params`, который должен быть последним в списке параметров.

Как и можно было догадаться, данный прием — всего лишь удобство для вызывающего кода, потому что среда CLR самостоятельно создает массив по мере необходимости.

ти. В момент, когда массив окажется внутри области видимости вызываемого метода, его можно трактовать как полноценный массив .NET, обладающий всей функциональностью базового библиотечного класса `System.Array`. Взгляните на вывод:

```
You sent me 5 doubles.
Average of data is: 32.864
You sent me 3 doubles.
Average of data is: 4.3
You sent me 0 doubles.
Average of data is: 0
```

Определение необязательных параметров

Язык C# дает возможность создавать методы, которые могут принимать *необязательные аргументы*. Такой прием позволяет вызывать метод, опуская ненужные аргументы, при условии, что подходят указанные для них стандартные значения.

На заметку! В главе 16 вы увидите, что главной побудительной причиной для добавления в язык C# необязательных аргументов послужило стремление упростить взаимодействие с объектами COM. Некоторые объектные модели Microsoft (например, Microsoft Office) открывают доступ к своей функциональности через объекты COM; многие из этих моделей были написаны давно и рассчитаны на использование необязательных параметров, не поддерживаемых в ранних версиях C#.

Для иллюстрации работы с необязательными аргументами предположим, что имеет-ся метод по имени `EnterLogData()` с одним необязательным параметром:

```
static void EnterLogData(string message, string owner = "Programmer")
{
    Console.Beep();
    Console.WriteLine("Error: {0}", message);
    Console.WriteLine("Owner of Error: {0}", owner);
}
```

Здесь последнему аргументу `string` было присвоено стандартное значение "Programmer" через операцию присваивания внутри определения параметров. В результате метод `EnterLogData()` можно вызывать из `Main()` двумя способами:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
    EnterLogData("Oh no! Grid can't find data");
    EnterLogData("Oh no! I can't find the payroll data", "CFO");
    Console.ReadLine();
}
```

Из-за того, что в первом вызове `EnterLogData()` не был указан второй аргумент `string`, будет использоваться его стандартное значение — "Programmer". Во втором вызове `EnterLogData()` для второго аргумента передано значение "CFO".

Важно понимать, что значение, присваиваемое необязательному параметру, должно быть известно на этапе компиляции и не может вычисляться во время выполнения (если вы попытаетесь сделать это, то компилятор сообщит об ошибке). В целях иллюстрации модифицируем метод `EnterLogData()`, добавив к нему дополнительный необязательный параметр:

```
// Ошибка! Стандартное значение для необязательного
// аргумента должно быть известно на этапе компиляции!
```



```
static void EnterLogData(string message,
    string owner = "Programmer", DateTime timeStamp = DateTime.Now)
{
    Console.Beep();
    Console.WriteLine("Error: {0}", message);
    Console.WriteLine("Owner of Error: {0}", owner);
    Console.WriteLine("Time of Error: {0}", timeStamp);
}
```

Такой код не скомпилируется, поскольку значение свойства `Now` класса `DateTime` вычисляется во время выполнения, а не на этапе компиляции.

На заметку! Во избежание неоднозначности необязательные параметры должны всегда помещаться в *конец* сигнатуры метода. Если необязательные параметры обнаруживаются перед обязательными, тогда компилятор сообщит об ошибке.

Вызов методов с использованием именованных параметров

Еще одним полезным языковым средством C# является поддержка *именованных аргументов*. По правде говоря, на первый взгляд может показаться, что данная языковая конструкция способна лишь порождать запутанный код. И по большому счету так действительно может произойти! Подобно необязательным аргументам причиной включения поддержки именованных параметров отчасти было желание упростить работу с уровнем взаимодействия с COM (см. главу 16).

Именованные аргументы позволяют вызывать метод с указанием значений параметров в любом желаемом порядке. Таким образом, вместо передачи параметров исключительно по позициям (как делается в большинстве случаев) можно указывать имя каждого аргумента, двоеточие и конкретное значение. Чтобы продемонстрировать использование именованных аргументов, добавим в класс `Program` следующий метод:

```
static void DisplayFancyMessage(ConsoleColor textColor,
    ConsoleColor backgroundColor, string message)
{
    // Сохранить старые цвета для их восстановления после вывода сообщения.
    ConsoleColor oldTextColor = Console.ForegroundColor;
    ConsoleColor oldbackgroundColor = Console.BackgroundColor;

    // Установить новые цвета и вывести сообщение.
    Console.ForegroundColor = textColor;
    Console.BackgroundColor = backgroundColor;
    Console.WriteLine(message);

    // Восстановить предыдущие цвета.
    Console.ForegroundColor = oldTextColor;
    Console.BackgroundColor = oldbackgroundColor;
}
```

Теперь, когда метод `DisplayFancyMessage()` написан, можно было бы ожидать, что при его вызове будут передаваться две переменные типа `ConsoleColor`, за которыми следует переменная типа `string`. Однако с помощью именованных аргументов метод `DisplayFancyMessage()` допустимо вызывать и так, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Methods *****");
    ...
}
```

```

DisplayFancyMessage(message: "Wow! Very Fancy indeed!",
    textColor: ConsoleColor.DarkRed,
    backgroundColor: ConsoleColor.White);

DisplayFancyMessage(backgroundColor: ConsoleColor.Green,
    message: "Testing...",
    textColor: ConsoleColor.DarkBlue);
Console.ReadLine();
}

```

С именованными аргументами связана одна особенность — при вызове метода позиционные параметры должны находиться перед всеми именованными параметрами. Другими словами, именованные аргументы должны всегда размещаться в конце вызова метода. Вот пример:

```

// Здесь все в порядке, т.к. позиционные аргументы находятся перед именованными.
DisplayFancyMessage(ConsoleColor.Blue,
    message: "Testing...",
    backgroundColor: ConsoleColor.White);

// ОШИБКА в вызове, поскольку позиционные аргументы идут после именованных.
DisplayFancyMessage(message: "Testing...",
    backgroundColor: ConsoleColor.White,
    ConsoleColor.Blue);

```

Даже если оставить в стороне указанное ограничение, то все равно может возникать вопрос: при каких условиях вообще требуется такая языковая конструкция? В конце концов, для чего нужно менять позиции аргументов метода?

Как выясняется, при наличии метода, в котором определены необязательные аргументы, данное средство может оказаться по-настоящему полезным. Предположим, что метод `DisplayFancyMessage()` переписан с целью поддержки необязательных аргументов, для которых указаны подходящие стандартные значения:

```

static void DisplayFancyMessage(ConsoleColor textColor = ConsoleColor.Blue,
    ConsoleColor backgroundColor = ConsoleColor.White,
    string message = "Test Message")
{
    ...
}

```

Учитывая, что каждый аргумент имеет стандартное значение, именованные аргументы позволяют указывать в вызывающем коде только те параметры, которые не должны принимать стандартные значения. Следовательно, если нужно, чтобы значение "Hello!" появлялось в виде текста синего цвета на белом фоне, то в вызывающем коде можно просто записать так:

```
DisplayFancyMessage(message: "Hello!");
```

Если же необходимо, чтобы строка "Test Message" выводилась синим цветом на зеленом фоне, тогда должен применяться такой вызов:

```
DisplayFancyMessage(backgroundColor: ConsoleColor.Green);
```

Как видите, необязательные аргументы и именованные параметры часто работают бок о бок. В завершение темы построения методов C# необходимо ознакомиться с концепцией *перегрузки методов*.

Понятие перегрузки методов

Подобно другим современным языкам объектно-ориентированного программирования в C# разрешена *перегрузка* методов. Выражаясь просто, когда определяется набор идентично именованных методов, которые отличаются друг от друга количеством (или типами) параметров, то говорят, что такой метод был *перегружен*.

Чтобы оценить удобство перегрузки методов, давайте представим себя на месте разработчика, использующего Visual Basic 6.0 (VB6). Предположим, что на языке VB6 создается набор методов, возвращающих сумму значений разнообразных типов (Integer, Double и т.д.). С учетом того, что VB6 не поддерживает перегрузку методов, придется определить уникальный набор методов, каждый из которых будет делать по существу одно и то же (возвращать сумму значений аргументов):

' Примеры кода VB6.

```
Public Function AddInts(ByVal x As Integer, ByVal y As Integer) As Integer
    AddInts = x + y
End Function

Public Function AddDoubles(ByVal x As Double, ByVal y As Double) As Double
    AddDoubles = x + y
End Function

Public Function AddLongs(ByVal x As Long, ByVal y As Long) As Long
    AddLongs = x + y
End Function
```

Такой код не только становится трудным в сопровождении, но и заставляет помнить имена всех методов. Применяя перегрузку, вызывающему коду можно предоставить возможность обращения к единственному методу по имени Add(). Ключевой аспект в том, чтобы обеспечить для каждой версии метода отличающийся набор аргументов (различий только в возвращаемом типе не достаточно).

На заметку! Как объясняется в главе 9, существует возможность построения обобщенных методов, которые переносят концепцию перегрузки на новый уровень. Используя обобщения, можно определять *заполнители типов* для реализации метода, которая указывается во время его вызова.

Чтобы попрактиковаться с перегруженными методами, создадим новый проект консольного приложения по имени MethodOverloading и добавим в него следующее определение класса:

```
// Код C#.
class Program
{
    static void Main(string[] args)
    {
    }

    // Перегруженный метод Add().
    static int Add(int x, int y)
    { return x + y; }

    static double Add(double x, double y)
    { return x + y; }

    static long Add(long x, long y)
    { return x + y; }
}
```

Теперь вызывающий код может просто обращаться к методу `Add()` с требуемыми аргументами, а компилятор будет самостоятельно находить подходящую для вызова реализацию на основе предоставленных аргументов:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Method Overloading *****\n");
    // Вызов int-версии Add().
    Console.WriteLine(Add(10, 10));
    // Вызов long-версии Add().
    Console.WriteLine(Add(900_000_000_000, 900_000_000_000));
    // Вызов double-версии Add().
    Console.WriteLine(Add(4.3, 4.4));
    Console.ReadLine();
}
```

Среда Visual Studio оказывает помощь при вызове перегруженных методов. Когда вводится имя перегруженного метода (такого как хорошо знакомый метод `Console.WriteLine()`), средство IntelliSense отображает список всех его доступных версий. Обратите внимание, что по списку можно перемещаться с применением клавиш со стрелками вниз и вверх (рис. 4.1).

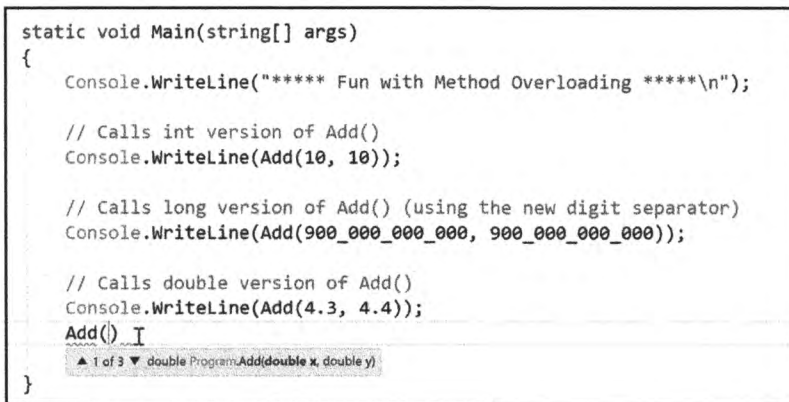


Рис. 4.1. Средство IntelliSense в Visual Studio для перегруженных методов

Исходный код. Проект `MethodOverloading` доступен в подкаталоге `Chapter_4`.

Локальные функции (нововведение)

Еще одним новым средством, появившимся в C# 7, является возможность создавать методы внутри методов, что официально называется *локальными функциями*. Локальная функция — это функция, объявленная внутри другой функции.

На заметку! До настоящего момента мы использовали термин *метод*. Почему вдруг введен термин *функция*? Отличается ли она чем-то от метода? Выражаясь формально, можно привести аргументы в пользу того, что функция и метод отличаются друг от друга. Но на практике они применяются взаимозаменяемо. В данной книге метод и функция считаются эквивалентными. Новое средство носит официальное название *локальные функции*, а потому мы не хотели его изменять, стремясь соблюсти согласованность в тексте. В дальнейшем мы будем называть их *методами*.

Чтобы взглянуть на работу нового средства, создадим проект консольного приложения по имени `FunWithLocalFunctions`. В качестве примера предположим, что нужно расширить метод `Add()` из предыдущего раздела, включив в него проверку достоверности вводимых данных. Достичь цели можно многими способами, один из которых предусматривает добавление проверки достоверности прямо в метод `Add()`. Давайте воспользуемся им и обновим предыдущий пример, как показано ниже:

```
static int Add(int x, int y)
{
    // Выполнить здесь проверку достоверности
    return x + y;
}
```

Как видите, крупные изменения отсутствуют. Есть только комментарий, который указывает, что реальный код должен предпринимать какое-то действие. Что, если мы захотим отделить фактическую задачу метода (возвращение суммы значений аргументов) от проверки достоверности данных аргументов? Можно было бы создать дополнительные методы и вызывать их из метода `Add()`. Но это потребовало бы создания еще одного метода, предназначенного для применения только одним другим методом. Выглядит как излишество. Новое средство позволяет сначала выполнить проверку достоверности и затем инкапсулировать фактическую задачу метода, определенного внутри метода `AddWrapper()`:

```
static int AddWrapper(int x, int y)
{
    // Выполнить здесь проверку достоверности
    return Add();
    int Add()
    {
        return x + y;
    }
}
```

Содержащийся внутри метод `Add()` может быть вызван только из охватывающего метода `AddWrapper()`. Возникает вопрос: что это нам дало? В контексте данного примера ответ довольно прост: немного (если вообще что-либо). Средство было добавлено в спецификацию C# для специальных итераторных методов (глава 8) и асинхронных методов (глава 19); вы увидите его преимущества при ознакомлении с указанными темами.

Исходный код. Проект `LocalFunctions` доступен в подкаталоге `Chapter_4`.

Итак, начальный обзор построения методов с использованием синтаксиса C# завершен. Теперь давайте посмотрим, каким образом создавать и манипулировать массивами, перечислениями и структурами.

Тип `enum`

Вспомните из главы 1, что система типов .NET состоит из классов, структур, перечислений, интерфейсов и делегатов. Чтобы начать исследование таких типов, рассмотрим роль *перечисления* (`enum`), создав новый проект консольного приложения по имени `FunWithEnums`.

На заметку! Не путайте термины *перечисление* и *перечислитель*; они обозначают совершенно разные концепции. Перечисление — специальный тип данных, состоящих из пар “имя-значение”. Перечислитель — тип класса или структуры, который реализует интерфейс .NET по имени `IEnumerable`. Обычно упомянутый интерфейс реализуется классами коллекций, а также классом `System.Array`. Как будет показано в главе 8, поддерживающие `IEnumerable` объекты могут работать с циклами `foreach`.

При построении какой-либо системы часто удобно создавать набор символических имен, которые отображаются на известные числовые значения. Например, в случае создания системы начисления заработной платы может возникнуть необходимость в ссылке на типы сотрудников с применением констант вроде `VicePresident` (вице-президент), `Manager` (менеджер), `Contractor` (подрядчик) и `Grunt` (рядовой сотрудник). Для этой цели в С# поддерживается понятие специальных перечислений. Например, далее представлено специальное перечисление по имени `EmpType` (его можно определить в том же файле, где находится класс `Program`, прямо перед определением класса):

```
// Специальное перечисление.
enum EmpType
{
    Manager,           // = 0
    Grunt,             // = 1
    Contractor,        // = 2
    VicePresident       // = 3
}
```

В перечислении `EmpType` определены четыре именованные константы, которые соответствуют дискретным числовым значениям. По умолчанию первому элементу присваивается значение 0, а остальным элементам значения устанавливаются по схеме $n+1$. При желании исходное значение можно изменять подходящим образом. Например, если имеет смысл нумеровать члены `EmpType` со значения 102 до 105, тогда можно поступить следующим образом:

```
// Начать нумерацию со значения 102.
enum EmpType
{
    Manager = 102,
    Grunt,           // = 103
    Contractor,      // = 104
    VicePresident     // = 105
}
```

Нумерация в перечислениях не обязательно должна быть последовательной и содержать только уникальные значения. Если (по той или иной причине) перечисление `EmpType` необходимо сконфигурировать так, как показано ниже, то компиляция пройдет гладко и без ошибок:

```
// Значения элементов в перечислении не обязательно должны быть
// последовательными!
enum EmpType
{
    Manager = 10,
    Grunt = 1,
    Contractor = 100,
    VicePresident = 9
}
```

Управление хранилищем, лежащим в основе перечисления

По умолчанию для хранения значений перечисления используется тип `System.Int32` (`int` в языке С#); тем не менее, при желании его легко заменить. Перечисления в С# можно определять в похожей манере для любых основных системных типов (`byte`, `short`, `int` или `long`). Например, чтобы значения перечисления `EmpType` хранились с применением типа `byte`, а не `int`, можно записать так:

```
// На этот раз для элементов EmpType используется тип byte.
enum EmpType : byte
{
    Manager = 10,
    Grunt = 1,
    Contractor = 100,
    VicePresident = 9
}
```

Изменение типа, лежащего в основе перечисления, может быть полезным при построении приложения .NET, которое планируется развертывать на устройствах с небольшим объемом памяти, а потому необходимо экономить память везде, где только возможно. Конечно, если в качестве типа хранилища для перечисления указан `byte`, то каждое значение должно входить в диапазон его допустимых значений. Например, следующая версия `EmpType` приведет к ошибке на этапе компиляции, т.к. значение 999 не умещается в диапазон допустимых значений типа `byte`:

```
// Ошибка на этапе компиляции! Значение 999 слишком велико для типа byte!
enum EmpType : byte
{
    Manager = 10,
    Grunt = 1,
    Contractor = 100,
    VicePresident = 999
}
```

Объявление переменных типа перечисления

После установки диапазона и типа хранилища перечисление можно использовать вместо так называемых “магических чисел”. Поскольку перечисления — всего лишь определяемые пользователем типы данных, их можно применять как возвращаемые значения функций, параметры методов, локальные переменные и т.д. Предположим, что есть метод по имени `AskForBonus()`, который принимает в качестве единственного параметра переменную `EmpType`. На основе значения входного параметра в окно консоли будет выводиться подходящий ответ на запрос о надбавке к зарплате.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("**** Fun with Enums ****");
        // Создать переменную типа EmpType.
        EmpType emp = EmpType.Contractor;
        AskForBonus(emp);
        Console.ReadLine();
    }

    // Перечисления как параметры.
    static void AskForBonus(EmpType e)
    {
        switch (e)
        {
            case EmpType.Manager:
                Console.WriteLine("How about stock options instead?");
                // Не желаете ли взамен фондовые опционы?
                break;
        }
    }
}
```

```

case EmpType.Grunt:
    Console.WriteLine("You have got to be kidding...");
    // Вы должно быть шутите...
break;
case EmpType.Contractor:
    Console.WriteLine("You already get enough cash...");
    // Вы уже получаете вполне достаточно...
break;
case EmpType.VicePresident:
    Console.WriteLine("VERY GOOD, Sir!");
    // Очень хорошо, сэр!
break;
    }
}
}

```

Обратите внимание, что когда переменной `enum` присваивается значение, вы должны указывать перед этим значением (`Grunt`) имя самого перечисления (`EmpType`). Из-за того, что перечисления представляют собой фиксированные наборы пар "имя-значение", установка переменной `enum` в значение, которое не определено прямо в перечислимом типе, не допускается:

```

static void ThisMethodWillNotCompile()
{
    // Ошибка! SalesManager отсутствует в перечислении EmpType!
    EmpType emp = EmpType.SalesManager;

    // Ошибка! Не указано имя EmpType перед значением Grunt!
    emp = Grunt;
}

```

Тип System.Enum

С перечислениями .NET связан один интересный аспект — они получают свою функциональность от класса `System.Enum`. В классе `System.Enum` определено множество методов, которые позволяют исследовать и трансформировать заданное перечисление. Одним из них является метод `Enum.GetUnderlyingType()`, который возвращает тип данных, используемый для хранения значений перечислимого типа (`System.Byte` в текущем объявлении `EmpType`):

```

static void Main(string[] args)
{
    Console.WriteLine("**** Fun with Enums ****");
    // Создать переменную типа EmpType.
    EmpType emp = EmpType.Contractor;
    AskForBonus(emp);
    // Вывести тип хранилища для значений перечисления.
    Console.WriteLine("EmpType uses a {0} for storage",
        Enum.GetUnderlyingType(emp.GetType()));
    Console.ReadLine();
}

```

Заглянув в браузер объектов Visual Studio, можно удостовериться, что метод `Enum.GetUnderlyingType()` требует передачи `System.Type` в качестве первого параметра. В главе 15 будет показано, что класс `Type` представляет описание метаданных для конкретной сущности .NET.

Один из возможных способов получения метаданных (как демонстрировалось ранее) предусматривает применение метода `GetType()`, который является общим для всех типов

в библиотеках базовых классов .NET. Другой подход заключается в использовании операции `typeof` языка C#. Преимущество такого способа связано с тем, что он не требует объявления переменной сущности, описание метаданных которой требуется получить:

```
// На этот раз для получения информации о типе используется операция typeof.
Console.WriteLine("EmpType uses a {0} for storage",
    Enum.GetUnderlyingType(typeof(EmpType)));
```

Динамическое выяснение пар “имя-значение” перечисления

Кроме метода `Enum.GetUnderlyingType()` все перечисления C# поддерживают метод по имени `ToString()`, который возвращает строковое имя текущего значения перечисления. Ниже приведен пример:

```
static void Main(string[] args)
{
    Console.WriteLine("**** Fun with Enums ****");
    EmpType emp = EmpType.Contractor;
    AskForBonus(emp);

    // Выводит строку "emp is a Contractor".
    Console.WriteLine("emp is a {0}.", emp.ToString());
    Console.ReadLine();
}
```

Если интересует не имя, а значение заданной переменной перечисления, то можно просто привести ее к лежащему в основе типу хранения, например:

```
static void Main(string[] args)
{
    Console.WriteLine("**** Fun with Enums ****");
    EmpType emp = EmpType.Contractor;
    ...
    // Выводит строку "Contractor = 100".
    Console.WriteLine("{0} = {1}", emp.ToString(), (byte)emp);
    Console.ReadLine();
}
```

На заметку! Статический метод `Enum.Format()` предлагает более высокий уровень форматирования за счет указания флага желаемого формата. Более подробную информацию о методе `System.Enum.Format()` ищите в документации .NET Framework 4.7 SDK.

В типе `System.Enum` определен еще один статический метод по имени `GetValues()`, возвращающий экземпляр класса `System.Array`. Каждый элемент в массиве соответствует члену в указанном перечислении. Рассмотрим следующий метод, который выводит на консоль пары “имя-значение” из перечисления, переданного в качестве параметра:

```
// Этот метод выводит детали любого перечисления.
static void EvaluateEnum(System.Enum e)
{
    Console.WriteLine("=> Information about {0}", e.GetType().Name);
    Console.WriteLine("Underlying storage type: {0}",
        Enum.GetUnderlyingType(e.GetType()));

    // Получить все пары "имя-значение" для входного параметра.
    Array enumData = Enum.GetValues(e.GetType());
    Console.WriteLine("This enum has {0} members.", enumData.Length);
    // Вывести строковое имя и ассоциированное значение,
    // используя флаг формата D (см. главу 3).
```

```

for(int i = 0; i < enumData.Length; i++)
{
    Console.WriteLine("Name: {0}, Value: {0:D}",
        enumData.GetValue(i));
}
Console.WriteLine();
}

```

Чтобы протестировать метод `EvaluateEnum()`, модифицируем `Main()` для создания переменных нескольких типов перечислений, объявленных в пространстве имен `System` (вместе с перечислением `EmpType`):

```

static void Main(string[] args)
{
    Console.WriteLine("**** Fun with Enums ****");
    ...
    EmpType e2 = EmpType.Contractor;
    // Эти типы являться перечислениями из пространства имен System.
    DayOfWeek day = DayOfWeek.Monday;
    ConsoleColor cc = ConsoleColor.Gray;

    EvaluateEnum(e2);
    EvaluateEnum(day);
    EvaluateEnum(cc);
    Console.ReadLine();
}

```

Ниже показана часть вывода:

```

=> Information about DayOfWeek
Underlying storage type: System.Int32
This enum has 7 members.
Name: Sunday, Value: 0
Name: Monday, Value: 1
Name: Tuesday, Value: 2
Name: Wednesday, Value: 3
Name: Thursday, Value: 4
Name: Friday, Value: 5
Name: Saturday, Value: 6

```

В ходе чтения книги вы увидите, что перечисления широко применяются во всех библиотеках базовых классов .NET. Например, в ADO.NET используются разнообразные перечисления для представления состояния подключения к базе данных (открыто или закрыто) либо состояния строки в объекте `DataTable` (изменена, новая или отсоединена). Таким образом, при работе с любым перечислением всегда помните о возможности взаимодействия с парами “имя-значение”, применяя члены класса `System.Enum`.

Исходный код. Проект `FunWithEnums` доступен в подкаталоге `Chapter_4`.

Понятие структуры (как типа значения)

Теперь, когда вы понимаете роль типов перечислений, давайте посмотрим, как использовать *структуры* (`struct`) .NET. Типы структур хорошо подходят для моделирования в приложении математических, геометрических и других “атомарных” сущностей. Структура (такая как перечисление) — это определяемый пользователем тип; тем не менее, структура не является просто коллекцией пар “имя-значение”. Взамен структуры

представляют собой типы, которые могут содержать любое количество полей данных и членов, действующих на таких полях.

На заметку! Если вы имеете опыт объектно-ориентированного программирования, тогда можете считать структуры “легковесными типами классов”, т.к. они предоставляют способ определения типа, который поддерживает инкапсуляцию, но не может использоваться для построения семейства взаимосвязанных типов. Когда возникает потребность в создании семейства типов, связанных отношением наследования, необходимо применять классы.

На первый взгляд процесс определения и использования структур выглядит простым, но, как часто бывает, самое сложное скрыто в деталях. Чтобы приступить к изучению основ типов структур, создадим новый проект по имени FunWithStructures. В языке C# структуры определяются с применением ключевого слова `struct`. Определим новую структуру по имени `Point`, которая содержит две переменные типа `int` и набор методов для взаимодействия с ними:

```
struct Point
{
    // Поля структуры.
    public int X;
    public int Y;

    // Добавить 1 к позиции (X, Y).
    public void Increment()
    {
        X++; Y++;
    }

    // Вычесть 1 из позиции (X, Y).
    public void Decrement()
    {
        X--; Y--;
    }

    // Отобразить текущую позицию.
    public void Display()
    {
        Console.WriteLine("X = {0}, Y = {1}", X, Y);
    }
}
```

Здесь мы определили два целочисленных поля (`X` и `Y`), используя ключевое слово `public`, которое является модификатором управления доступом (обсуждение будет продолжено в главе 5). Объявление данных с ключевым словом `public` обеспечивает вызывающему коду возможность прямого доступа к таким данным через переменную `Point` (посредством операции точки).

На заметку! Определение открытых данных внутри класса или структуры обычно считается плохим стилем кодирования. Взамен рекомендуется определять *закрытые* данные, доступ и изменение которых производится с применением *открытых* свойств. Более подробные сведения приведены в главе 5.

Вот метод `Main()`, который позволяет протестировать тип `Point`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** A First Look at Structures *****\n");
```

```
// Создать начальную переменную типа Point.
Point myPoint;
myPoint.X = 349;
myPoint.Y = 76;
myPoint.Display();
// Скорректировать значения X и Y.
myPoint.Increment();
myPoint.Display();
Console.ReadLine();
}
```

Вывод выглядит вполне ожидаемо:

```
***** A First Look at Structures *****
X = 349, Y = 76
X = 350, Y = 77
```

Создание переменных типа структур

Для создания переменной типа структуры на выбор доступно несколько вариантов. В следующем коде мы просто создаем переменную типа `Point` и присваиваем значения каждому ее открытому полю данных до того, как обращаться к членам переменной. Если не присвоить значения открытым полям данных (`X` и `Y` в данном случае) перед использованием структуры, то компилятор сообщит об ошибке:

```
// Ошибка! Полю Y не присвоено значение.
Point p1;
p1.X = 10;
p1.Display();

// Все в порядке! Перед использованием значения присвоены обоим полям.
Point p2;
p2.X = 10;
p2.Y = 10;
p2.Display();
```

В качестве альтернативы переменные типа структур можно создавать с применением ключевого слова `new` языка C#, что приводит к вызову *стандартного конструктора* структуры. По определению стандартный конструктор не принимает аргументов. Преимущество вызова стандартного конструктора структуры в том, что каждое поле данных автоматически получает свое стандартное значение:

```
// Установить для всех полей стандартные значения,
// используя стандартный конструктор.
Point p1 = new Point();

// Выводит X=0, Y=0
p1.Display();
```

Допускается также проектировать структуры со *специальным конструктором*, что позволяет указывать значения для полей данных при создании переменной, а не устанавливать их по отдельности. Конструкторы подробно рассматриваются в главе 5; однако в целях иллюстрации изменим структуру `Point` следующим образом:

```
struct Point
{
    // Поля структуры.
    public int X;
    public int Y;
```

```
// Специальный конструктор.
public Point(int XPos, int YPos)
{
    X = XPos;
    Y = YPos;
}
...
}
```

Затем переменные типа `Point` можно создавать так:

```
// Вызвать специальный конструктор.
Point p2 = new Point(50, 60);

// Выводит X=50, Y=60
p2.Display();
```

Как упоминалось ранее, работа со структурами на первый взгляд довольно проста. Тем не менее, чтобы углубить понимание особенностей этого типа, необходимо ознакомиться с отличиями между типами значений и ссылочными типами .NET.

Исходный код. Проект `FunWithStructures` доступен в подкаталоге `Chapter_4`.

Типы значений и ссылочные типы

На заметку! В последующем обсуждении типов значений и ссылочных типов предполагается наличие у вас базовых знаний объектно-ориентированного программирования. Если это не так, тогда имеет смысл перейти к чтению раздела "Понятие типов C#, допускающих `null`" далее в главе и возвратиться к настоящему разделу после изучения глав 5 и 6.

В отличие от массивов, строк и перечислений структуры C# не имеют идентично именованного представления в библиотеке .NET (т.е. класс вроде `System.Structure` отсутствует), но они являются неявно производными от абстрактного класса `System.ValueType`. Выражаясь просто, роль класса `System.ValueType` заключается в обеспечении размещения экземпляра производного типа (например, любой структуры) в *стеке*, а не в *куче* с автоматической сборкой мусора. Данные, размещаемые в стеке, могут создаваться и уничтожаться быстро, т.к. время их жизни определяется областью видимости, в которой они объявлены. С другой стороны, данные, размещаемые в куче, отслеживаются сборщиком мусора .NET и имеют время жизни, которое определяется большим числом факторов, объясняемых в главе 13.

С точки зрения функциональности единственное назначение класса `System.ValueType` — переопределение функциональных методов, объявленных в классе `System.Object`, с целью использования семантики на основе значений, а не ссылок. Вероятно, вы уже знаете, что переопределение представляет собой процесс изменения реализации виртуального (или возможно абстрактного) метода, определенного внутри базового класса. Базовым классом для `ValueType` является `System.Object`. В действительности методы экземпляра, определенные в `System.ValueType`, идентичны методам экземпляра, которые определены в `System.Object`:

```
// Структуры и перечисления неявно расширяют класс System.ValueType.
public abstract class ValueType : object
{
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
}
```

```

    public Type GetType();
    public virtual string ToString();
}

```

Учитывая, что типы значений применяют семантику на основе значений, время жизни структуры (что относится ко всем числовым типам данных (int, float), а также к любому перечислению или структуре) предсказуемо. Когда переменная типа структуры покидает область определения, она немедленно удаляется из памяти:

```

// Локальные структуры извлекаются из стека,
// когда метод возвращает управление.
static void LocalValueTypes()
{
    // Вспомните, что int - на самом деле структура System.Int32.
    int i = 0;
    // Вспомните, что Point - в действительности тип структуры.
    Point p = new Point();
} // Здесь i и p покидают стек!

```

Типы значений, ссылочные типы и операция присваивания

Когда переменная одного типа значения присваивается переменной другого типа значения, выполняется почленное копирование полей данных. В случае простого типа данных, такого как System.Int32, единственным копируемым членом будет числовое значение. Однако для типа Point в новую переменную структуры будут копироваться значения полей X и Y. В целях демонстрации создадим новый проект консольного приложения по имени ValueAndReferenceTypes и скопируем предыдущее определение Point в новое пространство имен, после чего добавим к типу Program следующий метод:

```

// Присваивание двух внутренних типов значений дает
// в результате две независимые переменные в стеке.
static void ValueTypeAssignment()
{
    Console.WriteLine("Assigning value types\n");

    Point p1 = new Point(10, 10);
    Point p2 = p1;

    // Вывести значения обеих переменных Point.
    p1.Display();
    p2.Display();

    // Изменить p1.X и снова вывести значения переменных. Значение p2.X не изменилось.
    p1.X = 100;
    Console.WriteLine("\n=> Changed p1.X\n");
    p1.Display();
    p2.Display();
}

```

Здесь создается переменная типа Point (p1), которая присваивается другой переменной типа Point (p2). Поскольку Point — тип значения, в стеке находятся две копии Point, каждой из которых можно манипулировать независимым образом. Поэтому при изменении значения p1.X значение p2.X остается незатронутым:

```

Assigning value types
X = 10, Y = 10
X = 10, Y = 10
=> Changed p1.X
X = 100, Y = 10
X = 10, Y = 10

```

По контрасту с типами значений, когда операция присваивания применяется к переменным ссылочных типов (т.е. экземплярам всех классов), происходит перенаправление на то, на что ссылочная переменная указывает в памяти. В целях иллюстрации создадим новый класс по имени `PointRef` с теми же членами, что и у структуры `Point`, но только переименуем конструктор в соответствии с именем данного класса:

```
// Классы всегда являются ссылочными типами.
class PointRef
{
    // Те же самые члены, что и в структуре Point...
    // Не забудьте изменить имя конструктора на PointRef!
    public PointRef(int XPos, int YPos)
    {
        X = XPos;
        Y = YPos;
    }
}
```

Задействуем готовый тип `PointRef` в следующем новом методе. Обратите внимание, что помимо использования вместо структуры `Point` класса `PointRef` код идентичен коду метода `ValueTypeAssignment()`:

```
static void ReferenceTypeAssignment()
{
    Console.WriteLine("Assigning reference types\n");
    PointRef p1 = new PointRef(10, 10);
    PointRef p2 = p1;

    // Вывести значения обеих переменных PointRef.
    p1.Display();
    p2.Display();

    // Изменить p1.X и снова вывести значения.
    p1.X = 100;
    Console.WriteLine("\n=> Changed p1.X\n");
    p1.Display();
    p2.Display();
}
```

В рассматриваемом случае есть две ссылки, указывающие на тот же самый объект в управляемой куче. Таким образом, когда изменяется значение `X` с использованием ссылки `p1`, изменится также и значение `p2.X`. Вот вывод, получаемый в результате вызова метода `ReferenceTypeAssignment()` внутри `Main()`:

```
Assigning reference types
X = 10, Y = 10
X = 10, Y = 10
=> Changed p1.X
X = 100, Y = 10
X = 100, Y = 10
```

Типы значений, содержащие ссылочные типы

Теперь, когда вы лучше понимаете базовые отличия между типами значений и ссылочными типами, давайте обратимся к более сложному примеру. Предположим, что имеется следующий ссылочный тип (класс), который поддерживает информационную строку (`InfoString`), устанавливаемую с применением специального конструктора:

```
class ShapeInfo
{
    public string InfoString;
    public ShapeInfo(string info)
    {
        InfoString = info;
    }
}
```

Далее представим, что переменная типа ShapeInfo должна содержаться внутри типа значения по имени Rectangle. Кроме того, в типе Rectangle предусмотрен специальный конструктор, который позволяет вызывающему коду указывать значение для внутренней переменной-члена типа ShapeInfo. Вот полное определение типа Rectangle:

```
struct Rectangle
{
    // Структура Rectangle содержит член ссылочного типа.
    public ShapeInfo RectInfo;
    public int RectTop, RectLeft, RectBottom, RectRight;
    public Rectangle(string info, int top, int left, int bottom, int right)
    {
        RectInfo = new ShapeInfo(info);
        RectTop = top; RectBottom = bottom;
        RectLeft = left; RectRight = right;
    }
    public void Display()
    {
        Console.WriteLine("String = {0}, Top = {1}, Bottom = {2}, " +
            "Left = {3}, Right = {4}",
            RectInfo.infoString, RectTop, RectBottom, RectLeft, RectRight);
    }
}
```

Здесь ссылочный тип содержится внутри типа значения. Возникает важный вопрос: что произойдет в результате присваивания одной переменной типа Rectangle другой переменной того же типа? Учитывая то, что уже известно о типах значений, можно корректно предположить, что целочисленные данные (которые на самом деле являются структурой — System.Int32) должны быть независимой сущностью для каждой переменной Rectangle. Но что можно сказать о внутреннем ссылочном типе? Будет ли полностью скопировано *состояние* этого объекта или же только ссылка на него? Чтобы получить ответ, определим новый метод и вызовем его внутри Main():

```
static void ValueTypeContainingRefType()
{
    // Создать первую переменную Rectangle.
    Console.WriteLine("-> Creating r1");
    Rectangle r1 = new Rectangle("First Rect", 10, 10, 50, 50);
    // Присвоить новой переменной Rectangle переменную r1.
    Console.WriteLine("-> Assigning r2 to r1");
    Rectangle r2 = r1;
    // Изменить некоторые значения в r2.
    Console.WriteLine("-> Changing values of r2");
    r2.RectInfo.InfoString = "This is new info!";
    r2.RectBottom = 4444;
    // Вывести значения из обеих переменных Rectangle.
    r1.Display();
    r2.Display();
}
```


Вывод будет таким:

```
-> Creating r1
-> Assigning r2 to r1
-> Changing values of r2
String = This is new info!, Top = 10, Bottom = 50, Left = 10, Right = 50
String = This is new info!, Top = 10, Bottom = 4444, Left = 10, Right = 50
```

Как видите, когда мы модифицируем значение информационной строки с использованием ссылки r2, для ссылки r1 отображается то же самое значение. По умолчанию, если тип значения содержит другие ссылочные типы, то присваивание приводит к копированию ссылок. В результате получаются две независимые структуры, каждая из которых содержит ссылку, указывающую на один и тот же объект в памяти (т.е. создается поверхностная копия). Для выполнения глубокого копирования, при котором в новый объект полностью копируется состояние внутренних ссылок, можно реализовать интерфейс `ICloneable` (что будет показано в главе 8).

Исходный код. Проект `ValueAndReferenceTypes` доступен в подкаталоге `Chapter_4`.

Передача ссылочных типов по значению

Вполне очевидно, что ссылочные типы и типы значений могут передаваться методам в виде параметров. Тем не менее, передача ссылочного типа (например, класса) по ссылке совершенно отличается от его передачи по значению. Чтобы понять разницу, предположим, что есть простой класс `Person`, определенный в новом проекте консольного приложения по имени `RefTypeValTypeParams`:

```
class Person
{
    public string personName;
    public int personAge;

    // Конструкторы.
    public Person(string name, int age)
    {
        personName = name;
        personAge = age;
    }
    public Person() {}
    public void Display()
    {
        Console.WriteLine("Name: {0}, Age: {1}", personName, personAge);
    }
}
```

А что если мы создадим метод, который позволит вызывающему коду передавать объект `Person` по значению (обратите внимание на отсутствие модификаторов параметров, таких как `out` или `ref`)?

```
static void SendAPersonByValue(Person p)
{
    // Изменить значение возраста в p?
    p.personAge = 99;

    // Увидит ли вызывающий код это изменение?
    p = new Person("Nikk1", 99);
}
```

Здесь видно, что метод `SendAPersonByValue()` пытается присвоить входной ссылке на `Person` новый объект `Person`, а также изменить некоторые данные состояния. Протестируем этот метод, вызвав его внутри `Main()`.

```
static void Main(string[] args)
{
    // Передача ссылочных типов по значению.
    Console.WriteLine("***** Passing Person object by value *****");
    Person fred = new Person("Fred", 12);
    Console.WriteLine("\nBefore by value call, Person is:"); // перед вызовом
    fred.Display();

    SendAPersonByValue(fred);
    Console.WriteLine("\nAfter by value call, Person is:"); // после вызова
    fred.Display();
    Console.ReadLine();
}
```

Ниже показан результирующий вывод:

```
***** Passing Person object by value *****
Before by value call, Person is:
Name: Fred, Age: 12
After by value call, Person is:
Name: Fred, Age: 99
```

Легко заметить, что значение `PersonAge` было изменено. Кажется, что такое поведение противоречит смыслу передачи параметра по значению. Учитывая, что попытка изменения состояния входного объекта `Person` прошла успешно, возникает вопрос: что же тогда было скопировано? Ответ: была получена копия ссылки на объект из вызывающего кода. Следовательно, раз уж метод `SendAPersonByValue()` указывает на тот же самый объект, что и вызывающий код, становится возможным изменение данных состояния этого объекта. Нельзя лишь переустанавливать ссылку так, чтобы она указывала на какой-то другой объект.

Передача ссылочных типов по ссылке

Предположим, что имеется метод `SendAPersonByReference()`, в котором ссылочный тип передается по ссылке (обратите внимание на наличие модификатора параметра `ref`):

```
static void SendAPersonByReference(ref Person p)
{
    // Изменить некоторые данные в p.
    p.personAge = 555;

    // p теперь указывает на новый объект в куче!
    p = new Person("Nikki", 999);
}
```

Как и можно было ожидать, вызываемому коду предоставлена полная свобода в плане манипулирования входным параметром. Вызываемый код не только может изменять состояние объекта, но и переопределять ссылку так, чтобы она указывала на новый объект `Person`. Давайте протестируем метод `SendAPersonByReference()`, вызвав его внутри `Main()`:

```
static void Main(string[] args)
{
    // Передача ссылочных типов по ссылке.
```

```
Console.WriteLine("***** Passing Person object by reference *****");
...
Person mel = new Person("Mel", 23);
Console.WriteLine("Before by ref call, Person is:"); // перед вызовом
mel.Display();

SendAPersonByReference(ref mel);
Console.WriteLine("After by ref call, Person is:"); // после вызова
mel.Display();
Console.ReadLine();
}
```

Вывод выглядит следующим образом:

```
***** Passing Person object by reference *****
Before by ref call, Person is:
Name: Mel, Age: 23
After by ref call, Person is:
Name: Nikki, Age: 999
```

Здесь видно, что после вызова объект по имени Mel возвращается как объект по имени Nikki, поскольку метод имел возможность изменить то, на что указывала в памяти входная ссылка. Ниже представлены основные правила, которые необходимо соблюдать при передаче ссылочных типов.

- Если ссылочный тип передается по ссылке, тогда вызываемый код может изменять значения данных состояния объекта, а также объект, на который указывает ссылка.
- Если ссылочный тип передается по значению, то вызываемый код может изменять значения данных состояния объекта, но не объект, на который указывает ссылка.

Исходный код. Проект RefTypeValTypeParams доступен в подкаталоге Chapter_4.

Заключительные детали относительно типов значений и ссылочных типов

В завершение данной темы взгляните на табл. 4.3 со сводкой по основным отличиям между типами значений и ссылочными типами.

Таблица 4.3. Отличия между типами значений и ссылочными типами

Интересующий вопрос	Тип значения	Ссылочный тип
Где размещены объекты?	Размещаются в стеке	Размещаются в управляемой куче
Как представлена переменная?	Переменные типов значений являются локальными копиями	Переменные ссылочных типов указывают на память, занимаемую размещенным экземпляром
Какой тип является базовым?	Неявно расширяет System.ValueType	Может быть производным от любого другого типа (кроме System.ValueType), если только этот тип не запечатан (см. главу 6)

Окончание табл. 4.3

Интересующий вопрос	Тип значения	Ссылочный тип
Может ли этот тип выступать в качестве базового для других типов?	Нет. Типы значений всегда запечатаны, и наследовать от них нельзя	Да. Если тип не запечатан, то он может выступать в качестве базового для других типов
Каково стандартное поведение передачи параметров?	Переменные передаются по значению (т.е. вызываемой функции передается копия переменной)	Для ссылочных типов ссылка копируется по значению
Можно ли переопределять метод <code>System.Object.Finalize()</code> в этом типе?	Нет	Да, косвенно (как показано в главе 13)
Можно ли определять конструкторы для этого типа?	Да, но стандартный конструктор является зарезервированным (т.е. все специальные конструкторы должны иметь аргументы)	Безусловно!
Когда переменные этого типа прекращают свое существование?	Когда покидают область видимости, в которой они были определены	Когда объект подвергается сборке мусора

Несмотря на различия, типы значений и ссылочные типы имеют возможность реализовывать интерфейсы и могут поддерживать любое количество полей, методов, перегруженных операций, констант, свойств и событий.

Понятие типов C#, допускающих `null`

Давайте исследуем роль *типов данных, допускающих значение `null`*, с применением консольного приложения по имени `NullableTypes`. Как вам уже известно, типы данных C# обладают фиксированным диапазоном значений и представлены в виде типов пространства имен `System`. Например, тип данных `System.Boolean` может принимать только значения из набора `{true, false}`. Вспомните, что все числовые типы данных (а также `Boolean`) являются *типами значений*. Типам значений никогда не может быть присвоено значение `null`, потому что оно служит для представления пустой объектной ссылки.

```
static void Main(string[] args)
{
    // Ошибка на этапе компиляции!
    // Типы значений не могут быть установлены в null!
    bool myBool = null;
    int myInt = null;
    // Все в порядке! Строки являются ссылочными типами.
    string myString = null;
}
```

Язык C# поддерживает концепцию *типов данных, допускающих значение `null`*. Выражаясь просто, допускающий `null` тип может представлять все значения лежащего в основе типа плюс `null`. Таким образом, если вы объявите переменную типа `bool`, допускающего `null`, то ей можно будет присваивать значение из набора `{true, false, null}`. Это может быть чрезвычайно удобно при работе с реляционными базами данных, поскольку в таблицах баз данных довольно часто встречаются столбцы, для которых значе-

ния не определены. Без концепции типов данных, допускающих `null`, в C# не было бы удобного способа для представления числовых элементов данных без значений.

Чтобы определить переменную типа, допускающего `null`, необходимо добавить к имени интересующего типа данных суффикс в виде знака вопроса (?). Обратите внимание, что такой синтаксис законен, только когда применяется к типам значений. При попытке создать ссылочный тип, допускающий `null` (включая `string`), компилятор сообщит об ошибке. Как и переменным с типами, не допускающими `null`, локальным переменным, которые имеют типы, допускающие `null`, должно присваиваться начальное значение, прежде чем ими можно будет пользоваться:

```
static void LocalNullableVariables()
{
    // Определить несколько локальных переменных, допускающих null.
    int? nullableInt = 10;
    double? nullableDouble = 3.14;
    bool? nullableBool = null;
    char? nullableChar = 'a';
    int?[] arrayOfNullableInts = new int?[10];

    // Ошибка! Строки являются ссылочными типами!
    // string? s = "oops";
}
```

В языке C# система обозначений в форме суффикса ? представляет собой сокращение для создания экземпляра обобщенного типа структуры `System.Nullable<T>`. Хотя подробное исследование обобщений мы отложим до главы 9, сейчас важно понимать, что тип `System.Nullable<T>` предоставляет набор членов, которые могут применяться всеми типами, допускающими `null`.

Например, с помощью свойства `HasValue` или операции `!=` можно программно выяснять, действительно ли переменной, допускающей `null`, было присвоено значение `null`. Значение, которое присвоено типу, допускающему `null`, можно получать напрямую или через свойство `Value`. Учитывая, что суффикс ? является просто сокращением для использования `Nullable<T>`, метод `LocalNullableVariables()` можно было бы реализовать следующим образом:

```
static void LocalNullableVariablesUsingNullable()
{
    // Определить несколько типов, допускающих null, с применением Nullable<T>.
    Nullable<int> nullableInt = 10;
    Nullable<double> nullableDouble = 3.14;
    Nullable<bool> nullableBool = null;
    Nullable<char> nullableChar = 'a';
    Nullable<int>[] arrayOfNullableInts = new Nullable<int>[10];
}
```

Работа с типами, допускающими `null`

Как отмечалось ранее, типы данных, допускающие `null`, особенно полезны при взаимодействии с базами данных, потому что столбцы в таблицах данных могут быть намеренно оставлены пустыми (скажем, быть неопределенными). В целях демонстрации рассмотрим показанный далее класс, эмулирующий процесс доступа к базе данных с таблицей, в которой два столбца могут принимать значения `null`. Обратите внимание, что метод `GetIntFromDatabase()` не присваивает значение члену целочисленного типа, допускающего `null`, тогда как метод `GetBoolFromDatabase()` присваивает допустимое значение члену типа `bool?`.

```

class DatabaseReader
{
    // Поле данных типа, допускающего null.
    public int? numericValue = null;
    public bool? boolValue = true;

    // Обратите внимание на возвращаемый тип, допускающий null.
    public int? GetIntFromDatabase()
    { return numericValue; }

    // Обратите внимание на возвращаемый тип, допускающий null.
    public bool? GetBoolFromDatabase()
    { return boolValue; }
}

```

Теперь предположим, что в следующем методе Main() происходит обращение к каждому члену класса DatabaseReader и выяснение присвоенных значений с применением членов HasValue и Value, а также операции равенства C# (точнее операции “не равно”):

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Nullable Data *****\n");
    DatabaseReader dr = new DatabaseReader();

    // Получить значение int из "базы данных".
    int? i = dr.GetIntFromDatabase();
    if (i.HasValue)
        Console.WriteLine("Value of 'i' is: {0}", i.Value);
        // вывод значения переменной i
    else
        Console.WriteLine("Value of 'i' is undefined.");
        // значение переменной i не определено

    // Получить значение bool из "базы данных".
    bool? b = dr.GetBoolFromDatabase();
    if (b != null)
        Console.WriteLine("Value of 'b' is: {0}", b.Value);
        // вывод значения переменной b
    else
        Console.WriteLine("Value of 'b' is undefined.");
        // значение переменной b не определено
    Console.ReadLine();
}

```

Операция объединения с null

Следующий важный аспект связан с тем, что любая переменная, которая может иметь значение null (т.е. переменная ссылочного типа или переменная типа, допускающего null), может использоваться с операцией ?? языка C#, формально называемой *операцией объединения с null*. Операция ?? позволяет присваивать значение типу, допускающему null, если извлеченное значение на самом деле равно null. В рассматриваемом примере мы предположим, что в случае возвращения методом GetIntFromDatabase() значения null (конечно, данный метод запрограммирован так, что он *всегда* возвращает null, но общую идею вы должны уловить) локальной переменной целочисленного типа, допускающего null, необходимо присвоить значение 100:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Nullable Data *****\n");
    DatabaseReader dr = new DatabaseReader();
    ...
    // Если значение, возвращаемое из GetIntFromDatabase(), равно
    // null, то присвоить локальной переменной значение 100.
    int myData = dr.GetIntFromDatabase() ?? 100;
    Console.WriteLine("Value of myData: {0}", myData);
    Console.ReadLine();
}
```

Преимущество применения операции `??` заключается в том, что она дает более компактную версию кода, чем традиционный условный оператор `if/else`. Однако при желании можно было бы написать показанный ниже функционально эквивалентный код, который в случае возвращения `null` обеспечит установку переменной в значение 100:

```
// Более длинный код, в котором не используется синтаксис ??
int? moreData = dr.GetIntFromDatabase();
if (!moreData.HasValue)
    moreData = 100;
Console.WriteLine("Value of moreData: {0}", moreData);
```

null-условная операция

При разработке программного обеспечения распространенная практика предусматривает проверку на предмет `null` входных параметров, которым передаются значения, возвращаемые членами типов (методами, свойствами, индексаторами). Например, пусть имеется метод, который принимает в качестве единственного параметра строковый массив. В целях безопасности его желательно проверять на предмет `null`, прежде чем приступить к обработке. Поступая подобным образом, мы не получим ошибку во время выполнения, если массив окажется пустым. Следующий код демонстрирует традиционный способ реализации такой проверки:

```
static void TesterMethod(string[] args)
{
    // Перед доступом к данным массива мы должны проверить его на равенство null!
    if (args != null)
    {
        Console.WriteLine($"You sent me {args.Length} arguments.");
    }
}
```

Чтобы устранить обращение к свойству `Length` массива `string` в случае, когда он равен `null`, здесь используется условный оператор. Если вызывающий код не создаст массив данных и вызовет метод `TesterMethod()` примерно так, как показано ниже, то никаких ошибок во время выполнения не возникнет:

```
TesterMethod(null);
```

В текущей версии языка C# есть возможность задействовать *null-условную операцию* (знак вопроса, находящийся после типа переменной, но перед операцией доступа к члену), которая позволяет упростить представленную ранее проверку на предмет `null`. Вместо явного условного оператора, проверяющего на неравенство значению `null`, теперь можно написать такой код:

```
static void TesterMethod(string[] args)
{
```

```
// Мы должны проверять на предмет null перед доступом к данным массива!
Console.WriteLine($"You sent me {args?.Length} arguments.");
}
```

В этом случае условный оператор не применяется. Взамен к переменной массива `string` в качестве суффикса добавлена операция `?.` Если `args` окажется `null`, тогда обращение к свойству `Length` не приведет к ошибке во время выполнения. Чтобы вывести действительное значение, можно было бы воспользоваться операцией объединения с `null` и установить стандартное значение:

```
Console.WriteLine($"You sent me {args?.Length ?? 0} arguments.");
```

Существуют дополнительные области кодирования, в которых `null`-условная операция будет очень удобной, особенно при работе с делегатами и событиями. Тем не менее, данные темы рассматриваются позже, в главе 10, где вы найдете соответствующие сценарии применения.

Исходный код. Проект `NullableTypes` доступен в подкаталоге `Chapter_4`.

Кортежи (нововведение)

В завершение главы мы исследуем роль кортежей, используя финальный проект консольного приложения под названием `FunWithTuples`. Как упоминалось ранее в главе, одна из целей применения параметров `out` — получение более одного значения из вызова метода. Хотя прием определенно работает, в какой-то мере он представляет собой уловку. Гораздо лучше использовать конструкцию, которая специально предназначена для таких ситуаций.

Кортежи, которые являются легковесными структурами данных, содержащими множество полей, фактически появились в версии C# 6, но применяться могли в очень ограниченной манере. Поля не проверялись на предмет достоверности, нельзя было определять собственные методы и (возможно) наиболее важно то, что каждое свойство относилось к ссылочному типу, потенциально приводя к проблемам с памятью и производительностью.

В версии C# 7 кортежи вместо ссылочных типов используют новый тип данных `ValueTuple`, сберегая значительных объем памяти. Тип данных `ValueTuple` создает разные структуры на основе количества свойств для кортежа. Кроме того, в C# 7 каждому свойству кортежа можно назначать специфическое имя (подобно переменным), что значительно повышает удобство работы с ними.

Начало работы с кортежами

Итак, достаточно теории, давайте напишем какой-нибудь код! Чтобы создать кортеж, просто повестите значения, подлежащие присваиванию, в круглые скобки:

```
("a", 5, "c")
```

Обратите внимание, что все значения не обязаны относиться к тому же самому типу данных. Конструкция с круглыми скобками также применяется для присваивания кортежа переменной (или можно использовать ключевое слово `var` и тогда компилятор назначит типы данных самостоятельно). Показанные далее две строки кода делают одно и то же — присваивают предыдущий пример кортежа переменной. Переменная `values` будет кортежем с двумя свойствами `string` и одним свойством `int`.

```
(string, int, string) values = ("a", 5, "c");
var values = ("a", 5, "c");
```


На заметку! Если приведенный выше код не компилируется, тогда понадобится установить NuGet-пакет `System.ValueTuple`. Щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer, выберите в контекстном меню пункт `Manage NuGet Packages` (Управление пакетами NuGet) и в открывшемся окне `NuGet Package Manager` (Диспетчер пакетов NuGet) щелкните на кнопке `Browse` (Обзор), расположенной в левом верхнем углу. Затем введите `System.ValueTuple` в поле поиска и щелкните на кнопке `Install` (Установить). В случае применения редакции VS 2017 Community в среде Windows 10 установка поддержки кортежей не требуется.

По умолчанию компилятор назначает каждому свойству имя `ItemX`, где `X` представляет позицию свойства в кортеже, начиная с 1. В предыдущем примере свойства именуются как `Item1`, `Item2` и `Item3`. Доступ к ним осуществляется следующим образом:

```
Console.WriteLine($"First item: {values.Item1}");
Console.WriteLine($"Second item: {values.Item2}");
Console.WriteLine($"Third item: {values.Item3}");
```

К каждому свойству кортежа можно также добавлять специфическое имя либо в правой, либо в левой части оператора. Хотя назначение имен в обеих частях оператора не приводит к ошибке на этапе компиляции, имена в правой части игнорируются, а использоваться будут имена в левой части. Показанные ниже две строки кода демонстрируют установку имен в левой и правой частях оператора, давая тот же самый результат:

```
(string FirstLetter, int TheNumber, string SecondLetter)
    valuesWithNames = ("a", 5, "c");
var valuesWithNames2 = (FirstLetter: "a", TheNumber: 5, SecondLetter: "c");
```

Теперь доступ к свойствам кортежа возможен с применением имен полей, а также системы обозначений `ItemX`:

```
Console.WriteLine($"First item: {valuesWithNames.FirstLetter}");
Console.WriteLine($"Second item: {valuesWithNames.TheNumber}");
Console.WriteLine($"Third item: {valuesWithNames.SecondLetter}");
// Система обозначений ItemX по-прежнему работает!
Console.WriteLine($"First item: {valuesWithNames.Item1}");
Console.WriteLine($"Second item: {valuesWithNames.Item2}");
Console.WriteLine($"Third item: {valuesWithNames.Item3}");
```

Обратите внимание, что при назначении имен в правой части оператора должно использоваться ключевое слово `var`. Установка типов данных специальным образом (даже без специфических имен) заставляет компилятор применять синтаксис в левой части оператора, назначать свойствам имена согласно системе обозначений `ItemX` и игнорировать имена, указанные в правой части. В следующих двух операторах имена `Custom1` и `Custom2` игнорируются:

```
(int, int) example = (Custom1:5, Custom2:7);
(int Field1, int Field2) example = (Custom1:5, Custom2:7);
```

Важно также понимать, что специальные имена полей существуют только на этапе компиляции и не доступны при инспектировании кортежа во время выполнения с использованием рефлексии (рефлексия раскрывается в главе 15).

Выведение имен свойств кортежей (C# 7.1)

В C# 7.1 появилась возможность выводить имена свойств кортежей при определенных обстоятельствах. Однако чтобы это работало, должна быть включено применение версии C# 7.1.

Например, показанный ниже код первоначально приводит к ошибке на этапе компиляции, связанной с именами свойств кортежа (в последней строке):

```
Console.WriteLine("=> Inferred Tuple Names");
var foo = new {Prop1 = "first", Prop2 = "second"};
var bar = (foo.Prop1, foo.Prop2);
Console.WriteLine($"{bar.Prop1};{bar.Prop2}");
```

Наведите курсор на сообщение об ошибке и позвольте среде Visual Studio обновить проект для использования C# 7.1 (или модифицируйте файл проекта вручную, как объяснялось в главе 2). После обновления проекта имена свойств кортежа станут выводиться при его создании.

Кортежи как возвращаемые значения методов

Ранее в главе для возвращения из вызова метода более одного значения применялись параметры `out`. Для этого существуют другие способы вроде создания класса или структуры специально для возвращения значений. Но если такой класс или структура используется только в целях передачи данных для одного метода, тогда нет нужды выполнять излишнюю работу и писать добавочный код. Кортежи прекрасно подходят для решения задачи, т.к. они легковесны, просты в объявлении и несложны в применении.

Ниже представлен один из примеров, рассмотренных в разделе о параметрах `out`. Метод `FillTheseValues()` возвращает три значения, но требует использования в вызывающем коде трех параметров как механизма передачи.

```
static void FillTheseValues(out int a, out string b, out bool c)
{
    a = 9;
    b = "Enjoy your string.";
    c = true;
}
```

За счет применения кортежа от параметров можно избавиться и все равно получать обратно три значения:

```
static (int a, string b, bool c) FillTheseValues()
{
    return (9, "Enjoy your string.", true);
}
```

Вызывать новый метод не сложнее любого другого метода:

```
var samples = FillTheseValues();
Console.WriteLine($"Int is: {samples.a}");
Console.WriteLine($"String is: {samples.b}");
Console.WriteLine($"Boolean is: {samples.c}");
```

Возможно, даже лучшим примером будет разбор полного имени на отдельные части (имя, отчество, фамилия). Следующий метод `SplitNames()` получает полное имя и возвращает кортеж с составными частями:

```
static (string first, string middle, string last) SplitNames(string fullName)
{
    // Действия, необходимые для расщепления полного имени.
    return ("Philip", "F", "Japikse");
}
```

Использование отбрасывания с кортежами

Продолжим пример с методом `SplitNames()`. Пусть известно, что требуются только имя и фамилия, но не отчество. В таком случае можно указать имена свойств для значений, которые необходимо возвращать, а ненужные значения заменить заполнителем в виде подчеркивания (`_`):

```
var (first, _, last) = SplitNames("Philip F Japikse");
Console.WriteLine($"{first}:{last}");
```

Значение, соответствующее отчеству, в кортеже отбрасывается.

Деконструирование кортежей

Деконструирование является термином, описывающим отделение свойств кортежа друг от друга с целью применения по одному. Именно это делает метод `FillTheseValues()`. Но есть и другой случай использования такого приема — деконструирование специальных типов.

Возьмем укороченную версию структуры `Point`, которая применялась ранее в главе. В нее был добавлен новый метод по имени `Deconstruct()`, возвращающий индивидуальные свойства экземпляра `Point` в виде кортежа со свойствами `XPos` и `YPos`.

```
struct Point
{
    // Поля структуры.
    public int X;
    public int Y;
    // Специальный конструктор.
    public Point(int XPos, int YPos)
    {
        X = XPos;
        Y = YPos;
    }
    public (int XPos, int YPos) Deconstruct() => (X, Y);
}
```

Новый метод `Deconstruct()` выделен полужирным. Его можно именовать как угодно, но обычно он имеет имя `Deconstruct()`. В результате с помощью единственного вызова метода можно получить индивидуальные значения структуры путем возвращения кортежа:

```
Point p = new Point(7, 5);
var pointValues = p.Deconstruct();
Console.WriteLine($"X is: {pointValues.XPos}");
Console.WriteLine($"Y is: {pointValues.YPos}");
```

Исходный код. Проект `FunWithTuples` доступен в подкаталоге `Chapter_4`.

На этом первоначальное знакомство с языком программирования C# завершено. В главе 5 вы начнете погружаться в детали объектно-ориентированной разработки.

Резюме

Глава начиналась с исследования массивов. Затем мы обсудили ключевые слова C#, которые позволяют строить специальные методы. Вспомните, что по умолчанию параметры передаются по значению; тем не менее, параметры можно передавать и по ссылке, пометив их модификаторами `ref` или `out`. Кроме того, вы узнали о роли необязательных и именованных параметров, а также о том, как определять и вызывать методы, принимающие массивы параметров.

После рассмотрения темы перегрузки методов в главе приводились подробные сведения, касающиеся способов определения перечислений и структур в C# и их представления в библиотеках базовых классов .NET. Попутно были исследованы основные характеристики типов значений и ссылочных типов, включая их поведение при передаче в качестве параметров методам, а также способы взаимодействия с типами данных, допускающими `null`, и переменными, которые могут иметь значение `null` (например, переменными ссылочных типов и переменными типов значений, допускающих `null`), с использованием операций `?` и `??`. Финальный раздел был посвящен давно ожидаемому средству в языке C# — кортежам. После выяснения, что они собой представляют и как работают, кортежи применялись для возвращения множества значений из методов и для деконструирования специальных типов.

часть III

Объектно-ориентированное программирование на C#

В этой части

Глава 5. Инкапсуляция

Глава 6. Наследование и полиморфизм

Глава 7. Структурированная обработка исключений

Глава 8. Работа с интерфейсами

ГЛАВА 5

Инкапсуляция

В главах 3 и 4 мы исследовали несколько основных синтаксических конструкций, присущих любому приложению .NET, которое вам придется разрабатывать. Начиная с данной главы, мы приступаем к изучению объектно-ориентированных возможностей C#. Первым, что вам предстоит узнать, будет процесс построения четко определенных типов классов, которые поддерживают любое количество *конструкторов*. После ознакомления с основами определения классов и размещения объектов мы посвятим остаток главы теме *инкапсуляции*. В ходе изложения вы научитесь определять свойства классов, а также получите подробные сведения о ключевом слове `static`, синтаксисе инициализации объектов, полях только для чтения, константных данных и частичных классах.

Знакомство с типом класса C#

С точки зрения платформы .NET наиболее фундаментальной программной конструкцией является *тип класса*. Формально класс — это определяемый пользователем тип, состоящий из полей данных (часто называемых *переменными-членами*) и членов, которые оперируют полями данных (к ним относятся конструкторы, свойства, методы, события и т.д.). Коллективно набор полей данных представляет “состояние” экземпляра класса (по-другому называемого *объектом*). Мощь объектно-ориентированных языков, таких как C#, заключается в том, что за счет группирования данных и связанной с ними функциональности в унифицированное определение класса вы получаете возможность моделировать свое программное обеспечение в соответствии с сущностями реального мира.

Для начала создадим новый проект консольного приложения C# по имени `SimpleClassExample`. Затем добавим в проект новый файл класса (`Car.cs`), используя пункт меню `Project⇒Add Class` (Проект⇒Добавить класс). В открывшемся диалоговом окне понадобится выбрать значок `Class` (Класс), как показано на рис. 5.1, и щелкнуть на кнопке `Add` (Добавить).

Класс определяется в C# с применением ключевого слова `class`. Вот как выглядит простейшее из возможных объявление класса:

```
class Car
{
}
```

После определения типа класса необходимо определить набор переменных-членов, которые будут использоваться для представления его состояния.

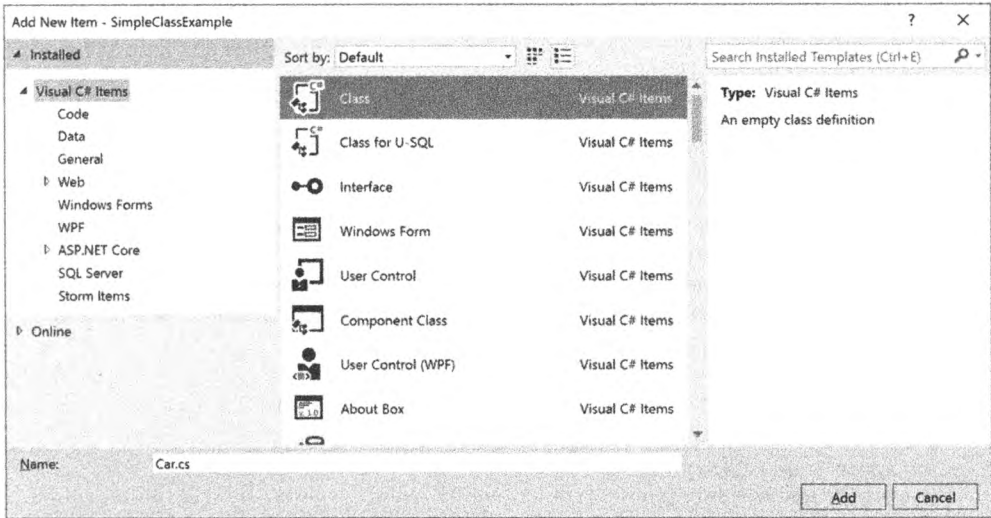


Рис. 5.1. Добавление нового типа класса C#

Например, вы можете принять решение, что объекты *Car* (автомобили) должны иметь поле данных типа *int*, представляющее текущую скорость, и поле данных типа *string* для представления дружественного названия автомобиля. С учетом таких начальных проектных положений класс *Car* будет выглядеть следующим образом:

```
class Car
{
    // 'Состояние' объекта Car.
    public string petName;
    public int currSpeed;
}
```

Обратите внимание, что переменные-члены объявлены с применением модификатора доступа *public*. Открытые (*public*) члены класса доступны напрямую после того, как создан объект этого типа. Вспомните, что термин *объект* используется для описания экземпляра заданного типа класса, который создан с помощью ключевого слова *new*.

На заметку! Поля данных класса редко (если вообще когда-нибудь) должны определяться как открытые. Чтобы обеспечить целостность данных состояния, намного лучше объявлять данные закрытыми (*private*) или возможно защищенными (*protected*) и разрешать контролируемый доступ к данным через свойства (как будет показано далее в главе). Однако для максимального упрощения первого примера мы определили поля данных как открытые.

После определения набора переменных-членов, представляющих состояние класса, следующим шагом в проектировании будет установка членов, которые моделируют его поведение. Для этого примера в классе *Car* определены методы по имени *SpeedUp()* и *PrintState()*. Модифицируйте код класса, как показано ниже:

```
class Car
{
    // 'Состояние' объекта Car.
    public string petName;
    public int currSpeed;
```

```
// Функциональность Car.
// Использовать синтаксис членов, сжатых до выражений, который появился в C# 6.
public void PrintState()
    => Console.WriteLine("{0} is going {1} MPH.", petName, currSpeed);

public void SpeedUp(int delta)
    => currSpeed += delta;
}
```

Метод `PrintState()` — простая диагностическая функция, которая выводит текущее состояние объекта `Car` в окно командной строки. Метод `SpeedUp()` увеличивает скорость автомобиля, представляемого объектом `Car`, на величину, которая передается во входном параметре типа `int`. Обновите код метода `Main()` следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    // Разместить в памяти и сконфигурировать объект Car.
    Car myCar = new Car();
    myCar.petName = "Henry";
    myCar.currSpeed = 10;

    // Увеличить скорость автомобиля в несколько раз и вывести новое состояние.
    for (int i = 0; i <= 10; i++)
    {
        myCar.SpeedUp(5);
        myCar.PrintState();
    }
    Console.ReadLine();
}
```

Запустив программу, вы увидите, что переменная `Car` (`myCar`) поддерживает свое текущее состояние на протяжении жизни приложения:

```
***** Fun with Class Types *****
Henry is going 15 MPH.
Henry is going 20 MPH.
Henry is going 25 MPH.
Henry is going 30 MPH.
Henry is going 35 MPH.
Henry is going 40 MPH.
Henry is going 45 MPH.
Henry is going 50 MPH.
Henry is going 55 MPH.
Henry is going 60 MPH.
Henry is going 65 MPH.
```

Размещение объектов с помощью ключевого слова `new`

Как было показано в предыдущем примере кода, объекты должны размещаться в памяти с применением ключевого слова `new`. Если вы не укажете ключевое слово `new` и попытаетесь использовать переменную класса в последующем операторе кода, то получите ошибку на этапе компиляции. Например, приведенный ниже метод `Main()` не скомпилируется:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
```

```
// Ошибка на этапе компиляции! Забыли использовать new для создания объекта!
Car myCar;
myCar.petName = "Fred";
}
```

Чтобы корректно создать объект с применением ключевого слова `new`, можно определить и разместить в памяти объект `Car` в одной строке кода:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    Car myCar = new Car();
    myCar.petName = "Fred";
}
```

В качестве альтернативы определение и размещение в памяти экземпляра класса может осуществляться в отдельных строках кода:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    Car myCar;
    myCar = new Car();
    myCar.petName = "Fred";
}
```

Здесь первый оператор кода просто объявляет ссылку на определяемый объект типа `Car`. Ссылка будет указывать на действительный объект в памяти только после ее явного присваивания.

В любом случае к настоящему моменту мы имеем простейший класс, в котором определено несколько элементов данных и ряд базовых операций. Чтобы расширить функциональность текущего класса `Car`, необходимо разобраться с ролью *конструкторов*.

Понятие конструкторов

Учитывая наличие у объекта состояния (представленного значениями его переменных-членов), обычно желательно присвоить подходящие значения полям объекта перед тем, как работать с ним. В настоящее время класс `Car` требует присваивания значений полям `petName` и `currSpeed` по отдельности. Для текущего примера такое действие не слишком проблематично, поскольку открытых элементов данных всего два. Тем не менее, зачастую класс содержит несколько десятков полей, с которыми надо что-то делать. Ясно, что было бы нежелательно писать 20 операторов инициализации для всех 20 элементов данных.

К счастью, язык `C#` поддерживает использование *конструкторов*, которые позволяют устанавливать состояние объекта в момент его создания. Конструктор — это специальный метод класса, который неявно вызывается при создании объекта с применением ключевого слова `new`. Однако в отличие от “нормального” метода конструктор никогда не имеет возвращаемого значения (даже `void`) и всегда именуется идентично имени класса, объекты которого он конструирует.

Роль стандартного конструктора

Каждый класс `C#` снабжается “бесплатным” *стандартным конструктором*, который в случае необходимости может быть переопределен. По определению стандартный конструктор никогда не принимает аргументов. После размещения нового объекта в памяти стандартный конструктор гарантирует установку всех полей данных в соответс-

твующие стандартные значения (стандартные значения для типов данных C# были описаны в главе 3).

Если вас не устраивают такие стандартные присваивания, тогда можете переопределить стандартный конструктор в соответствии со своими нуждами. В целях иллюстрации модифицируем класс C# следующим образом:

```
class Car
{
    // 'Состояние' объекта Car.
    public string petName;
    public int currSpeed;

    // Специальный стандартный конструктор.
    public Car()
    {
        petName = "Chuck";
        currSpeed = 10;
    }
    ...
}
```

В данном случае мы заставляем объекты Car начинать свое существование под именем Chuck и со скоростью 10 миль в час. Создать объект Car со стандартными значениями можно так:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    // Вызов стандартного конструктора.
    Car chuck = new Car();
    // Выводит строку "Chuck is going 10 MPH."
    chuck.PrintState();
    ...
}
```

Определение специальных конструкторов

Обычно помимо стандартного конструктора в классах определяются дополнительные конструкторы. Тем самым пользователю объекта предоставляется простой и согласованный способ инициализации состояния объекта прямо во время его создания. Взгляните на следующее изменение класса Car, который теперь поддерживает в совокупности три конструктора:

```
class Car
{
    // 'Состояние' объекта Car.
    public string petName;
    public int currSpeed;

    // Специальный стандартный конструктор.
    public Car()
    {
        petName = "Chuck";
        currSpeed = 10;
    }

    // Здесь currSpeed получает стандартное значение для типа int (0).
    public Car(string pn)
```

```

{
    petName = pn;
}

// Позволяет вызывающему коду установить полное состояние объекта Car.
public Car(string pn, int cs)
{
    petName = pn;
    currSpeed = cs;
}
...
}

```

Имейте в виду, что один конструктор отличается от другого (с точки зрения компилятора C#) числом и/или типами аргументов. Вспомните из главы 4, что определение метода с тем же самым именем, но разным количеством или типами аргументов, называется *перегрузкой* метода. Таким образом, конструктор класса Car перегружен, чтобы предложить несколько способов создания объекта во время объявления. В любом случае теперь есть возможность создавать объекты Car, используя любой из его открытых конструкторов. Вот пример:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    // Создать объект Car по имени Chuck со скоростью 10 миль в час.
    Car chuck = new Car();
    chuck.PrintState();

    // Создать объект Car по имени Mary со скоростью 0 миль в час.
    Car mary = new Car("Mary");
    mary.PrintState();

    // Создать объект Car по имени Daisy со скоростью 75 миль в час.
    Car daisy = new Car("Daisy", 75);
    daisy.PrintState();
    ...
}

```

Конструкторы как члены, сжатые до выражений (нововведение)

Версия языка C# 7 базируется на стиле членов, сжатых до выражений, из C# 6 и добавляет дополнительные случаи употребления для нового стиля. Новый синтаксис теперь применим к конструкторам, финализаторам, а также к средствам доступа get/set для свойств и индексов. С учетом сказанного предыдущий конструктор можно переписать следующим образом:

```

// Здесь currSpeed получит стандартное
// значение для типа int (0).
public Car(string pn) => petName = pn;

```

Второй конструктор не подходит, т.к. члены, сжатые до выражений, предназначены для однострочных методов.

Еще раз о стандартном конструкторе

Как вы только что узнали, все классы снабжаются стандартным конструктором. Следовательно, если добавить в текущий проект новый класс по имени Motorcycle, определенный примерно так:

```
class Motorcycle
{
    public void PopAWheely()
    {
        Console.WriteLine("Yeeeeeeee Haaaaaeewww!");
    }
}
```

то сразу появится возможность создавать экземпляры `Motorcycle` с помощью стандартного конструктора:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    Motorcycle mc = new Motorcycle();
    mc.PopAWheely();
    ...
}
```

Тем не менее, как только определен специальный конструктор с любым числом параметров, стандартный конструктор молча удаляется из класса и перестает быть доступным. Воспринимайте это так: если вы не определили специальный конструктор, тогда компилятор C# снабжает класс стандартным конструктором, давая возможность пользователю размещать в памяти экземпляр вашего класса с набором полей данных, которые установлены в корректные стандартные значения. Однако когда вы определяете уникальный конструктор, то компилятор предполагает, что вы решили взять власть в свои руки.

Следовательно, если вы хотите позволить пользователю создавать экземпляр вашего типа с помощью стандартного конструктора, а также специального конструктора, то должны явно переопределить стандартный конструктор. Важно понимать, что в подавляющем большинстве случаев реализация стандартного конструктора класса намеренно оставляется пустой, т.к. требуется только создание объекта со стандартными значениями. Изменим класс `Motorcycle`:

```
class Motorcycle
{
    public int driverIntensity;
    public void PopAWheely()
    {
        for (int i = 0; i <= driverIntensity; i++)
        {
            Console.WriteLine("Yeeeeeeee Haaaaaeewww!");
        }
    }

    // Вернуть стандартный конструктор, который будет
    // устанавливать все члены данных в стандартные значения.
    public Motorcycle() {}

    // Специальный конструктор.
    public Motorcycle(int intensity)
    {
        driverIntensity = intensity;
    }
}
```

На заметку! Теперь, когда вы лучше понимаете роль конструкторов класса, полезно узнать об одном удобном сокращении. В IDE-среде Visual Studio предлагается фрагмент кода `ctor`. Если вы наберете `ctor` и два раза нажмете клавишу `<Tab>`, тогда IDE-среда автоматически определит специальный стандартный конструктор. Затем можно добавить нужные параметры и логику реализации. Испытайте такой прием.

Роль ключевого слова `this`

В языке C# имеется ключевое слово `this`, которое обеспечивает доступ к текущему экземпляру класса. Один из возможных сценариев использования `this` предусматривает устранение неоднозначности с областью видимости, которая может возникнуть, когда входной параметр имеет такое же имя, как и поле данных класса. Разумеется, вы могли бы просто придерживаться соглашения об именовании, которое не приводит к такой неоднозначности; тем не менее, чтобы проиллюстрировать такой сценарий, добавим в класс `Motorcycle` новое поле типа `string` (под названием `name`), предназначенное для представления имени водителя. Затем добавим метод `SetDriverName()` со следующей реализацией:

```
class Motorcycle
{
    public int driverIntensity;

    // Новые члены для представления имени водителя.
    public string name;
    public void SetDriverName(string name)
    {
        name = name;
    }
    ...
}
```

Хотя приведенный код нормально скомпилируется, Visual Studio отобразит сообщение с предупреждением о том, что переменная присваивается сама себе! В целях иллюстрации добавим в метод `Main()` вызов `SetDriverName()` и выведем значение поля `name`. Вы можете быть удивлены, обнаружив, что значением поля `name` является пустая строка!

```
// Создать объект Motorcycle с мотоциклистом по имени Tiny?
Motorcycle c = new Motorcycle(5);
c.SetDriverName("Tiny");
c.PopAWheely();
Console.WriteLine("Rider name is {0}", c.name); // Выводит пустое значение name!
```

Проблема в том, что реализация метода `SetDriverName()` присваивает входному параметру значение *его самого*, т.к. компилятор предполагает, что `name` ссылается на переменную, находящуюся в области действия метода, а не на поле `name` из области видимости класса. Для информирования компилятора о том, что необходимо установить поле данных `name` текущего объекта в значение входного параметра `name`, просто используйте ключевое слово `this`, чтобы устранить такую неоднозначность:

```
public void SetDriverName(string name)
{
    this.name = name;
}
```

Имейте в виду, что если неоднозначность отсутствует, тогда применять ключевое слово `this` для доступа класса к собственным данным или членам вовсе не обязательно. Например, если вы переименуете член данных типа `string` с `name` на `driverName` (что также повлечет за собой модификацию кода в методе `Main()`), то потребность в использовании `this` отпадет, поскольку неоднозначности с областью видимости больше нет:

```
class Motorcycle
{
    public int driverIntensity;
    public string driverName;

    public void SetDriverName(string name)
    {
        // Эти два оператора функционально эквивалентны.
        driverName = name;
        this.driverName = name;
    }
    ...
}
```

Несмотря на то что применение ключевого слова `this` в неоднозначных ситуациях дает не особенно большой выигрыш, вы можете счесть его удобным при реализации членов класса, т.к. IDE-среды, подобные Visual Studio, будут активизировать средство IntelliSense, когда присутствует `this`. Это может оказаться полезным, если вы забыли имя члена класса и хотите быстро вспомнить его определение. Взгляните на рис. 5.2.



Рис. 5.2. Активизация средства IntelliSense для `this`

Построение цепочки вызовов конструкторов с использованием `this`

Еще один сценарий применения ключевого слова `this` касается проектирования класса с использованием приема, который называется *построением цепочки конструкторов*. Такой паттерн проектирования полезен при наличии класса, определяющего множество конструкторов. Учитывая тот факт, что конструкторы нередко проверяют входные аргументы на соблюдение разнообразных бизнес-правил, довольно часто внутри набора конструкторов обнаруживается избыточная логика проверки достоверности.

Рассмотрим следующее измененное определение класса `Motorcycle`:

```
class Motorcycle
{
    public int driverIntensity;
    public string driverName;

    public Motorcycle() { }

    // Избыточная логика конструктора!
    public Motorcycle(int intensity)
    {
        if (intensity > 10)
        {
            intensity = 10;
        }
        driverIntensity = intensity;
    }

    public Motorcycle(int intensity, string name)
    {
        if (intensity > 10)
        {
            intensity = 10;
        }
        driverIntensity = intensity;
        driverName = name;
    }
    ...
}
```

Здесь (возможно в попытке обеспечить безопасность мотоциклиста) внутри каждого конструктора производится проверка того, что уровень мощности не превышает значения 10. Наряду с тем, что это правильно, в двух конструкторах присутствует избыточный код. Подход далек от идеала, поскольку в случае изменения правил (например, если уровень мощности не должен превышать значение 5 вместо 10) код придется модифицировать в нескольких местах.

Один из способов улучшить создавшуюся ситуацию предусматривает определение в классе `Motorcycle` метода, который будет выполнять проверку входных аргументов. Если вы решите поступить так, тогда каждый конструктор сможет вызывать такой метод перед присваиванием значений полям. Хотя описанный подход позволяет изолировать код, который придется обновлять при изменении бизнес-правил, теперь появилась другая избыточность:

```
class Motorcycle
{
    public int driverIntensity;
    public string driverName;

    // Конструкторы.
    public Motorcycle() { }

    public Motorcycle(int intensity)
    {
        SetIntensity(intensity);
    }

    public Motorcycle(int intensity, string name)
    {
        SetIntensity(intensity);
```

```

        driverName = name;
    }
    public void SetIntensity(int intensity)
    {
        if (intensity > 10)
        {
            intensity = 10;
        }
        driverIntensity = intensity;
    }
    ...
}

```

Более совершенный подход предполагает назначение конструктора, который принимает *наибольшее количество аргументов*, в качестве “главного конструктора” и выполнение требуемой логики проверки достоверности внутри его реализации. Остальные конструкторы могут применять ключевое слово `this` для передачи входных аргументов главному конструктору и при необходимости предоставлять любые дополнительные параметры. В таком случае вам придется беспокоиться только о поддержке единственного конструктора для всего класса, в то время как оставшиеся конструкторы будут в основном пустыми.

Ниже представлена финальная реализация класса `Motorcycle` (с одним дополнительным конструктором в целях иллюстрации). При связывании конструкторов в цепочку обратите внимание, что ключевое слово `this` располагается за пределами самого конструктора и отделяется от его объявления двоеточием:

```

class Motorcycle
{
    public int driverIntensity;
    public string driverName;

    // Связывание конструкторов в цепочку.
    public Motorcycle() {}
    public Motorcycle(int intensity)
        : this(intensity, "") {}
    public Motorcycle(string name)
        : this(0, name) {}

    // Это 'главный' конструктор, выполняющий всю реальную работу.
    public Motorcycle(int intensity, string name)
    {
        if (intensity > 10)
        {
            intensity = 10;
        }
        driverIntensity = intensity;
        driverName = name;
    }
    ...
}

```

Имейте в виду, что использовать ключевое слово `this` для связывания вызовов конструкторов в цепочку вовсе не обязательно. Однако такой подход позволяет получить лучше сопровождаемое и более краткое определение класса. Применяя данный прием, также можно упростить решение задач программирования, потому что реальная работа делегируется единственному конструктору (обычно принимающему большую часть параметров), тогда как остальные просто “перекладывают на него ответственность”.

На заметку! Вспомните из главы 4, что в языке C# поддерживаются необязательные параметры. Если вы будете использовать в конструкторах своих классов необязательные параметры, то сможете добиться тех же преимуществ, что и при связывании конструкторов в цепочку, но со значительно меньшим объемом кода. Вскоре вы увидите, как это делается.

Изучение потока управления конструкторов

Напоследок отметим, что как только конструктор передал аргументы выделенному главному конструктору (и главный конструктор обработал данные), первоначально вызванный конструктор продолжит выполнение всех оставшихся операторов кода. В целях прояснения модифицируем конструкторы класса `Motorcycle`, добавив в них вызов метода `Console.WriteLine()`:

```
class Motorcycle
{
    public int driverIntensity;
    public string driverName;

    // Связывание конструкторов в цепочку.
    public Motorcycle()
    {
        Console.WriteLine("In default ctor");
        // Внутри стандартного конструктора
    }

    public Motorcycle(int intensity)
        : this(intensity, "")
    {
        Console.WriteLine("In ctor taking an int");
        // Внутри конструктора, принимающего int
    }

    public Motorcycle(string name)
        : this(0, name)
    {
        Console.WriteLine("In ctor taking a string");
        // Внутри конструктора, принимающего string
    }

    // Это 'главный' конструктор, выполняющий всю реальную работу.
    public Motorcycle(int intensity, string name)
    {
        Console.WriteLine("In master ctor ");
        // Внутри главного конструктора
        if (intensity > 10)
        {
            intensity = 10;
        }
        driverIntensity = intensity;
        driverName = name;
    }
    ...
}
```

Теперь изменим метод `Main()`, чтобы он работал с объектом `Motorcycle` следующим образом:


```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with class Types *****\n");
    // Создать объект Motorcycle.
    Motorcycle c = new Motorcycle(5);
    c.SetDriverName("Tiny");
    c.PopAWheely();
    Console.WriteLine("Rider name is {0}", c.driverName); // вывод имени гонщика
    Console.ReadLine();
}
```

Взгляните на вывод, полученный из показанного выше метода Main():

```
***** Fun with class Types *****

In master ctor
In ctor taking an int
Yeeeeeeee Haaaaaeewww!
Yeeeeeeee Haaaaaeewww!
Yeeeeeeee Haaaaaeewww!
Yeeeeeeee Haaaaaeewww!
Yeeeeeeee Haaaaaeewww!
Yeeeeeeee Haaaaaeewww!
Rider name is Tiny
```

Ниже описан поток логики конструкторов.

- Прежде всего, создается объект путем вызова конструктора, принимающего один аргумент типа `int`.
- Этот конструктор передает полученные данные главному конструктору и предоставляет любые дополнительные начальные аргументы, не указанные вызывающим кодом.
- Главный конструктор присваивает входные данные полям данных объекта.
- Управление возвращается первоначально вызванному конструктору, который выполняет оставшиеся операторы кода.

В построении цепочек конструкторов примечательно то, что данный шаблон программирования будет работать с любой версией языка C# и платформой .NET. Тем не менее, если целевой платформой является .NET 4.0 или последующая версия, то решение задач можно дополнительно упростить, применяя необязательные аргументы в качестве альтернативы построению традиционных цепочек конструкторов.

Еще раз о необязательных аргументах

В главе 4 вы изучили необязательные и именованные аргументы. Вспомните, что необязательные аргументы позволяют определять стандартные значения для входных аргументов. Если вызывающий код устраивают стандартные значения, то указывать уникальные значения не обязательно, но это нужно делать, чтобы снабдить объект специальными данными. Рассмотрим следующую версию класса `Motorcycle`, которая теперь предлагает несколько возможностей конструирования объектов, используя *единственное* определение конструктора:

```
class Motorcycle
{
    // Единственный конструктор, использующий необязательные аргументы.
    public Motorcycle(int intensity = 0, string name = "")
    {
```

```

    if (intensity > 10)
    {
        intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
}
...
}

```

С помощью такого единственного конструктора можно создавать объект `Motorcycle`, указывая ноль, один или два аргумента. Вспомните, что синтаксис именованных аргументов по существу позволяет пропускать подходящие стандартные установки (см. главу 4).

```

static void MakeSomeBikes()
{
    // driverName = "", driverIntensity = 0
    Motorcycle m1 = new Motorcycle();
    Console.WriteLine("Name= {0}, Intensity= {1}",
        m1.driverName, m1.driverIntensity);

    // driverName = "Tiny", driverIntensity = 0
    Motorcycle m2 = new Motorcycle(name: "Tiny");
    Console.WriteLine("Name= {0}, Intensity= {1}",
        m2.driverName, m2.driverIntensity);

    // driverName = "", driverIntensity = 7
    Motorcycle m3 = new Motorcycle(7);
    Console.WriteLine("Name= {0}, Intensity= {1}",
        m3.driverName, m3.driverIntensity);
}

```

В любом случае к настоящему моменту вы способны определить класс с полями данными (т.е. переменными-членами) и разнообразными операциями, такими как методы и конструкторы. Теперь давайте формализуем роль ключевого слова `static`.

Исходный код. Проект `SimpleClassExample` доступен в подкаталоге `Chapter_5`.

Понятие ключевого слова `static`

Класс C# может определять любое количество *статических членов*, которые объявляются с применением ключевого слова `static`. В таком случае интересующий член должен вызываться прямо на уровне класса, а не через переменную со ссылкой на объект. Чтобы проиллюстрировать разницу, обратимся к нашему старому знакомому классу `System.Console`. Как вы уже видели, метод `WriteLine()` не вызывается на уровне объекта:

```

// Ошибка на этапе компиляции! WriteLine() - не метод уровня объекта!
Console c = new Console();
c.WriteLine("I can't be printed...");

```

Взамен статический член `WriteLine()` предваряется именем класса:

```

// Правильно! WriteLine() - статический метод.
Console.WriteLine("Much better! Thanks...");

```

Попросту говоря, статические члены — это элементы, которые проектировщик класса посчитал настолько общими, что перед обращением к ним даже нет нужды создавать экземпляр класса. Наряду с тем, что определять статические члены можно в любом классе, чаще всего они обнаруживаются внутри *обслуживающих классов*. По определению обслуживающий класс представляет собой такой класс, который не поддерживает какое-либо состояние на уровне объектов и не предполагает создание своих экземпляров с помощью ключевого слова `new`. Взамен обслуживающий класс открывает доступ ко всей функциональности посредством членов уровня класса (также известных под названием статических).

Например, если бы вы воспользовались браузером объектов Visual Studio (выбрав пункт меню View⇒Object Browser (Вид⇒Браузер объектов)) для просмотра пространства имен `System` из сборки `mscorlib.dll`, то увидели бы, что все члены классов `Console`, `Math`, `Environment` и `GC` (среди прочих) открывают доступ к своей функциональности через статические члены. Они являются лишь несколькими обслуживающими классами, которые можно найти в библиотеках базовых классов .NET.

И снова следует отметить, что статические члены находятся не только в обслуживающих классах; они могут быть частью в принципе любого определения класса. Просто запомните, что статические члены продвигают отдельный элемент на уровень класса вместо уровня объектов. Как будет показано в нескольких последующих разделах, ключевое слово `static` может применяться к перечисленным ниже конструкциям:

- данные класса;
- методы класса;
- свойства класса;
- конструктор;
- полное определение класса;
- в сочетании с ключевым словом `using`.

Давайте рассмотрим все варианты, начав с концепции статических данных.

На заметку! Роль статических свойств будет объясняться позже в главе во время исследования самих свойств.

Определение статических полей данных

При проектировании класса в большинстве случаев данные определяются на уровне экземпляра — другими словами, как нестатические данные. Когда определяются данные уровня экземпляра, то известно, что каждый создаваемый новый объект поддерживает собственную независимую копию этих данных. По контрасту при определении *статических* данных класса выделенная под них память разделяется всеми объектами этой категории.

Чтобы увидеть разницу, создадим новый проект консольного приложения под названием `StaticDataAndMembers` и добавим в него новый класс по имени `SavingsAccount`. Начнем с определения элемента данных уровня экземпляра (для моделирования текущего баланса) и специального конструктора для установки начального баланса:

```
// Простой класс депозитного счета.
class SavingsAccount
{
    // Данные уровня экземпляра.
    public double currBalance;
```

```

public SavingsAccount(double balance)
{
    currBalance = balance;
}
}

```

При создании объектов `SavingsAccount` память под поле `currBalance` выделяется для каждого объекта. Таким образом, можно было бы создать пять разных объектов `SavingsAccount`, каждый с собственным уникальным балансом. Более того, в случае изменения баланса в одном объекте счета другие объекты не затрагиваются.

С другой стороны, память под статические данные распределяется однажды и разделяется всеми объектами того же самого класса. Добавьте в класс `SavingsAccount` статический элемент данных по имени `currInterestRate`, который устанавливается в стандартное значение 0.04:

```

// Простой класс депозитного счета.
class SavingsAccount
{
    // Данные уровня экземпляра.
    public double currBalance;
    // Статический элемент данных.
    public static double currInterestRate = 0.04;
    public SavingsAccount(double balance)
    {
        currBalance = balance;
    }
}

```

После создания трех экземпляров класса `SavingsAccount`, как показано ниже, размещение данных в памяти будет выглядеть примерно так, как иллюстрируется на рис. 5.3:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Static Data *****\n");
    SavingsAccount s1 = new SavingsAccount(50);
    SavingsAccount s2 = new SavingsAccount(100);
    SavingsAccount s3 = new SavingsAccount(10000.75);
    Console.ReadLine();
}

```

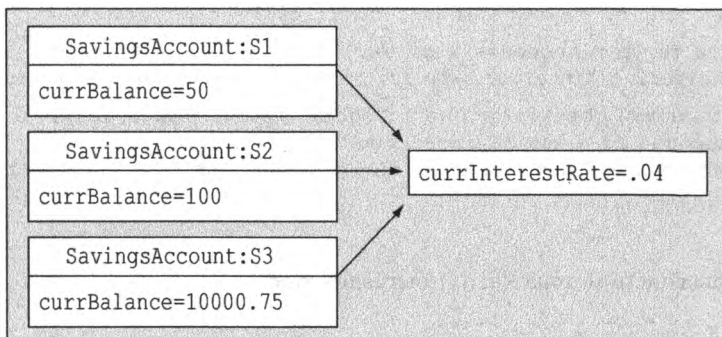


Рис. 5.3. Память под статические данные распределяется один раз и разделяется между всеми экземплярами класса

Здесь предполагается, что все депозитные счета должны иметь одну и ту же процентную ставку. Поскольку статические данные разделяются всеми объектами той же самой категории, если вы измените процентную ставку каким-либо образом, тогда все объекты будут "видеть" новое значение при следующем доступе к статическим данным, т.к. все они по существу просматривают одну и ту же ячейку памяти. Чтобы понять, как изменять (или получать) статические данные, понадобится рассмотреть роль статических методов.

Определение статических методов

Модифицируем класс `SavingsAccount`, определив в нем два статических метода. Первый статический метод (`GetInterestRate()`) будет возвращать текущую процентную ставку, а второй (`SetInterestRate()`) позволит изменять эту процентную ставку:

```
// Простой класс депозитного счета.
class SavingsAccount
{
    // Данные уровня экземпляра.
    public double currBalance;

    // Статический элемент данных.
    public static double currInterestRate = 0.04;

    public SavingsAccount(double balance)
    {
        currBalance = balance;
    }

    // Статические члены для установки/получения процентной ставки.
    public static void SetInterestRate(double newRate)
    { currInterestRate = newRate; }

    public static double GetInterestRate()
    { return currInterestRate; }
}
```

Рассмотрим показанный ниже сценарий использования класса:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Static Data *****\n");
    SavingsAccount s1 = new SavingsAccount(50);
    SavingsAccount s2 = new SavingsAccount(100);

    // Вывести текущую процентную ставку.
    Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());

    // Создать новый объект; это не 'сбросит' процентную ставку.
    SavingsAccount s3 = new SavingsAccount(10000.75);
    Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());

    Console.ReadLine();
}
```

Вывод предыдущего метода `Main()` выглядит так:

```
***** Fun with Static Data *****
Interest Rate is: 0.04
Interest Rate is: 0.04
```

Как видите, при создании новых экземпляров класса `SavingsAccount` значение статических данных не сбрасывается, поскольку среда CLR выделяет для них место в памяти только один раз. Затем все объекты типа `SavingsAccount` имеют дело с одним и тем же значением в статическом поле `currInterestRate`.

Когда проектируется любой класс C#, одна из задач связана с выяснением того, какие порции данных должны быть определены как статические члены, а какие — нет. Хотя строгих правил не существует, запомните, что поле статических данных разделяется между всеми объектами конкретного класса. Поэтому, если необходимо, чтобы часть данных совместно использовалась всеми объектами, то статические члены будут самым подходящим вариантом.

Посмотрим, что произойдет, если поле `currInterestRate` не определено с ключевым словом `static`. Это означает, что каждый объект `SavingsAccount` будет иметь собственную копию поля `currInterestRate`. Предположим, что вы создали сто объектов `SavingsAccount` и нуждаетесь в изменении размера процентной ставки. Такое действие потребовало бы вызова метода `SetInterestRate()` сто раз! Ясно, что подобный способ моделирования "разделяемых данных" трудно считать удобным. Статические данные безупречны, когда есть значение, которое должно быть общим для всех объектов заданной категории.

На заметку! Ссылка на нестатические члены внутри реализации статического члена приводит к ошибке на этапе компиляции. В качестве связанного замечания: ошибкой также будет применение ключевого слова `this` к статическому члену, потому что `this` подразумевает объект!

Определение статических конструкторов

Типичный конструктор используется для установки значений данных уровня экземпляра во время его создания. Однако что произойдет, если вы попытаетесь присвоить значение статическому элементу данных в типичном конструкторе? Вы можете быть удивлены, обнаружив, что значение сбрасывается каждый раз, когда создается новый объект!

В целях иллюстрации изменим код конструктора класса `SavingsAccount`, как показано ниже (также обратите внимание, что поле `currInterestRate` больше не устанавливается при объявлении):

```
class SavingsAccount
{
    public double currBalance;
    public static double currInterestRate;

    // Обратите внимание, что наш конструктор устанавливает
    // значение статического поля currInterestRate.
    public SavingsAccount(double balance)
    {
        currInterestRate = 0.04; // Это статические данные!
        currBalance = balance;
    }
    ...
}
```

Теперь поместим в метод `Main()` следующий код:

```
static void Main( string[] args )
{
    Console.WriteLine("***** Fun with Static Data *****\n");
```

```
// Создать объект счета.
SavingsAccount s1 = new SavingsAccount(50);

// Вывести текущую процентную ставку.
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());

// Попытаться изменить процентную ставку через свойство.
SavingsAccount.SetInterestRate(0.08);

// Создать второй объект счета.
SavingsAccount s2 = new SavingsAccount(100);

// Должно быть выведено 0.08, не так ли?
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());
Console.ReadLine();
}
```

При выполнении этого метода Main() вы увидите, что переменная currInterestRate сбрасывается каждый раз, когда создается новый объект SavingsAccount, и она всегда установлена в 0.04. Очевидно, что установка значений статических данных в нормальном конструкторе уровня экземпляра сводит на нет все их предназначение. Когда бы ни создавался новый объект, данные уровня класса сбрасываются! Один из подходов к установке статического поля предполагает применение синтаксиса инициализации членов, как делалось изначально:

```
class SavingsAccount
{
    public double currBalance;

    // Статические данные.
    public static double currInterestRate = 0.04;
    ...
}
```

Такой подход обеспечит установку статического поля только один раз независимо от того, сколько объектов создается. Но что, если значение статических данных необходимо получать во время выполнения? Например, в типичном банковском приложении значение переменной, представляющей процентную ставку, будет читаться из базы данных или внешнего файла. Решение задач подобного рода требует области действия метода, такого как конструктор, для выполнения соответствующих операторов кода.

По этой причине язык C# позволяет определять статический конструктор, который дает возможность безопасно устанавливать значения статических данных. Взгляните на следующее изменение в коде класса:

```
class SavingsAccount
{
    public double currBalance;
    public static double currInterestRate;
    public SavingsAccount(double balance)
    {
        currBalance = balance;
    }

    // Статический конструктор!
    static SavingsAccount()
    {
        Console.WriteLine("In static ctor!"); // в статическом конструкторе
        currInterestRate = 0.04;
    }
    ...
}
```

Выражаясь просто, статический конструктор представляет собой специальный конструктор, который является идеальным местом для инициализации значений статических данных, если их значения не известны на этапе компиляции (например, когда значения нужно прочитать из внешнего файла или базы данных, сгенерировать случайные числа либо еще каким-то образом получить значения). Если вы снова запустите предыдущий метод `Main()`, то увидите ожидаемый вывод. Обратите внимание, что сообщение `"In static ctor!"` выводится только один раз, т.к. среда CLR вызывает все статические конструкторы перед первым использованием (и никогда не вызывает их заново для данного экземпляра приложения):

```
***** Fun with Static Data *****
In static ctor!
Interest Rate is: 0.04
Interest Rate is: 0.08
```

Ниже приведено несколько интересных моментов, касающихся статических конструкторов.

- В отдельно взятом классе может быть определен только один статический конструктор. Другими словами, перегружать статический конструктор нельзя.
- Статический конструктор не имеет модификатора доступа и не может принимать параметры.
- Статический конструктор выполняется только один раз вне зависимости от количества создаваемых объектов заданного класса.
- Исполняющая система вызывает статический конструктор, когда создает экземпляр класса или перед доступом к первому статическому члену из вызывающего кода.
- Статический конструктор выполняется перед любым конструктором уровня экземпляра.

С учетом такой модификации при создании новых объектов `SavingsAccount` значения статических данных предохраняются, поскольку статический член устанавливается только один раз внутри статического конструктора независимо от количества созданных объектов.

Исходный код. Проект `StaticDataAndMembers` доступен в подкаталоге `Chapter_5`.

Определение статических классов

Ключевое слово `static` допускается также применять прямо на уровне класса. Когда класс определен как статический, его экземпляры нельзя создавать с использованием ключевого слова `new`, и он может содержать только члены или поля данных, помеченные ключевым словом `static`. В случае нарушения этого правила возникают ошибки на этапе компиляции.

На заметку! Вспомните, что класс (или структура), который открывает доступ только к статической функциональности, часто называется *обслуживающим классом*. При проектировании обслуживающего класса рекомендуется применять ключевое слово `static` к самому определению класса.

На первый взгляд такое средство может показаться довольно странным, учитывая невозможность создания экземпляров класса. Тем не менее, в первую очередь класс, который содержит только статические члены и/или константные данные, не нуждается

в выделении для него памяти. Чтобы продемонстрировать сказанное, создадим новый проект консольного приложения по имени SimpleUtilityClass. Определим в нем следующий класс:

```
// Статические классы могут содержать только статические члены!
static class TimeUtilClass
{
    public static void PrintTime()
    { Console.WriteLine(DateTime.Now.ToShortTimeString()); }

    public static void PrintDate()
    { Console.WriteLine(DateTime.Today.ToShortDateString()); }
}
```

Так как класс TimeUtilClass определен с ключевым словом static, создавать его экземпляры с помощью ключевого слова new нельзя. Взамен вся функциональность доступна на уровне класса:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Static Classes *****\n");

    // Это работает нормально.
    TimeUtilClass.PrintDate();
    TimeUtilClass.PrintTime();

    // Ошибка на этапе компиляции! Создавать экземпляры статического
    // класса невозможно!
    TimeUtilClass u = new TimeUtilClass ();

    Console.ReadLine();
}
```

Импортирование статических членов с применением ключевого слова using языка C#

В версии C# 6 появилась поддержка импортирования статических членов с помощью ключевого слова using. В целях иллюстрации предположим, что в файле C# определен обслуживающий класс. Поскольку в нем делаются вызовы метода WriteLine() класса Console и обращения к свойству Now класса DateTime, должен быть предусмотрен оператор using для пространства имен System. Из-за того, что все члены упомянутых классов являются статическими, в файле кода можно указать следующие директивы using static:

```
// Импортировать статические члены классов Console и DateTime.
using static System.Console;
using static System.DateTime;
```

После такого "статического импортирования" в файле кода появляется возможность напрямую применять статические методы классов Console и DateTime, не снабжая их префиксом в виде имени класса, в котором они определены (хотя можно по-прежнему поступать так при условии, что было импортировано пространство имен System). Например, модифицируем наш обслуживающий класс TimeUtilClass, как показано ниже:

```
static class TimeUtilClass
{
    public static void PrintTime() => WriteLine(Now.ToShortTimeString());
    public static void PrintDate() => WriteLine(Today.ToShortDateString());
}
```

Можно было бы утверждать, что данная версия класса несколько привлекательнее, т.к. немного уменьшилась кодовая база. В более реалистичном примере упрощения кода мог бы участвовать класс C#, интенсивно использующий класс `System.Math` (или какой-то другой обслуживающий класс). Поскольку этот класс содержит только статические члены, отчасти было бы проще указать для него оператор `using static` и затем напрямую обращаться членам класса `Math` в своем файле кода.

Однако имейте в виду, что злоупотребление операторами статического импортирования может привести в результате к путанице. Во-первых, как быть, если метод `WriteLine()` определен сразу в нескольких классах? Будет сбивать с толку как компилятор, так и другие программисты, читающие ваш код. Во-вторых, если разработчик не особенно хорошо знаком с библиотеками кода .NET, то он может не знать о том, что `WriteLine()` является членом класса `Console`. До тех пор, пока разработчик не заметит набор операторов статического импортирования в начале файла кода C#, он не может быть полностью уверен в том, где данный метод фактически определен. По указанным причинам применение операторов `using static` в книге ограничено.

К настоящему моменту вы должны уметь определять простые типы классов, содержащие конструкторы, поля и разнообразные статические (и нестатические) члены. Обладая такими базовыми знаниями о конструкции классов, можно приступить к ознакомлению с тремя основными принципами объектно-ориентированного программирования (ООП).

Исходный код. Проект `SimpleUtilityClass` доступен в подкаталоге `Chapter_5`.

Основные принципы объектно-ориентированного программирования

Все объектно-ориентированные языки (C#, Java, C++, Visual Basic и т.д.) должны поддерживать три основных принципа ООП.

- **Инкапсуляция.** Каким образом язык скрывает детали внутренней реализации объектов и предохраняет целостность данных?
- **Наследование.** Каким образом язык стимулирует многократное использование кода?
- **Полиморфизм.** Каким образом язык позволяет трактовать связанные объекты в сходной манере?

Прежде чем погрузиться в синтаксические детали каждого принципа, важно понять их базовую роль. Ниже предлагается обзор всех принципов, а в оставшейся части этой и во всей следующей главе приведены подробные сведения, связанные с ними.

Роль инкапсуляции

Первый основной принцип ООП называется *инкапсуляцией*. Такая характерная черта описывает способность языка скрывать излишние детали реализации от пользователя объекта. Например, предположим, что вы имеете дело с классом по имени `DatabaseReader`, в котором определены два главных метода: `Open()` и `Close()`.

```
// Пусть этот класс инкапсулирует детали открытия и закрытия базы данных.
DatabaseReader dbReader = new DatabaseReader();
dbReader.Open(@"C:\AutoLot.mdf");

// Сделать что-то с файлом данных и закрыть файл.
dbReader.Close();
```

Вымышленный класс `DatabaseReader` инкапсулирует внутренние детали нахождения, загрузки, манипулирования и закрытия файла данных. Программистам нравится инкапсуляция, т.к. этот основной принцип ООП упрощает задачи кодирования. Отсутствует необходимость беспокоиться о многочисленных строках кода, которые работают “за кулисами”, чтобы обеспечить функционирование класса `DatabaseReader`. Все, что понадобится — создать экземпляр и отправить ему подходящие сообщения (например, открыть файл по имени `AutoLot.mdf`, расположенный на диске `C:`).

С понятием инкапсуляции программной логики тесно связана идея защиты данных. В идеале данные состояния объекта должны быть определены с применением ключевого слова `private` (или, возможно, `protected`). В итоге внешний мир должен вежливо попросить об изменении либо извлечении лежащего в основе значения, что крайне важно, т.к. открыто объявленные элементы данных легко могут стать поврежденными (конечно, лучше случайно, чем намеренно). Вскоре будет дано формальное определение такого аспекта инкапсуляции.

Роль наследования

Следующий принцип ООП — *наследование* — отражает возможность языка разрешать построение определений новых классов на основе определений существующих классов. По сути, наследование позволяет расширять поведение базового (или *родительского*) класса за счет наследования его основной функциональности производным подклассом (также называемым *дочерним классом*). На рис. 5.4 показан простой пример.

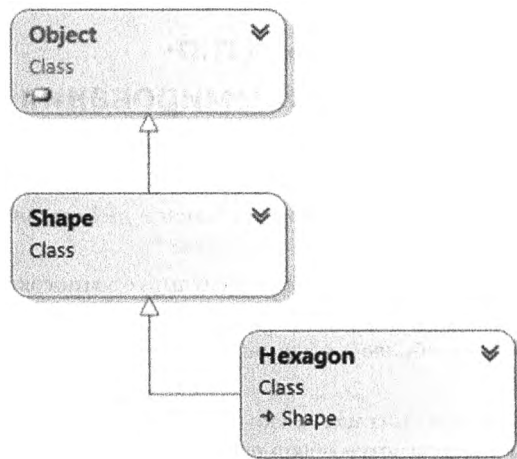


Рис. 5.4. Отношение “является”

Диаграмма на рис. 5.4 читается так: “шестиугольник (Hexagon) является фигурой (Shape), которая является объектом (Object)”. При наличии классов, связанных такой формой наследования, между типами устанавливается *отношение “является”* (“is-a”). Отношение “является” называется *наследованием*.

Здесь можно предположить, что класс `Shape` определяет некоторое количество членов, являющихся общими для всех наследников (скажем, значение для представления цвета фигуры, а также значения для высоты и ширины). Учитывая, что класс `Hexagon` расширяет `Shape`, он наследует основную функциональность, определяемую классами `Shape` и `Object`, и вдобавок сам определяет дополнительные детали, связанные с шестиугольником (какими бы они ни были).

На заметку! В рамках платформы .NET класс `System.Object` всегда находится на вершине любой иерархии классов, являясь первоначальным родительским классом, и определяет общую функциональность для всех типов (как показано в главе 6).

В мире ООП существует еще одна форма повторного использования кода: модель включения/делегации, также известная как *отношение “имеет”* (“has-a”) или *агрегация*. Такая форма повторного использования не применяется для установки отношений “родительский-дочерний”. На самом деле отношение “имеет” позволяет одному классу определять переменную-член другого класса и опосредованно (когда требуется) открывать доступ к его функциональности пользователю объекта.

Например, предположим, что снова моделируется автомобиль. Может возникнуть необходимость выразить идею, что автомобиль “имеет” радиоприемник. Было бы нелогично пытаться наследовать класс `Car` (автомобиль) от класса `Radio` (радиоприемник) или наоборот (ведь `Car` не “является” `Radio`). Взамен есть два независимых класса, работающих совместно, где класс `Car` создает и открывает доступ к функциональности класса `Radio`:

```
class Radio
{
    public void Power(bool turnOn)
    {
        Console.WriteLine("Radio on: {0}", turnOn);
    }
}

class Car
{
    // Car 'имеет' Radio.
    private Radio myRadio = new Radio();

    public void TurnOnRadio(bool onOff)
    {
        // Делегировать вызов внутреннему объекту.
        myRadio.Power(onOff);
    }
}
```

Обратите внимание, что пользователю объекта ничего не известно о том, что класс `Car` использует внутренний объект `Radio`:

```
static void Main(string[] args)
{
    // Внутренне вызов передается объекту Radio.
    Car viper = new Car();
    viper.TurnOnRadio(false);
}
```

Роль полиморфизма

Последним основным принципом ООП является *полиморфизм*. Указанная характерная черта обозначает способность языка трактовать связанные объекты в сходной манере. В частности, данный принцип ООП позволяет базовому классу определять набор членов (формально называемый *полиморфным интерфейсом*), которые доступны всем наследникам. Полиморфный интерфейс класса конструируется с применением любого количества *виртуальных* или *абстрактных* членов (подробности ищите в главе 6).

Выражаясь кратко, *виртуальный член* — это член базового класса, определяющий стандартную реализацию, которую можно изменять (или более формально *переопределять*) в производном классе. В отличие от него *абстрактный метод* — это член базового класса, который не предоставляет стандартную реализацию, а предлагает только сигнатуру. Если класс унаследован от базового класса, в котором определен абстрактный метод, то такой метод *должен* быть переопределен в производном классе. В любом случае, когда производные классы переопределяют члены, определенные в базовом классе, по существу они переопределяют свою реакцию на тот же самый запрос.

Чтобы увидеть полиморфизм в действии, давайте предоставим некоторые детали иерархии фигур, показанной на рис. 5.4. Предположим, что в классе Shape определен виртуальный метод Draw(), не принимающий параметров. С учетом того, что каждой фигуре необходимо визуализировать себя уникальным образом, подклассы вроде Hexagon и Circle могут переопределять метод Draw() по своему усмотрению (рис. 5.5).

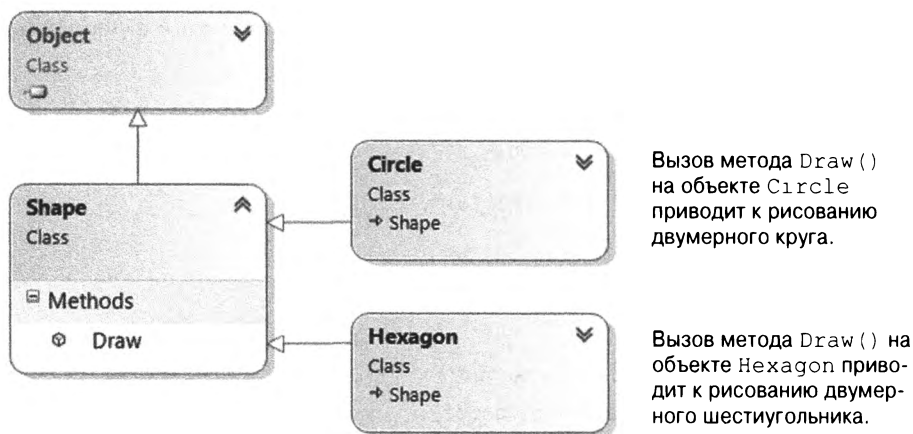


Рис. 5.5. Классический полиморфизм

После того как полиморфный интерфейс спроектирован, можно начинать делать разнообразные предположения в коде. Например, так как классы Hexagon и Circle унаследованы от общего родителя (Shape), массив элементов типа Shape может содержать любые объекты классов, производных от этого базового класса. Более того, поскольку класс Shape определяет полиморфный интерфейс для всех производных типов (метод Draw() в данном примере), уместно предположить, что каждый член массива обладает такой функциональностью.

Рассмотрим следующий метод Main(), который заставляет массив элементов производных от Shape типов визуализировать себя с использованием метода Draw():

```

class Program
{
    static void Main(string[] args)
    {
        Shape[] myShapes = new Shape[3];
        myShapes[0] = new Hexagon();
        myShapes[1] = new Circle();
        myShapes[2] = new Hexagon();

        foreach (Shape s in myShapes)
        {

```

```

        // Использовать полиморфный интерфейс!
        s.Draw();
    }
    Console.ReadLine();
}
}

```

На этом краткий обзор основных принципов ООП завершен. Остальной материал главы посвящен дальнейшим подробностям поддержки инкапсуляции в языке C#. Детали наследования и полиморфизма обсуждаются в главе 6.

Модификаторы доступа C#

При работе с инкапсуляцией вы должны всегда принимать во внимание то, какие аспекты типа являются видимыми различным частям приложения. В частности, типы (классы, интерфейсы, структуры, перечисления и делегаты) и их члены (свойства, методы, конструкторы и поля) определяются с использованием специального ключевого слова, управляющего “видимостью” элемента для других частей приложения. Хотя в C# для управления доступом предусмотрены многочисленные ключевые слова, они отличаются в том, к чему могут успешно применяться (к типу или члену). Модификаторы доступа и особенности их использования описаны в табл. 5.1.

Таблица 5.1. Модификаторы доступа C#

Модификатор доступа	К чему может быть применен	Практический смысл
public	Типы или члены типов	Открытые (public) элементы не имеют ограничений доступа. Открытый член может быть доступен из объекта, а также из любого производного класса. Открытый тип может быть доступен из других внешних сборок
private	Члены типов или вложенные типы	Закрытые (private) элементы могут быть доступны только классу (или структуре), где они определены
protected	Члены типов или вложенные типы	Защищенные (protected) элементы могут использоваться классом, который их определяет, и любым дочерним классом. Однако защищенные элементы не доступны внешнему миру через операцию точки (.)
internal	Типы или члены типов	Внутренние (internal) элементы доступны только в рамках текущей сборки. Таким образом, если в библиотеке классов .NET определен набор внутренних типов, то другие сборки не смогут ими пользоваться
protected internal	Члены типов или вложенные типы	Когда в объявлении элемента указана комбинация ключевых слов protected и internal, то такой элемент будет доступен внутри определяющей его сборки, внутри определяющего класса и для всех его наследников

В текущей главе рассматриваются только ключевые слова `public` и `private`. В последующих главах будет исследована роль модификаторов `internal` и `protected internal` (удобных при построении библиотек кода .NET) и модификатора `protected` (полезного при создании иерархий классов).

Стандартные модификаторы доступа

По умолчанию члены типов являются *неявно закрытыми* (`private`), тогда как типы — *неявно внутренними* (`internal`). Таким образом, следующее определение класса автоматически устанавливается как `internal`, а стандартный конструктор типа — как `private` (тем не менее, как и можно было предполагать, закрытые конструкторы классов нужны редко):

```
// Внутренний класс с закрытым стандартным конструктором.
class Radio
{
    Radio() {}
}
```

Если вы предпочитаете явное объявление, тогда можете добавить соответствующие ключевые слова без каких-либо негативных последствий (помимо дополнительных усилий по набору):

```
// Внутренний класс с закрытым стандартным конструктором.
internal class Radio
{
    private Radio() {}
}
```

Чтобы позволить другим частям программы обращаться к членам объекта, вы должны определить эти члены с ключевым словом `public` (или возможно с ключевым словом `protected`, которое объясняется в следующей главе). Вдобавок, если вы хотите открыть доступ к `Radio` внешним сборкам (что удобно при построении библиотек кода .NET, как показано в главе 14), то к нему придется добавить модификатор `public`:

```
// Открытый класс с открытым стандартным конструктором.
public class Radio
{
    public Radio() {}
}
```

Модификаторы доступа и вложенные типы

Как упоминалось в табл. 5.1, модификаторы доступа `private`, `protected` и `protected internal` могут применяться к *вложенному типу*. Вложение типов будет подробно рассматриваться в главе 6, а пока достаточно знать, что вложенный тип — это тип, объявленный прямо внутри области видимости класса или структуры. Для примера ниже приведено закрытое перечисление (по имени `CarColor`), вложенное в открытый класс (по имени `SportsCar`):

```
public class SportsCar
{
    // Нормально! Вложенные типы могут быть помечены как private.
    private enum CarColor
    {
        Red, Green, Blue
    }
}
```

Здесь допустимо применять модификатор доступа `private` к вложенному типу. Однако невложенные типы (вроде `SportsCar`) могут определяться только с модификатором `public` или `internal`. Таким образом, следующее определение класса незаконно:

```
// Ошибка! Невложенный тип не может быть помечен как private!
private class SportsCar
{ }
```

Первый принцип ООП: службы инкапсуляции C#

Концепция инкапсуляции вращается вокруг идеи о том, что данные класса не должны быть напрямую доступными через его экземпляр. Наоборот, данные класса определяются как закрытые. Если пользователь объекта желает изменить его состояние, тогда он должен делать это косвенно, используя открытые члены. Чтобы проиллюстрировать необходимость в службах инкапсуляции, предположим, что создано такое определение класса:

```
// Класс с единственным открытым полем.
class Book
{
    public int numberOfPages;
}
```

Проблема с открытыми данными заключается в том, что сами по себе они неспособны “понять”, является ли присваиваемое значение допустимым с точки зрения текущих бизнес-правил системы. Как известно, верхний предел значений для типа `int` в C# довольно высок (2 147 483 647), поэтому компилятор разрешит следующее присваивание:

```
// Хм... Ничего себе мини-новелла!
static void Main(string[] args)
{
    Book miniNovel = new Book();
    miniNovel.numberOfPages = 30_000_000;
}
```

Хотя границы типа данных `int` не превышены, понятно, что мини-новелла на 30 миллионов страниц выглядит несколько неправдоподобно. Как видите, открытые поля не предоставляют способа ограничения значений верхними (или нижними) логическими пределами. Если в системе установлено текущее бизнес-правило, которое регламентирует, что книга должна иметь от 1 до 1000 страниц, то совершенно неясно, как обеспечить его выполнение программным образом. Именно потому открытым полям обычно нет места в определениях классов производственного уровня.

На заметку! Говоря точнее, члены класса, которые представляют состояние объекта, не должны помечаться как `public`. В то же время позже в главе вы увидите, что вполне нормально иметь открытые константы и открытые поля, допускающие только чтение.

Инкапсуляция предлагает способ предохранения целостности данных состояния для объекта. Вместо определения открытых полей (которые могут легко привести к повреждению данных) необходимо выработать у себя привычку определять *закрытые данные*, управление которыми осуществляется опосредованно с применением одного из двух главных приемов:

- определение пары открытых методов доступа и изменения;
- определение открытого свойства .NET.

Независимо от выбранного приема идея заключается в том, что хорошо инкапсулированный класс должен защищать свои данные и скрывать подробности своего функционирования от любопытных глаз из внешнего мира. Это часто называют *программированием в стиле черного ящика*. Преимущество такого подхода в том, что объект может свободно изменять внутреннюю реализацию любого метода. Работа существующего кода, который использует данный метод, не нарушается при условии, что параметры и возвращаемые значения методов остаются неизменными.

Инкапсуляция с использованием традиционных методов доступа и изменения

В оставшейся части главы будет построен довольно полный класс, моделирующий обычного сотрудника. Для начала создадим новый проект консольного приложения по имени EmployeeApp и добавим в него новый файл класса (Employee.cs) с помощью пункта меню Project→Add class. Дополним класс Employee следующими полями, методами и конструкторами:

```
class Employee
{
    // Поля данных.
    private string empName;
    private int empID;
    private float currPay;

    // Конструкторы.
    public Employee() {}
    public Employee(string name, int id, float pay)
    {
        empName = name;
        empID = id;
        currPay = pay;
    }

    // Методы.
    public void GiveBonus(float amount)
    {
        currPay += amount;
    }

    public void DisplayStats()
    {
        Console.WriteLine("Name: {0}", empName); // имя сотрудника
        Console.WriteLine("ID: {0}", empID); // идентификационный номер сотрудника
        Console.WriteLine("Pay: {0}", currPay); // текущая выплата
    }
}
```

Обратите внимание, что поля класса Employee в текущий момент определены с применением ключевого слова `private`. Учитывая это, поля `empName`, `empID` и `currPay` не доступны напрямую через объектную переменную. Таким образом, показанная ниже логика в `Main()` приведет к ошибкам на этапе компиляции:

```
static void Main(string[] args)
{
    Employee emp = new Employee();
    // Ошибка! Невозможно напрямую обращаться к закрытым полям объекта!
    emp.empName = "Marv";
}
```

Если нужно, чтобы внешний мир взаимодействовал с полным именем сотрудника, то традиционный подход (который распространен в Java) предусматривает определение методов доступа (метод `get`) и изменения (метод `set`). Роль метода `get` заключается в возвращении вызывающему коду текущего значения лежащих в основе данных состояния. Метод `set` позволяет вызывающему коду изменять текущее значение лежащих в основе данных состояния при условии удовлетворения бизнес-правил.

В целях иллюстрации давайте инкапсулируем поле `empName`, для чего к существующему классу `Employee` необходимо добавить показанные ниже открытые методы. Обратите внимание, что метод `SetName()` выполняет проверку входных данных, чтобы удостовериться, что строка имеет длину 15 символов или меньше. Если это не так, тогда на консоль выводится сообщение об ошибке и происходит возврат без внесения изменений в поле `empName`.

На заметку! В случае класса производственного уровня проверку длины строки с именем сотрудника следовало бы предусмотреть также и внутри логики конструктора. Мы пока проигнорируем указанную деталь, но улучшим код позже, во время исследований синтаксиса свойств .NET.

```
class Employee
{
    // Поля данных.
    private string empName;
    ...
    // Метод доступа (метод get) .
    public string GetName()
    {
        return empName;
    }
    // Метод изменения (метод set) .
    public void SetName(string name)
    {
        // Перед присваиванием проверить входное значение.
        if (name.Length > 15)
            Console.WriteLine("Error! Name length exceeds 15 characters!");
            // Ошибка! Длина имени превышает 15 символов!
        else
            empName = name;
    }
}
```

Такой подход требует наличия двух уникально именованных методов для управления единственным элементом данных. Чтобы протестировать новые методы, модифицируем метод `Main()` следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Encapsulation *****\n");
    Employee emp = new Employee("Marvin", 456, 30000);
    emp.GiveBonus(1000);
    emp.DisplayStats();

    // Использовать методы get/set для взаимодействия
    // с именем сотрудника, представленного объектом.
    emp.SetName("Marv");
    Console.WriteLine("Employee is named: {0}", emp.GetName());
    Console.ReadLine();
}
```

Благодаря коду в методе `SetName()` попытка указать для имени строку, содержащую более 15 символов (как показано ниже), приводит к выводу на консоль жестко закодированного сообщения об ошибке:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Encapsulation *****\n");
    ...
    // Длиннее 15 символов! На консоль выводится сообщение об ошибке.
    Employee emp2 = new Employee();
    emp2.SetName("Xena the warrior princess");

    Console.ReadLine();
}
```

Пока все идет хорошо. Мы инкапсулировали закрытое поле `empName` с использованием двух открытых методов с именами `GetName()` и `SetName()`. Для дальнейшей инкапсуляции данных в классе `Employee` понадобится добавить разнообразные дополнительные методы (такие как `GetID()`, `SetID()`, `GetCurrentPay()`, `SetCurrentPay()`). В каждом методе, изменяющем данные, может содержаться несколько строк кода, в которых реализована проверка дополнительных бизнес-правил. Несмотря на то что это определенно достижимо, для инкапсуляции данных класса в языке C# имеется удобная альтернативная система записи.

Инкапсуляция с использованием свойств .NET

Хотя инкапсулировать поля данных можно с применением традиционной пары методов `get` и `set`, в языках .NET предпочтение отдается обеспечению инкапсуляции данных с использованием *свойств*. Прежде всего, имейте в виду, что свойства — всего лишь упрощенное представление “настоящих” методов доступа и изменения. Следовательно, проектировщик класса по-прежнему может выполнить любую внутреннюю логику перед присваиванием значения (например, преобразовать в верхний регистр, избавиться от недопустимых символов, проверить вхождение внутрь границ и т.д.).

Ниже приведен измененный класс `Employee`, который теперь обеспечивает инкапсуляцию каждого поля с использованием синтаксиса свойств вместо традиционных методов `get` и `set`.

```
class Employee
{
    // Поля данных.
    private string empName;
    private int empID;
    private float currPay;

    // Свойства.
    public string Name
    {
        get { return empName; }
        set
        {
            if (value.Length > 15)
                Console.WriteLine("Error! Name length exceeds 15 characters!");
            // Ошибка! Длина имени превышает 15 символов!

            else
                empName = value;
        }
    }
}
```

```
// Можно было бы добавить дополнительные бизнес-правила для установки
// данных свойств, но в настоящем примере в этом нет необходимости.
public int ID
{
    get { return empID; }
    set { empID = value; }
}
public float Pay
{
    get { return currPay; }
    set { currPay = value; }
}
...
}
```

Свойство C# состоит из определений областей `get` (метод доступа) и `set` (метод изменения) прямо внутри самого свойства. Обратите внимание, что свойство указывает тип инкапсулируемых им данных способом, который выглядит как возвращаемое значение. Кроме того, в отличие от метода при определении свойства не применяются круглые скобки (даже пустые). Взгляните на следующий комментарий к текущему свойству `ID`:

```
// int представляет тип данных, инкапсулируемых этим свойством.
public int ID // Обратите внимание на отсутствие круглых скобок.
{
    get { return empID; }
    set { empID = value; }
}
```

В области `set` свойства используется лексема `value`, которая представляет входное значение, присваиваемое свойству вызывающим кодом. Лексема `value` не является настоящим ключевым словом C#, а представляет собой то, что называется *контекстным ключевым словом*. Когда лексема `value` находится внутри области `set`, она всегда обозначает значение, присваиваемое вызывающим кодом, и всегда имеет тип, совпадающий с типом самого свойства. Таким образом, вот как свойство `Name` может проверить допустимую длину строки:

```
public string Name
{
    get { return empName; }
    set
    {
        // Здесь value на самом деле имеет тип string.
        if (value.Length > 15)
            Console.WriteLine("Error! Name length exceeds 15 characters!");
            // Ошибка! Длина имени превышает 15 символов!
        else
            empName = value;
    }
}
```

После определения свойств подобного рода вызывающему коду кажется, что он имеет дело с *открытым элементом данных*; однако “за кулисами” при каждом обращении к ним вызывается корректный блок `get` или `set`, предохраняя инкапсуляцию:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Encapsulation *****\n");
    Employee emp = new Employee("Marvin", 456, 30000);
}
```

```

emp.GiveBonus(1000);
emp.DisplayStats();

// Переустановка и затем получение свойства Name.
emp.Name = "Marv";
Console.WriteLine("Employee is named: {0}", emp.Name);
Console.ReadLine();
}

```

Свойства (как противоположность методам доступа и изменения) также облегчают манипулирование типами, поскольку способны реагировать на внутренние операции C#. В целях иллюстрации предположим, что тип класса `Employee` имеет внутреннюю закрытую переменную-член, представляющую возраст сотрудника. Ниже показаны необходимые изменения (обратите внимание на применение цепочки вызовов конструкторов):

```

class Employee
{
    ...
    // Новое поле и свойство.
    private int empAge;
    public int Age
    {
        get { return empAge; }
        set { empAge = value; }
    }

    // Обновленные конструкторы.
    public Employee() {}
    public Employee(string name, int id, float pay)
        :this(name, 0, id, pay){}
    public Employee(string name, int age, int id, float pay)
    {
        empName = name;
        empID = id;
        empAge = age;
        currPay = pay;
    }

    // Обновленный метод DisplayStats() теперь учитывает возраст.
    public void DisplayStats()
    {
        Console.WriteLine("Name: {0}", empName);    // имя сотрудника
        Console.WriteLine("ID: {0}", empID);        // идентификационный номер сотрудника
        Console.WriteLine("Age: {0}", empAge);      // возраст сотрудника
        Console.WriteLine("Pay: {0}", currPay);     // текущая выплата
    }
}

```

Теперь пусть создан объект `Employee` по имени `joe`. Необходимо сделать так, чтобы в день рождения сотрудника возраст увеличивался на 1 год. Используя традиционные методы `set` и `get`, пришлось бы написать приблизительно такой код:

```

Employee joe = new Employee();
joe.SetAge(joe.GetAge() + 1);

```

Тем не менее, если `empAge` инкапсулируется посредством свойства по имени `Age`, то код будет проще:

```

Employee joe = new Employee();
joe.Age++;

```

Свойства, сжатые до выражений (нововведение)

Как упоминалось ранее, методы `set` и `get` свойств также могут записываться в виде членов, сжатых до выражений. Правила и синтаксис те же: однострочные метода могут быть записаны с применением нового синтаксиса. Таким образом, свойство `Age` можно было бы переписать следующим образом:

```
public int Age
{
    get => empAge;
    set => empAge = value;
}
```

Оба варианта кода компилируются в одинаковый набор инструкций IL, поэтому выбор используемого синтаксиса зависит только от ваших предпочтений. В книге будут сочетаться оба стиля, чтобы подчеркнуть, что мы не придерживаемся какого-то специфического стиля написания кода.

Использование свойств внутри определения класса

Свойства, в частности их порция `set`, являются общепринятым местом для размещения бизнес-правил класса. В текущий момент класс `Employee` имеет свойство `Name`, которое гарантирует, что длина имени не превышает 15 символов. Остальные свойства (`ID`, `Pay` и `Age`) также могут быть обновлены соответствующей логикой.

Хотя все это хорошо, но необходимо также принимать во внимание и то, что обычно происходит внутри конструктора класса. Конструктор получает входные параметры, проверяет данные на предмет допустимости и затем присваивает значения внутренним закрытым полям. Пока что главный конструктор не проверяет входные строковые данные на вхождение в диапазон допустимых значений, а потому его можно было бы изменить следующим образом:

```
public Employee(string name, int age, int id, float pay)
{
    // Похоже на проблему...
    if (name.Length > 15)
        Console.WriteLine("Error! Name length exceeds 15 characters!");
    // Ошибка! Длина имени превышает 15 символов!

    else
        empName = name;

    empID = id;
    empAge = age;
    currPay = pay;
}
```

Наверняка вы заметили проблему, связанную с таким подходом. Свойство `Name` и главный конструктор выполняют одну и ту же проверку на наличие ошибок. Реализуя проверки для других элементов данных, есть реальный шанс столкнуться с дублированием кода. Стремясь рационализировать код и изолировать всю проверку, касающуюся ошибок, в каком-то центральном местоположении, вы добьетесь успеха, если для получения и установки значений внутри класса *всегда* будете применять свойства. Взгляните на показанный ниже модифицированный конструктор:

```
public Employee(string name, int age, int id, float pay)
{
    // Уже лучше! Используйте свойства для установки данных класса.
    // Это сократит количество дублированных проверок на предмет ошибок.
```

```

    Name = name;
    Age = age;
    ID = id;
    Pay = pay;
}

```

Помимо обновления конструкторов для применения свойств при присваивании значений рекомендуется повсюду в реализации класса использовать свойства, чтобы гарантировать неизменное соблюдение бизнес-правил. Во многих случаях прямая ссылка на лежащие в основе закрытые данные производится только внутри самого свойства. Имея все сказанное в виду, модифицируем класс `Employee`:

```

class Employee
{
    // Поля данных.
    private string empName;
    private int empID;
    private float currPay;
    private int empAge;
    // Конструкторы.
    public Employee() { }
    public Employee(string name, int id, float pay)
        :this(name, 0, id, pay){}
    public Employee(string name, int age, int id, float pay)
    {
        Name = name;
        Age = age;
        ID = id;
        Pay = pay;
    }
    // Методы.
    public void GiveBonus(float amount)
    { Pay += amount; }
    public void DisplayStats()
    {
        Console.WriteLine("Name: {0}", Name);
        Console.WriteLine("ID: {0}", ID);
        Console.WriteLine("Age: {0}", Age);
        Console.WriteLine("Pay: {0}", Pay);
    }
    // Свойства остаются прежними...
    ...
}

```

Свойства, допускающие только чтение и только запись

При инкапсуляции данных может возникнуть желание сконфигурировать *свойство, допускающее только чтение*, для чего нужно просто опустить блок `set`. Аналогично, если необходимо *свойство, доступное только для записи*, то следует опустить блок `get`. Например, пусть имеется новое свойство по имени `SocialSecurityNumber`, которое инкапсулирует закрытую строковую переменную `empSSN`. Вот как превратить его в свойство, доступное только для чтения:

```

public string SocialSecurityNumber
{
    get { return empSSN; }
}

```

Теперь предположим, что конструктор класса принимает новый параметр, который дает возможность указывать в вызывающем коде номер карточки социального страхования для объекта сотрудника. Поскольку свойство `SocialSecurityNumber` допускает только чтение, устанавливать значение так, как показано ниже, нельзя:

```
public Employee(string name, int age, int id, float pay, string ssn)
{
    Name = name;
    Age = age;
    ID = id;
    Pay = pay;

    // Если свойство предназначено только для чтения, это больше невозможно!
    SocialSecurityNumber = ssn;
}
```

Если только вы не готовы переделать данное свойство в поддерживающее чтение и запись, тогда единственным выбором будет применение лежащей в основе переменной-члена `empSSN` внутри логики конструктора:

```
public Employee(string name, int age, int id, float pay, string ssn)
{
    ...
    // Проверить надлежащим образом входной параметр ssn и затем установить значение.
    empSSN = ssn;
}
```

Исходный код. Проект `EmployeeApp` доступен в подкаталоге `Chapter_5`.

Еще раз о ключевом слове `static`: определение статических свойств

Ранее в главе рассказывалось о роли ключевого слова `static`. Теперь, когда вы научились использовать синтаксис свойств C#, мы можем формализовать статические свойства. В проекте `StaticDataAndMembers` класс `SavingsAccount` имел два открытых статических метода для получения и установки процентной ставки. Однако более стандартный подход предусматривает помещение такого элемента данных в статическое свойство. Ниже приведен пример (обратите внимание на применение ключевого слова `static`):

```
// Простой класс депозитного счета.
class SavingsAccount
{
    // Данные уровня экземпляра.
    public double currBalance;

    // Статический элемент данных.
    private static double currInterestRate = 0.04;

    // Статическое свойство.
    public static double InterestRate
    {
        get { return currInterestRate; }
        set { currInterestRate = value; }
    }
    ...
}
```


Если вы хотите использовать свойство `InterestRate` вместо предыдущих статических методов, то понадобится модифицировать метод `Main()`:

```
// Вывести текущую процентную ставку через свойство.
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.InterestRate);
```

Понятие автоматических свойств

При создании свойств для инкапсуляции данных часто обнаруживается, что области `set` содержат код для применения бизнес-правил программы. Тем не менее, в некоторых случаях нужна только простая логика извлечения или установки значения. В результате получается большой объем кода следующего вида:

```
// Тип Car, использующий стандартный синтаксис свойств.
class Car
{
    private string carName = "";
    public string PetName
    {
        get { return carName; }
        set { carName = value; }
    }
}
```

В подобных случаях многократное определение закрытых поддерживающих полей и простых свойств может стать слишком громоздким. Например, при построении класса, которому нужны 9 закрытых элементов данных, в итоге получаются 9 связанных с ними свойств, которые представляют собой не более чем тонкие оболочки для служб инкапсуляции.

Чтобы упростить процесс обеспечения простой инкапсуляции данных полей, можно использовать *синтаксис автоматических свойств*. Как следует из названия, это средство перекладывает работу по определению закрытых поддерживающих полей и связанных с ними свойств C# на компилятор за счет применения небольшого нововведения в синтаксисе. В целях иллюстрации создадим новый проект консольного приложения по имени `AutoProps`. Взгляните на переделанный класс `Car`, в котором данный синтаксис используется для быстрого создания трех свойств:

```
class Car
{
    // Автоматические свойства! Нет нужды определять поддерживающие поля.
    public string PetName { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }
}
```

На заметку! Среда Visual Studio предлагает фрагмент кода `prop`. Если вы наберете слово `prop` и два раза нажмете клавишу `<Tab>`, то IDE-среда сгенерирует начальный код для нового автоматического свойства. Затем с помощью клавиши `<Tab>` можно циклически проходить по всем частям определения и заполнять необходимые детали. Испытайте описанный прием.

При определении автоматического свойства вы просто указываете модификатор доступа, лежащий в основе тип данных, имя свойства и пустые области `get/set`. Во время компиляции тип будет оснащен автоматически сгенерированным поддерживающим полем и подходящей реализацией логики `get/set`.

На заметку! Имя автоматически сгенерированного закрытого поддерживающего поля будет невидимым для кодовой базы C#. Просмотреть его можно только с помощью инструмента вроде `ildasm.exe`.

Начиная с версии C# 6, разрешено определять "автоматическое свойство только для чтения", опуская область `set`. Тем не менее, определять свойство, предназначенное только для записи, нельзя. Вот пример:

```
// Свойство только для чтения? Допустимо!
public int MyReadOnlyProp { get; }

// Свойство только для записи? Ошибка!
public int MyWriteOnlyProp { set; }
```

Взаимодействие с автоматическими свойствами

Поскольку компилятор будет определять закрытые поддерживающие поля на этапе компиляции (и учитывая, что эти поля в коде C# непосредственно не доступны), в классе, который имеет автоматические свойства, для установки и чтения лежащих в их основе значений всегда должен применяться синтаксис свойств. Указанный факт важно отметить, т.к. многие программисты напрямую используют закрытые поля *внутри* определения класса, что в данном случае невозможно. Например, если бы класс `Car` содержал метод `DisplayStats()`, то в его реализации пришлось бы применять имена свойств:

```
class Car
{
    // Автоматические свойства!
    public string PetName { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public void DisplayStats()
    {
        Console.WriteLine("Car Name: {0}", PetName);
        Console.WriteLine("Speed: {0}", Speed);
        Console.WriteLine("Color: {0}", Color);
    }
}
```

При использовании экземпляра класса, определенного с автоматическими свойствами, присваивать и получать значения можно с помощью вполне ожидаемого синтаксиса свойств:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Automatic Properties *****\n");

    Car c = new Car();
    c.PetName = "Frank";
    c.Speed = 55;
    c.Color = "Red";

    Console.WriteLine("Your car is named {0}? That's odd...",
        c.PetName);
    c.DisplayStats();

    Console.ReadLine();
}
```

Автоматические свойства и стандартные значения

Когда автоматические свойства применяются для инкапсуляции числовых и булевских данных, их можно использовать прямо внутри кодовой базы, поскольку скрытым поддерживающим полям будут присваиваться безопасные стандартные значения (`false` для булевских и `0` для числовых данных). Но имейте в виду, что когда синтаксис автоматического свойства применяется для упаковки переменной другого класса, то скрытое поле ссылочного типа также будет установлено в стандартное значение `null` (и это может привести к проблеме, если не проявить должную осторожность).

Давайте добавим в текущий проект новый класс по имени `Garage` (представляющий гараж), в котором используются два автоматических свойства (разумеется, реальный класс гаража может поддерживать коллекцию объектов `Car`; однако в данный момент проигнорируем такую деталь):

```
class Garage
{
    // Скрытое поддерживающее поле int установлено в 0!
    public int NumberOfCars { get; set; }

    // Скрытое поддерживающее поле Car установлено в null!
    public Car MyAuto { get; set; }
}
```

Имея стандартные значения C# для полей данных, значение `NumberOfCars` можно вывести в том виде, как есть (поскольку ему автоматически присвоено значение `0`). Но если напрямую обратиться к `MyAuto`, то во время выполнения сгенерируется исключение ссылки на `null`, потому что лежащей в основе переменной-члену типа `Car` не был присвоен новый объект.

```
static void Main(string[] args)
{
    ...
    Garage g = new Garage();
    // Нормально, выводится стандартное значение 0.
    Console.WriteLine("Number of Cars: {0}", g.NumberOfCars);
    // Ошибка во время выполнения! Поддерживающее поле в данный момент равно null!
    Console.WriteLine(g.MyAuto.PetName);
    Console.ReadLine();
}
```

Чтобы решить проблему, можно модифицировать конструкторы класса, обеспечив безопасное создание объекта. Ниже показан пример:

```
class Garage
{
    // Скрытое поддерживающее поле установлено в 0!
    public int NumberOfCars { get; set; }

    // Скрытое поддерживающее поле установлено в null!
    public Car MyAuto { get; set; }

    // Для переопределения стандартных значений, присвоенных скрытым
    // поддерживающим полям, должны использоваться конструкторы.
    public Garage()
    {
        MyAuto = new Car();
        NumberOfCars = 1;
    }
}
```

```

public Garage(Car car, int number)
{
    MyAuto = car;
    NumberOfCars = number;
}
}

```

После такого изменения объект Car теперь можно помещать в объект Garage:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Automatic Properties *****\n");
    // Создать объект автомобиля.
    Car c = new Car();
    c.PetName = "Frank";
    c.Speed = 55;
    c.Color = "Red";
    c.DisplayStats();

    // Поместить автомобиль в гараж.
    Garage g = new Garage();
    g.MyAuto = c;
    // Вывести количество автомобилей в гараже.
    Console.WriteLine("Number of Cars in garage: {0}", g.NumberOfCars);
    // Вывести название автомобиля.
    Console.WriteLine("Your car is named: {0}", g.MyAuto.PetName);
    Console.ReadLine();
}

```

Инициализация автоматических свойств

Наряду с тем, что предыдущий подход работает вполне нормально, в версии C# 6 появилась языковая возможность, которая содействует упрощению способа присваивания автоматическим свойствам их начальных значений. Как упоминалось ранее в главе, полю данных в классе можно напрямую присваивать начальное значение при его объявлении. Например:

```

class Car
{
    private int numberOfDoors = 2;
}

```

В похожей манере язык C# теперь позволяет присваивать начальные значения лежащим в основе поддерживающим полям, которые генерируются компилятором. В результате смягчаются трудности, присущие добавлению дополнительных операторов кода в конструкторы класса, которые обеспечивают корректную установку данных свойств.

Ниже приведена модифицированная версия класса Garage с инициализацией автоматических свойств подходящими значениями. Обратите внимание, что больше нет необходимости в добавлении к стандартному конструктору класса дополнительной логики для выполнения безопасного присваивания. В коде свойству MyAuto напрямую присваивается новый объект Car.

```

class Garage
{
    // Скрытое поддерживающее поле установлено в 1.
    public int NumberOfCars { get; set; } = 1;
}

```

```
// Скрытое поддерживающее поле установлено в новый объект Car.
public Car MyAuto { get; set; } = new Car();

public Garage(){}
public Garage(Car car, int number)
{
    MyAuto = car;
    NumberOfCars = number;
}
}
```

Вы наверняка согласитесь с тем, что автоматические свойства — очень полезное средство языка программирования C#, т.к. отдельные свойства в классе можно определять с применением модернизированного синтаксиса. Конечно, если вы создаете свойство, которое помимо получения и установки закрытого поддерживающего поля требует дополнительного кода (такого как логика проверки достоверности, регистрация в журнале событий, взаимодействие с базой данных и т.д.), то его придется определять как "нормальное" свойство .NET вручную. Автоматические свойства C# не делают ничего кроме обеспечения простой инкапсуляции для лежащей в основе порции (сгенерированных компилятором) закрытых данных.

Исходный код. Проект AutoProps доступен в подкаталоге Chapter_5.

Понятие синтаксиса инициализации объектов

На протяжении всей главы можно заметить, что при создании нового объекта конструктор позволяет указывать начальные значения. Вдобавок свойства позволяют безопасным образом получать и устанавливать лежащие в основе данные. При работе со сторонними классами, включая классы из библиотеки базовых классов .NET, нередко обнаруживается, что в них отсутствует конструктор, который позволял бы устанавливать абсолютно все порции данных состояния. В итоге программист обычно вынужден выбирать наилучший конструктор из числа возможных, а затем присваивать остальные значения с использованием предоставляемого набора свойств.

Для упрощения процесса создания и подготовки объекта в C# предлагается *синтаксис инициализации объектов*. Такой прием делает возможным создание новой объектной переменной и присваивание значений многочисленным свойствам и/или открытым полям в нескольких строках кода. Синтаксически инициализатор объекта выглядит как список разделенных запятыми значений, помещенный в фигурные скобки ({}). Каждый элемент в списке инициализации отображается на имя открытого поля или открытого свойства инициализируемого объекта.

Чтобы увидеть данный синтаксис в действии, создадим новый проект консольного приложения по имени ObjectInitializers. Ниже показан класс Point, в котором присутствуют автоматические свойства (для синтаксиса инициализации объектов они не обязательны, но помогают получить более лаконичный код):

```
class Point
{
    public int X { get; set; }
    public int Y { get; set; }
    public Point(int xVal, int yVal)
    {
        X = xVal;
        Y = yVal;
    }
}
```

```

public Point() { }
public void DisplayStats()
{
    Console.WriteLine("[{0}, {1}]", X, Y);
}
}

```

А теперь посмотрим, как создавать объекты `Point`, с применением любого из следующих подходов:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Init Syntax *****\n");
    // Создать объект Point, устанавливая каждое свойство вручную.
    Point firstPoint = new Point();
    firstPoint.X = 10;
    firstPoint.Y = 10;
    firstPoint.DisplayStats();

    // Создать объект Point посредством специального конструктора.
    Point anotherPoint = new Point(20, 20);
    anotherPoint.DisplayStats();

    // Создать объект Point, используя синтаксис инициализации объектов.
    Point finalPoint = new Point { X = 30, Y = 30 };
    finalPoint.DisplayStats();
    Console.ReadLine();
}

```

При создании последней переменной `Point` специальный конструктор не используется (как делается традиционно), а взамен устанавливаются значения открытых свойств `X` и `Y`. “За кулисами” вызывается стандартный конструктор типа, за которым следует установка значений указанных свойств. В таком отношении синтаксис инициализации объектов представляет собой просто сокращение синтаксиса для создания переменной класса с применением стандартного конструктора и установки данных состояния свойство за свойством.

Вызов специальных конструкторов с помощью синтаксиса инициализации

В предшествующих примерах объекты типа `Point` инициализировались путем неявного вызова стандартного конструктора этого типа:

```

// Здесь стандартный конструктор вызывается неявно.
Point finalPoint = new Point { X = 30, Y = 30 };

```

При желании стандартный конструктор допускается вызывать и явно:

```

// Здесь стандартный конструктор вызывается явно.
Point finalPoint = new Point() { X = 30, Y = 30 };

```

Имейте в виду, что при конструировании объекта типа с использованием синтаксиса инициализации можно вызывать любой конструктор, определенный в классе. В настоящий момент в типе `Point` определен конструктор с двумя аргументами для установки позиции (x , y). Таким образом, следующее объявление переменной `Point` приведет к установке `X` в 100 и `Y` в 100 независимо от того факта, что в аргументах конструктора указаны значения 10 и 16:

```

// Вызов специального конструктора.
Point pt = new Point(10, 16) { X = 100, Y = 100 };

```

Имея текущее определение типа `Point`, вызов специального конструктора с применением синтаксиса инициализации не особенно полезен (и излишне многословен). Тем не менее, если тип `Point` предоставляет новый конструктор, который позволяет вызывающему коду устанавливать цвет (через специальное перечисление `PointColor`), тогда комбинация специальных конструкторов и синтаксиса инициализации объектов становится ясной. Модифицируем тип `Point`, как показано далее:

```
enum PointColor
{ LightBlue, BloodRed, Gold }

class Point
{
    public int X { get; set; }
    public int Y { get; set; }
    public PointColor Color { get; set; }

    public Point(int xVal, int yVal)
    {
        X = xVal;
        Y = yVal;
        Color = PointColor.Gold;
    }

    public Point(PointColor ptColor)
    {
        Color = ptColor;
    }

    public Point()
        : this(PointColor.BloodRed){ }

    public void DisplayStats()
    {
        Console.WriteLine("[{0}, {1}]", X, Y);
        Console.WriteLine("Point is {0}", Color);
    }
}
```

Посредством нового конструктора теперь можно создавать точку золотистого цвета (в позиции (90, 20)):

```
// Вызов более интересного специального конструктора
// с помощью синтаксиса инициализации.
Point goldPoint = new Point(PointColor.Gold) { X = 90, Y = 20 };
goldPoint.DisplayStats();
```

Инициализация данных с помощью синтаксиса инициализации

Как кратко упоминалось ранее в главе (и будет подробно обсуждаться в главе 6), отношение “имеет” позволяет формировать новые классы, определяя переменные-члены существующих классов. Например, пусть определен класс `Rectangle`, в котором для представления координат верхнего левого и нижнего правого углов используется тип `Point`. Так как автоматические свойства устанавливают все переменные с типами классов в `null`, новый класс будет реализован с применением “традиционного” синтаксиса свойств:

```
class Rectangle
{
    private Point topLeft = new Point();
    private Point bottomRight = new Point();
```

```

public Point TopLeft
{
    get { return topLeft; }
    set { topLeft = value; }
}
public Point BottomRight
{
    get { return bottomRight; }
    set { bottomRight = value; }
}
public void DisplayStats()
{
    Console.WriteLine("[TopLeft: {0}, {1}, {2} BottomRight: {3}, {4}, {5}]",
        topLeft.X, topLeft.Y, topLeft.Color,
        bottomRight.X, bottomRight.Y, bottomRight.Color);
}
}

```

С помощью синтаксиса инициализации объектов можно было бы создать новую переменную `Rectangle` и установить внутренние объекты `Point` следующим образом:

```

// Создать и инициализировать объект Rectangle.
Rectangle myRect = new Rectangle
{
    TopLeft = new Point { X = 10, Y = 10 },
    BottomRight = new Point { X = 200, Y = 200 }
};

```

Преимущество синтаксиса инициализации объектов в том, что он по существу сокращает объем вводимого кода (предполагая отсутствие подходящего конструктора). Вот как выглядит традиционный подход к созданию похожего экземпляра `Rectangle`:

```

// Традиционный подход.
Rectangle r = new Rectangle();
Point p1 = new Point();
p1.X = 10;
p1.Y = 10;
r.TopLeft = p1;
Point p2 = new Point();
p2.X = 200;
p2.Y = 200;
r.BottomRight = p2;

```

Поначалу синтаксис инициализации объектов может показаться несколько непривычным, но как только вы освоитесь с кодом, то будете приятно поражены тем, насколько быстро и с минимальными усилиями можно устанавливать состояние нового объекта.

Исходный код. Проект `ObjectInitializers` доступен в подкаталоге `Chapter_5`.

Работа с данными константных полей

Язык C# предлагает ключевое слово `const`, предназначенное для определения константных данных, которые после начальной установки больше никогда не могут быть изменены. Как нетрудно догадаться, оно полезно при определении набора известных значений для использования в приложениях, логически связанных с заданным классом или структурой.

Предположим, что вы строите обслуживающий класс по имени `MyMathClass`, в котором нужно определить значение числа π (для простоты будем считать его равным 3.14). Начнем с создания нового проекта консольного приложения по имени `ConstData`. Учитывая, что давать возможность другим разработчикам изменять это значение в коде нежелательно, число π можно смоделировать с помощью следующей константы:

```
namespace ConstData
{
    class MyMathClass
    {
        public const double PI = 3.14;
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Fun with Const *****\n");
            Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
            // Ошибка! Константу изменять нельзя!
            // MyMathClass.PI = 3.1444;
            Console.ReadLine();
        }
    }
}
```

Обратите внимание, что ссылка на константные данные, определенные в классе `MyMathClass`, производится с применением префикса в виде имени класса (т.е. `MyMathClass.PI`). Причина в том, что константные поля класса являются неявно *статическими*. Однако допустимо определять локальные константные данные и обращаться к ним внутри области действия метода или свойства, например:

```
static void LocalConstStringVariable()
{
    // Доступ к локальным константным данным можно получать напрямую.
    const string fixedStr = "Fixed string Data";
    Console.WriteLine(fixedStr);
    // Ошибка!
    // fixedStr = "This will not work!";
}
```

Независимо от того, где вы определяете константную часть данных, всегда помните о том, что начальное значение, присваиваемое константе, должно быть указано в момент ее определения. Следовательно, если вы модифицируете класс `MyMathClass` так, чтобы значение `PI` присваивалось внутри конструктора класса, то получите ошибку на этапе компиляции:

```
class MyMathClass
{
    // Попытка установить PI в конструкторе?
    public const double PI;
    public MyMathClass()
    {
        // Невозможно - присваивание должно осуществляться в момент объявления.
        PI = 3.14;
    }
}
```

Такое ограничение связано с тем фактом, что значение константных данных должно быть известно на этапе компиляции. Как известно, конструкторы (или любые другие методы) вызываются во время выполнения.

Понятие полей, допускающих только чтение

С константными данными тесно связано понятие *полей данных, доступных только для чтения* (которое не следует путать со свойствами только для чтения). Подобно константе поле только для чтения не может быть изменено после первоначального присваивания. Тем не менее, в отличие от константы значение, присваиваемое такому полю, может быть определено во время выполнения и потому может на законном основании присваиваться внутри конструктора, но больше нигде.

Поле только для чтения полезно в ситуации, когда значение не известно вплоть до стадии выполнения (возможно из-за того, что для его получения необходимо прочитать внешний файл), но нужно гарантировать, что впоследствии оно не будет изменяться. В целях иллюстрации рассмотрим следующую модификацию класса `MyMathClass`:

```
class MyMathClass
{
    // Поля только для чтения могут присваиваться
    // в конструкторах, но больше нигде.
    public readonly double PI;
    public MyMathClass ()
    {
        PI = 3.14;
    }
}
```

Любая попытка выполнить присваивание полю, помеченному как `readonly`, за пределами конструктора приведет к ошибке на этапе компиляции:

```
class MyMathClass
{
    public readonly double PI;
    public MyMathClass ()
    {
        PI = 3.14;
    }
    // Ошибка!
    public void ChangePI()
    { PI = 3.144444;}
}
```

Статические поля, допускающие только чтение

В отличие от константных полей поля, допускающие только чтение, не являются неявно статическими. Таким образом, если необходимо предоставить доступ к `PI` на уровне класса, то придется явно использовать ключевое слово `static`. Если значение статического поля только для чтения известно на этапе компиляции, тогда начальное присваивание выглядит очень похожим на такое присваивание в случае константы (однако в этой ситуации проще применить ключевое слово `const`, потому что поле данных присваивается в момент его объявления):

```
class MyMathClass
{
    public static readonly double PI = 3.14;
}
class Program
{
    static void Main(string[] args)
    {
```

```

        Console.WriteLine("***** Fun with Const *****");
        Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
        Console.ReadLine();
    }
}

```

Тем не менее, если значение статического поля только для чтения не известно вплоть до времени выполнения, то должен использоваться статический конструктор, как было описано ранее в главе:

```

class MyMathClass
{
    public static readonly double PI;
    static MyMathClass()
    { PI = 3.14; }
}

```

Исходный код. Проект ConstData доступен в подкаталоге Chapter_5.

Понятие частичных классов

Последняя, но не менее важная тема связана с пониманием роли ключевого слова `partial` языка C#. Класс производственного уровня вполне может насчитывать многие сотни (если не тысячи) строк кода внутри единственного файла *.cs. Как обнаруживается, при создании классов нередко порядочная часть стереотипного кода, будучи однажды обдуманной, может по существу игнорироваться. Например, поля данных, свойства и конструкторы обычно остаются неизменными во время эксплуатации, в то время как методы имеют тенденцию довольно часто пересматриваться из-за обновления алгоритмов и т.п.

В языке C# одиночный класс можно разносить по нескольким файлам кода для отделения стереотипного кода от более полезных (и сложных) членов. Чтобы продемонстрировать ситуацию, когда частичные классы могут быть удобными, загрузим ранее созданный проект EmployeeApp в Visual Studio и откроем файл Employee.cs для редактирования. Как вы помните, этот единственный файл содержит код для всех аспектов класса:

```

class Employee
{
    // Поля данных
    // Конструкторы
    // Методы
    // Свойства
}

```

Применяя частичные классы, вы могли бы перенести (скажем) свойства, конструкторы и поля данных в новый файл по имени Employee.Core.cs (имя файла к делу не относится). Первый шаг предусматривает добавление ключевого слова `partial` к текущему определению класса и вырезание кода, подлежащего помещению в новый файл:

```

// Employee.cs
partial class Employee
{
    // Методы
    // Свойства
}

```

Далее предположив, что к проекту был добавлен новый файл класса, в него можно переместить поля данных и конструкторы с помощью простой операции вырезания и вставки. Кроме того, вы *должны* добавить ключевое слово `partial` к этому аспекту определения класса. Вот пример:

```
// Employee.Core.cs
partial class Employee
{
    // Поля данных
    // Конструкторы
}
```

На заметку! Не забывайте, что каждый аспект определения частичного класса должен быть помечен ключевым словом `partial`!

После компиляции модифицированного проекта вы не должны заметить вообще никакой разницы. Вся идея, положенная в основу частичного класса, касается только стадии проектирования. Как только приложение скомпилировано, в сборке оказывается один целостный класс. Единственное требование при определении частичных классов связано с тем, что разные части должны иметь одно и то же имя класса и находиться внутри того же самого пространства имен `.NET`.

Сценарии использования для частичных классов?

Теперь, когда вы понимаете механизм определения частичного класса, вас может интересовать, когда (и почему) он может потребоваться. Откровенно говоря, определения частичных классов применяются не слишком часто. Однако среда Visual Studio постоянно использует их в фоновом режиме. Например, если вы строите графический пользовательский интерфейс с применением инфраструктуры Windows Presentation Foundation (WPF), то заметите, что Visual Studio помещает весь сгенерированный визуальным конструктором код в отдельный файл частичного класса, позволяя вам сосредоточиться на специальной программной логике (не отвлекаясь на код, сгенерированный конструктором).

Исходный код. Проект `EmployeeAppPartial` доступен в подкаталоге `Chapter_5`.

Резюме

Целью главы было ознакомление вас с ролью типа класса `C#`. Вы видели, что классы могут иметь любое количество *конструкторов*, которые позволяют пользователю объекта устанавливать состояние объекта при его создании. В главе также было продемонстрировано несколько приемов проектирования классов (и связанных с ними ключевых слов). Вспомните, что ключевое слово `this` используется для получения доступа к текущему объекту, ключевое слово `static` дает возможность определять поля и члены, привязанные к уровню класса (в отличие от объекта), а ключевое слово `const` и модификатор `readonly` позволяют определять элементы данных, которые никогда не изменяются после первоначальной установки.

Большая часть главы была посвящена деталям первого принципа ООП — инкапсуляции. Вы узнали о модификаторах доступа `C#` и роли свойств типа, о синтаксисе инициализации объектов и о частичных классах. Теперь вы готовы перейти к чтению следующей главы, в которой речь пойдет о построении семейства взаимосвязанных классов с применением наследования и полиморфизма.

ГЛАВА 6

Наследование и полиморфизм

В главе 5 рассматривался первый основной принцип объектно-ориентированного программирования (ООП) — инкапсуляция. Вы узнали, как строить отдельный четко определенный тип класса с конструкторами и разнообразными членами (полями, свойствами, методами, константами и полями только для чтения). В настоящей главе мы сосредоточим внимание на оставшихся двух принципах ООП: наследовании и полиморфизме.

Прежде всего, вы научитесь строить семейства связанных классов с применением наследования. Как будет показано, такая форма многократного использования кода позволяет определять в родительском классе общую функциональность, которая может быть задействована, а возможно и модифицирована в дочерних классах. В ходе изложения вы узнаете, как устанавливать *полиморфный интерфейс* в иерархиях классов, используя виртуальные и абстрактные члены, а также о роли явного приведения.

Глава завершится исследованием роли изначального родительского класса в библиотеках базовых классов .NET — `System.Object`.

Базовый механизм наследования

Вспомните из главы 5, что *наследование* — это аспект ООП, упрощающий повторное использование кода. Говоря более точно, встречаются две разновидности повторного использования кода: наследование (отношение “является”) и модель включения/делегации (отношение “имеет”). Давайте начнем текущую главу с рассмотрения классической модели наследования, т.е. отношения “является”.

Когда вы устанавливаете между классами отношение “является”, то тем самым строите зависимость между двумя или более типами классов. Основная идея, лежащая в основе классического наследования, состоит в том, что новые классы могут создаваться с применением существующих классов как отправной точки. В качестве простого примера создадим новый проект консольного приложения по имени `BasicInheritance`. Предположим, что спроектирован класс `Car`, который моделирует ряд базовых деталей автомобиля:

```
// Простой базовый класс.  
class Car  
{  
    public readonly int maxSpeed;  
    private int currSpeed;
```

```

public Car(int max)
{
    maxSpeed = max;
}
public Car()
{
    maxSpeed = 55;
}
public int Speed
{
    get { return currSpeed; }
    set
    {
        currSpeed = value;
        if (currSpeed > maxSpeed)
        {
            currSpeed = maxSpeed;
        }
    }
}
}

```

Обратите внимание, что класс `Car` использует службы инкапсуляции для управления доступом к закрытому полю `currSpeed` посредством открытого свойства по имени `Speed`. В данный момент с типом `Car` можно работать следующим образом:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Basic Inheritance *****\n");
    // Создать объект Car и установить максимальную скорость.
    Car myCar = new Car(80);
    // Установить текущую скорость и вывести ее на консоль.
    myCar.Speed = 50;
    Console.WriteLine("My car is going {0} MPH", myCar.Speed);
    Console.ReadLine();
}

```

Указание родительского класса для существующего класса

Теперь предположим, что планируется построить новый класс по имени `MiniVan`. Подобно базовому классу `Car` вы хотите определить класс `MiniVan` так, чтобы он поддерживал данные для максимальной и текущей скоростей и свойство по имени `Speed`, которое позволило бы пользователю модифицировать состояние объекта. Очевидно, что классы `Car` и `MiniVan` взаимосвязаны; фактически можно сказать, что `MiniVan` “является” разновидностью `Car`. Отношение “является” (формально называемое *классическим наследованием*) позволяет строить новые определения классов, которые расширяют функциональность существующих классов.

Существующий класс, который будет служить основой для нового класса, называется *базовым классом*, *суперклассом* или *родительским классом*. Роль базового класса заключается в определении всех общих данных и членов для классов, которые его расширяют. Расширяющие классы формально называются *производными* или *дочерними* классами. В языке C# для установления между классами отношения “является” применяется операция двоеточия в определении класса. Пусть написан следующий новый класс `MiniVan`:

```
// MiniVan "является" Car.
class MiniVan : Car
{
}
```

В текущее время в новом классе какие-либо члены не определены. Так чего же мы достигли за счет наследования MiniVan от базового класса Car? Выразаясь просто, объекты MiniVan теперь имеют доступ ко всем открытым членам, определенным внутри базового класса.

На заметку! Несмотря на то что конструкторы обычно определяются как открытые, производный класс никогда не наследует конструкторы родительского класса. Конструкторы используются для создания только экземпляра класса, внутри которого они определены, но к ним можно обращаться в производном классе через построение цепочки вызовов конструкторов, как будет показано далее.

Учитывая отношение между этими двумя типами классов, вот как можно работать с классом MiniVan:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Basic Inheritance *****\n");
    ...
    // Создать объект MiniVan.
    MiniVan myVan = new MiniVan();
    myVan.Speed = 10;
    Console.WriteLine("My van is going {0} MPH",
        myVan.Speed);
    Console.ReadLine();
}
```

Обратите внимание, что хотя в класс MiniVan никакие члены не добавлялись, в нем есть прямой доступ к открытому свойству Speed родительского класса; тем самым обеспечивается повторное использование кода. Такой подход намного лучше, чем создание класса MiniVan, который имеет те же самые члены, что и класс Car, скажем, свойство Speed. Дублирование кода в двух классах приводит к необходимости сопровождения двух порций кода, что определенно будет непродуктивным расходом времени.

Всегда помните о том, что наследование предохраняет инкапсуляцию, а потому следующий код вызовет ошибку на этапе компиляции, т.к. закрытые члены не могут быть доступны через объектную ссылку:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Basic Inheritance *****\n");
    ...
    // Создать объект MiniVan.
    MiniVan myVan = new MiniVan();
    myVan.Speed = 10;
    Console.WriteLine("My van is going {0} MPH",
        myVan.Speed);
    // Ошибка! Доступ к закрытым членам невозможен!
    myVan.currSpeed = 55;
    Console.ReadLine();
}
```

В качестве связанного замечания: даже когда класс MiniVan определяет собственный набор членов, он по-прежнему не будет располагать возможностью доступа к любым

закрытым членам базового класса `Car`. Не забывайте, что закрытые члены доступны только внутри класса, в котором они определены. Например, показанный ниже метод в `MiniVan` приведет к ошибке на этапе компиляции:

```
// Класс MiniVan является производным от Car.
class MiniVan : Car
{
    public void TestMethod()
    {
        // Нормально! Доступ к открытым членам родительского
        // типа в производном типе возможен.
        Speed = 10;

        // Ошибка! Нельзя обращаться к закрытым членам
        // родительского типа из производного типа!
        currSpeed = 10;
    }
}
```

Замечание относительно множества базовых классов

Говоря о базовых классах, важно иметь в виду, что язык C# требует, чтобы отдельно взятый класс имел в точности один непосредственный базовый класс. Создать тип класса, который был бы производным напрямую от двух и более базовых классов, невозможно (такой прием, поддерживаемый в неуправляемом языке C++, известен как *множественное наследование*). Попытка создать класс, для которого указаны два непосредственных родительских класса, как продемонстрировано в следующем коде, приведет к ошибке на этапе компиляции:

```
// Недопустимо! Множественное наследование
// классов в языке C# не разрешено!
class WontWork
    : BaseClassOne, BaseClassTwo
{ }
```

В главе 8 вы увидите, что платформа .NET позволяет классу или структуре реализовывать любое количество дискретных интерфейсов. Таким способом тип C# может поддерживать несколько линий поведения, одновременно избегая сложностей, которые связаны с множественным наследованием. К слову, в то время как класс может иметь только один непосредственный базовый класс, интерфейс разрешено наследовать от множества других интерфейсов. Применяя этот подход, можно строить развитые иерархии интерфейсов, которые моделируют сложные линии поведения (см. главу 8).

Ключевое слово `sealed`

Язык C# предлагает еще одно ключевое слово, `sealed`, которое предотвращает наследование. Когда класс помечен как `sealed` (запечатанный), компилятор не позволяет создавать классы, производные от него. Например, пусть вы приняли решение о том, что дальнейшее расширение класса `MiniVan` не имеет смысла:

```
// Класс MiniVan не может быть расширен!
sealed class MiniVan : Car
{
}
```

Если вы или ваш коллега попытаетесь унаследовать от запечатанного класса `MiniVan`, то получите ошибку на этапе компиляции:


```
// Ошибка! Нельзя расширять класс, помеченный ключевым словом sealed'
class DeluxeMiniVan
    : MiniVan
{ }
```

Чаще всего запечатывание класса имеет наибольший смысл, когда проектируется обслуживающий класс. Скажем, в пространстве имен `System` определены многочисленные запечатанные классы. В этом несложно убедиться, открыв браузер объектов в Visual Studio (через меню View (Вид)) и выбрав класс `String`, который определен в пространстве имен `System` внутри сборки `mscorlib.dll`. На рис. 6.1 обратите внимание на значок, используемый для обозначения класса `sealed`.

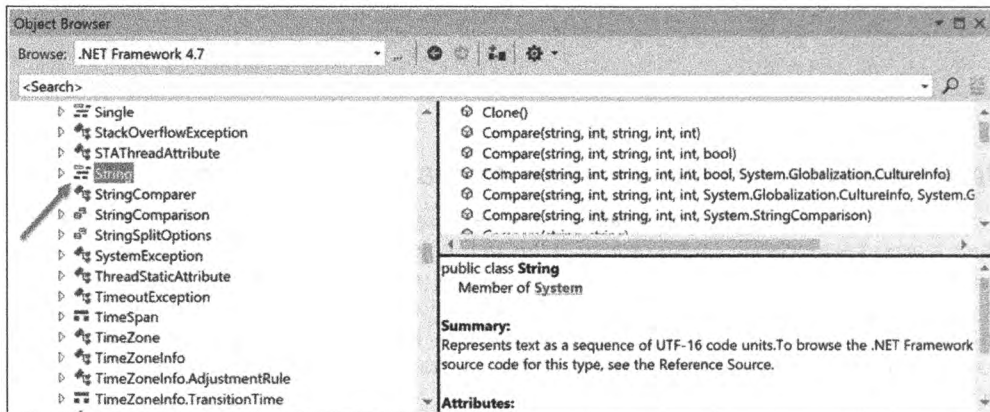


Рис. 6.1. В библиотеках базовых классов определено множество запечатанных классов, таких как `System.String`

Таким образом, как и в случае `MiniVan`, если вы попытаетесь построить новый класс, который расширял бы `System.String`, то получите ошибку на этапе компиляции:

```
// Еще одна ошибка! Нельзя расширять класс, помеченный как sealed!
class MyString
    : String
{ }
```

На заметку! В главе 4 вы узнали о том, что структуры C# всегда неявно запечатаны (см. табл. 4.3). Следовательно, создать структуру, производную от другой структуры, класс, производный от структуры, или структуру, производную от класса, невозможно. Структуры могут применяться для моделирования только отдельных, атомарных, определяемых пользователем типов. Если вы хотите задействовать отношение “является”, тогда должны использовать классы.

Нетрудно догадаться, что есть многие другие детали наследования, о которых вы узнаете в оставшемся материале главы. Пока просто примите к сведению, что операция двоеточия позволяет устанавливать отношения “базовый–производный” между классами, а ключевое слово `sealed` предотвращает последующее наследование.

Корректировка диаграмм классов Visual Studio

В главе 2 кратко упоминалось о том, что среда Visual Studio позволяет устанавливать отношения “базовый–производный” между классами визуальным образом во вре-

мя проектирования. Для работы с указанным аспектом IDE-среды сначала понадобится добавить в текущий проект новый файл диаграммы классов. Выберите пункт меню Project⇒Add New Item (Проект⇒Добавить новый элемент) и щелкните на значке Class Diagram (Диаграмма классов); на рис. 6.2 файл был переименован с ClassDiagram1.cd на Cars.cd.

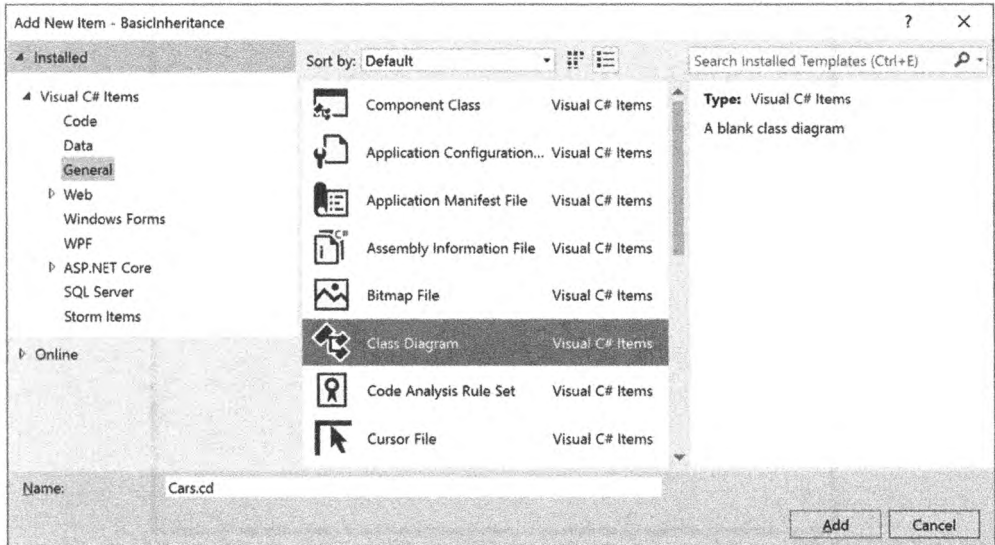


Рис. 6.2. Добавление новой диаграммы классов

После щелчка на кнопке Add (Добавить) отобразится пустая поверхность проектирования. Чтобы добавить типы в визуальный конструктор классов, просто перетаскивайте на эту поверхность каждый файл из окна Solution Explorer (Проводник решений). Также вспомните, что удаление элемента из визуального конструктора (путем его выбора и нажатия клавиши <Delete>) не приводит к уничтожению ассоциированного с ним исходного кода, а просто убирает элемент из поверхности конструктора. Текущая иерархия классов показана на рис. 6.3.

Как говорилось в главе 2, помимо простого отображения отношений между типами внутри текущего приложения можно также создавать новые типы и наполнять их членами, применяя панель инструментов конструктора классов и окно Class Details (Детали класса).

При желании можете свободно использовать указанные визуальные инструменты во время проработки оставшихся глав книги. Однако всегда анализируйте сгенерированный код, чтобы четко понимать, что эти инструменты для вас сделали.

Исходный код. Проект BasicInheritance доступен в подкаталоге Chapter_6.

Второй принцип ООП: детали наследования

Теперь, когда вы видели базовый синтаксис наследования, давайте построим более сложный пример и рассмотрим многочисленные детали построения иерархий классов. Мы снова обратимся к классу Employee, который был спроектирован в главе 5. Первым делом необходимо создать новый проект консольного приложения C# по имени Employees.

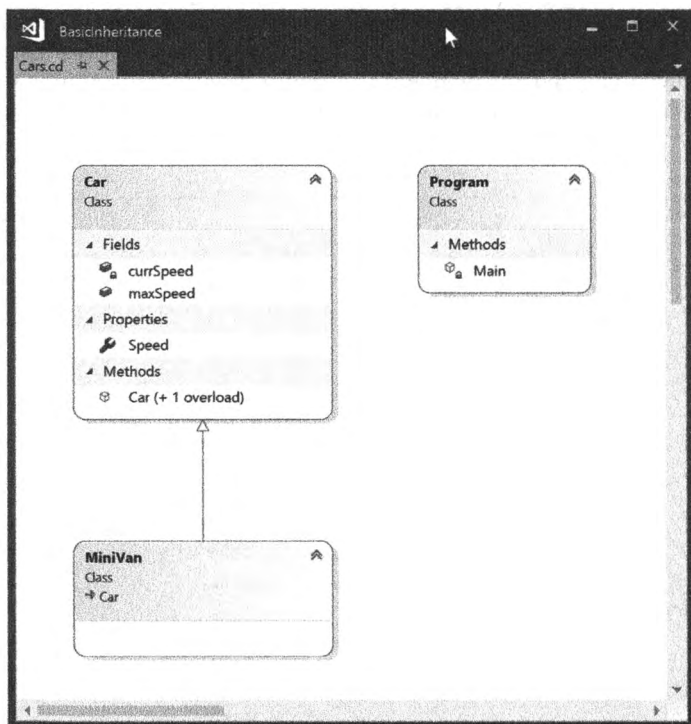


Рис. 6.3. Визуальный конструктор Visual Studio

Далее выберите пункт меню **Project** ⇒ **Add Existing Item** (Проект ⇒ Добавить существующий элемент) и перейдите к местоположению файлов `Employee.cs` и `Employee.Core.cs`, созданных в примере `EmployeeApp` из главы 5. Выделите оба файла (щелчком при нажатой клавише **<Ctrl>**) и щелкните на кнопке **Add** (Добавить). Среда Visual Studio отреагирует копированием каждого файла в текущий проект (имея дело с копиями, вам не придется беспокоиться о том, что исходные файлы проекта из главы 5 случайно будут изменены).

Прежде чем приступить к построению каких-то производных классов, следует уделить внимание одной детали. Поскольку первоначальный класс `Employee` был создан в проекте по имени `EmployeeApp`, он находится внутри идентично названного пространства имен `.NET`. Пространства имен подробно рассматриваются в главе 14; тем не менее, ради простоты переименуйте текущее пространство имен (в обоих файлах) на `Employees`, чтобы оно совпадало с именем нового проекта:

```
// Не забудьте изменить название пространства имен в обоих файлах C#!
namespace Employees
{
    partial class Employee
    { ... }
}
```

На заметку! В качестве проверки работоспособности скомпилируйте и запустите новый проект, нажав **<Ctrl+F5>**. Пока что программа ничего не делает, но это позволит удостовериться в отсутствии ошибок на этапе компиляции.

Нашей целью является создание семейства классов, которые моделируют разнообразные типы сотрудников в компании. Предположим, что необходимо задействовать

функциональность класса Employee при создании двух новых классов (SalesPerson и Manager). Новый класс SalesPerson “является” Employee (как и Manager). Вспомните, что в модели классического наследования базовые классы (вроде Employee) обычно применяются для определения характеристик, общих для всех наследников. Подклассы (такие как SalesPerson и Manager) расширяют общую функциональность, добавляя к ней специфическую функциональность.

В настоящем примере мы будем считать, что класс Manager расширяет Employee, сохраняя количество фондовых опционов, тогда как класс SalesPerson поддерживает хранение количества продаж. Добавьте новый файл класса (Manager.cs), в котором определяется класс Manager со следующим автоматическим свойством:

```
// Менеджерам нужно знать количество их фондовых опционов.
class Manager : Employee
{
    public int StockOptions { get; set; }
}
```

Затем добавьте еще один новый файл класса (SalesPerson.cs), в котором определен класс SalesPerson с подходящим автоматическим свойством:

```
// Продавцам нужно знать количество продаж.
class SalesPerson : Employee
{
    public int SalesNumber { get; set; }
}
```

После того как отношение “является” установлено, классы SalesPerson и Manager автоматически наследуют все открытые члены базового класса Employee. В целях иллюстрации модифицируем метод Main(), как показано ниже:

```
// Создание объекта подкласса и доступ к функциональности базового класса.
static void Main(string[] args)
{
    Console.WriteLine("***** The Employee Class Hierarchy *****\n");
    SalesPerson fred = new SalesPerson();
    fred.Age = 31;
    fred.Name = "Fred";
    fred.SalesNumber = 50;
    Console.ReadLine();
}
```

Управление созданием объектов базового класса с помощью ключевого слова base

В настоящий момент объекты классов SalesPerson и Manager могут создаваться только с использованием “бесплатно полученного” стандартного конструктора (см. главу 5). Памятуя о данном факте, предположим, что в класс Manager добавлен новый конструктор с шестью аргументами, который вызывается следующим образом:

```
static void Main(string[] args)
{
    ...
    // Предположим, что у Manager есть конструктор с такой сигнатурой:
    // (string fullName, int age, int empID,
    // float currPay, string ssn, int numbofOpts)
    Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
    Console.ReadLine();
}
```

Взглянув на список параметров, легко заметить, что большинство аргументов должно быть сохранено в переменных-членах, определенных в базовом классе `Employee`. Чтобы сделать это, в классе `Manager` можно было бы реализовать показанный ниже специальный конструктор:

```
public Manager(string fullName, int age, int empID,
               float currPay, string ssn, int numbofOpts)
{
    // Это свойство определено в классе Manager.
    StockOptions = numbofOpts;

    // Присвоить входные параметры, используя
    // унаследованные свойства родительского класса.
    ID = empID;
    Age = age;
    Name = fullName;
    Pay = currPay;

    // Если свойство SSN окажется доступным только для чтения,
    // тогда здесь возникнет ошибка на этапе компиляции!
    SocialSecurityNumber = ssn;
}
```

Первая проблема с таким подходом в том, что если любое свойство определено как допускающее только чтение (например, свойство `SocialSecurityNumber`), то присвоить значение входного параметра `string` данному полю не удастся, как можно видеть в финальном операторе специального конструктора.

Вторая проблема связана с тем, что был косвенно создан довольно неэффективный конструктор, учитывая тот факт, что в C# стандартный конструктор базового класса вызывается автоматически перед выполнением логики конструктора производного класса, если не указано иначе. После этого момента текущая реализация имеет доступ к многочисленным открытым свойствам базового класса `Employee` для установки его состояния. Таким образом, во время создания объекта `Manager` на самом деле выполнялось семь действий (обращения к пяти унаследованным свойствам и двум конструкторам).

Для оптимизации создания объектов производного класса необходимо корректно реализовать конструкторы подкласса, чтобы они явно вызывали подходящий специальный конструктор базового класса вместо стандартного конструктора. Подобным образом можно сократить количество вызовов инициализации унаследованных членов (что уменьшит время обработки). Первым делом понадобится обеспечить в родительском классе `Employee` следующий конструктор с пятью параметрами:

```
// Добавить в базовый класс Employee.
public Employee(string name, int age, int id, float pay, string ssn)
    :this(name, age, id, pay)
{
    empSSN = ssn;
}
```

Модифицируем специальный конструктор в классе `Manager`, применив ключевое слово `base`:

```
public Manager(string fullName, int age, int empID,
               float currPay, string ssn, int numbofOpts)
    : base(fullName, age, empID, currPay, ssn)
{
    // Это свойство определено в классе Manager.
    StockOptions = numbofOpts;
}
```

Здесь ключевое слово `base` ссылается на сигнатуру конструктора (подобно синтаксису, используемому для объединения конструкторов одиночного класса в цепочку через ключевое слово `this`, как обсуждалось в главе 5), что всегда указывает производному конструктору на необходимость передачи данных конструктору непосредственного родительского класса. В рассматриваемой ситуации явно вызывается конструктор с пятью параметрами, определенный в `Employee`, что избавляет от излишних обращений во время создания объекта дочернего класса. Специальный конструктор класса `SalesPerson` выглядит в основном идентично:

```
// В качестве общего правила запомните, что все подклассы должны
// явно вызывать подходящий конструктор базового класса.
public SalesPerson(string fullName, int age, int empID, float currPay,
    string ssn, int numbOfSales) : base(fullName, age, empID, currPay, ssn)
{
    // Это принадлежит нам!
    SalesNumber = numbOfSales;
}
```

На заметку! Ключевое слово `base` можно применять всякий раз, когда подкласс желает обратиться к открытому или защищенному члену, определенному в родительском классе. Использование этого ключевого слова не ограничивается логикой конструктора. Вы увидите примеры применения ключевого слова `base` в подобной манере позже в главе при рассмотрении полиморфизма.

Наконец, вспомните, что после добавления к определению класса специального конструктора стандартный конструктор молча удаляется. Следовательно, не забудьте переопределить стандартный конструктор для классов `SalesPerson` и `Manager`. Вот пример:

```
// Аналогичным образом переопределите стандартный
// конструктор также и в классе Manager.
public SalesPerson() {}
```

Хранение секретов семейства: ключевое слово `protected`

Как вы уже знаете, открытые элементы напрямую доступны отовсюду, в то время как закрытые элементы могут быть доступны только в классе, где они определены. Вспомните из главы 5, что C# опережает многие другие современные объектные языки и предоставляет дополнительное ключевое слово для определения доступности членов — `protected` (защищенный).

Когда базовый класс определяет защищенные данные или защищенные члены, он устанавливает набор элементов, которые могут быть непосредственно доступны любому наследнику. Если вы хотите разрешить дочерним классам `SalesPerson` и `Manager` напрямую обращаться к разделу данных, который определен в `Employee`, то модифицируйте исходный класс `Employee`, как показано ниже:

```
// Защищенные данные состояния.
partial class Employee
{
    // Производные классы теперь могут иметь прямой доступ к этой информации.
    protected string empName;
    protected int empID;
    protected float currPay;
    protected int empAge;
    protected string empSSN;
    ...
}
```

Преимущество определения защищенных членов в базовом классе заключается в том, что производным классам больше не придется обращаться к данным косвенно, ис-

пользуя открытые методы и свойства. Разумеется, подходу присущ и недостаток: когда производный класс имеет прямой доступ к внутренним данным своего родителя, есть вероятность непредумышленного обхода существующих бизнес-правил, которые реализованы внутри открытых свойств. Определяя защищенные члены, вы создаете уровень доверия между родительским классом и дочерним классом, т.к. компилятор не будет перехватывать какие-либо нарушения бизнес-правил, предусмотренных для типа.

Наконец, имейте в виду, что с точки зрения пользователя объекта защищенные данные расцениваются как закрытые (поскольку пользователь находится "снаружи" семейства). По указанной причине следующий код недопустим:

```
static void Main(string[] args)
{
    // Ошибка! Доступ к защищенным данным из клиентского кода невозможен!
    Employee emp = new Employee();
    emp.empName = "Fred";
}
```

На заметку! Несмотря на то что защищенные поля данных могут нарушить инкапсуляцию, определять защищенные методы вполне безопасно (и полезно). При построении иерархий классов обычно приходится определять набор методов, которые предназначены для применения только производными типами, но не внешним миром.

Добавление запечатанного класса

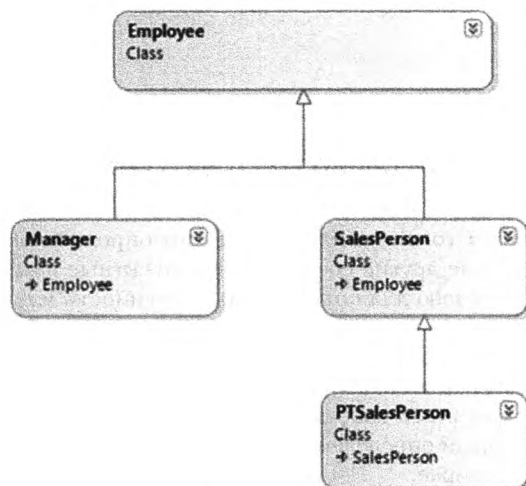


Рис. 6.4. Класс PTSalesPerson

Вспомните, что *запечатанный* класс не может быть расширен другими классами. Как уже упоминалось, такой прием чаще всего используется при проектировании обслуживающих классов. Тем не менее, при построении иерархий классов вы можете обнаружить, что определенная ветвь в цепочке наследования нуждается в "отсечении", т.к. дальнейшее ее расширение не имеет смысла. Для примера предположим, что вы добавили в приложение еще один класс (PTSalesPerson), который расширяет существующий тип SalesPerson. На рис. 6.4 показано текущее обновление.

Класс PTSalesPerson представляет продавца, который работает на условиях частичной занятости. В качестве варианта скажем, что нужно гаранти-

ровать отсутствие возможности создания подкласса PTSalesPerson. (В конце концов, какой смысл в дополнительной "частичной занятости" от имеющейся "частичной занятости"?). Чтобы предотвратить наследование от класса, необходимо применить ключевое слово `sealed`:

```
sealed class PTSalesPerson : SalesPerson
{
    public PTSalesPerson(string fullName, int age, int empID,
        float currPay, string ssn, int numbofSales)
        :base (fullName, age, empID, currPay, ssn, numbofSales)
    {
    }
}
```

```
// Остальные члены класса...
}
```

Реализация модели включения/делегации

Вы уже знаете, что повторное использование кода встречается в двух видах. Только что было продемонстрировано классическое отношение “является”. Перед тем, как мы начнем исследование третьего принципа ООП (полиморфизма), давайте взглянем на отношение “имеет” (также известное как *модель включения/делегации* или *агрегация*). Предположим, что создан новый класс, который моделирует пакет льгот для сотрудников:

```
// Этот новый тип будет функционировать как включаемый класс.
class BenefitPackage
{
    // Предположим, что есть другие члены, представляющие
    // медицинские/стоматологические программы и т.д.
    public double ComputePayDeduction()
    {
        return 125.0;
    }
}
```

Очевидно, что было бы довольно странно устанавливать отношение “является” между классом `BenefitPackage` и типами сотрудников. (Разве `Employee` “является” `BenefitPackage`? Вряд ли.) Однако должно быть ясно, что какое-то отношение между ними должно быть установлено. Короче говоря, нужно выразить идею о том, что каждый сотрудник “имеет” `BenefitPackage`. Для этого можно модифицировать определение класса `Employee` следующим образом:

```
// Теперь сотрудники имеют льготы.
partial class Employee
{
    // Содержит объект BenefitPackage.
    protected BenefitPackage empBenefits = new BenefitPackage();
    ...
}
```

На данной стадии вы имеете объект, который благополучно содержит в себе другой объект. Тем не менее, открытие доступа к функциональности содержащегося объекта внешнему миру требует делегации. *Делегация* — просто действие по добавлению во включающий класс открытых членов, которые работают с функциональностью содержащегося внутри объекта. Например, можно было бы изменить класс `Employee` так, чтобы он открывал доступ к включенному объекту `empBenefits` с применением специального свойства, а также использовать его функциональность внутренне посредством нового метода по имени `GetBenefitCost()`:

```
partial class Employee
{
    // Содержит объект BenefitPackage.
    protected BenefitPackage empBenefits = new BenefitPackage();

    // Открывает доступ к некоторому поведению, связанному со льготами.
    public double GetBenefitCost()
    { return empBenefits.ComputePayDeduction(); }

    // Открывает доступ к объекту через специальное свойство.
    public BenefitPackage Benefits
    {
```



```

        get { return empBenefits; }
        set { empBenefits = value; }
    }
    ...
}

```

В следующем обновленном методе Main() обратите внимание на взаимодействие с внутренним типом BenefitsPackage, который определен в типе Employee:

```

static void Main(string[] args)
{
    Console.WriteLine("***** The Employee Class Hierarchy *****\n");
    ...
    Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
    double cost = chucky.GetBenefitCost();
    Console.ReadLine();
}

```

Определения вложенных типов

В главе 5 кратко упоминалась концепция вложенных типов, которая является развитием рассмотренного выше отношения "имеет". В C# (а также в других языках .NET) допускается определять тип (перечисление, класс, интерфейс, структуру или делегат) прямо внутри области действия класса либо структуры. В таком случае вложенный (или "внутренний") тип считается членом охватывающего (или "внешнего") типа, и в глазах исполняющей системы им можно манипулировать как любым другим членом (полем, свойством, методом и событием). Синтаксис, применяемый для вложения типа, достаточно прост:

```

public class OuterClass
{
    // Открытый вложенный тип может использоваться кем угодно.
    public class PublicInnerClass {}

    // Закрытый вложенный тип может использоваться
    // только членами включающего класса.
    private class PrivateInnerClass {}
}

```

Хотя синтаксис довольно ясен, ситуации, в которых это может понадобиться, не настолько очевидны. Для того чтобы понять данный прием, рассмотрим характерные черты вложенных типов.

- Вложенные типы позволяют получить полный контроль над уровнем доступа внутреннего типа, потому что они могут быть объявлены как закрытые (вспомните, что невложенные классы нельзя объявлять с ключевым словом private).
- Поскольку вложенный тип является членом включающего класса, он может иметь доступ к закрытым членам этого включающего класса.
- Часто вложенный тип полезен только как вспомогательный для внешнего класса и не предназначен для использования во внешнем мире.

Когда тип включает в себя другой тип класса, он может создавать переменные-члены этого типа, как в случае любого другого элемента данных. Однако если с вложенным типом нужно работать за пределами включающего типа, тогда его придется уточнять именем включающего типа. Взгляните на приведенный ниже код:

```

static void Main(string[] args)
{
    // Создать и использовать объект открытого вложенного класса. Нормально!
}

```

```

OuterClass.PublicInnerClass inner;
inner = new OuterClass.PublicInnerClass();

// Ошибка на этапе компиляции! Доступ к закрытому вложенному классу невозможен!
OuterClass.PrivateInnerClass inner2;
inner2 = new OuterClass.PrivateInnerClass();
}

```

Для применения такой концепции в примере с сотрудниками предположим, что определение `BenefitPackage` теперь вложено непосредственно в класс `Employee`:

```

partial class Employee
{
    public class BenefitPackage
    {
        // Предположим, что есть другие члены, представляющие
        // медицинские/стоматологические программы и т.д.
        public double ComputePayDeduction()
        {
            return 125.0;
        }
    }
    ...
}

```

Процесс вложения может распространяться настолько “глубоко”, насколько требуется. Например, пусть необходимо создать перечисление по имени `BenefitPackageLevel`, документирующее разнообразные уровни льгот, которые может выбирать сотрудник. Чтобы программно обеспечить тесную связь между типами `Employee`, `BenefitPackage` и `BenefitPackageLevel`, перечисление можно вложить следующим образом:

```

// В класс Employee вложен класс BenefitPackage.
public partial class Employee
{
    // В класс BenefitPackage вложено перечисление BenefitPackageLevel.
    public class BenefitPackage
    {
        public enum BenefitPackageLevel
        {
            Standard, Gold, Platinum
        }
        public double ComputePayDeduction()
        {
            return 125.0;
        }
    }
    ...
}

```

Из-за отношений вложения вот как приходится использовать перечисление `BenefitPackageLevel`:

```

static void Main(string[] args)
{
    ...
    // Определить уровень льгот.
    Employee.BenefitPackage.BenefitPackageLevel myBenefitLevel =
        Employee.BenefitPackage.BenefitPackageLevel.Platinum;
    Console.ReadLine();
}

```

Итак, к настоящему моменту вы ознакомились с несколькими ключевыми словами (и концепциями), которые позволяют строить иерархии типов, связанных посредством классического наследования, включения и вложения. Не беспокойтесь, если пока еще не все детали ясны. На протяжении оставшихся глав книги будет построено немало иерархий. А теперь давайте перейдем к исследованию последнего принципа ООП — полиморфизма.

Третий принцип ООП: поддержка полиморфизма в C#

Вспомните, что в базовом классе `Employee` определен метод по имени `GiveBonus()`, который первоначально был реализован так:

```
public partial class Employee
{
    public void GiveBonus(float amount)
    {
        Pay += amount;
    }
    ...
}
```

Поскольку метод `GiveBonus()` был определен с ключевым словом `public`, бонусы можно раздавать продавцам и менеджерам (а также продавцам с частичной занятостью):

```
static void Main(string[] args)
{
    Console.WriteLine("***** The Employee Class Hierarchy *****\n");
    // Выдать каждому сотруднику бонус?
    Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
    chucky.GiveBonus(300);
    chucky.DisplayStats();
    Console.WriteLine();

    SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);
    fran.GiveBonus(200);
    fran.DisplayStats();
    Console.ReadLine();
}
```

Проблема с текущим проектным решением заключается в том, что открыто унаследованный метод `GiveBonus()` функционирует идентично для всех подклассов. В идеале при подсчете бонуса для штатного продавца и частично занятого продавца должно приниматься во внимание количество продаж. Возможно, менеджеры вместе с денежным вознаграждением должны получать дополнительные фондовые опционы. Учитывая это, вы однажды столкнетесь с интересным вопросом: “Как сделать так, чтобы связанные типы реагировали по-разному на один и тот же запрос?”. Попробуем найти на него ответ.

Ключевые слова `virtual` и `override`

Полиморфизм предоставляет подклассу способ определения собственной версии метода, определенного в его базовом классе, с применением процесса, который называется *переопределением метода*. Чтобы модернизировать текущее проектное решение, необходимо понимать смысл ключевых слов `virtual` и `override`. Если базовый класс желает определить метод, который *может быть* (но не обязательно) переопределен в подклассе, то он должен пометить его ключевым словом `virtual`:

```
partial class Employee
{
    // Теперь этот метод может быть переопределен в производном классе.
    public virtual void GiveBonus(float amount)
    {
        Pay += amount;
    }
    ...
}
```

На заметку! Методы, помеченные ключевым словом `virtual`, называются *виртуальными методами*.

Когда подкласс желает изменить реализацию деталей виртуального метода, он прибегает к помощи ключевого слова `override`. Например, классы `SalesPerson` и `Manager` могли бы переопределять метод `GiveBonus()`, как показано ниже (предположим, что класс `PTSalesPerson` не будет переопределять `GiveBonus()`, а потому просто наследует его версию из `SalesPerson`):

```
class SalesPerson : Employee
{
    ...
    // Бонус продавца зависит от количества продаж.
    public override void GiveBonus(float amount)
    {
        int salesBonus = 0;
        if (SalesNumber >= 0 && SalesNumber <= 100)
            salesBonus = 10;
        else
        {
            if (SalesNumber >= 101 && SalesNumber <= 200)
                salesBonus = 15;
            else
                salesBonus = 20;
        }
        base.GiveBonus(amount * salesBonus);
    }
}

class Manager : Employee
{
    ...
    public override void GiveBonus(float amount)
    {
        base.GiveBonus(amount);
        Random r = new Random();
        StockOptions += r.Next(500);
    }
}
```

Обратите внимание, что каждый переопределенный метод может задействовать стандартное поведение посредством ключевого слова `base`.

Таким образом, полностью повторять реализацию логики метода `GiveBonus()` вовсе не обязательно, а взамен можно повторно использовать (и расширять) стандартное поведение родительского класса.

Также предположим, что текущий метод `DisplayStats()` класса `Employee` объявлен виртуальным:

```
public virtual void DisplayStats()
{
    Console.WriteLine("Name: {0}", Name);
    Console.WriteLine("ID: {0}", ID);
    Console.WriteLine("Age: {0}", Age);
    Console.WriteLine("Pay: {0}", Pay);
    Console.WriteLine("SSN: {0}", SocialSecurityNumber);
}
```

Тогда каждый подкласс может переопределять метод `DisplayStats()` с целью отображения количества продаж (для продавцов) и текущих фондовых опционов (для менеджеров). Например, рассмотрим версию метода `DisplayStats()` из класса `Manager` (класс `SalesPerson` реализовывал бы метод `DisplayStats()` в похожей манере, выводя на консоль количество продаж):

```
public override void DisplayStats()
{
    base.DisplayStats();
    Console.WriteLine("Number of Stock Options: {0}", StockOptions);
}
```

Теперь, когда каждый подкласс может истолковывать эти виртуальные методы значащим для него образом, их экземпляры ведут себя как более независимые сущности:

```
static void Main(string[] args)
{
    Console.WriteLine("***** The Employee Class Hierarchy *****\n");
    // Лучшая система бонусов!
    Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
    chucky.GiveBonus(300);
    chucky.DisplayStats();
    Console.WriteLine();

    SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);
    fran.GiveBonus(200);
    fran.DisplayStats();
    Console.ReadLine();
}
```

Вот результат тестового запуска приложения в его текущем виде:

```
***** The Employee Class Hierarchy *****
Name: Chucky
ID: 92
Age: 50
Pay: 100300
SSN: 333-23-2322
Number of Stock Options: 9337

Name: Fran
ID: 93
Age: 43
Pay: 5000
SSN: 932-32-3232
Number of Sales: 31
```

Переопределение виртуальных членов в IDE-среде Visual Studio

Вы наверняка уже заметили, что при переопределении члена класса приходится вспоминать тип каждого параметра, не говоря уже об имени метода и соглашениях по

передаче параметров (`ref`, `out` и `params`). В Visual Studio доступно полезное средство IntelliSense, к которому можно обращаться при переопределении виртуального члена. Если вы наберете слово `override` внутри области действия типа класса (и затем нажмете клавишу пробела), то IntelliSense автоматически отобразит список всех допускающих переопределение членов родительского класса (рис. 6.5).

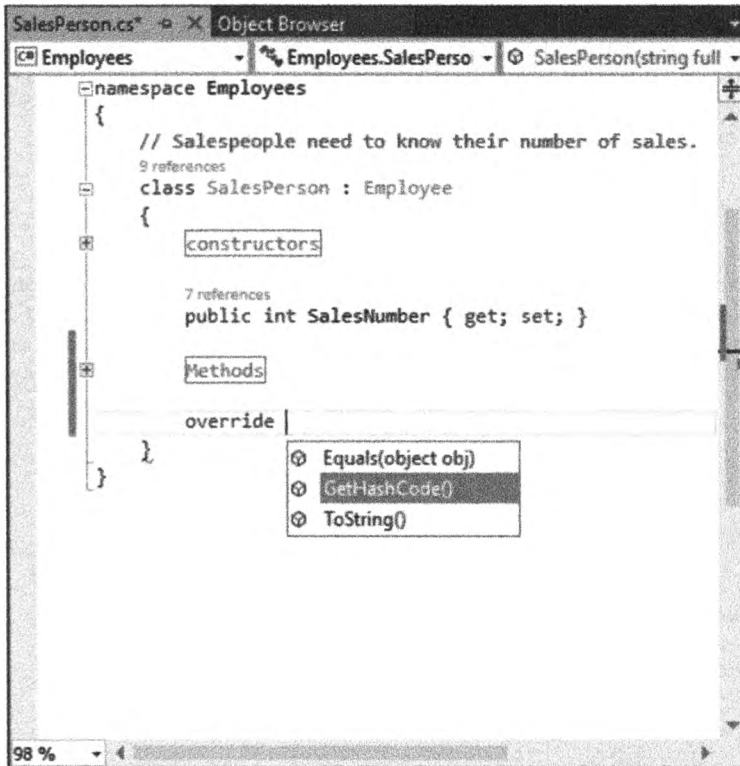


Рис. 6.5. Быстрый просмотр переопределяемых методов в Visual Studio

Если вы выберете член и нажмете клавишу <Enter>, то IDE-среда отреагирует автоматическим заполнением заглушки метода. Обратите внимание, что вы также получаете оператор кода, который вызывает родительскую версию виртуального члена (можете удалить эту строку, если она не нужна). Например, при использовании описанного приема для переопределения метода `DisplayStats()` вы обнаружите следующий автоматически сгенерированный код:

```

public override void DisplayStats()
{
    base.DisplayStats();
}
  
```

Запечатывание виртуальных членов

Вспомните, что к типу класса можно применить ключевое слово `sealed`, чтобы предотвратить расширение его поведения другими типами через наследование. Ранее класс `PTSalesPerson` был запечатан на основе предположения о том, что разработчикам не имеет смысла дальше расширять эту линию наследования.

Следует отметить, что временами желательно не запечатывать класс целиком, а просто предотвратить переопределение некоторых виртуальных методов в производных типах. В качестве примера предположим, что вы не хотите, чтобы продавцы с частичной занятостью получали специальные бонусы. Предотвратить переопределение виртуального метода `GiveBonus()` в классе `PTSalesPerson` можно, запечатав данный метод в классе `SalesPerson`:

```
// Класс SalesPerson запечатал метод GiveBonus()!
class SalesPerson : Employee
{
    ...
    public override sealed void GiveBonus(float amount)
    {
        ...
    }
}
```

Здесь класс `SalesPerson` на самом деле переопределяет виртуальный метод `GiveBonus()`, определенный в `Employee`, но явно помечает его как `sealed`. Таким образом, попытка переопределения метода `GiveBonus()` в классе `PTSalesPerson` приведет к ошибке на этапе компиляции:

```
sealed class PTSalesPerson : SalesPerson
{
    public PTSalesPerson(string fullName, int age, int empID,
        float currPay, string ssn, int numbOfSales)
        :base (fullName, age, empID, currPay, ssn, numbOfSales)
    {
    }

    // Ошибка на этапе компиляции! Переопределять этот метод
    // в классе PTSalesPerson нельзя, т.к. он был запечатан.
    public override void GiveBonus(float amount)
    {
    }
}
```

Абстрактные классы

В настоящий момент базовый класс `Employee` спроектирован так, что предоставляет различные данные-члены своим наследникам, а также предлагает два виртуальных метода (`GiveBonus()` и `DisplayStats()`), которые могут быть переопределены в наследниках. Хотя все это замечательно, у такого проектного решения имеется один весьма странный побочный эффект: можно напрямую создавать экземпляры базового класса `Employee`:

```
// Что это будет означать?
Employee X = new Employee();
```

В нашем примере базовый класс `Employee` служит единственной цели — определять общие члены для всех подклассов. По всем признакам мы не намерены позволять кому-либо создавать непосредственные экземпляры типа `Employee`, т.к. он концептуально чересчур общий. Например, если кто-то заявит, что он сотрудник, то тут же возникнет вопрос: сотрудник какого *рода* (консультант, инструктор, административный работник, литературный редактор, советник в правительстве)?

Учитывая, что многие базовые классы имеют тенденцию быть довольно расплывчатыми сущностями, намного более эффективным проектным решением для данного примера будет предотвращение возможности непосредственного создания в коде нового объекта `Employee`. В C# цели можно добиться за счет использования ключевого слова `abstract` в определении класса, создавая в итоге *абстрактный базовый класс*:

```
// Превращение класса Employee в абстрактный для
// предотвращения прямого создания его экземпляров.
abstract partial class Employee
{
    ...
}
```

Теперь попытка создания экземпляра класса `Employee` приводит к ошибке на этапе компиляции:

```
// Ошибка! Нельзя создавать экземпляр абстрактного класса!
Employee X = new Employee();
```

Определение класса, экземпляры которого нельзя создавать напрямую, на первый взгляд может показаться странным. Однако вспомните, что базовые классы (абстрактные или нет) полезны тем, что содержат все общие данные и функциональность для производных типов. Такая форма абстракции дает возможность считать, что “идея” сотрудника является полностью допустимой, просто это не конкретная сущность. Кроме того, необходимо понимать, что хотя *непосредственно* создавать экземпляры абстрактного класса невозможно, они все равно появляются в памяти при создании экземпляров производных классов. Таким образом, для абстрактных классов вполне нормально (и общепринято) определять любое количество конструкторов, которые вызываются косвенно, когда выделяется память под экземпляры производных классов.

На данной стадии у нас есть довольно интересная иерархия сотрудников. Мы добавим чуть больше функциональности к приложению позже, при рассмотрении правил приведения типов C#. А пока на рис. 6.6 представлено текущее проектное решение.

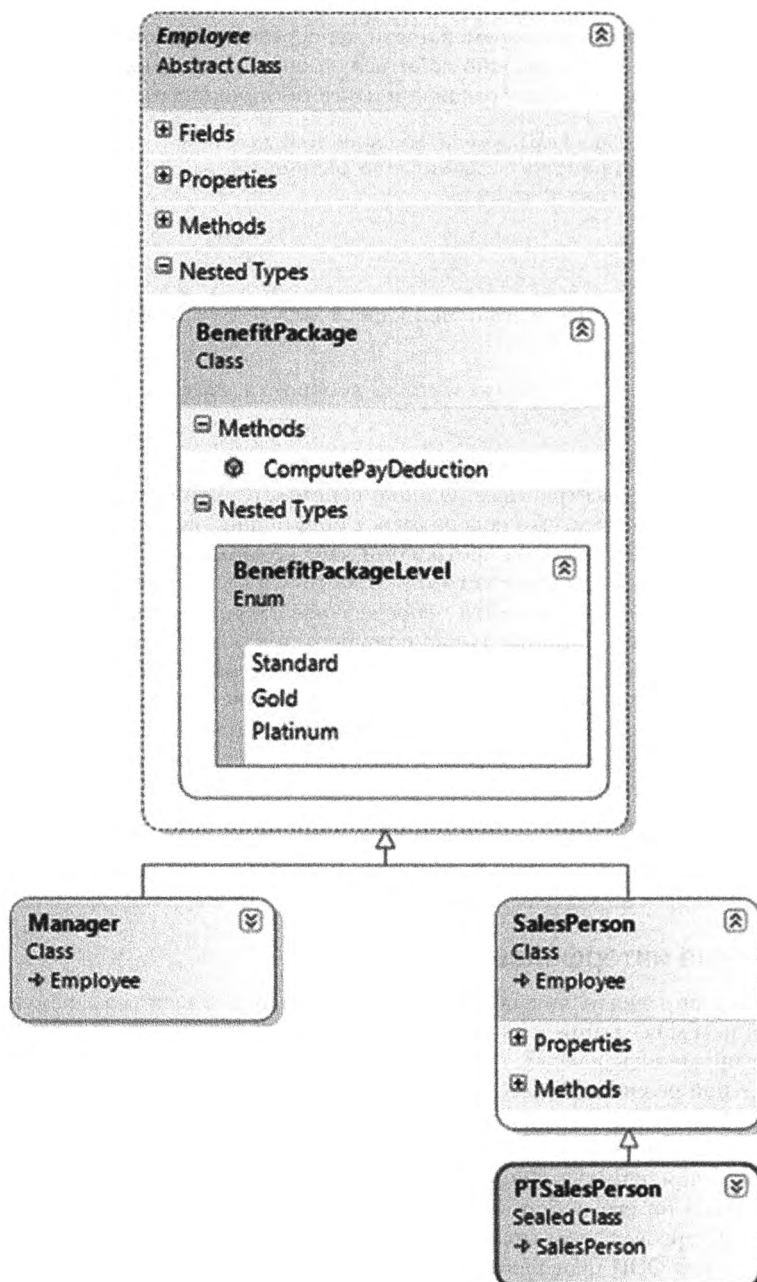
Исходный код. Проект `Employees` доступен в подкаталоге `Chapter_6`.

Полиморфные интерфейсы

Когда класс определен как абстрактный базовый (посредством ключевого слова `abstract`), в нем может определяться любое число *абстрактных членов*. Абстрактные члены могут применяться везде, где требуется определить член, который не предоставляет стандартной реализации, но *должен* приниматься во внимание каждым производным классом. Тем самым вы навязываете *полиморфный интерфейс* каждому наследнику, оставляя им задачу реализации конкретных деталей абстрактных методов.

Выражаясь упрощенно, полиморфный интерфейс абстрактного базового класса просто ссылается на его набор виртуальных и абстрактных методов. На самом деле это намного интереснее, чем может показаться на первый взгляд, поскольку данная характерная черта ООП позволяет строить легко расширяемые и гибкие приложения. В целях иллюстрации мы реализуем (и слегка модифицируем) иерархию фигур, кратко описанную в главе 5 во время обзора основных принципов ООП. Для начала создадим новый проект консольного приложения C# по имени `Shapes`.

На рис. 6.7 обратите внимание на то, что типы `Hexagon` и `Circle` расширяют базовый класс `Shape`. Как и любой базовый класс, `Shape` определяет набор членов (в данном случае свойство `PetName` и метод `Draw()`), общих для всех наследников.

Рис. 6.6. Иерархия классов **Employee**

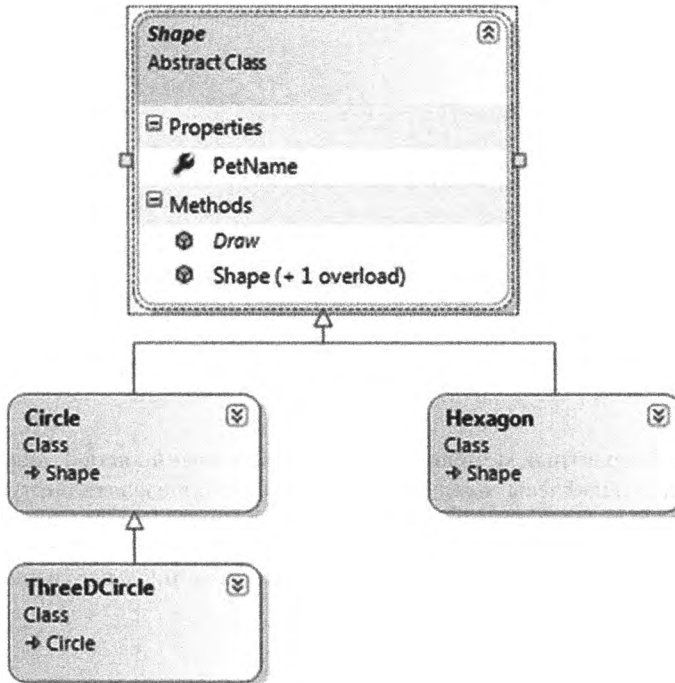


Рис. 6.7. Иерархия классов фигур

Во многом подобно иерархии классов сотрудников вы должны иметь возможность запретить создание экземпляров класса `Shape` напрямую, потому что он представляет слишком абстрактную концепцию. Чтобы предотвратить непосредственное создание экземпляров класса `Shape`, его можно определить как абстрактный класс. К тому же, учитывая, что производные типы должны уникальным образом реагировать на вызов метода `Draw()`, давайте пометим его как `virtual` и определим стандартную реализацию.

```
// Абстрактный базовый класс иерархии.
abstract class Shape
{
    public Shape(string name = "NoName")
    { PetName = name; }

    public string PetName { get; set; }

    // Единственный виртуальный метод.
    public virtual void Draw()
    {
        Console.WriteLine("Inside Shape.Draw()");
    }
}
```

Обратите внимание, что виртуальный метод `Draw()` предоставляет стандартную реализацию, которая просто выводит на консоль сообщение, информирующее о том, что вызван метод `Draw()` из базового класса `Shape`. Теперь вспомните, что когда метод помечен ключевым словом `virtual`, он поддерживает стандартную реализацию, которую автоматически наследуют все производные типы. Если дочерний класс так решит, то он *может* переопределить такой метод, но он не *обязан* это делать. Рассмотрим показанную ниже реализацию типов `Circle` и `Hexagon`:

```
// Класс Circle не переопределяет метод Draw().
class Circle : Shape
{
    public Circle() {}
    public Circle(string name) : base(name){}
}

// Класс Hexagon переопределяет метод Draw().
class Hexagon : Shape
{
    public Hexagon() {}
    public Hexagon(string name) : base(name){}
    public override void Draw()
    {
        Console.WriteLine("Drawing {0} the Hexagon", PetName);
    }
}
```

Полезность абстрактных методов становится совершенно ясной, как только вы снова вспомните, что подклассы *никогда не обязаны* переопределять виртуальные методы (как в случае Circle). Следовательно, если создать экземпляры типов Hexagon и Circle, то обнаружится, что Hexagon знает, как правильно “рисовать” себя (или, по крайней мере, выводить на консоль подходящее сообщение). Тем не менее, реакция Circle порядком сбивает с толку.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Polymorphism *****\n");
    Hexagon hex = new Hexagon("Beth");
    hex.Draw();
    Circle cir = new Circle("Cindy");
    // Вызывает реализацию базового класса!
    cir.Draw();
    Console.ReadLine();
}
```

Взгляните на вывод метода Main():

```
***** Fun with Polymorphism *****
```

```
Drawing Beth the Hexagon
```

```
Inside Shape.Draw()
```

Очевидно, что это не самое разумное проектное решение для текущей иерархии. Чтобы вынудить каждый дочерний класс переопределять метод Draw(), его можно определить как абстрактный метод класса Shape, т.е. какая-либо стандартная реализация вообще не предлагается. Для пометки метода как абстрактного в C# используется ключевое слово `abstract`. Обратите внимание, что абстрактные методы не предоставляют никакой реализации:

```
abstract class Shape
{
    // Вынудить все дочерние классы определять способ своей визуализации.
    public abstract void Draw();
    ...
}
```

На заметку! Абстрактные методы могут быть определены только в абстрактных классах, иначе возникнет ошибка на этапе компиляции.

Методы, помеченные как `abstract`, являются чистым протоколом. Они просто определяют имя, возвращаемый тип (если есть) и набор параметров (при необходимости). Здесь абстрактный класс `Shape` информирует производные типы о том, что у него есть метод по имени `Draw()`, который не принимает аргументов и ничего не возвращает. О необходимых деталях должен позаботиться производный класс.

С учетом сказанного метод `Draw()` в классе `Circle` теперь должен быть обязательно переопределен. В противном случае `Circle` также должен быть абстрактным классом и декорироваться ключевым словом `abstract` (что очевидно не подходит в настоящем примере). Вот изменения в коде:

```
// Если не реализовать здесь абстрактный метод Draw(), то Circle
// также должен считаться абстрактным и быть помечен как abstract!
class Circle : Shape
{
    public Circle() {}
    public Circle(string name) : base(name) {}
    public override void Draw()
    {
        Console.WriteLine("Drawing {0} the Circle", PetName);
    }
}
```

Итак, теперь можно предполагать, что любой класс, производный от `Shape`, действительно имеет уникальную версию метода `Draw()`. Для демонстрации полной картины полиморфизма рассмотрим следующий код:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Polymorphism *****\n");
    // Создать массив совместимых с Shape объектов.
    Shape[] myShapes = {new Hexagon(), new Circle(), new Hexagon("Mick"),
        new Circle("Beth"), new Hexagon("Linda")};

    // Пройти в цикле по всем элементам и взаимодействовать
    // с полиморфным интерфейсом.
    foreach (Shape s in myShapes)
    {
        s.Draw();
    }
    Console.ReadLine();
}
```

Ниже показан вывод из этого метода `Main()`:

```
***** Fun with Polymorphism *****

Drawing NoName the Hexagon
Drawing NoName the Circle
Drawing Mick the Hexagon
Drawing Beth the Circle
Drawing Linda the Hexagon
```

Данный метод `Main()` иллюстрирует полиморфизм в чистом виде. Хотя *напрямую* создавать экземпляры абстрактного базового класса (`Shape`) невозможно, с помощью абстрактной базовой переменной допускается хранить ссылки на объекты любого подкласса. Таким образом, созданный массив объектов `Shape` способен хранить объекты классов, производных от базового класса `Shape` (попытка добавления в массив объектов, несовместимых с `Shape`, приведет к ошибке на этапе компиляции).

С учетом того, что все элементы в массиве `myShapes` на самом деле являются производными от `Shape`, вы знаете, что все они поддерживают один и тот же "полиморфный интерфейс" (или, говоря проще, все они имеют метод `Draw()`). Во время итерации по массиву ссылок `Shape` исполняющая система самостоятельно определяет лежащий в основе тип элемента. В этот момент и вызывается корректная версия метода `Draw()`.

Такой прием также делает простым безопасное расширение текущей иерархии. Например, пусть вы унаследовали от абстрактного базового класса `Shape` дополнительные классы (`Triangle`, `Square` и т.д.). Благодаря полиморфному интерфейсу код внутри цикла `foreach` не потребует никаких изменений, т.к. компилятор обеспечивает помещение внутрь массива `myShapes` только совместимых с `Shape` типов.

Соккрытие членов

Язык C# предоставляет возможность, которая логически противоположна переопределению методов и называется *сокрытием*. Выражаясь формально, если производный класс определяет член, который идентичен члену, определенному в базовом классе, то производный класс *скрывает* версию члена из родительского класса. В реальном мире такая ситуация чаще всего возникает, когда вы создаете подкласс от класса, который разрабатывали не вы (или ваша команда); например, такой класс может входить в состав пакета программного обеспечения .NET, приобретенного у независимого поставщика.

В целях иллюстрации предположим, что вы получили от коллеги на доработку класс по имени `ThreeDCircle`, в котором определен метод `Draw()`, не принимающий аргументов:

```
class ThreeDCircle
{
    public void Draw()
    {
        Console.WriteLine("Drawing a 3D Circle");
    }
}
```

Вы полагаете, что `ThreeDCircle` "является" `Circle`, поэтому решаете унаследовать его от своего существующего типа `Circle`:

```
class ThreeDCircle : Circle
{
    public void Draw()
    {
        Console.WriteLine("Drawing a 3D Circle");
    }
}
```

После перекомпиляции вы обнаруживаете следующее предупреждение:

'ThreeDCircle.Draw()' hides inherited member 'Circle.Draw()'. To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.

`Shapes.ThreeDCircle.Draw()` скрывает унаследованный член `Shapes.Circle.Draw()`. Чтобы текущий член переопределял эту реализацию, добавьте ключевое слово `override`. В противном случае добавьте ключевое слово `new`.

Дело в том, что у вас есть производный класс (`ThreeDCircle`), который содержит метод, идентичный унаследованному методу. Решить проблему можно несколькими способами. Вы могли бы просто модифицировать версию метода `Draw()` из родительского класса, добавив ключевое слово `override` (как предлагает компилятор). При таком под-

ходе у типа `ThreeDCircle` появляется возможность расширять стандартное поведение родительского типа, как и требовалось. Однако если у вас нет доступа к файлу кода с определением базового класса (частый случай, когда приходится работать с библиотеками от независимых поставщиков), тогда нет и возможности изменить метод `Draw()`, превратив его в виртуальный член.

В качестве альтернативы вы можете добавить ключевое слово `new` к определению проблемного члена `Draw()` своего производного типа (`ThreeDCircle`). Поступая так, вы явно утверждаете, что реализация производного типа намеренно спроектирована для фактического игнорирования версии члена из родительского типа (в реальности это может оказаться полезным, если внешнее программное обеспечение .NET каким-то образом конфликтует с вашим программным обеспечением).

```
// Этот класс расширяет Circle и скрывает унаследованный метод Draw().
class ThreeDCircle : Circle
{
    // Скрыть любую реализацию Draw(), находящуюся выше в иерархии.
    public new void Draw()
    {
        Console.WriteLine("Drawing a 3D Circle");
    }
}
```

Вы можете также применить ключевое слово `new` к любому члену типа, который унаследован от базового класса (полю, константе, статическому члену или свойству). Продолжая пример, предположим, что в классе `ThreeDCircle` необходимо скрыть унаследованное свойство `PetName`:

```
class ThreeDCircle : Circle
{
    // Скрыть свойство PetName, определенное выше в иерархии.
    public new string PetName { get; set; }

    // Скрыть любую реализацию Draw(), находящуюся выше в иерархии.
    public new void Draw()
    {
        Console.WriteLine("Drawing a 3D Circle");
    }
}
```

Наконец, имейте в виду, что вы по-прежнему можете обратиться к реализации скрытого члена из базового класса, используя явное приведение, как описано в следующем разделе. Вот пример:

```
static void Main(string[] args)
{
    ...
    // Здесь вызывается метод Draw(), определенный в классе ThreeDCircle.
    ThreeDCircle o = new ThreeDCircle();
    o.Draw();

    // Здесь вызывается метод Draw(), определенный в родительском классе!
    ((Circle)o).Draw();
    Console.ReadLine();
}
```

Правила приведения для базовых и производных классов

Теперь, когда вы умеете строить семейства взаимосвязанных типов классов, нужно изучить правила, которым подчиняются *операции приведения классов*. Давайте возвратимся к иерархии классов сотрудников, созданной ранее в главе, и добавим несколько новых методов в класс Program (если вы прорабатываете примеры, тогда откройте проект Employees в Visual Studio). Как описано в последнем разделе настоящей главы, изначальным базовым классом в системе является System.Object. По указанной причине любой класс “является” Object и может трактоваться как таковой. Таким образом, внутри переменной типа object допускается хранить экземпляр любого типа:

```
static void CastingExamples()
{
    // Manager "является" System.Object, поэтому в переменной
    // типа object можно сохранять ссылку на Manager.
    object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
}
```

В проекте Employees классы Manager, SalesPerson и PTSalesPerson расширяют класс Employee, а потому допустимая ссылка на базовый класс может хранить любой из объектов указанных классов. Следовательно, приведенный далее код также законен:

```
static void CastingExamples()
{
    // Manager "является" System.Object, поэтому в переменной
    // типа object можно сохранять ссылку на Manager.
    object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
    // Manager также "является" Employee.
    Employee moonUnit = new Manager("MoonUnit Zappa", 2, 3001, 20000,
                                    "101-11-1321", 1);
    // PTSalesPerson "является" SalesPerson.
    SalesPerson jill = new PTSalesPerson("Jill", 834, 3002, 100000, "111-12-1119", 90);
}
```

Первое правило приведения между типами классов гласит, что когда два класса связаны отношением “является”, то всегда можно безопасно сохранить объект производного типа в ссылке базового класса. Формально это называется *неявным приведением*, поскольку оно “просто работает” в соответствии с законами наследования. В результате появляется возможность строить некоторые мощные программные конструкции. Например, предположим, что в текущем классе Program определен новый метод:

```
static void GivePromotion(Employee emp)
{
    // Повысить зарплату...
    // Предоставить место на парковке компании...
    Console.WriteLine("{0} was promoted!", emp.Name);
}
```

Из-за того, что данный метод принимает единственный параметр типа Employee, в сущности, ему можно передавать объект любого унаследованного от Employee класса, учитывая наличие отношения “является”:

```
static void CastingExamples()
{
    // Manager "является" System.Object, поэтому в переменной
    // типа object можно сохранять ссылку на Manager.
    object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
}
```

```
// Manager также "является" Employee.
Employee moonUnit = new Manager("MoonUnit Zappa", 2, 3001, 20000,
                                "101-11-1321", 1);

GivePromotion(moonUnit);

// PTSalesPerson "является" SalesPerson.
SalesPerson jill = new PTSalesPerson("Jill", 834, 3002, 100000,
                                      "111-12-1119", 90);

GivePromotion(jill);
}
```

Предыдущий код компилируется благодаря неявному приведению от типа базового класса (Employee) к производному классу. Но что, если вы хотите также вызвать метод GivePromotion() с объектом frank (хранящимся в общей ссылке System.Object)? Если вы передадите объект frank методу GivePromotion() напрямую, то получите ошибку на этапе компиляции:

```
object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
// Ошибка!
GivePromotion(frank);
```

Проблема в том, что вы пытаетесь передать переменную, которая объявлена как принадлежащая не к типу Employee, а к более общему типу System.Object. Учитывая, что в цепочке наследования он находится выше, чем Employee, компилятор не разрешит неявное приведение, стараясь сохранить ваш код насколько возможно безопасным в отношении типов.

Несмотря на то что сами вы можете выяснить, что ссылка object указывает в памяти на объект совместимого с Employee класса, компилятор сделать подобное не в состоянии, поскольку это не будет известно вплоть до времени выполнения. Чтобы удовлетворить компилятор, понадобится применить *явное приведение*, которое и является вторым правилом: в таких случаях вы можете явно приводить "вниз", используя операцию приведения C#.

Базовый шаблон, которому нужно следовать при выполнении явного приведения, выглядит так:

```
(класс_к_которому_нужно_привести) существующая_ссылка
```

Таким образом, чтобы передать переменную типа object методу GivePromotion(), потребуется написать следующий код:

```
// Правильно!
GivePromotion((Manager) frank);
```

Ключевое слово as

Имейте в виду, что явное приведение оценивается *во время выполнения*, а не на этапе компиляции. Ради иллюстрации предположим, что проект Employees содержит копию класса Hexagon, созданного ранее в главе. Для простоты мы могли бы добавить в текущий проект такой класс:

```
class Hexagon
{
    public void Draw() { Console.WriteLine("Drawing a hexagon!"); }
}
```

Хотя приведение объекта сотрудника к объекту фигуры абсолютно лишено смысла, код вроде показанного ниже скомпилируется без ошибок:


```
// Привести объект frank к типу Hexagon невозможно,
// но этот код нормально скомпилируется!
object frank = new Manager();
Hexagon hex = (Hexagon)frank;
```

Тем не менее, вы получите ошибку времени выполнения, или более формально — *исключение времени выполнения*. В главе 7 будут рассматриваться подробности структурированной обработки исключений, а пока полезно отметить, что при явном приведении можно перехватывать возможные ошибки с применением ключевых слов `try` и `catch`:

```
// Перехват возможной ошибки приведения.
object frank = new Manager();
Hexagon hex;
try
{
    hex = (Hexagon)frank;
}
catch (InvalidCastException ex)
{
    Console.WriteLine(ex.Message);
}
```

Очевидно, что показанный пример надуман; в такой ситуации вас никогда не будет беспокоить приведение между указанными типами. Однако предположим, что есть массив элементов `System.Object`, среди которых лишь малая толика содержит объекты, совместимые с `Employee`. В этом случае первым делом желательно определить, совместим ли элемент массива с типом `Employee`, и если да, то лишь тогда выполнить приведение.

Для быстрого определения совместимости одного типа с другим во время выполнения в C# предусмотрено ключевое слово `as`. С помощью ключевого слова `as` можно определить совместимость, проверив возвращаемое значение на предмет `null`. Взгляните на следующий код:

```
// Использование ключевого слова as для проверки совместимости.
object[] things = new object[4];
things[0] = new Hexagon();
things[1] = false;
things[2] = new Manager();
things[3] = "Last thing";

foreach (object item in things)
{
    Hexagon h = item as Hexagon;
    if (h == null)
        Console.WriteLine("Item is not a hexagon");
    else
    {
        h.Draw();
    }
}
```

Здесь производится проход в цикле по всем элементам в массиве объектов и проверка каждого из них на совместимость с классом `Hexagon`. Метод `Draw()` вызывается, если (и только если) обнаруживается объект, совместимый с `Hexagon`. В противном случае выводится сообщение о том, что элемент несовместим.

Ключевое слово **is** (обновление)

В дополнение к ключевому слову **as** язык C# предлагает ключевое слово **is**, предназначенное для определения совместимости типов двух элементов. Тем не менее, в отличие от ключевого слова **as**, если типы не совместимы, тогда ключевое слово **is** возвращает **false**, а не ссылку **null**. В текущий момент метод `GivePromotion()` спроектирован для приема любого возможного типа, производного от `Employee`. Взгляните на следующую его модификацию, в которой теперь осуществляется проверка, какой конкретно "тип сотрудника" был передан:

```
static void GivePromotion(Employee emp)
{
    Console.WriteLine("{0} was promoted!", emp.Name);
    if (emp is SalesPerson)
    {
        Console.WriteLine("{0} made {1} sale(s)!", emp.Name,
            ((SalesPerson)emp).SalesNumber);
        Console.WriteLine();
    }
    if (emp is Manager)
    {
        Console.WriteLine("{0} had {1} stock options...", emp.Name,
            ((Manager)emp).StockOptions);
        Console.WriteLine();
    }
}
```

Здесь во время выполнения производится проверка с целью выяснения, на что именно в памяти указывает входная ссылка типа базового класса. После определения, принят ли объект типа `SalesPerson` или `Manager`, можно применить явное приведение, чтобы получить доступ к специализированному члену данного типа. Также обратите внимание, что помещать операции приведения внутрь конструкции `try/catch` не обязательно, т.к. внутри раздела `if`, выполнившего проверку условия, уже известно, что приведение безопасно.

Нововведением версии C# 7 является то, что с помощью ключевого слова **is** переменной можно также присваивать объект преобразованного типа, если приведение работает. Это позволяет сделать предыдущий метод более ясным, устраняя проблему "двойного приведения". В предшествующем примере первое приведение выполняется, когда производится проверка совпадения типов, и если она проходит успешно, то переменную приходится приводить снова. Взгляните на следующее обновление предыдущего метода:

```
static void GivePromotion(Employee emp)
{
    Console.WriteLine("{0} was promoted!", emp.Name);
    // Если SalesPerson, тогда присвоить переменной s.
    if (emp is SalesPerson s)
    {
        Console.WriteLine("{0} made {1} sale(s)!", emp.Name, s.SalesNumber);
        Console.WriteLine();
    }
    // Если Manager, тогда присвоить переменной m.
    if (emp is Manager m)
    {
        Console.WriteLine("{0} had {1} stock options...", emp.Name, m.StockOptions);
        Console.WriteLine();
    }
}
```

Использование отбрасывания вместе с ключевым словом *is* (нововведение)

Ключевое слово *is* допускается также применять в сочетании с новым заполнителем для отбрасывания переменных. Вот как можно обеспечить перехват объектов всех типов в операторе *if* или *switch*:

```
if (obj is var _)
{
    // Делать что-то.
}
```

Такое условие в операторе *if* будет давать совпадение с чем угодно, а потому следует уделять внимание порядку, в котором используется блок сравнения с отбрасыванием.

Еще раз о сопоставлении с образцом (нововведение)

В главе 3 была представлена возможность сопоставления с образцом C# 7. Теперь, когда вы обрели прочное понимание приведения, наступило время для более удачного примера. Предыдущий пример можно модернизировать для применения оператора *switch*, сопоставляющего с образцом:

```
static void GivePromotion(Employee emp)
{
    Console.WriteLine("{0} was promoted!", emp.Name);
    switch (emp)
    {
        case SalesPerson s:
            Console.WriteLine("{0} made {1} sale(s)!", emp.Name, s.SalesNumber);
            break;
        case Manager m:
            Console.WriteLine("{0} had {1} stock options...", emp.Name, m.StockOptions);
            break;
    }
    Console.WriteLine();
}
```

Когда к оператору *case* добавляется конструкция *when*, для использования доступно полное определение объекта как он приводится. Например, свойство *SalesNumber* существует только в классе *SalesPerson*, но не в классе *Employee*. Если приведение в первом операторе *case* проходит успешно, то переменная *s* будет содержать экземпляр класса *SalesPerson*, так что оператор *case* можно было бы переписать следующим образом:

```
case SalesPerson s when s.SalesNumber > 5:
```

Такие новые добавления к *is* и *switch* обеспечивают хорошие улучшения, которые помогают сократить объем кода, выполняющего сопоставление, как демонстрировалось в предшествующих примерах.

Использование отбрасывания вместе с операторами *switch* (нововведение)

Отбрасывание также может применяться в операторах *switch*:

```
switch (emp)
{
    case SalesPerson s when s.SalesNumber > 5:
        Console.WriteLine("{0} made {1} sale(s)!", emp.Name, s.SalesNumber);
        break;
```

```

case Manager m:
    Console.WriteLine("{0} had {1} stock options...", emp.Name,
                      m.StockOptions);
    break;
case Intern _:
    // Игнорировать практикантов.
    break;
case null:
    // Предпринять какое-то действие в случае null.
    break;
}

```

Главный родительский класс `System.Object`

В завершение главы мы займемся исследованием главного родительского класса внутри платформы .NET — `Object`. При чтении предыдущих разделов вы могли заметить, что базовые классы во всех иерархиях (`Car`, `Shape`, `Employee`) никогда явно не указывали свои родительские классы:

```

// Какой класс является родительским для Car?
class Car
{...}

```

В мире .NET каждый тип в конечном итоге является производным от базового класса по имени `System.Object`, который в языке C# может быть представлен с помощью ключевого слова `object`. Класс `Object` определяет набор общих членов для каждого типа внутри платформы. По сути, когда вы строите класс, в котором явно не указан родительский класс, компилятор автоматически делает его производным от `Object`. Если вы хотите прояснить свои намерения, то можете определять классы, производные от `Object`, следующим образом (однако вы не обязаны поступать так):

```

// Явное наследование класса от System.Object.
class Car : object
{...}

```

Подобно любому классу в `System.Object` определен набор членов. В показанном ниже формальном определении C# обратите внимание, что некоторые члены объявлены как `virtual`, указывая на возможность их переопределения в подклассах, тогда как другие помечены ключевым словом `static` (и потому вызываются на уровне класса):

```

public class Object
{
    // Виртуальные члены.
    public virtual bool Equals(object obj);
    protected virtual void Finalize();
    public virtual int GetHashCode();
    public virtual string ToString();
    // Невиртуальные члены уровня экземпляра.
    public Type GetType();
    protected object MemberwiseClone();
    // Статические члены.
    public static bool Equals(object objA, object objB);
    public static bool ReferenceEquals(object objA, object objB);
}

```

В табл. 6.1 приведен обзор функциональности, предоставляемой некоторыми часто используемыми методами `System.Object`.

Таблица 6.1. Основные методы System.Object

Метод экземпляра	Описание
Equals()	<p>По умолчанию этот метод возвращает true, только если сравниваемые элементы ссылаются на один и тот же объект в памяти. Таким образом, Equals() применяется для сравнения объектных ссылок, а не состояния объектов. Обычно данный метод переопределяется, чтобы возвращать true, когда сравниваемые объекты имеют одинаковые значения внутреннего состояния (семантика, основанная на значениях).</p> <p>Имейте в виду, что если вы переопределяете Equals(), тогда должны также переопределить GetHashCode(), т.к. данные методы используются внутренне типами Hashtable для извлечения подобъектов из контейнера. Также вспомните из главы 4, что в классе ValueType этот метод переопределен для всех структур, чтобы выполнять сравнение на основе значений</p>
Finalize()	Пока можно считать, что этот метод (когда он переопределен) вызывается для освобождения любых выделенных ресурсов перед уничтожением объекта. Сборка мусора в среде CLR подробно рассматривается в главе 9
GetHashCode()	Этот метод возвращает значение int, которое идентифицирует конкретный объект
ToString()	Этот метод возвращает строковое представление объекта в формате <пространство имен>.<имя_типа> (называемое <i>полностью заданным именем</i>). Он будет часто переопределяться в подклассе, чтобы вместо полностью заданного имени возвращать строку, которая состоит из пар "имя-значение", представляющих внутреннее состояние объекта
GetType()	Этот метод возвращает объект Type, полностью описывающий объект, на который в текущий момент производится ссылка. Другими словами, он является методом идентификации типов во время выполнения (Runtime Type Identification — RTTI), доступным всем объектам (подробно обсуждается в главе 15)
MemberwiseClone()	Этот метод возвращает почленную копию текущего объекта и часто применяется для клонирования объектов (см. главу 8)

Чтобы проиллюстрировать стандартное поведение, обеспечиваемое базовым классом Object, мы создадим новый проект консольного приложения C# по имени ObjectOverrides. Добавим в проект новый тип класса C#, содержащий следующее пустое определение типа Person:

```
// Не забывайте, что класс Person расширяет Object.
class Person {
```

Теперь обновим метод Main() для взаимодействия с унаследованными членами System.Object:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with System.Object *****\n");
        Person p1 = new Person();

        // Использовать унаследованные члены System.Object.
        Console.WriteLine("ToString: {0}", p1.ToString());
        Console.WriteLine("Hash code: {0}", p1.GetHashCode());
        Console.WriteLine("Type: {0}", p1.GetType());
    }
}
```

```
// Создать другие ссылки на p1.
Person p2 = p1;
object o = p2;

// Указывают ли ссылки на один и тот же объект в памяти?
if (o.Equals(p1) && p2.Equals(o))
{
    Console.WriteLine("Same instance!");
}
Console.ReadLine();
}
}
```

Вот вывод, получаемый из этого метода Main():

```
***** Fun with System.Object *****
ToString: ObjectOverrides.Person
Hash code: 46104728
Type: ObjectOverrides.Person
Same instance!
```

Обратите внимание на то, что стандартная реализация ToString() возвращает полностью заданное имя текущего типа (ObjectOverrides.Person). Как будет показано в главе 14, где исследуется построение специальных пространств имен, каждый проект C# определяет “корневое пространство имен”, название которого совпадает с именем проекта. Здесь мы создали проект по имени ObjectOverrides, поэтому тип Person и класс Program помещены внутрь пространства имен ObjectOverrides.

Стандартное поведение метода Equals() заключается в проверке, указывают ли две переменные на один и тот же объект в памяти. В коде мы создаем новую переменную Person по имени p1. В этот момент новый объект Person помещается в управляемую кучу. Переменная p2 также относится к типу Person. Тем не менее, вместо создания нового экземпляра переменной p2 присваивается ссылка p1. Таким образом, переменные p1 и p2 указывают на один и тот же объект в памяти, как и переменная o (типа object). Учитывая, что p1, p2 и o указывают на одно и то же местоположение в памяти, проверка эквивалентности дает положительный результат.

Хотя готовое поведение System.Object в ряде случаев может удовлетворять всем потребностям, довольно часто в специальных типах часть этих унаследованных методов переопределяется. В целях иллюстрации модифицируем класс Person, добавив свойства, которые представляют имя, фамилию и возраст лица; все они могут быть установлены с помощью специального конструктора:

```
// Не забывайте, что класс Person расширяет Object.
class Person
{
    public string FirstName { get; set; } = "";
    public string LastName { get; set; } = "";
    public int Age { get; set; }

    public Person(string fName, string lName, int personAge)
    {
        FirstName = fName;
        LastName = lName;
        Age = personAge;
    }
    public Person() {}
}
```

Переопределение метода `System.Object.ToString()`

Многие создаваемые классы (и структуры) могут извлечь преимущества от переопределения метода `ToString()` для возвращения строки с текстовым представлением текущего состояния экземпляра типа. Помимо прочего, это полезно при отладке. То, как вы решите конструировать результирующую строку — дело личных предпочтений; однако рекомендуемый подход предусматривает отделение пар "имя-значение" друг от друга двоеточиями и помещение всей строки в квадратные скобки (такому принципу следуют многие типы из библиотек базовых классов .NET). Взгляните на следующую переопределенную версию `ToString()` для класса `Person`:

```
public override string ToString() =>
    $"[First Name: {FirstName}; Last Name: {LastName};
    Age: {Age}]";
```

Приведенная реализация метода `ToString()` довольно прямолинейна, потому что класс `Person` содержит всего три порции данных состояния. Тем не менее, всегда помните о том, что правильное переопределение `ToString()` должно также учитывать любые данные, определенные выше в цепочке наследования.

При переопределении метода `ToString()` для класса, расширяющего специальный базовый класс, первым делом необходимо получить возвращаемое значение `ToString()` из родительского класса, используя ключевое слово `base`. После получения строковых данных родительского класса их можно дополнить специальной информацией производного класса.

Переопределение метода `System.Object.Equals()`

Давайте также переопределим поведение метода `Object.Equals()`, чтобы работать с семантикой на основе значений. Вспомните, что по умолчанию `Equals()` возвращает `true`, только если два сравниваемых объекта ссылаются на один и тот же экземпляр объекта в памяти. Для класса `Person` может оказаться полезной такая реализация `Equals()`, которая возвращает `true`, если две сравниваемые переменные содержат те же самые значения состояния (например, фамилию, имя и возраст).

Прежде всего, обратите внимание, что входной аргумент метода `Equals()` имеет общий тип `System.Object`. В связи с этим первым делом необходимо удостовериться в том, что вызывающий код действительно передал экземпляр типа `Person`, и для дополнительной подстраховки проверить, что входной параметр не является ссылкой `null`.

После того, как вы установите, что вызывающий код передал выделенный экземпляр `Person`, один из подходов предусматривает реализацию метода `Equals()` для сравнения поле за полем данных входного объекта с данными текущего объекта:

```
public override bool Equals(object obj)
{
    if (obj is Person && obj != null)
    {
        Person temp;
        temp = (Person)obj;
        if (temp.FirstName == this.FirstName
            && temp.LastName == this.LastName
            && temp.Age == this.Age)
        {
            return true;
        }
    }
    else
    {

```

```

        return false;
    }
}
return false;
}

```

Здесь производится сравнение значений входного объекта с внутренними значениями текущего объекта (обратите внимание на применение ключевого слова `this`). Если имя, фамилия и возраст в двух объектах идентичны, то эти два объекта имеют одинаковые данные состояния и возвращается значение `true`. Любые другие результаты приводят к возвращению `false`.

Хотя такой подход действительно работает, вы определенно в состоянии представить, насколько трудоемкой была бы реализация специального метода `Equals()` для нетривиальных типов, которые могут содержать десятки полей данных. Распространенное сокращение предусматривает использование собственной реализации метода `ToString()`. Если класс располагает подходящей реализацией `ToString()`, в которой учитываются все поля данных вверх по цепочке наследования, тогда можно просто сравнивать строковые данные объектов (проверив на равенство `null`):

```

// Больше нет необходимости приводить obj к типу Person,
// т.к. у всех типов имеется метод ToString().
public override bool Equals(object obj) => obj?.ToString() == ToString();

```

Обратите внимание, что в этом случае нет необходимости проверять входной аргумент на принадлежность к корректному типу (`Person` в нашем примере), поскольку метод `ToString()` поддерживают все типы .NET. Еще лучше то, что больше не требуется выполнять проверку на предмет равенства свойство за свойством, т.к. теперь просто проверяются значения, возвращаемые методом `ToString()`.

Переопределение метода `System.Object.GetHashCode()`

В случае переопределения в классе метода `Equals()` вы также должны переопределить стандартную реализацию метода `GetHashCode()`. Выражаясь упрощенно, хеш-код — это числовое значение, которое представляет объект как специфическое состояние. Например, если вы создадите две переменные типа `string`, хранящие значение `Hello`, то они должны давать один и тот же хеш-код. Однако если одна из них хранит строку в нижнем регистре (`hello`), то должны получаться разные хеш-коды.

Для выдачи хеш-значения метод `System.Object.GetHashCode()` по умолчанию применяет адрес текущей ячейки памяти, где расположен объект. Тем не менее, если вы строите специальный тип, подлежащий хранению в экземпляре типа `Hashtable` (из пространства имен `System.Collections`), тогда всегда должны переопределять данный член, потому что для извлечения объекта тип `Hashtable` будет вызывать методы `Equals()` и `GetHashCode()`.

На заметку! Говоря точнее, класс `System.Collections.Hashtable` внутренне вызывает метод `GetHashCode()`, чтобы получить общее представление о местоположении объекта, а с помощью последующего (внутреннего) вызова метода `Equals()` определяет его точно.

Хотя мы не собираемся помещать объекты `Person` в `System.Collections.Hashtable`, ради полноты изложения давайте переопределим метод `GetHashCode()`. Существует много алгоритмов, которые можно применять для создания хеш-кода, как весьма изощренных, так и не очень. В большинстве ситуаций есть возможность генерировать значение хеш-кода, полагаясь на реализацию метода `GetHashCode()` из класса `System.String`.

Учитывая, что класс `String` уже имеет эффективный алгоритм хеширования, использующий для вычисления хеш-значения символьные данные объекта `String`, вы можете просто вызвать метод `GetHashCode()` с той частью полей данных, которая должна быть уникальной во всех экземплярах (вроде номера карточки социального страхования), если ее удастся идентифицировать. Таким образом, если определить в классе `Person` свойство `SSN`, то переопределить метод `GetHashCode()` можно было бы следующим образом:

```
// Предположим, что имеется свойство SSN.
class Person
{
    public string SSN {get; set;} = "";

    // Возвратить хеш-код на основе уникальных строковых данных.
    public override int GetHashCode()
    {
        return SSN.GetHashCode();
    }
}
```

Если выбрать часть уникальных строковых данных не удастся, но есть переопределенный метод `ToString()`, тогда можно вызвать `GetHashCode()` на собственном строковом представлении:

```
// Возвратить хеш-код на основе значения, возвращаемого методом ToString()
// для объекта Person.
public override int GetHashCode()
{
    return this.ToString().GetHashCode();
}
```

Тестирование модифицированного класса `Person`

Теперь, когда виртуальные члены класса `Object` переопределены, обновим `Main()` для тестирования внесенных изменений:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with System.Object *****\n");

    // ПРИМЕЧАНИЕ: эти объекты идентичны и предназначены
    // для тестирования методов Equals() и GetHashCode().
    Person p1 = new Person("Homer", "Simpson", 50);
    Person p2 = new Person("Homer", "Simpson", 50);

    // Получить строковые версии объектов.
    Console.WriteLine("p1.ToString() = {0}", p1.ToString());
    Console.WriteLine("p2.ToString() = {0}", p2.ToString());

    // Протестировать переопределенный метод Equals().
    Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));

    // Проверить хеш-коды.
    Console.WriteLine("Same hash codes?: {0}",
        p1.GetHashCode() == p2.GetHashCode());
    Console.WriteLine();

    // Изменить возраст p2 и протестировать снова.
    p2.Age = 45;
    Console.WriteLine("p1.ToString() = {0}", p1.ToString());
    Console.WriteLine("p2.ToString() = {0}", p2.ToString());
}
```

```

Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));
Console.WriteLine("Same hash codes?: {0}",
    p1.GetHashCode() == p2.GetHashCode());
Console.ReadLine();
}

```

Ниже показан вывод:

***** Fun with System.Object *****

```

p1.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p2.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p1 = p2?: True
Same hash codes?: True

p1.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p2.ToString() = [First Name: Homer; Last Name: Simpson; Age: 45]
p1 = p2?: False
Same hash codes?: False

```

Статические члены класса System.Object

В дополнение к только что рассмотренным членам уровня экземпляра класс `System.Object` определяет два (очень полезных) статических члена, которые также проверяют эквивалентность на основе значений или на основе ссылок. Взгляните на следующий код:

```

static void StaticMembersOfObject()
{
    // Статические члены System.Object.
    Person p3 = new Person("Sally", "Jones", 4);
    Person p4 = new Person("Sally", "Jones", 4);
    Console.WriteLine("P3 and P4 have same state: {0}", object.Equals(p3, p4));
    Console.WriteLine("P3 and P4 are pointing to same object: {0}",
        object.ReferenceEquals(p3, p4));
}

```

Здесь можно просто передать методу `ReferenceEquals()` два объекта (любого типа) и позволить классу `System.Object` выяснить их эквивалентность.

Исходный код. Проект `ObjectOverrides` доступен в подкаталоге `Chapter_6`.

Резюме

В настоящей главе объяснялась роль и детали наследования и полиморфизма. В ней были представлены многочисленные новые ключевые слова и лексемы для поддержки каждого приема. Например, вспомните, что с помощью двоеточия указывается родительский класс для создаваемого типа. Родительские типы способны определять любое количество виртуальных и/или абстрактных членов для установления полиморфного интерфейса. Производные типы переопределяют эти члены с применением ключевого слова `override`.

Вдобавок к построению множества иерархий классов в главе также исследовалось явное приведение между базовыми и производными типами. В завершение главы рассматривались особенности главного родительского класса в библиотеках базовых классов .NET — `System.Object`.

глава 7

Структурированная обработка исключений

В настоящей главе вы узнаете о том, как иметь дело с аномалиями, возникающими во время выполнения кода C#, с использованием *структурированной обработки исключений*. Будут описаны не только ключевые слова C#, предназначенные для этих целей (try, catch, throw, finally, when), но и разница между исключениями уровня приложения и уровня системы, а также роль базового класса System.Exception. Кроме того, будет показано, как создавать специальные исключения, и рассмотрены некоторые инструменты отладки в Visual Studio, связанные с исключениями.

Ода ошибкам, дефектам и исключениям

Что бы ни нашептывало наше (пороку завышенное) самолюбие, идеальных программистов не существует. Разработка программного обеспечения является сложным делом, и из-за такой сложности довольно часто даже самые лучшие программы поставляются с разнообразными *проблемами*. В одних случаях проблема возникает из-за “плохо написанного” кода (например, по причине выхода за границы массива), а в других — из-за ввода пользователем некорректных данных, которые не были учтены в кодовой базе приложения (скажем, когда в поле для телефонного номера вводится значение Chucky). Вне зависимости от причин проблемы в конечном итоге приложение не работает ожидаемым образом. Чтобы подготовить почву для предстоящего обсуждения структурированной обработки исключений, рассмотрим три распространенных термина, которые применяются для описания аномалий.

- **Дефекты.** Выражаясь просто, это ошибки, которые допустил программист. В качестве примера предположим, что вы программируете на неуправляемом C++. Если вы забудете освободить динамически выделенную память, что приводит к утечке памяти, тогда получите дефект.
- **Пользовательские ошибки.** С другой стороны, пользовательские ошибки обычно возникают из-за тех, кто запускает приложение, а не тех, кто его создает. Например, ввод конечным пользователем в текстовом поле неправильно сформированной строки с высокой вероятностью может привести к генерации ошибки, если в коде не была предусмотрена обработка некорректного ввода.
- **Исключения.** Исключениями обычно считаются аномалии во время выполнения, которые трудно (а то и невозможно) учесть на стадии программирования приложения. Примерами исключений могут быть попытка подключения к базе данных, которая больше не существует, открытие запароченного XML-файла или попытка установления связи с машиной, которая в текущий момент находится в автономном режиме. В каждом из упомянутых случаев программист (или конечный пользователь) обладает довольно низким контролем над такими “исключительными” обстоятельствами.

С учетом приведенных определений должно быть понятно, что структурированная обработка исключений в .NET — прием работы с *исключительными ситуациями* во время выполнения. Тем не менее, даже для дефектов и пользовательских ошибок, которые ускользнули от глаз программиста, среда CLR будет часто генерировать соответствующее исключение, идентифицирующее возникшую проблему. Скажем, в библиотеках базовых классов .NET определены многочисленные исключения, такие как `FormatException`, `IndexOutOfRangeException`, `FileNotFoundException`, `ArgumentOutOfRangeException` и т.д.

В рамках терминологии .NET исключение объясняется дефектами, некорректным пользовательским вводом и ошибками времени выполнения, даже если программисты могут трактовать каждую аномалию как отдельную проблему. Однако прежде чем погружаться в детали, формализуем роль структурированной обработки исключений и посмотрим, чем она отличается от традиционных приемов обработки ошибок.

На заметку! Чтобы сделать примеры кода максимально ясными, мы не будем перехватывать абсолютно все исключения, которые может выдавать заданный метод из библиотеки базовых классов. Разумеется, в своих проектах производственного уровня вы должны свободно использовать приемы, описанные в главе.

Роль обработки исключений .NET

До появления платформы .NET обработка ошибок в среде операционной системы Windows представляла собой запутанную смесь технологий. Многие программисты внедряли собственную логику обработки ошибок в контекст разрабатываемого приложения. Например, команда разработчиков могла определять набор числовых констант для представления известных условий возникновения ошибок и затем применять эти константы как возвращаемые значения методов. Взгляните на следующий фрагмент кода на языке C:

```
/* Типичный механизм перехвата ошибок в стиле C. */
#define E_FILENOTFOUND 1000

int UseFileSystem()
{
    // Предполагается, что в этой функции происходит нечто
    // такое, что приводит к возврату следующего значения.
    return E_FILENOTFOUND;
}

void main()
{
    int retVal = UseFileSystem();
    if(retVal == E_FILENOTFOUND)
        printf("Cannot find file..."); // Не удалось найти файл
}
```

Такой подход далек от идеала, учитывая тот факт, что константа `E_FILENOTFOUND` — всего лишь числовое значение, которое немного говорит о том, каким образом решить возникшую проблему. В идеале желательно, чтобы название ошибки, описательное сообщение и другая полезная информация об условиях возникновения ошибки были помещены в единственный четко определенный пакет (что как раз и происходит при структурированной обработке исключений). В дополнение к специальным приемам, к которым прибегают разработчики, внутри API-интерфейса Windows определены сотни кодов ошибок, которые поступают в виде определений `#define` и `HRESULT`, а также очень многих вариаций простых булевских значений (`bool`, `BOOL`, `VARIANT_BOOL` и т.д.).

Очевидной проблемой, присущей таким старым приемам, является полное отсутствие симметрии. Каждый подход более или менее подгоняется под заданную технологию, заданный язык и возможно даже заданный проект. Чтобы положить конец такому безумству, платформа .NET предложила стандартную методику для генерации и перехвата ошибок времени выполнения — структурированную обработку исключений. Достоинство этой методики в том, что разработчики теперь имеют унифицированный подход к обработке ошибок, который является общим для всех языков, ориентированных на .NET. Следовательно, способ обработки ошибок, используемый программистом на C#, синтаксически подобен способу, который применяет программист на VB или программист на C++, имеющий дело с C++/CLI.

Дополнительное преимущество связано с тем, что синтаксис, используемый для генерации и отлавливания исключений за пределами границ сборок и машины, идентичен. Скажем, если вы применяете язык C# при построении службы Windows Communication Foundation (WCF), то можете сгенерировать исключение SOAP для удаленного вызывающего кода, используя те же самые ключевые слова, которые позволяют генерировать исключения внутри методов одного приложения.

Еще одно преимущество исключений .NET состоит в том, что в отличие от загадочных числовых значений они представляют собой объекты, в которых содержится читабельное описание проблемы, а также детальный снимок стека вызовов на момент первоначального возникновения исключения. Более того, конечному пользователю можно предоставить справочную ссылку, которая указывает на URL-адрес с подробностями об ошибке, а также специальные данные, определенные программистом.

Строительные блоки обработки исключений в .NET

Программирование со структурированной обработкой исключений предусматривает применение четырех взаимосвязанных сущностей:

- тип класса, который представляет детали исключения;
- член, способный генерировать экземпляр класса исключения в вызывающем коде при соответствующих обстоятельствах;
- блок кода на вызывающей стороне, который обращается к члену, предрасположенному к возникновению исключения;
- блок кода на вызывающей стороне, который будет обрабатывать (или перехватывать) исключение, если оно произойдет.

Язык программирования C# предлагает пять ключевых слов (`try`, `catch`, `throw`, `finally` и `when`), которые позволяют генерировать и обрабатывать исключения. Объект, представляющий текущую проблему, относится к классу, который расширяет класс `System.Exception` (или производный от него класс). С учетом сказанного давайте исследуем роль данного базового класса, касающегося исключений.

Базовый класс `System.Exception`

Все исключения в конечном итоге происходят от базового класса `System.Exception`, который в свою очередь является производным от `System.Object`. Ниже показана основная часть этого класса (обратите внимание, что некоторые его члены являются виртуальными и, следовательно, могут быть переопределены в производных классах):

```
public class Exception : ISerializable, _Exception
{
    // Открытые конструкторы.
    public Exception(string message, Exception innerException);
    public Exception(string message);
```

```

public Exception();
...
// Методы.
public virtual Exception GetBaseException();
public virtual void GetObjectData(SerializationInfo info,
    StreamingContext context);
// Свойства.
public virtual IDictionary Data { get; }
public virtual string HelpLink { get; set; }
public Exception InnerException { get; }
public virtual string Message { get; }
public virtual string Source { get; set; }
public virtual string StackTrace { get; }
public MethodBase TargetSite { get; }
...
}

```

Как видите, многие свойства, определенные в классе `System.Exception`, по своей природе допускают только чтение. Причина в том, что стандартные значения для каждого из них обычно будут предоставляться производными типами. Например, стандартное сообщение типа `IndexOutOfRangeException` выглядит так: "Index was outside the bounds of the array" ("Индекс вышел за границы массива").

На заметку! Класс `Exception` реализует два интерфейса .NET. Хотя интерфейсы вы пока еще не изучали (см. главу 8), просто имейте в виду, что интерфейс `_Exception` позволяет исключению .NET обрабатываться управляемой кодовой базой (такой как приложение COM), в то время как интерфейс `ISerializable` дает возможность объекту исключения пересекать определенные границы (скажем, границы машины).

В табл. 7.1 описаны наиболее важные члены класса `System.Exception`.

Таблица 7.1. Основные члены типа `System.Exception`

Свойство <code>System.Exception</code>	Описание
<code>Data</code>	Это свойство только для чтения позволяет извлекать коллекцию пар "ключ-значение" (представленную объектом, реализующим <code>IDictionary</code>), которая предоставляет дополнительную определяемую программистом информацию об исключении. По умолчанию коллекция пуста
<code>HelpLink</code>	Это свойство позволяет получать или устанавливать URL для доступа к справочному файлу или веб-сайту с подробным описанием ошибки
<code>InnerException</code>	Это свойство только для чтения может использоваться для получения информации о предыдущих исключениях, которые послужили причиной возникновения текущего исключения. Запись предыдущих исключений осуществляется путем их передачи конструктору самого последнего сгенерированного исключения
<code>Message</code>	Это свойство только для чтения возвращает текстовое описание заданной ошибки. Само сообщение об ошибке устанавливается как параметр конструктора
<code>Source</code>	Это свойство позволяет получать либо устанавливать имя сборки или объекта, который привел к генерации исключения
<code>StackTrace</code>	Это свойство только для чтения содержит строку, идентифицирующую последовательность вызовов, которая привела к возникновению исключения. Как нетрудно догадаться, данное свойство очень полезно во время отладки или для сохранения информации об ошибке во внешнем журнале ошибок
<code>TargetSite</code>	Это свойство только для чтения возвращает объект <code>MethodBase</code> с описанием многочисленных деталей о методе, который привел к генерации исключения (вызов <code>ToString()</code> будет идентифицировать этот метод по имени)

Простейший пример

Чтобы продемонстрировать полезность структурированной обработки исключений, мы должны создать класс, который будет генерировать исключение в надлежащих (или, можно сказать, *исключительных*) обстоятельствах. Создадим новый проект консольного приложения C# по имени SimpleException и определим в нем два класса (Car (автомобиль) и Radio (радиоприемник)), связав их между собой отношением "имеет". В классе Radio определен единственный метод, который отвечает за включение и выключение радиоприемника:

```
class Radio
{
    public void TurnOn(bool on)
    {
        Console.WriteLine(on ? "Jamming..." : "Quiet time...");
    }
}
```

В дополнение к использованию класса Radio через включение/делегацию класс Car (его код показан ниже) определен так, что если пользователь превышает предопределенную максимальную скорость (заданную с помощью константного члена MaxSpeed), тогда двигатель выходит из строя, приводя объект Car в нерабочее состояние (отражается закрытой переменной-членом типа bool по имени carIsDead).

Кроме того, класс Car имеет несколько свойств для представления текущей скорости и указанного пользователем "дружественного названия" автомобиля, а также различные конструкторы для установки состояния нового объекта Car. Ниже приведено полное определение Car вместе с поясняющими комментариями.

```
class Car
{
    // Константа для представления максимальной скорости.
    public const int MaxSpeed = 100;

    // Свойства автомобиля.
    public int CurrentSpeed {get; set;} = 0;
    public string PetName {get; set;} = "";

    // Не вышел ли автомобиль из строя?
    private bool carIsDead;

    // В автомобиле имеется радиоприемник.
    private Radio theMusicBox = new Radio();

    // Конструкторы.
    public Car() {}
    public Car(string name, int speed)
    {
        CurrentSpeed = speed;
        PetName = name;
    }

    public void CrankTunes(bool state)
    {
        // Делегировать запрос внутреннему объекту.
        theMusicBox.TurnOn(state);
    }

    // Проверить, не перегрелся ли автомобиль.
    public void Accelerate(int delta)
```

```

{
    if (carIsDead)
        Console.WriteLine("{0} is out of order...", PetName);
    else
    {
        CurrentSpeed += delta;
        if (CurrentSpeed > MaxSpeed)
        {
            Console.WriteLine("{0} has overheated!", PetName);
            CurrentSpeed = 0;
            carIsDead = true;
        }
        else
            Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
    }
}
}

```

Теперь реализуем метод `Main()`, в котором объект `Car` будет превышать заранее заданную максимальную скорость (установленную в 100 внутри класса `Car`):

```

static void Main(string[] args)
{
    Console.WriteLine("***** Simple Exception Example *****");
    Console.WriteLine("=> Creating a car and stepping on it!");
    Car myCar = new Car("Zippy", 20);
    myCar.CrankTunes(true);

    for (int i = 0; i < 10; i++)
        myCar.Accelerate(10);
    Console.ReadLine();
}

```

Вывод будет выглядеть следующим образом:

```

***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
=> CurrentSpeed = 100
Zippy has overheated!
Zippy is out of order...

```

Генерация общего исключения (обновление)

Располагая функциональным классом `Car`, рассмотрим простейший способ генерации исключения. Текущая реализация метода `Accelerate()` просто отображает сообщение об ошибке, если вызывающий код пытается разогнать автомобиль до скорости, превышающей верхний предел.

Чтобы модернизировать метод `Accelerate()` для генерации исключения, когда пользователь пытается разогнать автомобиль до скорости, которая превышает установленный предел, потребуется создать и сконфигурировать новый экземпляр класса

`System.Exception`, установив значение доступного только для чтения свойства `Message` через конструктор класса. Для отправки объекта ошибки обратно вызывающему коду применяется ключевое слово `throw` языка C#. Ниже приведен обновленный код метода `Accelerate()`:

```
// На этот раз генерировать исключение, если пользователь
// превышает предел, указанный в MaxSpeed.
public void Accelerate(int delta)
{
    if (carIsDead)
        Console.WriteLine("{0} is out of order...", PetName);
    else
    {
        CurrentSpeed += delta;
        if (CurrentSpeed >= MaxSpeed)
        {
            carIsDead = true;
            CurrentSpeed = 0;

            // Использовать ключевое слово throw для генерации исключения.
            throw new Exception($"{PetName} has overheated!");
        }
        else
            Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
    }
}
```

Прежде чем выяснять, каким образом вызывающий код будет перехватывать данное исключение, необходимо отметить несколько интересных моментов. Для начала, если вы генерируете исключение, то всегда самостоятельно решаете, как вводится в действие ошибка и когда должно генерироваться исключение. Здесь мы предполагаем, что при попытке увеличить скорость объекта `Car` за пределы максимума должен быть сгенерирован объект `System.Exception` для уведомления о невозможности продолжить выполнение метода `Accelerate()` (в зависимости от создаваемого приложения такое предположение может быть как допустимым, так и нет).

В качестве альтернативы метод `Accelerate()` можно было бы реализовать так, чтобы он производил автоматическое восстановление, не генерируя перед этим исключение. По большому счету исключения должны генерироваться только при возникновении более критичного условия (например, отсутствие нужного файла, невозможность подключения к базе данных и т.п.). Принятие решения о том, что должно служить причиной генерации исключения, требует серьезного обдумывания и поиска веских оснований на стадии проектирования. Для преследуемых сейчас целей будем считать, что попытка увеличить скорость автомобиля выше максимально допустимой является вполне оправданной причиной для выдачи исключения.

В любом случае, если вы снова запустите приложение с показанной выше логикой в методе `Main()`, то исключение, в конце концов, будет сгенерировано. В следующем выводе видно, что результат отсутствия обработки этой ошибки нельзя назвать идеальным, учитывая получение многословного сообщения об ошибке, за которым следует прекращение работы программы:

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
```

```
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
```

```
Unhandled Exception: System.Exception: Zippy has overheated!
  at SimpleException.Car.Accelerate(Int32 delta) in C:\MyBooks\C# Book (7th ed)
\Code\Chapter_7\SimpleException\Car.cs:line 62
  at SimpleException.Program.Main(String[] args) in C:\MyBooks\C# Book (7th ed)
\Code\Chapter_7\SimpleException\Program.cs:line 20
Press any key to continue . . .
```

В версиях, предшествующих C# 7, конструкция `throw` была оператором, что означало возможность генерации исключения только там, где разрешены операторы. С выходом C# 7 конструкция `throw` стала доступной также как выражение и может быть указана везде, где разрешены выражения.

Перехват исключений

На заметку! Те, кто пришел в .NET из мира Java, должны помнить о том, что члены типа не прототипируются набором исключений, которые они могут генерировать (другими словами, платформа .NET не поддерживает проверяемые исключения). Лучше это или хуже, но вы не обязаны обрабатывать каждое исключение, генерируемое отдельно взятым членом.

Поскольку метод `Accelerate()` теперь генерирует исключение, вызывающий код должен быть готов обработать его, если оно возникнет. При вызове метода, который может сгенерировать исключение, должен использоваться блок `try/catch`. После перехвата объекта исключения можно обращаться к различным его членам и извлекать детальную информацию о проблеме.

Дальнейшие действия с такими данными в значительной степени зависят от вас. Вы можете зафиксировать их в файле отчета, записать в журнал событий Windows, отправить по электронной почте системному администратору или отобразить конечному пользователю сообщение о проблеме. Здесь мы просто выводим детали исключения в окно консоли:

```
// Обработка сгенерированного исключения.
static void Main(string[] args)
{
    Console.WriteLine("***** Simple Exception Example *****");
    Console.WriteLine("=> Creating a car and stepping on it!");
    Car myCar = new Car("Zippy", 20);
    myCar.CrankTunes(true);

    // Разогнаться до скорости, превышающей максимальный
    // предел автомобиля, с целью выдачи исключения.
    try
    {
        for(int i = 0; i < 10; i++)
            myCar.Accelerate(10);
    }
    catch(Exception e)
    {
        Console.WriteLine("\n*** Error! ***");           // ошибка
        Console.WriteLine("Method: {0}", e.TargetSite); // метод
        Console.WriteLine("Message: {0}", e.Message);   // сообщение
        Console.WriteLine("Source: {0}", e.Source);     // источник
    }
}
```

```
// Ошибка была обработана, продолжается выполнение следующего оператора.
Console.WriteLine("\n***** Out of exception logic *****");
Console.ReadLine();
}
```

По существу блок `try` представляет собой раздел операторов, которые в ходе выполнения могут генерировать исключение. Если исключение обнаруживается, тогда управление переходит к соответствующему блоку `catch`. С другой стороны, если код внутри блока `try` исключение не сгенерировал, то блок `catch` полностью пропускается, и выполнение проходит обычным образом. Ниже представлен вывод, полученный в результате тестового запуска данной программы:

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90

*** Error! ***
Method: Void Accelerate(Int32)
Message: Zippy has overheated!
Source: SimpleException

***** Out of exception logic *****
```

Как видите, после обработки исключения приложение может продолжать свое функционирование с оператора, находящегося после блока `catch`. В некоторых обстоятельствах исключение может оказаться достаточно критическим для того, чтобы служить основанием завершения работы приложения. Тем не менее, во многих случаях логика внутри обработчика исключений позволяет приложению спокойно продолжить выполнение (хотя, может быть, с несколько меньшим объемом функциональности, например, без возможности взаимодействия с удаленным источником данных).

Конфигурирование состояния исключения

В настоящий момент объект `System.Exception`, сконфигурированный внутри метода `Accelerate()`, просто устанавливает значение, доступное через свойство `Message` (посредством параметра конструктора). Как было показано ранее в табл. 7.1, класс `Exception` также предлагает несколько дополнительных членов (`TargetSite`, `StackTrace`, `HelpLink` и `Data`), которые полезны для дальнейшего уточнения природы возникшей проблемы. Чтобы усовершенствовать текущий пример, давайте по очереди рассмотрим возможности упомянутых членов.

Свойство `TargetSite`

Свойство `System.Exception.TargetSite` позволяет выяснить разнообразные детали о методе, который сгенерировал заданное исключение. Как демонстрировалось в предыдущем методе `Main()`, в результате вывода значения свойства `TargetSite` отобразится возвращаемое значение, имя и типы параметров метода, который сгенерировал исключение. Однако свойство `TargetSite` возвращает не простую строку, а строго типизированный объект `System.Reflection.MethodBase`. Данный тип можно применять

для сбора многочисленных деталей, касающихся проблемного метода, а также класса, в котором метод определен. В целях иллюстрации изменим предыдущую логику в блоке `catch` следующим образом:

```
static void Main(string[] args)
{
    ...
    // Свойство TargetSite в действительности возвращает объект MethodBase.
    catch(Exception e)
    {
        Console.WriteLine("\n*** Error! ***");
        Console.WriteLine("Member name: {0}", e.TargetSite); // имя члена
        Console.WriteLine("Class defining member: {0}",
            e.TargetSite.DeclaringType); // класс, определяющий член
        Console.WriteLine("Member type: {0}", e.TargetSite.MemberType); // тип члена
        Console.WriteLine("Message: {0}", e.Message); // сообщение
        Console.WriteLine("Source: {0}", e.Source); // источник
    }
    Console.WriteLine("\n***** Out of exception logic *****");
    Console.ReadLine();
}
```

На этот раз в коде используется свойство `MethodBase.DeclaringType` для выяснения полностью заданного имени класса, сгенерировавшего ошибку (в данном случае `SimpleException.Car`), а также свойство `MemberType` объекта `MethodBase` для идентификации типа члена (скажем, свойство или метод), в котором возникло исключение. Ниже показано, как будет выглядеть вывод в результате выполнения логики в блоке `catch`:

```
*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException
```

Свойство `StackTrace`

Свойство `System.Exception.StackTrace` позволяет идентифицировать последовательность вызовов, которая в результате привела к генерации исключения. Значение данного свойства никогда не устанавливается вручную — это делается автоматически во время создания объекта исключения. Чтобы подтвердить сказанное, модифицируем логику в блоке `catch`:

```
catch(Exception e)
{
    ...
    Console.WriteLine("Stack: {0}", e.StackTrace);
}
```

Снова запустив программу, в окне консоли можно обнаружить следующие данные трассировки стека (естественно, номера строк и пути к файлам у вас могут отличаться):

```
Stack: at SimpleException.Car.Accelerate(Int32 delta)
in c:\MyApps\SimpleException\car.cs:line 65 at SimpleException.Program.Main()
in c:\MyApps\SimpleException\Program.cs:line 21
```

Значение типа `string`, возвращаемое свойством `StackTrace`, отражает последовательность вызовов, которая привела к генерации данного исключения. Обратите внимание, что самый нижний номер строки в `string` указывает на место возникновения первого вызова в последовательности, а самый верхний — на место, где точно находится проблемный член. Очевидно, что такая информация очень полезна во время отладки или при ведении журнала для конкретного приложения, т.к. дает возможность отследить путь к источнику ошибки.

Свойство `HelpLink`

Хотя свойства `TargetSite` и `StackTrace` позволяют программистам выяснить, почему возникло конкретное исключение, информация подобного рода не особенно полезна для пользователей. Как уже было показано, с помощью свойства `System.Exception.Message` можно извлечь читабельную информацию и отобразить ее конечному пользователю. Вдобавок можно установить свойство `HelpLink` для указания на специальный URL или стандартный справочный файл Windows, где приводятся более подробные сведения о проблеме.

По умолчанию значением свойства `HelpLink` является пустая строка. Если вы хотите присвоить данному свойству какое-то более интересное значение, то должны делать это перед генерацией исключения `System.Exception`.

Изменим код метода `Car.Accelerate()`:

```
public void Accelerate(int delta)
{
    if (carIsDead)
        Console.WriteLine("{0} is out of order...", PetName);
    else
    {
        CurrentSpeed += delta;
        if (CurrentSpeed >= MaxSpeed)
        {
            carIsDead = true;
            CurrentSpeed = 0;

            // Мы хотим обращаться к свойству HelpLink, поэтому необходимо
            // создать локальную переменную перед генерацией объекта Exception.
            Exception ex =
                new Exception($"{PetName} has overheated!");
            ex.HelpLink = "http://www.CarsRUs.com";
            throw ex;
        }
        else
            Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed); // Вывод
                                                                            // текущей скорости
    }
}
```

Теперь можно обновить логику в блоке `catch` для вывода на консоль информации из свойства `HelpLink`:

```
catch (Exception e)
{
    ...
    Console.WriteLine("Help Link: {0}", e.HelpLink);
}
```

Свойство Data

Свойство `Data` класса `System.Exception` позволяет заполнить объект исключения подходящей вспомогательной информацией (такой как метка времени). Свойство `Data` возвращает объект, который реализует интерфейс по имени `IDictionary`, определенный в пространстве имен `System.Collections`. В главе 8 исследуется роль программирования на основе интерфейсов, а также рассматривается пространство имен `System.Collections`. В текущий момент важно понимать лишь то, что словарные коллекции позволяют создавать наборы значений, извлекаемых по ключу. Взгляните на очередное изменение метода `Car.Accelerate()`:

```
public void Accelerate(int delta)
{
    if (carIsDead)
        Console.WriteLine("{0} is out of order...", PetName);
    else
    {
        CurrentSpeed += delta;
        if (CurrentSpeed >= MaxSpeed)
        {
            carIsDead = true;
            CurrentSpeed = 0;

            // Мы хотим обращаться к свойству HelpLink, поэтому необходимо
            // создать локальную переменную перед генерацией объекта Exception.
            Exception ex = new Exception($"{PetName} has overheated!");
            ex.HelpLink = "http://www.CarsRUs.com";

            // Предоставить специальные данные, касающиеся ошибки.
            ex.Data.Add("TimeStamp", $"The car exploded at {DateTime.Now}");
            ex.Data.Add("Cause", "You have a lead foot.");
            throw ex;
        }
        else
            Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
    }
}
```

Для успешного прохода по парам "ключ-значение" нужно сначала указать директиву `using` для пространства имен `System.Collections`, т.к. в файле с классом, реализующим метод `Main()`, будет применяться тип `DictionaryEntry`:

```
using System.Collections;
```

Затем потребуется обновить логику в блоке `catch`, чтобы обеспечить проверку значения, возвращаемого из свойства `Data`, на равенство `null` (т.е. стандартному значению). После этого свойства `Key` и `Value` типа `DictionaryEntry` используются для вывода специальных данных на консоль:

```
catch (Exception e)
{
    ...
    Console.WriteLine("\n-> Custom Data:");
    foreach (DictionaryEntry de in e.Data)
        Console.WriteLine("-> {0}: {1}", de.Key, de.Value);
}
```

Вот как теперь выглядит финальный вывод программы:

```

***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90

*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException
Stack: at SimpleException.Car.Accelerate(Int32 delta)
      at SimpleException.Program.Main(String[] args)
Help Link: http://www.CarsRUs.com

-> Custom Data:
-> Timestamp: The car exploded at 9/12/2015 9:02:12 PM
-> Cause: You have a lead foot.

***** Out of exception logic *****

```

Свойство `Data` удобно в том смысле, что оно позволяет упаковывать специальную информацию об ошибке, не требуя построения нового типа класса для расширения базового класса `Exception`. Тем не менее, каким бы полезным ни было свойство `Data`, разработчики приложений .NET все равно обычно строят строго типизированные классы исключений, которые поддерживают специальные данные, применяя строго типизированные свойства.

Такой подход позволяет вызывающему коду перехватывать конкретный тип, производный от `Exception`, а не углубляться в коллекцию данных с целью получения дополнительных деталей. Чтобы понять, как это работает, необходимо разобраться с разницей между исключениями уровня системы и уровня приложения.

Исходный код. Проект `SimpleException` доступен в подкаталоге `Chapter_7`.

Исключения уровня системы (`System.SystemException`)

В библиотеках базовых классов .NET определено много классов, которые в конечном итоге являются производными от `System.Exception`. Например, в пространстве имен `System` определены основные объекты исключений, такие как `ArgumentOutOfRangeException`, `IndexOutOfRangeException`, `StackOverflowException` и т.п. В других пространствах имен есть исключения, которые отражают поведение этих пространств имен. Например, в `System.Drawing.Printing` определены исключения, связанные с выводом на печать, в `System.IO` — исключения, возникающие во время ввода-вывода, в `System.Data` — исключения, специфичные для баз данных, и т.д.

Исключения, которые генерируются самой платформой .NET, называются *системными исключениями*. Такие исключения в общем случае рассматриваются как неисправимые фатальные ошибки. Системные исключения унаследованы прямо от базо-

вого класса `System.SystemException`, который в свою очередь порожден от `System.Exception` (а тот — от класса `System.Object`):

```
public class SystemException : Exception
{
    // Разнообразные конструкторы.
}
```

Учитывая, что тип `System.SystemException` не добавляет никакой дополнительной функциональности кроме набора специальных конструкторов, вас может интересовать, по какой причине он вообще существует. Попросту говоря, когда тип исключения является производным от `System.SystemException`, то есть возможность выяснить, что исключение сгенерировала исполняющая среда .NET, а не кодовая база выполняющегося приложения. Это довольно легко проверить, используя ключевое слово `is`:

```
// Верно! NullReferenceException является SystemException.
NullReferenceException nullRefEx = new NullReferenceException();
Console.WriteLine("NullReferenceException is-a SystemException? : {0}",
    nullRefEx is SystemException);
```

Исключения уровня приложения (`System.ApplicationException`)

Поскольку все исключения .NET являются типами классов, вы можете создавать собственные исключения, специфичные для приложения. Однако из-за того, что базовый класс `System.SystemException` представляет исключения, генерируемые средой CLR, может сложиться впечатление, что вы должны порождать свои специальные исключения от типа `System.Exception`. Конечно, можно поступать и так, но взамен их лучше наследовать от класса `System.ApplicationException`:

```
public class ApplicationException : Exception
{
    // Разнообразные конструкторы.
}
```

Как и в `SystemException`, кроме набора конструкторов никаких дополнительных членов в классе `ApplicationException` не определено. С точки зрения функциональности единственная цель класса `System.ApplicationException` состоит в идентификации источника ошибки. При обработке исключения, производного от `System.ApplicationException`, можно предполагать, что исключение было сгенерировано кодовой базой выполняющегося приложения, а не библиотеками базовых классов .NET либо исполняющей средой .NET.

На заметку! На практике лишь немногие разработчики приложений .NET строят специальные исключения, которые расширяют класс `ApplicationException`. Взамен они чаще создают подкласс `System.Exception`, хотя формально допустимы оба подхода.

Построение специальных исключений, способ первый

Наряду с тем, что для сигнализации об ошибке во время выполнения можно всегда генерировать экземпляры `System.Exception` (как было показано в первом примере), иногда предпочтительнее создавать *строго типизированное исключение*, которое представляет уникальные детали, связанные с текущей проблемой. Например, предположим, что вы хотите построить специальное исключение (по имени `CarIsDeadException`)

для представления ошибки, которая возникает из-за увеличения скорости неисправного автомобиля. Первым делом создается новый класс, унаследованный от `System.Exception/System.ApplicationException` (по соглашению имена всех классов исключений заканчиваются суффиксом `Exception`; фактически это установившаяся практика в .NET).

На заметку! Согласно правилу все специальные классы исключений должны быть определены как открытые (вспомните, что стандартным модификатором доступа для невложенных типов является `internal`). Причина в том, что исключения часто передаются за границы сборок и потому должны быть доступны вызывающей кодовой базе.

Создадим новый проект консольного приложения по имени `CustomException` и скопируем в него предыдущие файлы `Car.cs` и `Radio.cs`, выбрав пункт меню `Project⇒Add Existing Item` (Проект⇒Добавить существующий элемент) и для ясности изменив название пространства имен, в котором определены типы `Car` и `Radio`, с `SimpleException` на `CustomException`. Затем добавим в него следующее определение класса:

```
// Это специальное исключение описывает детали условия выхода автомобиля из строя.
// (Не забывайте, что можно также просто расширить класс Exception.)
public class CarIsDeadException : ApplicationException
{
}
```

Как и с любым классом, вы можете создавать произвольное количество специальных членов, к которым можно обращаться внутри блока `catch` в вызывающем коде. Кроме того, вы можете также переопределять любые виртуальные члены, определенные в родительских классах. Например, вы могли бы реализовать `CarIsDeadException`, переопределив виртуальное свойство `Message`.

Вместо заполнения словаря данных (через свойство `Data`) при генерировании исключения конструктор позволяет указывать метку времени и причину возникновения ошибки. Наконец, метку времени и причину возникновения ошибки можно получить с применением строго типизированных свойств:

```
public class CarIsDeadException : ApplicationException
{
    private string messageDetails = String.Empty;
    public DateTime ErrorTimeStamp {get; set;}
    public string CauseOfError {get; set;}
    public CarIsDeadException(){}
    public CarIsDeadException(string message,
        string cause, DateTime time)
    {
        messageDetails = message;
        CauseOfError = cause;
        ErrorTimeStamp = time;
    }
    // Переопределить свойство Exception.Message.
    public override string Message => $"Car Error Message: {messageDetails}";
}
```

Здесь класс `CarIsDeadException` поддерживает закрытое поле (`messageDetails`), которое представляет данные, касающиеся текущего исключения; его можно устанавливать с использованием специального конструктора. Сгенерировать такое исключение в методе `Accelerate()` несложно. Понадобится просто создать, сконфигурировать и сгенерировать объект `CarIsDeadException`, а не `System.Exception` (обратите внимание, что в данном случае заполнять коллекцию данных вручную больше не требуется):

```
// Сгенерировать специальное исключение CarIsDeadException.
public void Accelerate(int delta)
{
    CarIsDeadException ex =
        new CarIsDeadException($"{PetName} has overheated!",
            "You have a lead foot", DateTime.Now);
    ex.HelpLink = "http://www.CarsRUs.com";
    throw ex;
    ...
}
```

Для перехвата такого входного исключения блок `catch` теперь можно модифицировать, чтобы в нем перехватывался конкретный тип `CarIsDeadException` (тем не менее, с учетом того, что `System.CarIsDeadException` “является” `System.Exception`, по-прежнему допустимо перехватывать `System.Exception`):

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Custom Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);

    try
    {
        // Отслеживать исключение.
        myCar.Accelerate(50);
    }
    catch (CarIsDeadException e)
    {
        Console.WriteLine(e.Message);
        Console.WriteLine(e.ErrorTimeStamp);
        Console.WriteLine(e.CauseOfError);
    }
    Console.ReadLine();
}
```

Итак, разобравшись в базовом процессе построения специального исключения, вас наверняка теперь интересует, когда может понадобиться поступать подобным образом. Обычно вам придется создавать специальные исключения только в случаях, если ошибка тесно связана с выдающим ее классом. В качестве примеров можно назвать специальный класс для работы с файлами, который генерирует набор ошибок, относящихся к файлам, класс `Car`, который генерирует ошибки, связанные с автомобилем, объект доступа к данным, который генерирует ошибки, имеющие отношение к определенной таблице базы данных, и т.д. Создавая специальные исключения, вы предоставляете вызывающему коду возможность обрабатывать многочисленные исключения по отдельности на описательной основе ошибка за ошибкой.

Построение специальных исключений, способ второй

Текущий класс `CarIsDeadException` переопределяет виртуальное свойство `System.Exception.Message`, чтобы сконфигурировать специальное сообщение об ошибке, и предоставляет два специальных свойства для учета дополнительных порций данных. Однако в реальности переопределять виртуальное свойство `Message` не обязательно, т.к. входное сообщение можно просто передать конструктору родительского класса:

```
public class CarIsDeadException : ApplicationException
{
    public DateTime ErrorTimeStamp { get; set; }
```

```

public string CauseOfError { get; set; }
public CarIsDeadException() { }
// Передача сообщения конструктору родительского класса.
public CarIsDeadException(string message, string cause, DateTime time)
    :base(message)
{
    CauseOfError = cause;
    ErrorTimeStamp = time;
}
}

```

Обратите внимание, что на этот раз не объявляется строковая переменная для представления сообщения и не переопределяется свойство `Message`. Взамен нужный параметр просто передается конструктору базового класса. При таком проектном решении специальный класс исключения является всего лишь уникально именованным классом, производным от `System.ApplicationException` (с дополнительными свойствами в случае необходимости), который не переопределяет какие-либо члены базового класса.

Не удивляйтесь, если большинство специальных классов исключений (а то и все) будет соответствовать такому простому шаблону. Во многих случаях роль специального исключения не обязательно связана с предоставлением дополнительной функциональности помимо той, что унаследована от базовых классов. На самом деле цель в том, чтобы предложить *строго именованный тип*, который четко идентифицирует природу ошибки, благодаря чему клиент может реализовать отличающуюся логику обработки для разных типов исключений.

Построение специальных исключений, способ третий

Если вы хотите создать по-настоящему интересный специальный класс исключения, тогда должны позаботиться о том, чтобы он следовал передовому опыту .NET. В частности, это предполагает, что класс обладает следующими характеристиками:

- является производным от класса `Exception/ApplicationException`;
- помечен атрибутом `[System.Serializable]`;
- определяет стандартный конструктор;
- определяет конструктор, который устанавливает значение унаследованного свойства `Message`;
- определяет конструктор для обработки “внутренних исключений”;
- определяет конструктор для поддержки сериализации типа.

При наличии только базовых знаний платформы .NET роль атрибутов и сериализации объектов может быть не ясна, что вполне нормально. Соответствующие темы будут подробно раскрыты далее в книге (в главе 15 объясняются атрибуты, а в главе 20 — службы сериализации). Тем не менее, чтобы завершить исследование специальных исключений, ниже приведена последняя версия класса `CarIsDeadException`, в которой реализованы все упомянутые выше специальные конструкторы (остальные специальные свойства и конструкторы выглядят так, как было показано в примере внутри раздела “Построение специальных исключений, способ второй”):

```

[System.Serializable]
public class CarIsDeadException : ApplicationException
{
    public CarIsDeadException() { }
    public CarIsDeadException(string message) : base( message ) { }
}

```

```

public CarIsDeadException(string message, System.Exception inner)
    : base( message, inner ) { }
protected CarIsDeadException(
    System.Runtime.Serialization.SerializationInfo info,
    System.Runtime.Serialization.StreamingContext context)
    : base( info, context ) { }
// Любые дополнительные специальные свойства, конструкторы и члены данных...
}

```

Поскольку создаваемые специальные исключения, следующие установившейся практике в .NET, на самом деле отличаются только своими именами, полезно знать, что среда Visual Studio предлагает фрагмент кода под названием Exception (рис. 7.1), который автоматически генерирует новый класс исключения, отвечающий рекомендациям .NET. (Вспомните из главы 2, что для активизации фрагмента кода необходимо ввести его имя, которым в данном случае является exception, и два раза нажать клавишу <Tab>.)

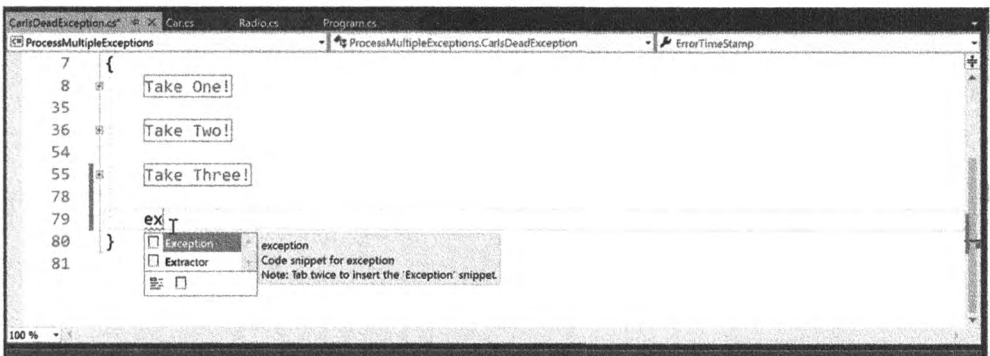


Рис. 7.1. Фрагмент кода Exception

Исходный код. Проект CustomException находится в подкаталоге Chapter_7.

Обработка множества исключений

В своей простейшей форме блок try сопровождается единственным блоком catch. Однако в реальности часто приходится сталкиваться с ситуациями, когда операторы внутри блока try могут генерировать многочисленные исключения. Создадим новый проект консольного приложения на C# по имени ProcessMultipleExceptions, добавим в него файлы Car.cs, Radio.cs и CarIsDeadException.cs из предыдущего проекта CustomException (посредством пункта меню Project → Add Existing Item) и надлежащим образом модифицируем название пространства имен.

Далее изменим метод Accelerate() класса Car так, чтобы он генерировал еще и предопределенное в библиотеках базовых классов исключение ArgumentOutOfRangeException, если передается недопустимый параметр (которым будет считаться любое значение меньше нуля). Обратите внимание, что конструктор этого класса исключения принимает имя проблемного аргумента в первом параметре типа string, за которым следует сообщение с описанием ошибки.

```

// Перед продолжением проверить аргумент на предмет допустимости.
public void Accelerate(int delta)
{

```

```

if(delta < 0)
    throw new
        ArgumentOutOfRangeException("delta", "Speed must be greater than zero!");
        // Значение скорости должно быть больше нуля!
...
}

```

Теперь логика в блоке `catch` может реагировать на каждый тип исключения специфическим образом:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    try
    {
        // Вызвать исключение выхода за пределы диапазона аргумента.
        myCar.Accelerate(-10);
    }
    catch (CarIsDeadException e)
    {
        Console.WriteLine(e.Message);
    }
    catch (ArgumentOutOfRangeException e)
    {
        Console.WriteLine(e.Message);
    }
    Console.ReadLine();
}

```

При написании множества блоков `catch` вы должны иметь в виду, что когда исключение сгенерировано, оно будет обрабатываться первым подходящим блоком `catch`. Чтобы проиллюстрировать, что означает “первый подходящий” блок `catch`, модифицируем предыдущий код, добавив еще один блок `catch`, который пытается обработать все остальные исключения кроме `CarIsDeadException` и `ArgumentOutOfRangeException` путем перехвата общего типа `System.Exception`:

```

// Этот код не скомпилируется!
static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    try
    {
        // Вызвать исключение выхода за пределы диапазона аргумента.
        myCar.Accelerate(-10);
    }
    catch(Exception e)
    {
        // Обработать все остальные исключения?
        Console.WriteLine(e.Message);
    }
    catch (CarIsDeadException e)
    {
        Console.WriteLine(e.Message);
    }
}

```

```

catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine(e.Message);
}
Console.ReadLine();
}

```

Такая логика обработки исключений приводит к ошибкам на этапе компиляции. Проблема в том, что первый блок `catch` может обрабатывать любые исключения, производные от `System.Exception` (с учетом отношения “является”), в том числе `CarIsDeadException` и `ArgumentOutOfRangeException`. Следовательно, два последних блока `catch` в принципе недостижимы!

Запомните эмпирическое правило: блоки `catch` должны быть структурированы так, чтобы первый `catch` перехватывал наиболее специфическое исключение (т.е. производный тип, расположенный ниже всех в цепочке наследования типов исключений), а последний `catch` — самое общее исключение (т.е. базовый класс имеющейся цепочки наследования: `System.Exception` в данном случае).

Таким образом, если вы хотите определить блок `catch`, который будет обрабатывать любые исключения помимо `CarIsDeadException` и `ArgumentOutOfRangeException`, то можно было бы написать следующий код:

```

// Этот код скомпилируется без проблем.
static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    try
    {
        // Вызвать исключение выхода за пределы диапазона аргумента.
        myCar.Accelerate(-10);
    }
    catch (CarIsDeadException e)
    {
        Console.WriteLine(e.Message);
    }
    catch (ArgumentOutOfRangeException e)
    {
        Console.WriteLine(e.Message);
    }
    // Этот блок будет перехватывать все остальные исключения
    // помимо CarIsDeadException и ArgumentOutOfRangeException.
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
    Console.ReadLine();
}

```

На заметку! Везде, где только возможно, отдавайте предпочтение перехвату специфичных классов исключений, а не общего класса `System.Exception`. Хотя может показаться, что это упрощает жизнь в краткосрочной перспективе (поскольку охватывает все исключения, которые пока не беспокоят), в долгосрочной перспективе могут возникать странные аварийные отказы во время выполнения, т.к. в коде не была предусмотрена непосредственная обработка более серьезной ошибки. Не забывайте, что финальный блок `catch`, который работает с `System.Exception`, на самом деле имеет тенденцию быть чрезвычайно общим.

Общие операторы catch

В языке C# также поддерживается “общий” контекст catch, который не получает явно объект исключения, сгенерированный заданным членом:

```
// Общий оператор catch.
static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    try
    {
        myCar.Accelerate(90);
    }
    catch
    {
        Console.WriteLine("Something bad happened...");
        // Произошло что-то плохое...
    }
    Console.ReadLine();
}
```

Очевидно, это не самый информативный способ обработки исключений, поскольку нет никакой возможности для получения содержательных данных о возникшей ошибке (таких как имя метода, стек вызовов или специальное сообщение). Тем не менее, в C# такая конструкция разрешена, потому что она может быть полезной, когда требуется обрабатывать все ошибки в обобщенной манере.

Повторная генерация исключений

Внутри логики блока try перехваченное исключение разрешено *повторно сгенерировать* для передачи вверх по стеку вызовов предшествующему вызывающему коду. Для этого просто используется ключевое слово throw в блоке catch. В итоге исключение передается вверх по цепочке вызовов, что может оказаться полезным, если блок catch способен обработать текущую ошибку только частично:

```
// Передача ответственности.
static void Main(string[] args)
{
    ...
    try
    {
        // Логика увеличения скорости автомобиля...
    }
    catch(CarIsDeadException e)
    {
        // Выполнить частичную обработку этой ошибки и передать ответственность.
        throw;
    }
    ...
}
```

Имейте в виду, что в данном примере кода конечным получателем исключения CarIsDeadException будет среда CLR, т.к. метод Main() генерирует его повторно. По указанной причине конечному пользователю будет отображаться системное диалоговое окно с информацией об ошибке. Обычно вы будете повторно генерировать частично обработанное исключение для передачи вызывающему коду, который имеет возможность обработать входное исключение более элегантным образом.

Также обратите внимание на неявную повторную генерацию объекта `CarIsDeadException` с помощью ключевого слова `throw` без аргументов. Дело в том, что здесь не создается новый объект исключения, а просто передается исходный объект исключения (со всей исходной информацией). Это позволяет сохранить контекст первоначального целевого объекта.

Внутренние исключения

Как нетрудно догадаться, вполне возможно, что исключение сгенерируется во время обработки другого исключения. Например, пусть вы обрабатываете исключение `CarIsDeadException` внутри отдельного блока `catch`, и в ходе этого процесса пытаетесь записать данные трассировки стека в файл `carErrors.txt` на диске `C:` (для получения доступа к типам, связанным с вводом-выводом, потребуется добавить директиву `using` с пространством имен `System.IO`):

```
catch (CarIsDeadException e)
{
    // Попытка открытия файла carErrors.txt, расположенного на диске C:.
    FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
    ...
}
```

Если указанный файл на диске `C:` отсутствует, тогда вызов метода `File.Open()` приведет к генерации исключения `FileNotFoundException`! Позже в книге, когда мы будем подробно рассматривать пространство имен `System.IO`, вы узнаете, как программно определить, существует ли файл на жестком диске, перед попыткой его открытия (тем самым вообще избегая исключения). Однако чтобы не отклоняться от темы исключений, предположим, что такое исключение было сгенерировано.

Когда во время обработки исключения вы сталкиваетесь с еще одним исключением, установившаяся практика предусматривает обязательное сохранение нового объекта исключения как "внутреннего исключения" в новом объекте того же типа, что и исходное исключение. Причина, по которой необходимо создавать новый объект обрабатываемого исключения, связана с тем, что единственным способом документирования внутреннего исключения является применение параметра конструктора. Взгляните на следующий код:

```
catch (CarIsDeadException e)
{
    try
    {
        FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
        ...
    }
    catch (Exception e2)
    {
        // Сгенерировать исключение, которое записывает новое
        // исключение, а также сообщение из первого исключения.
        throw new CarIsDeadException(e.Message, e2);
    }
}
```

Обратите внимание, что в данном случае конструктору `CarIsDeadException` во втором параметре передается объект `FileNotFoundException`. После настройки этого нового объекта он передается вверх по стеку вызовов следующему вызывающему коду, которым в рассматриваемой ситуации будет метод `Main()`.

Поскольку после `Main()` нет “следующего вызывающего кода”, который мог бы перехватить исключение, пользователю будет отображено системное диалоговое окно с сообщением об ошибке. Подобно повторной генерации исключения запись внутренних исключений обычно полезна, только если вызывающий код способен обработать исключение более элегантно. В таком случае внутри логики `catch` вызывающего кода можно использовать свойство `InnerException` для извлечения деталей внутреннего исключения.

Блок `finally`

В области действия `try/catch` можно также определять дополнительный блок `finally`. Целью блока `finally` является обеспечение того, что заданный набор операторов будет выполняться *всегда* независимо от того, возникло исключение (любого типа) или нет. Для иллюстрации предположим, что перед выходом из метода `Main()` радиоприемник в автомобиле должен всегда выключаться вне зависимости от обрабатываемого исключения:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    myCar.CrankTunes(true);
    try
    {
        // Логика, связанная с увеличением скорости автомобиля.
    }
    catch(CarIsDeadException e)
    {
        // Обработать объект CarIsDeadException.
    }
    catch(ArgumentOutOfRangeException e)
    {
        // Обработать объект ArgumentOutOfRangeException.
    }
    catch(Exception e)
    {
        // Обработать любой другой объект Exception.
    }
    finally
    {
        // Это код будет выполняться всегда независимо
        // от того, возникало исключение или нет.
        myCar.CrankTunes(false);
    }
    Console.ReadLine();
}
```

Если вы не определите блок `finally`, то в случае генерации исключения радиоприемник не выключится (что может быть или не быть проблемой). В более реалистичном сценарии, когда необходимо освободить объекты, закрыть файл либо отсоединиться от базы данных (или чего-то подобного), блок `finally` представляет собой подходящее место для выполнения надлежащей очистки.

Фильтры исключений

В версии C# 6 была введена новая конструкция, которая может быть помещена в блок `catch` посредством ключевого слова `when`. В случае ее добавления появляется возможность обеспечить выполнение операторов внутри блока `catch` только при удовлетворении некоторого условия в коде. Выражение условия должно давать в результате булевское значение (`true` или `false`) и может быть указано с применением простого выражения в самом определении `when` либо за счет вызова дополнительного метода в коде. Коротко говоря, такой подход позволяет добавлять “фильтры” к логике исключения.

Пусть в класс `CarIsDeadException` добавлено несколько специальных свойств:

```
public class CarIsDeadException : ApplicationException
{
    ...
    // Специальные члены для исключения.
    public DateTime ErrorTimeStamp { get; set; }
    public string CauseOfError { get; set; }
    public CarIsDeadException(string message,
        string cause, DateTime time)
        : base(message)
    {
        CauseOfError = cause;
        ErrorTimeStamp = time;
    }
}
```

Также предположим, что в методе `Accelerate()` для генерации исключения используется следующий новый конструктор:

```
CarIsDeadException ex =
    new CarIsDeadException($"{PetName} has overheated!",
        "You have a lead foot", DateTime.Now);
```

А теперь взгляните на показанную ниже модифицированную логику исключения. Здесь к обработчику `CarIsDeadException` добавлена конструкция `when`, которая гарантирует, что данный блок `catch` никогда не будет выполняться в пятницу (конечно, пример надуман, но кто захочет разбирать автомобиль на выходные?). Обратите внимание, что одиночное булевское выражение в конструкции `when` должно быть помещено в круглые скобки (кроме того, теперь внутри этого блока выводится новое сообщение, что будет происходить, только когда условие `when` дает `true`).

```
catch (CarIsDeadException e)
    when (e.ErrorTimeStamp.DayOfWeek != DayOfWeek.Friday)
{
    // Выводится, только если выражение в конструкции when вычисляется как true.
    Console.WriteLine("Catching car is dead!");
    Console.WriteLine(e.Message);
}
```

Несмотря на то что вы, скорее всего, просто предусмотрите блок `catch` для перехвата заданной ошибки при любых условиях, как видите, новое ключевое слово `when` позволяет добиться большей степени детализации при реагировании на ошибки времени выполнения.

Отладка необработанных исключений с использованием Visual Studio

Имейте в виду, что среда Visual Studio предлагает набор инструментов, которые помогают отлаживать необработанные специальные исключения. В целях иллюстрации предположим, что мы увеличили скорость объекта `Car` до значения, превышающего максимум, но на этот раз не позаботились о помещении вызова внутрь блока `try`:

```
Car myCar = new Car("Rusty", 90);  
myCar.Accelerate(2000);
```

Если вы запустите сеанс отладки в Visual Studio (выбрав пункт меню Debug⇒Start (Отладка⇒Начать)), то во время генерации необработанного исключения произойдет автоматический останов. Более того, откроется окно (рис. 7.2), отображающее значение свойства Message.

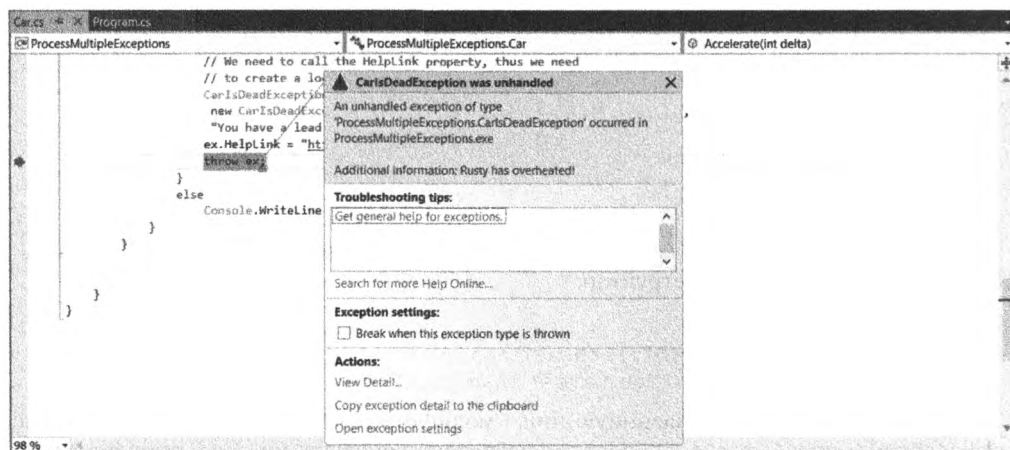


Рис. 7.2. Отладка необработанных специальных исключений в Visual Studio

На заметку! Если вы не обработали исключение, сгенерированное методом из библиотек базовых классов .NET, тогда отладчик Visual Studio остановит выполнение на операторе, который вызвал проблемный метод.

Щелкнув в этом окне на ссылке View Detail (Показать подробности), вы обнаружите подробную информацию о состоянии объекта (рис. 7.3).

Исходный код. Проект ProcessMultipleExceptions доступен в подкаталоге Chapter 7.

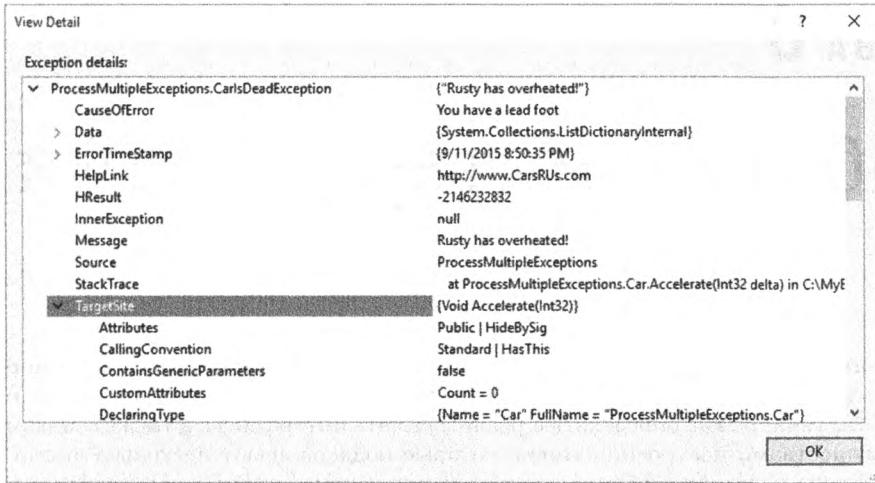


Рис. 7.3. Просмотр деталей исключения

Исходный код. Проект `ProcessMultipleExceptions` доступен в подкаталоге `Chapter_7`.

Резюме

В данной главе была раскрыта роль структурированной обработки исключений. Когда методу необходимо отправить объект ошибки вызывающему коду, он должен создать, сконфигурировать и сгенерировать специфичный объект производного от `System.Exception` типа посредством ключевого слова `throw` языка C#. Вызывающий код может обрабатывать любые входные исключения с применением ключевого слова `catch` и необязательного блока `finally`. В версии C# 6 появилась возможность создавать фильтры исключений с использованием дополнительного ключевого слова `when`, а в версии C# 7 расширен перечень мест, где можно генерировать исключения.

Когда вы строите собственные специальные исключения, то в конечном итоге создаете класс, производный от класса `System.ApplicationException`, который обозначает исключение, генерируемое текущим выполняющимся приложением. В противоположность этому объекты ошибок, производные от класса `System.SystemException`, представляют критические (и фатальные) ошибки, генерируемые средой CLR. Наконец, в главе были продемонстрированы разнообразные инструменты среды Visual Studio, которые можно применять для создания специальных исключений (согласно установившейся практике .NET), а также для отладки необработанных исключений.

ГЛАВА 8

Работа с интерфейсами

Материал настоящей главы опирается на ваши текущие знания объектно-ориентированной разработки и посвящен теме программирования на основе интерфейсов. Вы узнаете, как определять и реализовывать интерфейсы, а также ознакомитесь с преимуществами построения типов, которые поддерживают несколько линий поведения. В ходе изложения обсуждаются связанные темы, такие как получение ссылок на интерфейсы, явная реализация интерфейсов и построение иерархий интерфейсов. Будет исследовано несколько стандартных интерфейсов, определенных внутри библиотек базовых классов .NET. Как вы увидите, специальные классы и структуры могут реализовывать эти предопределенные интерфейсы для поддержки ряда полезных аспектов поведения, включая клонирование, перечисление и сортировку объектов.

Понятие интерфейсных типов

Для начала давайте ознакомимся с формальным определением *интерфейсного типа*. Интерфейс представляет собой всего лишь именованный набор *абстрактных членов*. Вспомните из главы 6, что абстрактные методы являются чистым протоколом, поскольку они не предоставляют свои стандартные реализации. Специфичные члены, определяемые интерфейсом, зависят от того, какое точно поведение он моделирует. Другими словами, интерфейс выражает *поведение*, которое заданный класс или структура может избрать для поддержки. Более того, далее в главе вы увидите, что класс или структура может реализовывать столько интерфейсов, сколько необходимо, и посредством этого поддерживать по существу множество линий поведения.

Как и можно было предположить, библиотеки базовых классов .NET поставляются с многочисленными предопределенными интерфейсными типами, которые реализуются разнообразными классами и структурами. Например, в главе 21 будет показано, что инфраструктура ADO.NET содержит множество поставщиков данных, которые позволяют взаимодействовать с определенной системой управления базами данных. Таким образом, в ADO.NET на выбор доступен обширный набор классов подключений (SqlConnection, OleDbConnection, OdbcConnection и т.д.). Вдобавок независимые поставщики баз данных (а также многие проекты с открытым кодом) предлагают библиотеки .NET для взаимодействия с большим числом других баз данных (MySQL, Oracle и т.д.), которые содержат объекты, реализующие упомянутые интерфейсы.

Невзирая на тот факт, что каждый класс подключения имеет уникальное имя, определен в отдельном пространстве имен и (в некоторых случаях) упакован в отдельную сборку, все они реализуют общий интерфейс под названием IDbConnection:

```
// Интерфейс IDbConnection определяет общий набор членов,  
// поддерживаемый всеми классами подключения.  
public interface IDbConnection : IDisposable  
{
```

```
// Методы.
IDbTransaction BeginTransaction();
IDbTransaction BeginTransaction(IsolationLevel il);
void ChangeDatabase(string databaseName);
void Close();
IDbCommand CreateCommand();
void Open();

// Свойства.
string ConnectionString { get; set; }
int ConnectionTimeout { get; }
string Database { get; }
ConnectionState State { get; }
}
```

На заметку! По соглашению имена всех интерфейсов .NET снабжаются префиксом в виде заглавной буквы I. При создании собственных специальных интерфейсов рекомендуется также следовать этому соглашению.

В настоящий момент детали того, что на самом деле делают члены интерфейса `IDbConnection`, не важны. Просто запомните, что в `IDbConnection` определен набор членов, которые являются общими для всех классов подключений ADO.NET. В итоге каждый класс подключения гарантированно поддерживает такие члены, как `Open()`, `Close()`, `CreateCommand()` и т.д. Кроме того, поскольку методы интерфейса `IDbConnection` всегда абстрактные, в каждом классе подключения они могут быть реализованы уникальным образом.

В оставшихся главах книги вы встретите десятки интерфейсов, поставляемых в библиотеках базовых классов .NET. Вы увидите, что эти интерфейсы могут быть реализованы в собственных специальных классах и структурах для определения типов, которые тесно интегрированы с платформой .NET. Вдобавок, как только вы оцените полезность интерфейсных типов, вы определенно найдете причины для построения собственных таких типов.

Сравнение интерфейсных типов и абстрактных базовых классов

Учитывая материалы главы 6, интерфейсный тип может выглядеть кое в чем похожим на абстрактный базовый класс. Вспомните, что когда класс помечен как абстрактный, он *может* определять любое количество абстрактных членов для предоставления полиморфного интерфейса всем производным типам. Однако даже если класс действительно определяет набор абстрактных членов, он также может определять любое количество конструкторов, полей данных, неабстрактных членов (с реализацией) и т.д. С другой стороны, интерфейсы содержат *только* определения членов.

Полиморфный интерфейс, устанавливаемый абстрактным родительским классом, обладает одним серьезным ограничением: члены, определенные абстрактным родительским классом, поддерживаются *только производными типами*. Тем не менее, в крупных программных системах часто разрабатываются многочисленные иерархии классов, не имеющие общего родителя кроме `System.Object`. Учитывая, что абстрактные члены в абстрактном базовом классе применимы только к производным типам, не существует какого-то способа конфигурирования типов в разных иерархиях для поддержки одного и того же полиморфного интерфейса. В качестве примера предположим, что определен следующий абстрактный класс:

```
public abstract class CloneableType
{
    // Поддерживать этот "полиморфный интерфейс" могут только производные типы.
    // Классы в других иерархиях не имеют доступа к данному абстрактному члену.
    public abstract object Clone();
}
```

При таком определении поддерживать метод `Clone()` способны только классы, расширяющие `CloneableType`. Если создается новый набор классов, которые не расширяют данный базовый класс, то извлечь пользу от такого полиморфного интерфейса не удастся. К тому же вы можете вспомнить, что язык C# не поддерживает множественное наследование для классов. По этой причине, если вы хотите создать класс `MiniVan`, который является и `Car`, и `CloneableType`, то поступить так, как показано ниже, не получится:

```
// Недопустимо! Множественное наследование для классов в C# не поддерживается.
public class MiniVan : Car, CloneableType
{
}
}
```

Как и можно было догадаться, здесь на помощь приходят интерфейсные типы. После того как интерфейс определен, он может быть реализован любым классом либо структурой, в любой иерархии и внутри любого пространства имен или сборки (написанной на любом языке программирования .NET). Как видите, интерфейсы являются *чрезвычайно* полиморфными. Рассмотрим стандартный интерфейс .NET под названием `ICloneable`, определенный в пространстве имен `System`. В нем определен единственный метод по имени `Clone()`:

```
public interface ICloneable
{
    object Clone();
}
```

Заглянув в документацию .NET Framework 4.7 SDK, вы обнаружите, что интерфейс `ICloneable` реализован очень многими на вид несвязанными типами (`System.Array`, `System.Data.SqlClient.SqlConnection`, `System.OperatingSystem`, `System.String` и т.д.). Хотя указанные типы не имеют общего родителя (кроме `System.Object`), их можно обрабатывать полиморфным образом посредством интерфейсного типа `ICloneable`.

Например, если есть метод по имени `CloneMe()`, принимающий параметр типа `ICloneable`, то такому методу можно передавать любой объект, который реализует указанный интерфейс. Рассмотрим следующий простой класс `Program`, определенный в проекте консольного приложения по имени `ICloneableExample`:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** A First Look at Interfaces *****\n");
        // Все эти классы поддерживают интерфейс ICloneable.
        string myStr = "Hello";
        OperatingSystem unixOS = new OperatingSystem(PlatformID.Unix, new Version());
        System.Data.SqlClient.SqlConnection sqlCnn =
            new System.Data.SqlClient.SqlConnection();
        // Следовательно, все они могут быть переданы методу,
        // принимающему параметр типа ICloneable.
        CloneMe(myStr);
        CloneMe(unixOS);
        CloneMe(sqlCnn);
        Console.ReadLine();
    }
}
```

```
private static void CloneMe(ICloneable c)
{
    // Клонировать то, что получено, и вывести его имя.
    object theClone = c.Clone();
    Console.WriteLine("Your clone is a: {0}", theClone.GetType().Name);
}
}
```

После запуска приложения в окне консоли выводится имя каждого класса, полученное с помощью метода `GetType()`, который унаследован от `System.Object`. Как объясняется в главе 15, этот метод и службы рефлексии .NET позволяют разбирать строение любого типа во время выполнения. Ниже показан вывод предыдущей программы:

```
***** A First Look at Interfaces *****

Your clone is a: String
Your clone is a: OperatingSystem
Your clone is a: SqlConnection
```

Исходный код. Проект `ICloneableExample` доступен в подкаталоге `Chapter_8`.

Еще одно ограничение абстрактных базовых классов связано с тем, что *каждый производный тип* должен предоставлять реализацию для всего набора абстрактных членов. Чтобы увидеть, в чем заключается проблема, вспомним иерархию фигур, которая была определена в главе 6. Предположим, что в базовом классе `Shape` определен новый абстрактный метод по имени `GetNumberOfPoints()`, который позволяет производным типам возвращать количество вершин, требуемых для визуализации фигуры:

```
abstract class Shape
{
    ...
    // Данный метод теперь должен поддерживать каждый производный класс!
    public abstract byte GetNumberOfPoints();
}
```

Очевидно, что единственным классом, который в принципе имеет вершины, будет `Hexagon`. Однако теперь из-за внесенного обновления *каждый* производный класс (`Circle`, `Hexagon` и `ThreeDCircle`) обязан предоставить конкретную реализацию метода `GetNumberOfPoints()`, даже если в этом нет никакого смысла. И снова интерфейсный тип предлагает решение. Если вы определите интерфейс, который представляет поведение "наличия вершин", то можно будет просто подключить его к классу `Hexagon`, оставив классы `Circle` и `ThreeDCircle` незатронутыми.

Определение специальных интерфейсов

Теперь, когда вы лучше понимаете общую роль интерфейсов, давайте рассмотрим пример определения и реализации специальных интерфейсов. Для начала создадим новый проект консольного приложения по имени `CustomInterface`. С помощью пункта меню `Project⇒Add Existing Item` (Проект⇒Добавить существующий элемент) вставим в него файл (или файлы) с определениями типов фигур (`Shapes.cs` в примерах кода), которые были созданы в главе 6. Затем переименуем пространство имен, в котором определяются типы, связанные с фигурами, на `CustomInterface` (просто чтобы избежать импортирования в новый проект определений пространства имен):

```
namespace CustomInterface
{
    // Здесь определяются типы фигур...
}
```


Теперь, выбрав пункт меню Project⇒Add Existing Item, вставим в проект новый интерфейс по имени IPointy (рис. 8.1).

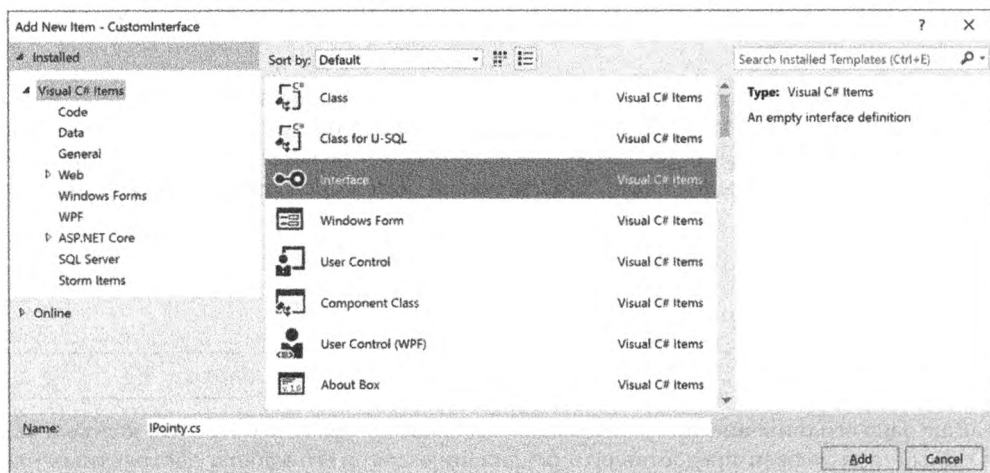


Рис. 8.1. Подобно классам интерфейсы могут определяться в файлах *.cs

На синтаксическом уровне интерфейс определяется с использованием ключевого слова `interface` языка C#. В отличие от классов для интерфейсов никогда не задается базовый класс (даже `System.Object`; тем не менее, как будет показано позже в главе, можно задавать базовые интерфейсы). Кроме того, для членов интерфейса никогда не указываются модификаторы доступа (т.к. все члены интерфейса являются неявно открытыми и абстрактными). Ниже приведен пример определения специального интерфейса в C#:

```
// Этот интерфейс определяет поведение "наличия вершин".
public interface IPointy
{
    // Неявно открытый и абстрактный.
    byte GetNumberOfPoints();
}
```

Запомните, что при определении членов интерфейса область реализации для них не определяется. Интерфейсы — это чистый протокол и потому реализация для них никогда не предоставляется (за нее отвечает поддерживающий класс или структура). Следовательно, показанная далее версия интерфейса `IPointy` в результате вызовет разнообразные ошибки на этапе компиляции:

```
// Внимание! В этом коде полно ошибок!
public interface IPointy
{
    // Ошибка! Интерфейсы не могут иметь поля данных!
    public int numbOfPoints;

    // Ошибка! Интерфейсы не могут иметь конструкторы!
    public IPointy() { numbOfPoints = 0; }

    // Ошибка! Интерфейсы не могут предоставлять реализацию своих членов!
    byte GetNumberOfPoints() { return numbOfPoints; }
}
```

В любом случае начальная версия интерфейса `IPointy` определяет единственный метод. Однако в интерфейсных типах .NET допускается также определять любое количество прототипов свойств. Например, модифицируем интерфейс `IPointy` так, чтобы в нем применялось свойство только для чтения, а не традиционный метод доступа:

```
// Поведение "наличия вершин" в виде свойства только для чтения.
public interface IPointy
{
    // Свойство, поддерживающее чтение и запись, в интерфейсе может выглядеть так:
    // retType PropName { get; set; }
    //
    // тогда как свойство только для записи - так:
    // retType PropName { set; }

    byte Points { get; }
}
```

На заметку! Интерфейсные типы также могут содержать определения событий (глава 10) и индексов (глава 11).

Сами по себе типы интерфейсов совершенно бесполезны, потому что они являются всего лишь именованными коллекциями абстрактных членов. Например, выделять память для типов интерфейсов, как делалось бы для класса или структуры, невозможно:

```
// Внимание! Выделять память для интерфейсных типов не допускается!
static void Main(string[] args)
{
    IPointy p = new IPointy(); // Ошибка на этапе компиляции!
}
```

Интерфейсы не привносят ничего особого до тех пор, пока не будут реализованы классом или структурой. Здесь `IPointy` представляет собой интерфейс, который выражает поведение "наличия вершин". Идея проста: одни классы в иерархии фигур (например, `Hexagon`) имеют вершины, в то время как другие (вроде `Circle`) — нет.

Реализация интерфейса

Когда решено расширить функциональность класса (или структуры) путем поддержки интерфейсов, к его определению добавляется список нужных интерфейсов, разделенных запятыми. Имейте в виду, что непосредственный базовый класс должен быть указан первым сразу после операции двоеточия. Если тип класса порождается напрямую от `System.Object`, то вы можете просто перечислить интерфейсы, поддерживаемые классом, т.к. компилятор C# будет считать, что типы расширяют `System.Object`, если не задано иначе. К слову, поскольку структуры всегда являются производными от класса `System.ValueType` (см. главу 4), достаточно указать список интерфейсов после определения структуры. Взгляните на приведенные ниже примеры:

```
// Этот класс является производными от System.Object
// и реализует единственный интерфейс.
public class Pencil : IPointy
{...}

// Этот класс также является производными от System.Object
// и реализует единственный интерфейс.
public class SwitchBlade : object, IPointy
{...}
```

```
// Этот класс является производными от специального базового
// класса и реализует единственный интерфейс.
public class Fork : Utensil, IPointy
{...}

// Эта структура неявно является производной
// от System.ValueType и реализует два интерфейса.
public struct PitchFork : ICloneable, IPointy
{...}
```

Важно понимать, что реализация интерфейса работает по принципу “все или ничего”. Поддерживающий тип не имеет возможности выборочно решать, какие члены он будет реализовывать. Учитывая, что интерфейс `IPointy` определяет единственное свойство только для чтения, накладные расходы невелики. Тем не менее, если вы реализуете интерфейс, который определяет десять членов (вроде показанного ранее `IDbConnection`), тогда тип отвечает за предоставление деталей для всех десяти абстрактных членов.

Вернемся к рассматриваемому примеру и добавим в проект новый тип класса по имени `Triangle`, который является `Shape` и поддерживает `IPointy`. Обратите внимание, что реализация доступного только для чтения свойства `Points` просто возвращает корректное количество вершин (3):

```
// Новый производный от Shape класс по имени Triangle.
class Triangle : Shape, IPointy
{
    public Triangle() { }
    public Triangle(string name) : base(name) { }
    public override void Draw()
    { Console.WriteLine("Drawing {0} the Triangle", PetName); }

    // Реализация IPointy.
    public byte Points
    {
        get { return 3; }
    }
}
```

Модифицируем существующий тип `Hexagon`, чтобы он также поддерживал интерфейс `IPointy`:

```
// Hexagon теперь реализует IPointy.
class Hexagon : Shape, IPointy
{
    public Hexagon() { }
    public Hexagon(string name) : base(name) { }
    public override void Draw()
    { Console.WriteLine("Drawing {0} the Hexagon", PetName); }

    // Реализация IPointy.
    public byte Points
    {
        get { return 6; }
    }
}
```

Чтобы подытожить сделанное к настоящему моменту, на рис. 8.2 приведена диаграмма классов Visual Studio, где все совместимые с `IPointy` классы представлены с помощью популярной системы обозначений в виде “леденца на палочке”. Еще раз обратите внимание, что `Circle` и `ThreeDCircle` не реализуют `IPointy`, поскольку такое поведение в данных классах не имеет смысла.

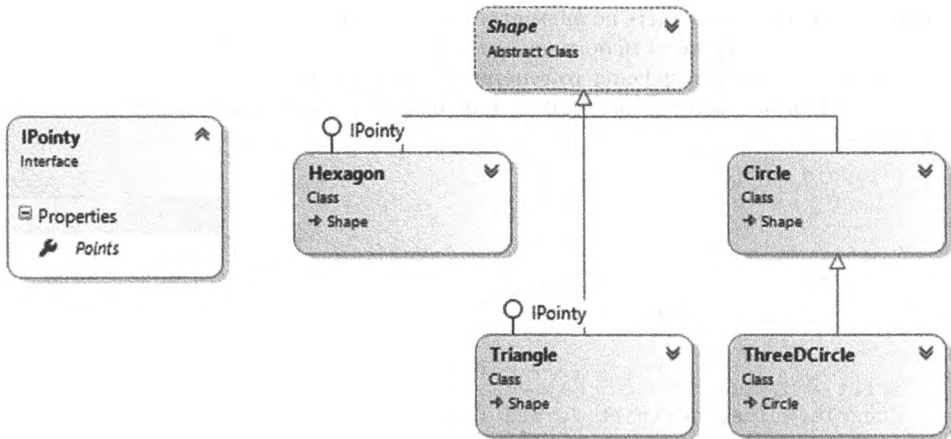


Рис. 8.2. Иерархия фигур, теперь с интерфейсами

На заметку! Чтобы скрыть или отобразить имена интерфейсов в визуальном конструкторе классов, щелкните правой кнопкой мыши на значке, представляющем интерфейс, и выберите в контекстном меню пункт Collapse (Свернуть) или Expand (Развернуть).

Обращение к членам интерфейса на уровне объектов

Теперь, имея несколько классов, которые поддерживают интерфейс `IPointy`, необходимо выяснить, каким образом взаимодействовать с новой функциональностью. Самый простой способ взаимодействия с функциональностью, предоставляемой заданным интерфейсом, заключается в обращении к его членам прямо на уровне объектов (при условии, что члены интерфейса не реализованы явно, о чем более подробно пойдет речь в разделе “Явная реализация интерфейсов” далее в главе). Например, рассмотрим следующий метод `Main()`:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interfaces *****\n");
    // Обратиться к свойству Points, определенному в интерфейсе IPointy.
    Hexagon hex = new Hexagon();
    Console.WriteLine("Points: {0}", hex.Points);
    Console.ReadLine();
}

```

Данный подход нормально работает в этом конкретном случае, т.к. здесь точно известно, что тип `Hexagon` реализует упомянутый интерфейс и, следовательно, имеет свойство `Points`. Однако в других случаях определить, какие интерфейсы поддерживаются заданным типом, может быть нереально. Для примера предположим, что есть массив, содержащий 50 объектов совместимых с `Shape` типов, и только некоторые из них поддерживают интерфейс `IPointy`. Очевидно, что если вы попытаетесь обратиться к свойству `Points` для типа, который не реализует `IPointy`, то возникнет ошибка. Как же динамически определить, поддерживает ли класс или структура подходящий интерфейс?

Один из способов выяснить во время выполнения, поддерживает ли тип конкретный интерфейс, предусматривает использование явного приведения. Если тип не поддерживает запрашиваемый интерфейс, то генерируется исключение `InvalidCastException`. В случае подобного рода необходимо использовать структурированную обработку исключений:

```
static void Main(string[] args)
{
    ...
    // Перехватить возможное исключение InvalidCastException.
    Circle c = new Circle("Lisa");
    IPointy itfPt = null;
    try
    {
        itfPt = (IPointy)c;
        Console.WriteLine(itfPt.Points);
    }
    catch (InvalidCastException e)
    {
        Console.WriteLine(e.Message);
    }
    Console.ReadLine();
}
```

Хотя можно было бы применить логику `try/catch` и надеяться на лучшее, в идеале хотелось бы определять, какие интерфейсы поддерживаются, до обращения к их членам. Давайте рассмотрим два способа, с помощью которых этого можно добиться.

Получение ссылок на интерфейсы: ключевое слово **as**

Для определения, поддерживает ли данный тип тот или иной интерфейс, можно использовать ключевое слово `as`, которое было представлено в главе 6. Если объект может трактоваться как указанный интерфейс, тогда возвращается ссылка на интересующий интерфейс, а если нет, то ссылка `null`. Таким образом, перед продолжением в коде необходимо реализовать проверку на предмет `null`:

```
static void Main(string[] args)
{
    ...
    // Можно ли hex2 трактовать как IPointy?
    Hexagon hex2 = new Hexagon("Peter");
    IPointy itfPt2 = hex2 as IPointy;
    if(itfPt2 != null)
        Console.WriteLine("Points: {0}", itfPt2.Points);
    else
        Console.WriteLine("OOPS! Not pointy...");
    Console.ReadLine();
}
```

Обратите внимание, что когда применяется ключевое слово `as`, отпадает необходимость в наличии логики `try/catch`, т.к. если ссылка не является `null`, то известно, что вызов происходит для действительной ссылки на интерфейс.

Получение ссылок на интерфейсы: ключевое слово **is** (обновление)

Проверить, реализован ли нужный интерфейс, можно также с помощью ключевого слова `is` (о котором впервые упоминалось в главе 6). Если интересующий объект не сов-

местим с указанным интерфейсом, тогда возвращается значение `false`. С другой стороны, если тип совместим с интерфейсом, то можно безопасно обращаться к его членам без использования логики `try/catch`.

В целях иллюстрации предположим, что имеется массив типов `Shape`, в котором определенные элементы реализуют интерфейс `IPointy`. Вот как с помощью ключевого слова `is` выяснить, какие элементы в массиве поддерживают данный интерфейс:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interfaces *****\n");
    // Создать массив элементов Shape.
    Shape[] myShapes = { new Hexagon(), new Circle(),
                        new Triangle("Joe"), new Circle("JoJo") };

    for(int i = 0; i < myShapes.Length; i++)
    {
        // Вспомните, что базовый класс Shape определяет абстрактный
        // член Draw(), поэтому все фигуры знают, как себя рисовать.
        myShapes[i].Draw();

        // У каких фигур есть вершины?
        if (myShapes[i] is IPointy ip)
            Console.WriteLine("-> Points: {0}", ip.Points); // есть вершины
        else
            Console.WriteLine("-> {0}\s not pointy!", myShapes[i].PetName);
                                // нет вершин

        Console.WriteLine();
    }
    Console.ReadLine();
}
```

Вывод выглядит следующим образом:

```
***** Fun with Interfaces *****

Drawing NoName the Hexagon
-> Points: 6

Drawing NoName the Circle
-> NoName's not pointy!

Drawing Joe the Triangle
-> Points: 3

Drawing JoJo the Circle
-> JoJo's not pointy'
```

На заметку! В примере задействовано новое средство C# 7 присваивания переменной (`ip`) экземпляра реализации интерфейса в сочетании с проверкой соответствия типу интерфейса. Все это — часть новых возможностей сопоставления с образцом, которые обсуждались в главах 3 и 6.

Использование интерфейсов в качестве параметров

Учитывая, что интерфейсы являются допустимыми типами .NET, можно строить методы, которые принимают интерфейсы в качестве параметров, как было проиллюстрировано на примере метода `CloneMe()` ранее в главе. Предположим, что в текущем примере определен еще один интерфейс по имени `IDraw3D`:

```
// Моделирует способность визуализации типа в трехмерном виде.
public interface IDraw3D
{
    void Draw3D();
}
```

Далее сконфигурируем две из трех наших фигур (Circle и Hexagon) с целью поддержки нового поведения:

```
// Circle поддерживает IDraw3D.
class ThreeDCircle : Circle, IDraw3D
{
    ...
    public void Draw3D()
    { Console.WriteLine("Drawing Circle in 3D!"); }
}
// Hexagon поддерживает IPointy и IDraw3D.
class Hexagon : Shape, IPointy, IDraw3D
{
    ...
    public void Draw3D()
    { Console.WriteLine("Drawing Hexagon in 3D!"); }
}
```

На рис. 8.3 показана обновленная диаграмма классов Visual Studio.

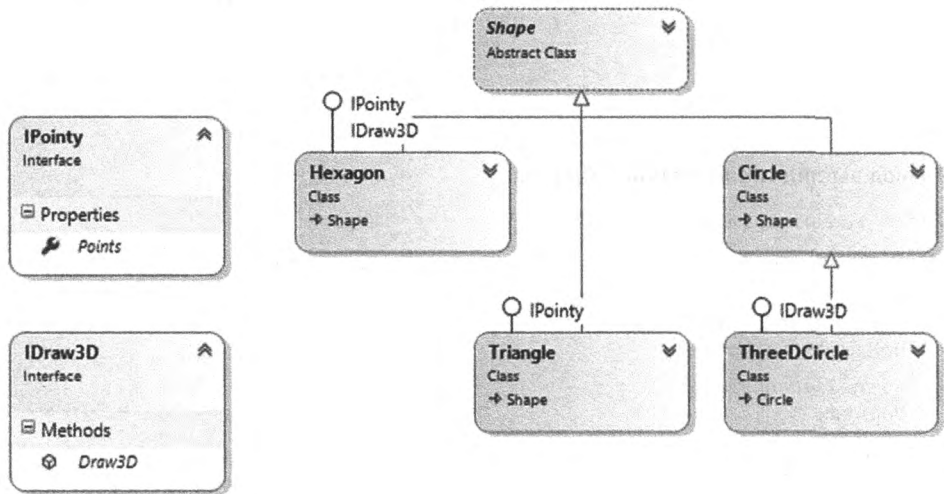


Рис. 8.3. Обновленная иерархия фигур

Если определить метод, принимающий интерфейс IDraw3D в качестве параметра, тогда ему можно будет передавать по существу любой объект, реализующий IDraw3D. (Попытка передачи типа, не поддерживающего необходимый интерфейс, приводит ошибке на этапе компиляции.) Взгляните на следующий метод, определенный в классе Program:

```
// Будет рисовать любую фигуру, поддерживающую IDraw3D.
static void DrawIn3D(IDraw3D itf3d)
{
    Console.WriteLine("-> Drawing IDraw3D compatible type");
    itf3d.Draw3D();
}
```

Теперь можно проверить, поддерживает ли элемент в массиве Shape новый интерфейс, и если поддерживает, то передать его методу DrawIn3D() на обработку:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interfaces *****\n");
    Shape[] myShapes = { new Hexagon(), new Circle(),
                        new Triangle("Joe"), new Circle("JoJo") };
    for(int i = 0; i < myShapes.Length; i++)
    {
        ...
        // Можно ли нарисовать эту фигуру в трехмерном виде?
        if(myShapes[i] is IDraw3D)
            DrawIn3D((IDraw3D)myShapes[i]);
    }
}
```

Ниже представлен вывод, полученный из модифицированной версии приложения. Обратите внимание, что в трехмерном виде отображается только объект Hexagon, т.к. все остальные члены массива Shape не реализуют интерфейс IDraw3D:

```
***** Fun with Interfaces *****
Drawing NoName the Hexagon
-> Points: 6
-> Drawing IDraw3D compatible type
Drawing Hexagon in 3D!
Drawing NoName the Circle
-> NoName's not pointy!
Drawing Joe the Triangle
-> Points: 3
Drawing JoJo the Circle
-> JoJo's not pointy!
```

Использование интерфейсов в качестве возвращаемых значений

Интерфейсы могут также применяться в качестве типов возвращаемых значений методов. Например, можно было бы написать метод, который получает массив объектов Shape и возвращает ссылку на первый элемент, поддерживающий IPointy:

```
// Этот метод возвращает первый объект в массиве,
// который реализует интерфейс IPointy.
static IPointy FindFirstPointyShape(Shape[] shapes)
{
    foreach (Shape s in shapes)
    {
        if (s is IPointy ip)
            return ip;
    }
    return null;
}
```

Взаимодействовать с методом FindFirstPointyShape() можно так:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interfaces *****\n");
```



```
// Создать массив элементов Shape.
Shape[] myShapes = { new Hexagon(), new Circle(),
                    new Triangle("Joe"), new Circle("JoJo")};
// Получить первый элемент, имеющий вершины.
// В целях безопасности не помешает проверить firstPointyItem на равенство null.
IPointy firstPointyItem = FindFirstPointyShape(myShapes);
Console.WriteLine("The item has {0} points", firstPointyItem.Points);
...
}
```

Массивы интерфейсных типов

Вспомните, что один интерфейс может быть реализован множеством типов, даже если они не находятся внутри той же самой иерархии классов и не имеют общего родительского класса помимо `System.Object`. Это позволяет формировать очень мощные программные конструкции. Например, пусть в текущем проекте разработаны три новых класса: два класса (`Knife` (нож) и `Fork` (вилка)) моделируют кухонные приборы, а третий (`PitchFork` (вилы)) — садовый инструмент (рис. 8.4).

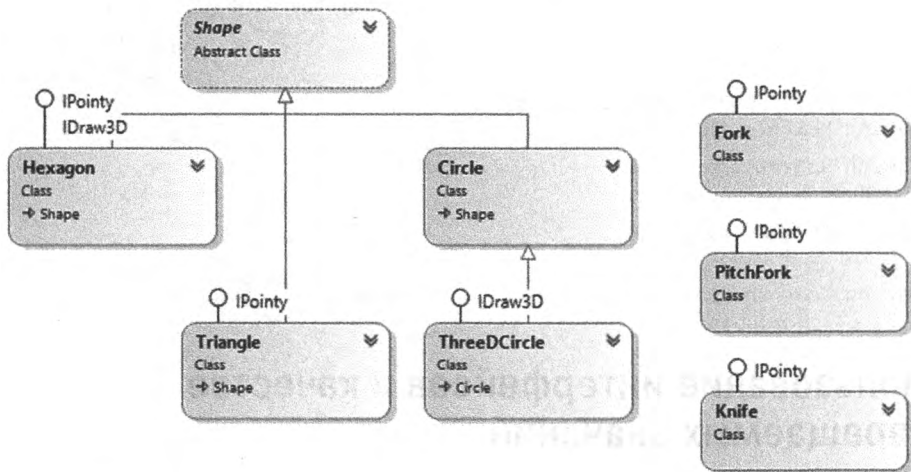


Рис. 8.4. Интерфейсы могут “подключаться” к любому типу внутри любой части иерархии классов

После того, как типы `PitchFork`, `Fork` и `Knife` определены, можно определить массив объектов, совместимых с `IPointy`. Поскольку все элементы поддерживают один и тот же интерфейс, можно выполнять проход по массиву и интерпретировать каждый его элемент как объект, совместимый с `IPointy`, несмотря на разнородность иерархий классов:

```
static void Main(string[] args)
{
    ...
    // Этот массив может содержать только типы,
    // которые реализуют интерфейс IPointy.
    IPointy[] myPointyObjects = {new Hexagon(), new Knife(),
                                new Triangle(), new Fork(), new PitchFork()};
    foreach(IPointy i in myPointyObjects)
        Console.WriteLine("Object has {0} points.", i.Points);
    Console.ReadLine();
}
```

Просто чтобы подчеркнуть важность продемонстрированного примера, запомните, что массив заданного интерфейсного типа может содержать элементы любых классов или структур, реализующих этот интерфейс.

Реализация интерфейсов с использованием Visual Studio

Хотя программирование на основе интерфейсов является мощным приемом, реализация интерфейсов может быть сопряжена с довольно большим объемом клавиатурного ввода. Учитывая, что интерфейсы являются именованными наборами абстрактных членов, для каждого метода интерфейса в каждом типе, который поддерживает данное поведение, потребуется вводить определение и реализацию. Следовательно, если вы хотите поддерживать интерфейс, который определяет пять методов и три свойства, тогда придется принять во внимание все восемь членов (иначе возникнут ошибки на этапе компиляции).

К счастью, в Visual Studio поддерживаются разнообразные инструменты, упрощающие задачу реализации интерфейсов. Для примера вставим в текущий проект еще один класс по имени `PointyTestClass`. Когда вы добавите к типу класса интерфейс, такой как `IPointy` (или любой другой подходящий интерфейс), то заметите, что по окончании ввода имени интерфейса (или при помещении на него курсора мыши в окне редактора кода) первая буква имени выделяется подчеркиванием (формально это называется *смарт-тегом*). Щелчок на смарт-теге приводит к отображению раскрывающегося списка, который позволяет реализовать интересующий интерфейс (рис. 8.5).

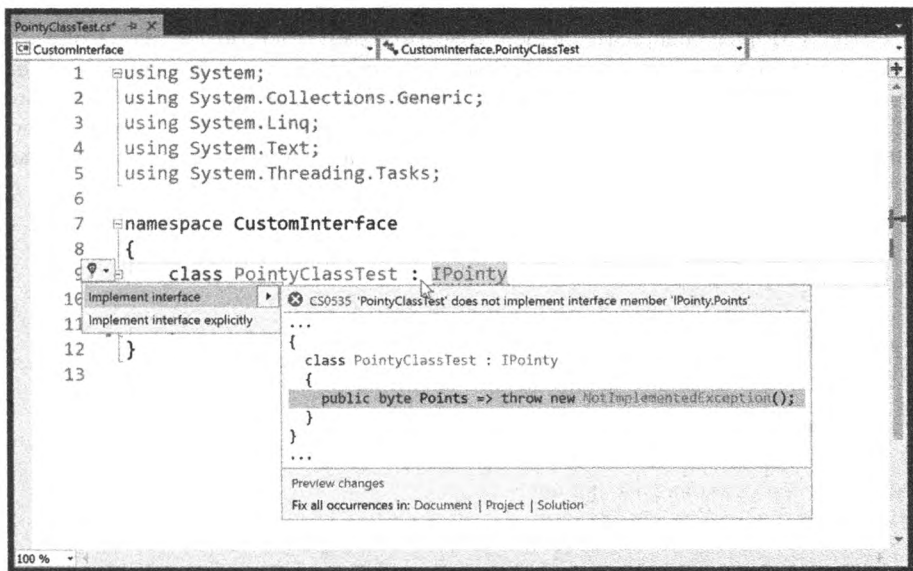


Рис. 8.5. Реализация интерфейсов в Visual Studio

Обратите внимание, что в списке предлагаются два пункта, из которых второй (явная реализация интерфейса) будет обсуждаться в следующем разделе. Для начала выберем первый пункт. Среда Visual Studio сгенерирует код заглушки, подлежащий обновлению (как видите, стандартная реализация генерирует исключение `System.NotImplementedException`, что вполне очевидно можно удалить):

```
namespace CustomInterface
{
    class PointyTestClass : IPointy
    {
        public byte Points
        {
            get { throw new NotImplementedException(); }
        }
    }
}
```

На заметку! Среда Visual Studio также поддерживает рефакторинг в форме извлечения интерфейса (Extract Interface), доступный через пункт Extract Interface (Извлечь интерфейс) меню Quick Actions (Быстрые действия). Такой рефакторинг позволяет извлечь новое определение интерфейса из существующего определения класса. Например, вы можете находиться где-то на полпути к завершению написания класса, но вдруг вас осеняет, что данное поведение можно обобщить в виде интерфейса (открывая возможность для альтернативных реализаций).

Исходный код. Проект CustomInterface доступен в подкаталоге Chapter_8.

Явная реализация интерфейсов

Как было показано ранее в главе, класс или структура может реализовывать любое количество интерфейсов. С учетом этого всегда существует возможность реализации интерфейсов, которые содержат члены с идентичными именами, из-за чего придется устранять конфликты имен. Чтобы проиллюстрировать разнообразные способы решения данной проблемы, создадим новый проект консольного приложения по имени InterfaceNameClash и добавим в него три специальных интерфейса, представляющих различные места, в которых реализующий их тип может визуализировать свой вывод:

```
// Вывести изображение на форму.
public interface IDrawToForm
{
    void Draw();
}

// Вывести изображение в буфер памяти.
public interface IDrawToMemory
{
    void Draw();
}

// Вывести изображение на принтер.
public interface IDrawToPrinter
{
    void Draw();
}
```

Обратите внимание, что в каждом интерфейсе определен метод по имени Draw() с идентичной сигнатурой (без аргументов). Если все объявленные интерфейсы необходимо поддерживать в одном классе Octagon, то компилятор разрешит следующее определение:

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
    public void Draw()
    {
        // Разделяемая логика вывода.
        Console.WriteLine("Drawing the Octagon...");
    }
}
```

Хотя компиляция такого кода пройдет гладко, здесь присутствует потенциальная проблема. Выражаясь просто, предоставление единственной реализации метода `Draw()` не позволяет предпринимать уникальные действия на основе того, какой интерфейс получен от объекта `Octagon`. Например, представленный ниже код будет приводить к вызову того же самого метода `Draw()` независимо от того, какой интерфейс получен:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
    // Все эти обращения приводят к вызову одного и того же метода Draw() !
    Octagon oct = new Octagon();

    IDrawToForm itfForm = (IDrawToForm)oct;
    itfForm.Draw();

    IDrawToPrinter itfPriner = (IDrawToPrinter)oct;
    itfPriner.Draw();

    IDrawToMemory itfMemory = (IDrawToMemory)oct;
    itfMemory.Draw();

    Console.ReadLine();
}
```

Очевидно, что код, требуемый для визуализации изображения в окне, значительно отличается от кода, который необходим для вывода изображения на сетевой принтер или в область памяти. При реализации нескольких интерфейсов, имеющих идентичные члены, разрешить подобный конфликт имен можно с применением синтаксиса *явной реализации интерфейсов*. Взгляните на следующую модификацию типа `Octagon`:

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
    // Явно привязать реализации Draw() к конкретным интерфейсам.
    void IDrawToForm.Draw()
    {
        Console.WriteLine("Drawing to form..."); // Вывод на форму
    }
    void IDrawToMemory.Draw()
    {
        Console.WriteLine("Drawing to memory..."); // Вывод в память
    }
    void IDrawToPrinter.Draw()
    {
        Console.WriteLine("Drawing to a printer..."); // Вывод на принтер
    }
}
```

Как видите, при явной реализации члена интерфейса общий шаблон выглядит следующим образом:

```
возвращаемыйТип ИмяИнтерфейса.ИмяМетода(параметры){ }
```

Обратите внимание, что при использовании такого синтаксиса модификатор доступа не указывается; явно реализованные члены автоматически будут закрытыми. Например, такой синтаксис недопустим:

```
// Ошибка! Модификатор доступа не может быть указан!
public void IDrawToForm.Draw()
{
    Console.WriteLine("Drawing to form...");
}
```

Поскольку явно реализованные члены всегда неявно закрыты, они перестают быть доступными на уровне объектов. Фактически, если вы примените к типу `Octagon` операцию точки, то обнаружите, что средство `IntelliSense` не отображает члены `Draw()`. Как и следовало ожидать, для доступа к требуемой функциональности должно использоваться явное приведение. Ниже представлен пример:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
    Octagon oct = new Octagon();

    // Теперь для доступа к членам Draw() должно использоваться приведение.
    IDrawToForm itfForm = (IDrawToForm)oct;
    itfForm.Draw();

    // Сокращенная форма, если переменная интерфейса не нужна.
    ((IDrawToPrinter)oct).Draw();

    // Можно также использовать ключевое слово is.
    if (oct is IDrawToMemory dtm)
        dtm.Draw();

    Console.ReadLine();
}
```

Наряду с тем, что этот синтаксис действительно полезен, когда необходимо устранить конфликты имен, явную реализацию интерфейсов можно применять и просто для сокрытия более “сложных” членов на уровне объектов. В таком случае при использовании операции точки пользователь объекта будет видеть только подмножество всей функциональности типа. Тем не менее, те, кому требуется более сложное поведение, могут извлекать желаемый интерфейс через явное приведение.

Исходный код. Проект `InterfaceNameClash` доступен в подкаталоге `Chapter_8`.

Проектирование иерархий интерфейсов

Интерфейсы могут быть организованы в иерархии. Подобно иерархии классов, когда интерфейс расширяет существующий интерфейс, он наследует все абстрактные члены, определяемые родителем (или родителями). Конечно, в отличие от наследования на основе классов производный интерфейс никогда не наследует действительную реализацию. Взамен он просто расширяет собственное определение дополнительными абстрактными членами.

Иерархии интерфейсов могут быть удобны, когда нужно расширить функциональность имеющегося интерфейса, не нарушая работу существующих кодовых баз. В целях иллюстрации создадим новый проект консольного приложения по имени `InterfaceHierarchy`. Затем так спроектируем новый набор интерфейсов, связанных с визуализацией, чтобы `IDrawable` был корневым интерфейсом в дереве семейства:

```
public interface IDrawable
{
    void Draw();
}
```

Учитывая, что интерфейс `IDrawable` определяет базовое поведение рисования, можно создать производный интерфейс, который расширяет `IDrawable` возможностью визуализации в других форматах, например:

```
public interface IAdvancedDraw : IDrawable
{
    void DrawInBoundingBox(int top, int left, int bottom, int right);
    void DrawUpsideDown();
}
```

При таком проектном решении, если класс реализует интерфейс `IAdvancedDraw`, тогда ему потребуется реализовать все члены, определенные в цепочке наследования (в частности, методы `Draw()`, `DrawInBoundingBox()` и `DrawUpsideDown()`):

```
public class BitmapImage : IAdvancedDraw
{
    public void Draw()
    {
        Console.WriteLine("Drawing...");
    }

    public void DrawInBoundingBox(int top, int left, int bottom, int right)
    {
        Console.WriteLine("Drawing in a box...");
    }

    public void DrawUpsideDown()
    {
        Console.WriteLine("Drawing upside down!");
    }
}
```

Теперь, когда применяется класс `BitmapImage`, появляется возможность вызывать каждый метод на уровне объекта (из-за того, что все они открыты), а также извлекать ссылку на каждый поддерживаемый интерфейс явным образом через приведение:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Simple Interface Hierarchy *****");

    // Вызвать на уровне объекта.
    BitmapImage myBitmap = new BitmapImage();
    myBitmap.Draw();
    myBitmap.DrawInBoundingBox(10, 10, 100, 150);
    myBitmap.DrawUpsideDown();

    // Получить IAdvancedDraw явным образом.
    IAdvancedDraw iAdvDraw = myBitmap as IAdvancedDraw;
    if (iAdvDraw != null)
        iAdvDraw.DrawUpsideDown();
    Console.ReadLine();
}
```

Множественное наследование с помощью интерфейсных типов

В отличие от типов классов интерфейс может расширять множество базовых интерфейсов, что позволяет проектировать мощные и гибкие абстракции. Создадим новый проект консольного приложения по имени MIInterfaceHierarchy. Здесь имеется еще одна коллекция интерфейсов, которые моделируют разнообразные абстракции, связанные с визуализацией и фигурами. Обратите внимание, что интерфейс IShape расширяет и IDrawable, и IPrintable:

```
// Множественное наследование для интерфейсных типов допускается.
interface IDrawable
{
    void Draw();
}

interface IPrintable
{
    void Print();
    void Draw(); // <-- Возможен конфликт имен!
}

// Множественное наследование интерфейсов. Разрешено!
interface IShape : IDrawable, IPrintable
{
    int GetNumberOfSides();
}
```

На рис. 8.6 показана текущая иерархия интерфейсов.

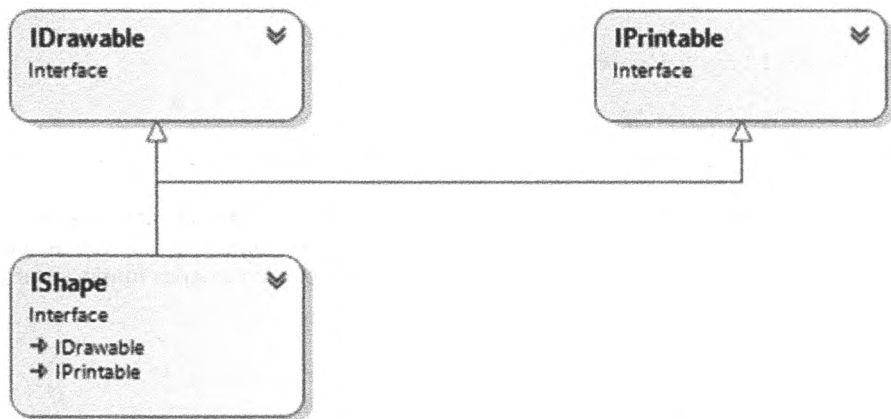


Рис. 8.6. В отличие от классов интерфейсы могут расширять сразу несколько базовых интерфейсов

Главный вопрос здесь в том, сколько методов должен реализовывать класс, поддерживающий IShape? Ответ: в зависимости от обстоятельств. Если вы хотите предоставить простую реализацию метода Draw(), тогда вам необходимо реализовать только три его члена, как иллюстрируется в следующем классе Rectangle:

```
class Rectangle : IShape
{
    public int GetNumberOfSides()
    { return 4; }
```

```

public void Draw()
{ Console.WriteLine("Drawing..."); }

public void Print()
{ Console.WriteLine("Printing..."); }
}

```

Если вы предпочитаете располагать специфическими реализациями для каждого метода `Draw()` (что в данном случае имеет смысл), то конфликт имен можно устранить с использованием явной реализации интерфейсов, как делается в представленном далее классе `Square`:

```

class Square : IShape
{
    // Использование явной реализации для устранения конфликта имен членов.
    void IPrintable.Draw()
    {
        // Вывести на принтер...
    }
    void IDrawable.Draw()
    {
        // Вывести на экран...
    }
    public void Print()
    {
        // Печатать...
    }

    public int GetNumberOfSides()
    { return 4; }
}

```

В идеале к данному моменту вы должны лучше понимать процесс определения и реализации специальных интерфейсов с применением синтаксиса C#. По правде говоря, привыкание к программированию на основе интерфейсов может занять определенное время, так что если вы находитесь в некотором замешательстве, то это совершенно нормальная реакция.

Однако имейте в виду, что интерфейсы являются фундаментальным аспектом .NET Framework. Независимо от типа разрабатываемого приложения (веб-приложение, настольное приложение с графическим пользовательским интерфейсом, библиотека доступа к данным и т.п.), работа с интерфейсами будет составной частью этого процесса. Подводя итог, запомните, что интерфейсы могут быть исключительно полезны в следующих ситуациях:

- существует единственная иерархия, в которой только подмножество производных типов поддерживает общее поведение;
- необходимо моделировать общее поведение, которое встречается в нескольких иерархиях, не имеющих общего родительского класса кроме `System.Object`.

Итак, вы ознакомились со спецификой построения и реализации специальных интерфейсов. Остаток главы посвящен исследованию нескольких предопределенных интерфейсов, содержащихся в библиотеках базовых классов .NET. Как будет показано, вы можете реализовывать стандартные интерфейсы .NET в своих специальных типах, обеспечивая их бесшовную интеграцию с платформой.

Интерфейсы IEnumerable и IEnumerator

Прежде чем приступить к исследованию процесса реализации существующих интерфейсов .NET, давайте сначала рассмотрим роль интерфейсов IEnumerable и IEnumerator. Вспомните, что язык C# поддерживает ключевое слово `foreach`, которое позволяет осуществлять проход по содержимому массива любого типа:

```
// Итерация по массиву элементов.
int[] myArrayOfInts = {10, 20, 30, 40};
foreach(int i in myArrayOfInts)
{
    Console.WriteLine(i);
}
```

Хотя может показаться, что данная конструкция подходит только для массивов, на самом деле `foreach` можно использовать с любым типом, который поддерживает метод `GetEnumerator()`. В целях иллюстрации создадим новый проект консольного приложения по имени `CustomEnumerator` и добавим в него файлы `Car.cs` и `Radio.cs`, которые были определены в примере `SimpleException` из главы 7 (через пункт меню `Project` → `Add Existing Item`).

На заметку! Во избежание импортирования в новый проект пространства имен `CustomException` имеет смысл переименовать пространство имен, содержащее типы `Car` и `Radio`, в `CustomEnumerator`.

Теперь вставим в проект новый класс `Garage` (гараж), который хранит набор объектов `Car` (автомобиль) внутри `System.Array`:

```
// Garage содержит набор объектов Car.
public class Garage
{
    private Car[] carArray = new Car[4];
    // При запуске заполнить несколькими объектами Car.
    public Garage()
    {
        carArray[0] = new Car("Rusty", 30);
        carArray[1] = new Car("Clunker", 55);
        carArray[2] = new Car("Zippy", 30);
        carArray[3] = new Car("Fred", 30);
    }
}
```

В идеальном случае было бы удобно проходить по внутренним элементам объекта `Garage` с применением конструкции `foreach` как в ситуации с массивом значений данных:

```
// Код выглядит корректным...
public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with IEnumerable / IEnumerator *****\n");
        Garage carLot = new Garage();

        // Проход по всем объектам Car в коллекции?
        foreach (Car c in carLot)
```

```

    {
        Console.WriteLine("{0} is going {1} MPH",
            c.PetName, c.CurrentSpeed);
    }
    Console.ReadLine();
}

```

К сожалению, компилятор информирует о том, что в классе `Garage` не реализован метод по имени `GetEnumerator()`, который формально определен в интерфейсе `IEnumerable`, находящемся в пространстве имен `System.Collections`.

На заметку! В главе 9 вы узнаете о роли обобщений и о пространстве имен `System.Collections.Generic`. Как будет показано, это пространство имен содержит обобщенные версии интерфейсов `IEnumerable/IEnumerator`, которые предлагают более безопасный к типам способ итерации по элементам.

Классы или структуры, которые поддерживают такое поведение, позиционируются как способные предоставлять вызывающему коду доступ к содержащимся внутри них элементам (в рассматриваемом примере самому ключевому слову `foreach`). Вот определение этого стандартного интерфейса `.NET`:

```

// Данный интерфейс информирует вызывающий код о том,
// что элементы объекта могут перечисляться.
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}

```

Как видите, метод `GetEnumerator()` возвращает ссылку на еще один интерфейс по имени `System.Collections.IEnumerator`, обеспечивающий инфраструктуру, которая позволяет вызывающему коду обходить внутренние объекты, содержащиеся в совместимом с `IEnumerable` контейнере:

```

// Этот интерфейс позволяет вызывающему коду получать элементы контейнера.
public interface IEnumerator
{
    bool MoveNext ();           // Переместить вперед внутреннюю позицию курсора.
    object Current { get; }     // Получить текущий элемент (свойство только
                                // для чтения).
    void Reset ();             // Сбросить курсор в позицию перед первым элементом.
}

```

Если вы хотите обновить тип `Garage` для поддержки этих интерфейсов, то можете пойти длинным путем и реализовать каждый метод вручную. Хотя вы определенно вольны предоставить специализированные версии методов `GetEnumerator()`, `MoveNext()`, `Current` и `Reset()`, существует более легкий путь. Поскольку тип `System.Array` (а также многие другие классы коллекций) уже реализует интерфейсы `IEnumerable` и `IEnumerator`, вы можете просто делегировать запрос к `System.Array` следующим образом (обратите внимание, что в файл кода понадобится импортировать пространство имен `System.Collections`):

```

using System.Collections;
...
public class Garage : IEnumerable
{
    // System.Array уже реализует IEnumerable!
    private Car[] carArray = new Car[4];
}

```

```

public Garage()
{
    carArray[0] = new Car("FeeFee", 200);
    carArray[1] = new Car("Clunker", 90);
    carArray[2] = new Car("Zippy", 30);
    carArray[3] = new Car("Fred", 30);
}

public IEnumerator GetEnumerator()
{
    // Возвратить IEnumerator объекта массива.
    return carArray.GetEnumerator();
}
}

```

После такого изменения тип `Garage` можно безопасно использовать внутри конструкции `foreach`. Более того, учитывая, что метод `GetEnumerator()` был определен как открытый, пользователь объекта может также взаимодействовать с типом `IEnumerator`:

```

// Вручную работать с IEnumerator.
IEnumerator i = carLot.GetEnumerator();
i.MoveNext();
Car myCar = (Car)i.Current;
Console.WriteLine("{0} is going {1} MPH", myCar.PetName, myCar.CurrentSpeed);

```

Тем не менее, если вы предпочитаете скрыть функциональность `IEnumerable` на уровне объектов, то просто примените явную реализацию интерфейса:

```

IEnumerator IEnumerable.GetEnumerator()
{
    // Возвратить IEnumerator объекта массива.
    return carArray.GetEnumerator();
}

```

В результате обычный пользователь объекта не обнаружит метод `GetEnumerator()` в классе `Garage`, в то время как конструкция `foreach` при необходимости будет получать интерфейс в фоновом режиме.

Исходный код. Проект `CustomEnumerator` доступен в подкаталоге `Chapter_8`.

Построение итераторных методов с использованием ключевого слова `yield`

Существует альтернативный способ построения типов, которые работают с циклом `foreach`, с использованием *итераторов*. Попросту говоря, *итератор* — это член, который указывает, каким образом должны возвращаться внутренние элементы контейнера во время обработки в цикле `foreach`. В целях иллюстрации создадим новый проект консольного приложения по имени `CustomEnumeratorWithYield` и вставим в него типы `Car`, `Radio` и `Garage` из предыдущего примера (снова при желании переименовав пространство имен для текущего проекта). Затем модифицируем тип `Garage`:

```

public class Garage : IEnumerable
{
    private Car[] carArray = new Car[4];
    ...
    // Итераторный метод.

```

```
public IEnumerator GetEnumerator()
{
    foreach (Car c in carArray)
    {
        yield return c;
    }
}
```

Обратите внимание, что показанная реализация метода `GetEnumerator()` осуществляет проход по элементам с применением внутренней логики `foreach` и возвращает каждый объект `Car` вызывающему коду, используя синтаксис `yield return`. Ключевое слово `yield` применяется для указания значения или значений, которые подлежат возвращению конструкцией `foreach` вызывающему коду. При достижении оператора `yield return` текущее местоположение в контейнере сохраняется и выполнение возобновляется с этого местоположения, когда итератор вызывается в следующий раз.

Итераторные методы не обязаны использовать ключевое слово `foreach` для возвращения своего содержимого. Итераторный метод допускается определять и так:

```
public IEnumerator GetEnumerator()
{
    yield return carArray[0];
    yield return carArray[1];
    yield return carArray[2];
    yield return carArray[3];
}
```

Обратите внимание в реализации на то, что метод `GetEnumerator()` явно возвращает вызывающему коду новое значение при каждом своем проходе. В рассматриваемом примере поступать подобным образом мало смысла, потому что если вы добавите дополнительные объекты к переменной-члену `carArray`, то метод `GetEnumerator()` станет рассогласованным. Тем не менее, такой синтаксис может быть полезен, когда вы хотите возвращать из метода локальные данные для обработки посредством `foreach`.

Использование локальной функции (нововведение)

Когда вызывается метод `GetEnumerator()`, код не выполняется до тех пор, пока не начнется перечисление по значению, возвращенному из метода. Модифицируем метод следующим образом, чтобы в первой строке генерировалось исключение:

```
public IEnumerator GetEnumerator()
{
    // Исключение не сгенерируется до тех пор, пока не будет вызван метод MoveNext().
    throw new Exception("This won't get called");
    foreach (Car c in carArray)
    {
        yield return c;
    }
}
```

Если функция вызывается, как показано далее, и больше ничего не делается, тогда исключение никогда не сгенерируется:

```
IEnumerator carEnumerator = carLot.GetEnumerator();
```

Код выполнится, а исключение сгенерируется не раньше, чем будет вызван метод `MoveNext()`. В зависимости от нужд программы ситуация может быть как вполне нормальной, так и нет. Вспомните средство локальных функций версии C# 7, представ-

ленное в главе 4; локальные функции — это закрытые функции, которые определены внутри других функций. Один из главных случаев применения данного нового средства связан с решением указанной проблемы (другой случай касается методов `async`, которые будут рассматриваться в главе 19).

Приведите метод к такому виду:

```
public IEnumerator GetEnumerator()
{
    // Это исключение сгенерируется немедленно.
    throw new Exception("This will get called");
    return actualImplementation();
    // Закрытая функция.
    IEnumerator actualImplementation()
    {
        foreach (Car c in carArray)
        {
            yield return c;
        }
    }
}
```

Построение именованного итератора

Также интересно отметить, что ключевое слово `yield` формально может применяться внутри любого метода независимо от его имени. Такие методы (которые официально называются *именованными итераторами*) уникальны тем, что способны принимать любое количество аргументов. При построении именованного итератора имейте в виду, что метод будет возвращать интерфейс `IEnumerable`, а не ожидаемый совместимый с `IEnumerator` тип. В целях иллюстрации добавим к типу `Garage` следующий метод (использующий локальную функцию для инкапсуляции функциональности итерации):

```
public IEnumerable GetTheCars(bool returnReversed)
{
    // Выполнить проверку на предмет ошибок.
    return actualImplementation();
    IEnumerable actualImplementation()
    {
        // Возвратить элементы в обратном порядке.
        if (returnReversed)
        {
            for (int i = carArray.Length; i != 0; i--)
            {
                yield return carArray[i - 1];
            }
        }
        else
        {
            // Возвратить элементы в том порядке, в каком они размещены в массиве.
            foreach (Car c in carArray)
            {
                yield return c;
            }
        }
    }
}
```

Обратите внимание, что новый метод позволяет вызывающему коду получать элементы в прямом, а также в обратном порядке, если во входном параметре указано значение `true`. Вот как взаимодействовать с методом `GetTheCars()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with the Yield Keyword *****\n");
    Garage carLot = new Garage();

    // Получить элементы, используя GetEnumerator().
    foreach (Car c in carLot)
    {
        Console.WriteLine("{0} is going {1} MPH",
            c.PetName, c.CurrentSpeed);
    }

    Console.WriteLine();

    // Получить элементы (в обратном порядке!)
    // с применением именованного итератора.
    foreach (Car c in carLot.GetTheCars(true))
    {
        Console.WriteLine("{0} is going {1} MPH",
            c.PetName, c.CurrentSpeed);
    }

    Console.ReadLine();
}
```

Наверняка вы согласитесь с тем, что именованные итераторы являются удобными конструкциями, поскольку они позволяют определять в единственном специальном контейнере множество способов запрашивания возвращаемого набора.

Итак, в завершение темы построения перечислимых объектов запомните: для того, чтобы специальные типы могли работать с ключевым словом `foreach` языка C#, контейнер должен определять метод по имени `GetEnumerator()`, который формально определен интерфейсным типом `IEnumerable`. Реализация этого метода обычно осуществляется просто путем делегирования работы внутреннему члену, который хранит подобъекты; однако допускается также использовать синтаксис `yield return`, чтобы предоставить множество методов "именованных итераторов".

Исходный код. Проект `CustomEnumeratorWithYield` доступен в подкаталоге `Chapter_8`.

Интерфейс `ICloneable`

Вспомните из главы 6, что в классе `System.Object` определен метод по имени `MemberwiseClone()`, который применяется для получения *поверхностной (неглубокой)* копии текущего объекта. Пользователи объекта не могут вызывать указанный метод напрямую, т.к. он является защищенным. Тем не менее, отдельный объект может самостоятельно вызывать `MemberwiseClone()` во время процесса *клонирования*. Для примера создадим новый проект консольного приложения по имени `CloneablePoint`, в котором определен класс `Point`:

```
// Класс по имени Point.
public class Point
{
    public int X {get; set;}
    public int Y {get; set;}
}
```

```

public Point(int xPos, int yPos) { X = xPos; Y = yPos;}
public Point(){}

// Переопределить Object.ToString().
public override string ToString() => $"X = {X}; Y = {Y}";
}

```

Учитывая имеющиеся у вас знания о ссылочных типах и типах значений (см. главу 4), должно быть понятно, что если вы присвоите одну переменную ссылочного типа другой такой переменной, то получите две ссылки, которые указывают на тот же самый объект в памяти. Таким образом, следующая операция присваивания в результате дает две ссылки на один и тот же объект `Point` в куче; модификация с использованием любой из ссылок оказывает воздействие на тот же самый объект в куче:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");
    // Две ссылки на один и тот же объект!
    Point p1 = new Point(50, 50);
    Point p2 = p1;
    p2.X = 0;
    Console.WriteLine(p1);
    Console.WriteLine(p2);
    Console.ReadLine();
}

```

Чтобы предоставить специальному типу возможность возвращения вызывающему коду идентичную копию самого себя, можно реализовать стандартный интерфейс `ICloneable`. Как было показано в начале главы, интерфейс `ICloneable` определяет единственный метод по имени `Clone()`:

```

public interface ICloneable
{
    object Clone();
}

```

Очевидно, что реализация метода `Clone()` варьируется от класса к классу. Однако базовая функциональность в основном остается неизменной: копирование значений переменных-членов в новый объект того же самого типа и возвращение его пользователю. Чтобы продемонстрировать сказанное, модифицируем класс `Point`:

```

// Теперь Point поддерживает способность клонирования.
public class Point : ICloneable
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int xPos, int yPos) { X = xPos; Y = yPos; }
    public Point() { }

    // Переопределить Object.ToString().
    public override string ToString() => $"X = {X}; Y = {Y}";

    // Возвратить копию текущего объекта.
    public object Clone() => new Point(this.X, this.Y);
}

```

Теперь можно создавать точные автономные копии типа `Point`, как показано далее:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");
}

```

```
// Обратите внимание, что Clone() возвращает простой тип object.
// Для получения производного типа требуется явное приведение.
Point p3 = new Point(100, 100);
Point p4 = (Point)p3.Clone();

// Изменить p4.X (что не приводит к изменению p3.x).
p4.X = 0;

// Вывести все объекты.
Console.WriteLine(p3);
Console.WriteLine(p4);
Console.ReadLine();
}
```

Несмотря на то что текущая реализация типа `Point` удовлетворяет всем требованиям, ее возможно немного улучшить. Поскольку `Point` не содержит никаких внутренних переменных ссылочного типа, реализацию метода `Clone()` можно упростить:

```
// Копировать все поля Point по очереди.
public object Clone() => this.MemberwiseClone();
```

Тем не менее, учтите, что если бы в типе `Point` содержались любые переменные-члены ссылочного типа, то метод `MemberwiseClone()` копировал бы ссылки на эти объекты (т.е. создавал бы *поверхностную копию*). Для поддержки подлинной *глубокой (детальной) копии* во время процесса клонирования понадобится создавать новые экземпляры каждой переменной-члена ссылочного типа. Давайте рассмотрим пример.

Более сложный пример клонирования

Теперь предположим, что класс `Point` содержит переменную-член ссылочного типа `PointDescription`. Данный класс представляет дружественное имя точки, а также ее идентификационный номер, выраженный как `System.Guid` (глобально уникальный идентификатор (globally unique identifier — GUID), т.е. статистически уникальное 128-битное число). Вот как выглядит реализация:

```
// Этот класс описывает точку.
public class PointDescription
{
    public string PetName { get; set; }
    public Guid PointID { get; set; }

    public PointDescription()
    {
        PetName = "No-name";
        PointID = Guid.NewGuid();
    }
}
```

Начальные изменения самого класса `Point` включают модификацию метода `ToString()` для учета новых данных состояния, а также определение и создание ссылочного типа `PointDescription`. Чтобы позволить внешнему миру устанавливать дружественное имя для `Point`, необходимо также изменить аргументы, передаваемые перегруженному конструктору:

```
public class Point : ICloneable
{
    public int X { get; set; }
    public int Y { get; set; }
    public PointDescription desc = new PointDescription();
```



```

public Point(int xPos, int yPos, string petName)
{
    X = xPos; Y = yPos;
    desc.PetName = petName;
}
public Point(int xPos, int yPos)
{
    X = xPos; Y = yPos;
}
public Point() { }

// Переопределить Object.ToString().
public override string ToString()
    => $"X = {X}; Y = {Y}; Name = {desc.PetName};\nID = {desc.PointID}\n";

// Возвратить копию текущего объекта.
public object Clone() => this.MemberwiseClone();
}

```

Обратите внимание, что метод `Clone()` пока еще не обновлялся. Следовательно, когда пользователь объекта запросит клонирование с применением текущей реализации, будет создана поверхностная (почленная) копия. В целях иллюстрации модифицируем метод `Main()`, как показано ниже:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");
    Console.WriteLine("Cloned p3 and stored new Point in p4");
    Point p3 = new Point(100, 100, "Jane");
    Point p4 = (Point)p3.Clone();

    Console.WriteLine("Before modification:");
    Console.WriteLine("p3: {0}", p3);
    Console.WriteLine("p4: {0}", p4);
    p4.desc.PetName = "My new Point";
    p4.X = 9;

    Console.WriteLine("\nChanged p4.desc.petName and p4.X");
    Console.WriteLine("After modification:");
    Console.WriteLine("p3: {0}", p3);
    Console.WriteLine("p4: {0}", p4);
    Console.ReadLine();
}

```

В приведенном выводе видно, что хотя типы значений действительно были изменены, внутренние ссылочные типы поддерживают одни и те же значения, т.к. они "указывают" на те же самые объекты в памяти (в частности, оба объекта имеют дружественное имя "My new Point"):

```

***** Fun with Object Cloning *****
Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509
p4: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509
Changed p4.desc.petName and p4.X
After modification:
p3: X = 100; Y = 100; Name = My new Point;

```

```
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509
p4: X = 9; Y = 100; Name = My new Point;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509
```

Чтобы заставить метод `Clone()` создавать полную глубокую копию внутренних ссылочных типов, нужно сконфигурировать объект, возвращаемый методом `MemberwiseClone()`, для учета имени текущего объекта `Point` (тип `System.Guid` на самом деле является структурой, так что числовые данные будут действительно копироваться). Вот одна из возможных реализаций:

```
// Теперь необходимо скорректировать код для учета члена PointDescription.
public object Clone()
{
    // Сначала получить поверхностную копию.
    Point newPoint = (Point)this.MemberwiseClone();

    // Затем восполнить пробелы.
    PointDescription currentDesc = new PointDescription();
    currentDesc.PetName = this.desc.PetName;
    newPoint.desc = currentDesc;
    return newPoint;
}
```

Если снова запустить приложение и просмотреть его вывод (показанный далее), то будет видно, что возвращаемый методом `Clone()` объект `Point` действительно копирует свои внутренние переменные-члены ссылочного типа (обратите внимание, что дружеские имена у `p3` и `p4` теперь уникальны):

```
***** Fun with Object Cloning *****

Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 100; Y = 100; Name = Jane;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a

Changed p4.desc.petName and p4.X
After modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 9; Y = 100; Name = My new Point;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a
```

Подведем итоги по процессу клонирования. При наличии класса или структуры, которая содержит только типы значений, необходимо реализовать метод `Clone()` с использованием метода `MemberwiseClone()`. Однако если есть специальный тип, поддерживающий ссылочные типы, тогда для построения глубокой копии может потребоваться создать новый объект, который учитывает каждую переменную-член ссылочного типа.

Исходный код. Проект `CloneablePoint` доступен в подкаталоге `Chapter_8`.

Интерфейс `Comparable`

Интерфейс `System.IComparable` описывает поведение, которое позволяет сортировать объекты на основе указанного ключа. Вот его формальное определение:

```
// Данный интерфейс позволяет объекту указывать
// его отношение с другими подобными объектами.
public interface IComparable
{
    int CompareTo(object o);
}
```

На заметку! Обобщенная версия этого интерфейса (`IComparable<T>`) предлагает более безопасный в отношении типов способ обработки операций сравнения объектов. Обобщения исследуются в главе 9.

Давайте создадим новый проект консольного приложения по имени `ComparableCar` и обновим класс `Car` из главы 7 (обратите внимание, что мы просто добавили новое свойство для представления уникального идентификатора каждого автомобиля и модифицированный конструктор):

```
public class Car
{
    ...
    public int CarID {get; set;}
    public Car(string name, int currSp, int id)
    {
        CurrentSpeed = currSp;
        PetName = name;
        CarID = id;
    }
    ...
}
```

Теперь предположим, что имеется следующий массив объектов `Car`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Sorting *****\n");
    // Создать массив объектов Car.
    Car[] myAutos = new Car[5];
    myAutos[0] = new Car("Rusty", 80, 1);
    myAutos[1] = new Car("Mary", 40, 234);
    myAutos[2] = new Car("Viper", 40, 34);
    myAutos[3] = new Car("Mel", 40, 4);
    myAutos[4] = new Car("Chuck", 40, 5);
    Console.ReadLine();
}
```

В классе `System.Array` определен статический метод по имени `Sort()`. Его вызов для массива внутренних типов (`int`, `short`, `string` и т.д.) приводит к сортировке элементов массива в числовом или алфавитном порядке, т.к. внутренние типы данных реализуют интерфейс `IComparable`. Но что произойдет, если передать методу `Sort()` массив объектов `Car`?

```
// Сортируются ли объекты Car? Пока еще нет!
Array.Sort(myAutos);
```

Запустив тестовый код, вы получите исключение времени выполнения, потому что класс `Car` не поддерживает необходимый интерфейс. При построении специальных типов вы можете реализовать интерфейс `IComparable`, чтобы позволить массивам, содержащим элементы этих типов, подвергаться сортировке. Когда вы реализуете детали

`CompareTo()`, то должны самостоятельно принять решение о том, что должно браться за основу в операции упорядочивания. Для типа `Car` вполне логичным кандидатом может служить внутреннее свойство `CarID`:

```
// Итерация по объектам Car может быть упорядочена на основе CarID.
public class Car : IComparable
{
    ...
    // Реализация интерфейса IComparable.
    int IComparable.CompareTo(object obj)
    {
        Car temp = obj as Car;
        if (temp != null)
        {
            if (this.CarID > temp.CarID)
                return 1;
            if (this.CarID < temp.CarID)
                return -1;
            else
                return 0;
        }
        else
            throw new ArgumentException("Parameter is not a Car!");
    }
    // Параметр не является объектом типа Car
}
```

Как видите, логика метода `CompareTo()` заключается в сравнении входного объекта с текущим экземпляром на основе специфичного элемента данных. Возвращаемое значение метода `CompareTo()` применяется для выяснения того, является текущий объект меньше, больше или равным объекту, с которым он сравнивается (табл. 8.1).

Таблица 8.1. Возвращаемые значения метода `CompareTo()`

Возвращаемое значение метода <code>CompareTo()</code>	Описание
Любое число меньше нуля	Этот экземпляр находится перед указанным объектом в порядке сортировки
Ноль	Этот экземпляр равен указанному объекту
Любое число больше нуля	Этот экземпляр находится после указанного объекта в порядке сортировки

Предыдущую реализацию метода `CompareTo()` можно усовершенствовать с учетом того факта, что тип данных `int` в C# (который представляет собой просто сокращенное обозначение для типа `System.Int32` в CLR) реализует интерфейс `IComparable`. Реализовать `CompareTo()` в `Car` можно было бы так:

```
int IComparable.CompareTo(object obj)
{
    Car temp = obj as Car;
    if (temp != null)
        return this.CarID.CompareTo(temp.CarID);
    else
        throw new ArgumentException("Parameter is not a Car!");
    // Параметр не является объектом типа Car
}
```

В любом случае, поскольку тип `Car` понимает, как сравнивать себя с подобными объектами, вы можете написать следующий тестовый код:

```
// Использование интерфейса IComparable.
static void Main(string[] args)
{
    // Создать массив объектов Car.
    ...
    // Отобразить текущее содержимое массива.
    Console.WriteLine("Here is the unordered set of cars:");
    foreach(Car c in myAutos)
        Console.WriteLine("{0} {1}", c.CarID, c.PetName);

    // Теперь отсортировать массив с применением IComparable!
    Array.Sort(myAutos);
    Console.WriteLine();

    // Отобразить отсортированное содержимое массива.
    Console.WriteLine("Here is the ordered set of cars:");
    foreach(Car c in myAutos)
        Console.WriteLine("{0} {1}", c.CarID, c.PetName);
    Console.ReadLine();
}
```

Ниже показан вывод, полученный в результате выполнения приведенного выше метода `Main()`:

```
***** Fun with Object Sorting *****
Here is the unordered set of cars:
1 Rusty
234 Mary
34 Viper
4 Mel
5 Chucky

Here is the ordered set of cars:
1 Rusty
4 Mel
5 Chucky
34 Viper
234 Mary
```

Указание множества порядков сортировки с помощью `IComparer`

В текущей версии класса `Car` в качестве основы для порядка сортировки используется идентификатор автомобиля (`CarID`). В другом проектном решении основой сортировки могло быть дружественное имя автомобиля (для вывода списка автомобилей в алфавитном порядке). А что если вы хотите построить класс `Car`, который можно было бы подвергать сортировке по идентификатору *и также* по дружественному имени? В таком случае вы должны ознакомиться с еще одним стандартным интерфейсом по имени `IComparer`, который определен в пространстве имен `System.Collections` следующим образом:

```
// Общий способ сравнения двух объектов.
interface IComparer
{
    int Compare(object o1, object o2);
}
```

На заметку! Обобщенная версия этого интерфейса (`IComparer<T>`) предоставляет более безопасный в отношении типов способ обработки операций сравнения объектов. Обобщения подробно рассматриваются в главе 9.

В отличие от `Comparable` интерфейс `Comparer` обычно не реализуется в типе, который вы пытаетесь сортировать (т.е. `Car`). Взамен данный интерфейс реализуется в любом количестве вспомогательных классов, по одному для каждого порядка сортировки (на основе дружественного имени, идентификатора автомобиля и т.д.). В настоящий момент типу `Car` уже известно, как сравнивать автомобили друг с другом по внутреннему идентификатору. Следовательно, чтобы позволить пользователю объекта сортировать массив объектов `Car` по дружественному имени, потребуется создать дополнительный вспомогательный класс, реализующий интерфейс `Comparer`. Вот необходимый код (не забудьте импортировать в файл кода пространство имен `System.Collections`):

```
// Этот вспомогательный класс используется для сортировки
// массива объектов Car по дружественному имени.
public class PetNameComparer : IComparer
{
    // Проверить дружественное имя каждого объекта.
    int IComparer.Compare(object o1, object o2)
    {
        Car t1 = o1 as Car;
        Car t2 = o2 as Car;
        if(t1 != null && t2 != null)
            return String.Compare(t1.PetName, t2.PetName);
        else
            throw new ArgumentException("Parameter is not a Car!");
        // Параметр не является объектом типа Car!
    }
}
```

Теперь вспомогательный класс `PetNameComparer` можно задействовать в коде. Класс `System.Array` содержит несколько перегруженных версий метода `Sort()`, одна из которых принимает объект, реализующий интерфейс `Comparer`:

```
static void Main(string[] args)
{
    ...
    // Теперь сортировать по дружественному имени.
    Array.Sort(myAutos, new PetNameComparer());

    // Вывести отсортированный массив.
    Console.WriteLine("Ordering by pet name:");
    foreach(Car c in myAutos)
        Console.WriteLine("{0} {1}", c.CarID, c.PetName);
    ...
}
```

Специальные свойства и специальные типы сортировки

Важно отметить, что вы можете применять специальное статическое свойство, вызывая пользователю объекта помощь с сортировкой типов `Car` по специфичному элементу данных. Предположим, что в класс `Car` добавлено статическое свойство только для чтения по имени `SortByPetName`, которое возвращает экземпляр класса, реализующего интерфейс `Comparer` (в этом случае `PetNameComparer`; не забудьте импортировать пространство имен `System.Collections`):

```
// Теперь мы поддерживаем специальное свойство для возвращения
// корректного экземпляра, реализующего интерфейс IComparer.
public class Car : IComparable
{
    ...
    // Свойство, возвращающее PetNameComparer.
    public static IComparer SortByPetName
    { get { return (IComparer)new PetNameComparer(); } }
}
```

Теперь в коде массив можно сортировать по дружественному имени, используя жестко ассоциированное свойство, а не автономный класс `PetNameComparer`:

```
// Сортировка по дружественному имени становится немного яснее.
Array.Sort(myAutos, Car.SortByPetName);
```

Исходный код. Проект `ComparableCar` доступен в подкаталоге `Chapter_8`.

К настоящему моменту вы должны не только понимать способы определения и реализации собственных интерфейсов, но и оценить их полезность. Конечно, интерфейсы встречаются внутри каждого важного пространства имен .NET, и в оставшихся главах книги вы продолжите работать с разнообразными стандартными интерфейсами.

Резюме

Интерфейс может быть определен как именованная коллекция *абстрактных членов*. Поскольку интерфейс не предоставляет никаких деталей реализации, общепринято расценивать его как поведение, которое может поддерживаться заданным типом. Когда два или больше число типов реализуют один и тот же интерфейс, каждый из них может трактоваться одинаковым образом (полиморфизм на основе интерфейсов), даже если типы определены в разных иерархиях.

Для определения новых интерфейсов в языке C# предусмотрено ключевое слово `interface`. Как было показано в главе, тип может поддерживать столько интерфейсов, сколько необходимо, и интерфейсы указываются в виде списка с разделителями-запятymi. Более того, разрешено создавать интерфейсы, которые являются производными от множества базовых интерфейсов.

В дополнение к построению специальных интерфейсов библиотеки .NET определяют набор стандартных (т.е. поставляемых вместе с платформой) интерфейсов. Вы видели, что можно создавать специальные типы, которые реализуют предопределенные интерфейсы с целью поддержки набора желательных возможностей, таких как клонирование, сортировка и перечисление.

часть IV

Дополнительные конструкции программирования на C#

В этой части

Глава 9. Коллекции и обобщения

Глава 10. Делегаты, события и лямбда-выражения

Глава 11. Расширенные средства языка C#

Глава 12. LINQ to Objects

Глава 13. Время существования объектов

ГЛАВА 9

Коллекции и обобщения

Любому приложению, создаваемому с помощью платформы .NET, потребуется решать вопросы поддержки и манипулирования набором значений данных в памяти. Значения данных могут поступать из множества местоположений, включая реляционную базу данных, локальный текстовый файл, XML-документ, вызов веб-службы, или через предоставляемый пользователем источник ввода.

В первом выпуске платформы .NET программисты часто применяли классы из пространства имен `System.Collections` для хранения и взаимодействия с элементами данных, используемыми внутри приложения. В версии .NET 2.0 язык программирования C# был расширен поддержкой средства, которое называется *обобщениями*, и вместе с этим изменением в библиотеках базовых классов появилось новое пространство имен — `System.Collections.Generic`.

В настоящей главе представлен обзор разнообразных пространств имен и типов коллекций (обобщенных и необобщенных), находящихся в библиотеках базовых классов .NET. Вы увидите, что обобщенные контейнеры часто превосходят свои необобщенные аналоги, поскольку они обычно обеспечивают лучшую безопасность в отношении типов и дают выигрыш в плане производительности. После того, как вы научитесь создавать и манипулировать обобщенными элементами внутри платформы, в оставшемся материале главы будет продемонстрировано создание собственных обобщенных методов и типов. Вы узнаете о роли *ограничений* (и соответствующего ключевого слова `where` языка C#), которые позволяют строить классы, исключительно безопасные к типам.

Побудительные причины создания классов коллекций

Несомненно, самым элементарным контейнером, который допускается применять для хранения данных приложения, считается массив. В главе 4 вы узнали, что массив C# позволяет определить набор идентично типизированных элементов (в том числе массив элементов типа `System.Object`, по существу представляющий собой массив данных любых типов) с фиксированным верхним пределом. Кроме того, вспомните из главы 4, что все переменные массивов C# получают много функциональных возможностей от класса `System.Array`. В качестве краткого напоминания взгляните на следующий метод `Main()`, который создает массив текстовых данных и манипулирует его содержимым разными способами:

```
static void Main(string[] args)
{
    // Создать массив строковых данных.
    string[] strArray = {"First", "Second", "Third"};

    // Отобразить количество элементов в массиве с помощью свойства Length.
    Console.WriteLine("This array has {0} items.", strArray.Length);
    Console.WriteLine();
}
```

```
// Отобразить содержимое массива, используя перечислитель.
foreach (string s in strArray)
{
    Console.WriteLine("Array Entry: {0}", s);
}
Console.WriteLine();

// Обратить массив и снова вывести его содержимое.
Array.Reverse(strArray);
foreach (string s in strArray)
{
    Console.WriteLine("Array Entry: {0}", s);
}
Console.ReadLine();
}
```

Хотя базовые массивы могут быть удобными для управления небольшими объемами данных фиксированного размера, есть немало случаев, когда требуются более гибкие структуры данных, такие как динамически расширяющийся и сокращающийся контейнер или контейнер, который может хранить только объекты, удовлетворяющие заданному критерию (например, объекты, производные от специфичного базового класса, или объекты, реализующие определенный интерфейс). Когда вы используете простой массив, всегда помните о том, что он имеет “фиксированный размер”. Если вы создали массив из трех элементов, то вы и получите только три элемента; следовательно, представленный далее код даст в результате исключение времени выполнения (конкретно — `IndexOutOfRangeException`):

```
static void Main(string[] args)
{
    // Создать массив строковых данных.
    string[] strArray = { "First", "Second", "Third" };

    // Попытка добавить новый элемент в конец массива?
    // Ошибка во время выполнения!
    strArray[3] = "new item?";
    ...
}
```

На заметку! На самом деле изменять размер массива можно с применением обобщенного метода `Resize()<T>`. Однако такое действие приведет к копированию данных в новый объект массива и может оказаться неэффективным.

Чтобы помочь в преодолении ограничений простого массива, библиотеки базовых классов .NET поставляются с несколькими пространствами имен, которые содержат классы коллекций. В отличие от простого массива C# классы коллекций построены с возможностью динамического изменения своих размеров на лету по мере вставки либо удаления из них элементов. Более того, многие классы коллекций предлагают улучшенную безопасность в отношении типов и всерьез оптимизированы для обработки содержащихся внутри данных в манере, эффективной с точки зрения затрат памяти. В ходе чтения главы вы быстро заметите, что класс коллекции может принадлежать к одной из двух обширных категорий:

- необобщенные коллекции (в основном находящиеся в пространстве имен `System.Collections`);
- обобщенные коллекции (в основном находящиеся в пространстве имен `System.Collections.Generic`).

Необобщенные коллекции обычно спроектированы для оперирования над типами `System.Object` и, следовательно, являются слабо типизированными контейнерами (тем не менее, некоторые необобщенные коллекции работают только со специфическим типом данных наподобие объектов `string`). По контрасту обобщенные коллекции являются намного более безопасными к типам, учитывая, что при создании вы должны указывать “вид типа” данных, которые они будут содержать. Как вы увидите, признаком любого обобщенного элемента является наличие “параметра типа”, обозначаемого с помощью угловых скобок (например, `List<T>`). Детали обобщений (в том числе связанные с ними преимущества) будут исследоваться позже в этой главе. А сейчас давайте ознакомимся с некоторыми ключевыми типами необобщенных коллекций из пространств имен `System.Collections` и `System.Collections.Specialized`.

Пространство имен `System.Collections`

С самого первого выпуска платформы .NET программисты часто использовали классы необобщенных коллекций из пространства имен `System.Collecitons`, которое содержит набор классов, предназначенных для управления и организации крупных объемов данных в памяти. В табл. 9.1 документированы распространенные классы коллекций, определенные в этом пространстве имен, а также основные интерфейсы, которые они реализуют.

Таблица 9.1. Полезные классы из пространства имен `System.Collections`

Класс <code>System.Collections</code>	Описание	Основные реализуемые интерфейсы
<code>ArrayList</code>	Представляет коллекцию с динамически изменяемым размером, выдающую объекты в последовательном порядке	<code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>
<code>BitArray</code>	Управляет компактным массивом битовых значений, которые представляются как булевские, где <code>true</code> обозначает установленный (1) бит, а <code>false</code> — неустановленный (0) бит	<code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>
<code>Hashtable</code>	Представляет коллекцию пар “ключ-значение”, организованных на основе хеш-кода ключа	<code>IDictionary</code> , <code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>
<code>Queue</code>	Представляет стандартную очередь объектов, работающую по принципу FIFO (“первый вошел — первый вышел”)	<code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>
<code>SortedList</code>	Представляет коллекцию пар “ключ-значение”, отсортированных по ключу и доступных по ключу и по индексу	<code>IDictionary</code> , <code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>
<code>Stack</code>	Представляет стек LIFO (“последний вошел — первый вышел”), поддерживающий функциональность заталкивания и выталкивания, а также считывания	<code>ICollection</code> , <code>IEnumerable</code> и <code>ICloneable</code>

Интерфейсы, реализованные перечисленными в табл. 9.1 классами коллекций, позволяют проникнуть в суть их общей функциональности. В табл. 9.2 представлено описание общей природы основных интерфейсов, часть из которых кратко обсуждалась в главе 8.

Таблица 9.2. Основные интерфейсы, поддерживаемые классами из пространства имен `System.Collections`

Интерфейс <code>System.Collections</code>	Описание
<code>ICollection</code>	Определяет общие характеристики (например, размер, перечисление и безопасность к потокам) для всех необобщенных типов коллекций
<code>ICloneable</code>	Позволяет реализующему объекту возвращать вызывающему коду копию самого себя
<code>IDictionary</code>	Позволяет объекту необобщенной коллекции представлять свое содержимое в виде пар "ключ-значение"
<code>IEnumerable</code>	Возвращает объект, реализующий интерфейс <code>IEnumerator</code> (см. следующую строку в таблице)
<code>IEnumerator</code>	Делает возможной итерацию в стиле <code>foreach</code> по элементам коллекции
<code>IList</code>	Обеспечивает поведение добавления, удаления и индексирования элементов в последовательном списке объектов

Иллюстративный пример: работа с `ArrayList`

Возможно, вы уже имеете начальный опыт применения (или реализации) некоторых из указанных выше классических структур данных, таких как стеки, очереди или списки. Если это не так, то при рассмотрении обобщенных аналогов таких структур позже в главе будут предоставлены дополнительные сведения об отличиях между ними. А пока что взгляните на метод `Main()`, в котором используется объект `ArrayList`. Обратите внимание, что мы можем добавлять (и удалять) элементы на лету, а контейнер автоматически соответствующим образом изменяет свой размер:

```
// Для доступа к ArrayList потребуется импортировать
// пространство имен System.Collections.
static void Main(string[] args)
{
    ArrayList strArray = new ArrayList();
    strArray.AddRange(new string[] { "First", "Second", "Third" });

    // Отобразить количество элементов в ArrayList.
    Console.WriteLine("This collection has {0} items.", strArray.Count);
    Console.WriteLine();

    // Добавить новый элемент и отобразить текущее их количество.
    strArray.Add("Fourth!");
    Console.WriteLine("This collection has {0} items.", strArray.Count);

    // Отобразить содержимое.
    foreach (string s in strArray)
    {
        Console.WriteLine("Entry: {0}", s);
    }
    Console.WriteLine();
}
```

Как вы могли догадаться, помимо свойства `Count` и методов `AddRange()` и `Add()` класс `ArrayList` имеет много полезных членов, которые полностью описаны в документации `.NET Framework`. К слову, другие классы `System.Collections` (`Stack`, `Queue` и т.д.) также подробно документированы в справочной системе `.NET`.

Однако важно отметить, что в большинстве ваших проектов .NET классы коллекций из пространства имен `System.Collections`, скорее всего, применяться не будут! В наши дни намного чаще используются их обобщенные аналоги, находящиеся в пространстве имен `System.Collections.Generic`. С учетом сказанного остальные не-обобщенные классы из `System.Collections` здесь не обсуждаются (и примеры работы с ними не приводятся).

Обзор пространства имен `System.Collections.Specialized`

`System.Collections` — не единственное пространство имен .NET, которое содержит необобщенные классы коллекций. В пространстве имен `System.Collections.Specialized` определено несколько специализированных типов коллекций. В табл. 9.3 описаны наиболее полезные типы в этом конкретном пространстве имен, которые все являются необобщенными.

Таблица 9.3. Полезные классы из пространства имен `System.Collections.Specialized`

Класс <code>System.Collections.Specialized</code>	Описание
<code>HybridDictionary</code>	Этот класс реализует интерфейс <code>IDictionary</code> за счет применения <code>ListDictionary</code> , пока коллекция мала, и переключения на <code>Hashtable</code> , когда коллекция становится большой
<code>ListDictionary</code>	Этот класс удобен, когда необходимо управлять небольшим количеством элементов (10 или около того), которые могут изменяться с течением времени. Для управления своими данными класс использует односвязный список
<code>StringCollection</code>	Этот класс обеспечивает оптимальный способ для управления крупными коллекциями строковых данных
<code>BitVector32</code>	Этот класс предоставляет простую структуру, которая хранит булевские значения и небольшие целые числа в 32 битах памяти

Кроме указанных конкретных типов классов пространство имен `System.Collections.Specialized` также содержит много дополнительных интерфейсов и абстрактных базовых классов, которые можно применять в качестве стартовых точек для создания специальных классов коллекций. Хотя в ряде ситуаций такие “специализированные” типы могут оказаться именно тем, что требуется в ваших проектах, здесь они рассматриваться не будут. И снова во многих ситуациях вы наверняка обнаружите, что пространство имен `System.Collections.Generic` предлагает классы с похожей функциональностью, но с добавочными преимуществами.

На заметку! В библиотеках базовых классов .NET доступны два дополнительных пространства имен, связанные с коллекциями (`System.Collections.ObjectModel` и `System.Collections.Concurrent`). Первое из них будет объясняться позже в главе, когда вы освоите тему обобщений. Пространство имен `System.Collections.Concurrent` предоставляет классы коллекций, хорошо подходящие для многопоточной среды (многопоточность обсуждается в главе 19).

Проблемы, присущие необобщенным коллекциям

Хотя на протяжении многих лет с использованием необобщенных классов коллекций (и интерфейсов) было построено немало успешных приложений .NET, опыт показал, что применение этих типов может привести к возникновению ряда проблем.

Первая проблема заключается в том, что использование классов коллекций `System.Collections` и `System.Collections.Specialized` в результате дает код с низкой производительностью, особенно в случае манипулирования числовыми данными (например, типами значений). Как вы вскоре увидите, когда структуры хранятся в любом необобщенном классе коллекции, прототипированном для оперирования с `System.Object`, среда CLR должна выполнять некоторое количество операций перемещения в памяти, что может нанести ущерб скорости выполнения.

Вторая проблема связана с тем, что большинство необобщенных классов коллекций не являются безопасными в отношении типов, т.к. они были созданы для работы с `System.Object` и потому могут содержать в себе вообще все что угодно. Если разработчик .NET нуждался в создании безопасной к типам коллекции (скажем, контейнера, который способен хранить объекты, реализующие только определенный интерфейс), то единственным реальным вариантом было создание нового класса коллекции вручную. Хотя задача не отличалась высокой трудоемкостью, она была несколько утомительной.

Прежде чем вы увидите, как применять обобщения в своих программах, полезно чуть глубже рассмотреть недостатки необобщенных классов коллекций, что поможет лучше понять проблемы, которые был призван решить механизм обобщений. Давайте создадим новый проект консольного приложения по имени `IssuesWithNongenericCollections`, после чего импортируем пространство имен `System.Collections` в файл кода C#:

```
using System.Collections;
```

Проблема производительности

Как уже было указано в главе 4, платформа .NET поддерживает две обширные категории данных: типы значений и ссылочные типы. Поскольку в .NET определены две основные категории типов, временами возникает необходимость представить переменную одной категории как переменную другой категории. Для этого в C# предлагается простой механизм, называемый *упаковкой* (boxing), который позволяет хранить данные типа значения внутри ссылочной переменной. Предположим, что в методе по имени `SimpleBoxUnboxOperation()` создана локальная переменная типа `int`. Если где-то в приложении понадобится представить такой тип значения как ссылочный тип, то значение придется *упаковать*:

```
static void SimpleBoxUnboxOperation()
{
    // Создать переменную ValueType (int).
    int myInt = 25;

    // Упаковать int в ссылку на object.
    object boxedInt = myInt;
}
```

Упаковку можно формально определить как процесс явного присваивания данных типа значения переменной `System.Object`. При упаковке значения среда CLR размещает в куче новый объект и копирует в него величину типа значения (в данном случае 25). В качестве результата возвращается ссылка на вновь размещенный в куче объект.

Противоположная операция также разрешена и называется *распаковкой* (unboxing). Распаковка представляет собой процесс преобразования значения, хранящегося в объ-

ектной ссылке, обратно в соответствующий тип значения в стеке. Синтаксически операция распаковки выглядит как обычная операция приведения, но ее семантика несколько отличается. Среда CLR начинает с проверки того, что полученный тип данных эквивалентен упакованному типу, и если это так, то копирует значение в переменную, находящуюся в стеке. Например, следующие операции распаковки работают успешно при условии, что лежащим в основе типом `boxedInt` действительно является `int`:

```
static void SimpleBoxUnboxOperation()
{
    // Создать переменную ValueType (int).
    int myInt = 25;

    // Упаковать int в ссылку на object.
    object boxedInt = myInt;

    // Распаковать ссылку обратно в int.
    int unboxedInt = (int)boxedInt;
}
```

Когда компилятор C# встречает синтаксис упаковки/распаковки, он выпускает код CIL, который содержит коды операций `box/unbox`. Если вы просмотрите сборку с помощью утилиты `ildasm.exe`, то обнаружите в ней показанный далее код CIL:

```
.method private hidebysig static void SimpleBoxUnboxOperation() cil managed
{
    // Code size 19 (0x13)
    .maxstack 1
    .locals init ([0] int32 myInt, [1] object boxedInt, [2] int32 unboxedInt)
    IL_0000: nop
    IL_0001: ldc.i4.s 25
    IL_0003: stloc.0
    IL_0004: ldloc.0
    IL_0005: box [mscorlib]System.Int32
    IL_000a: stloc.1
    IL_000b: ldloc.1
    IL_000c: unbox.any [mscorlib]System.Int32
    IL_0011: stloc.2
    IL_0012: ret
} // end of method Program::SimpleBoxUnboxOperation
```

Помните, что в отличие от обычного приведения распаковка *обязана* осуществляться только в подходящий тип данных. Попытка распаковать порцию данных в некорректный тип приводит к генерации исключения `InvalidCastException`. Для обеспечения высокой безопасности каждая операция распаковки должна быть помещена внутрь конструкции `try/catch`, но такое действие со всеми операциями распаковки в приложении может оказаться достаточно трудоемкой задачей. Ниже показан измененный код, который выдаст ошибку из-за того, что в нем предпринята попытка распаковки упакованного значения `int` в тип `long`:

```
static void SimpleBoxUnboxOperation()
{
    // Создать переменную ValueType (int).
    int myInt = 25;

    // Упаковать int в ссылку на object.
    object boxedInt = myInt;

    // Распаковать в неподходящий тип данных, чтобы
    // инициализировать исключение времени выполнения.
}
```

```

try
{
    long unboxedInt = (long)boxedInt;
}
catch (InvalidCastException ex)
{
    Console.WriteLine(ex.Message);
}
}

```

На первый взгляд упаковка/распаковка может показаться довольно непримечательным средством языка, с которым связан больше академический интерес, нежели практическая ценность. В конце концов, необходимость хранения локального типа значения в локальной переменной `object` будет возникать нечасто. Тем не менее, оказывается, что процесс упаковки/распаковки очень полезен, поскольку позволяет предполагать, что все можно трактовать как `System.Object`, а среда CLR самостоятельно позаботится о деталях, касающихся памяти.

Давайте обратимся к практическому использованию описанных приемов. Предположим, что создан необобщенный класс `System.Collections.ArrayList` для хранения множества числовых (расположенных в стеке) данных. Члены `ArrayList` прототипированы для работы с данными `System.Object`. Теперь рассмотрим методы `Add()`, `Insert()`, `Remove()`, а также индексатор класса:

```

public class ArrayList : object,
    IList, ICollection, IEnumerable, ICloneable
{
    ...
    public virtual int Add(object value);
    public virtual void Insert(int index, object value);
    public virtual void Remove(object obj);
    public virtual object this[int index] {get; set; }
}

```

Класс `ArrayList` был построен для оперирования с экземплярами `object`, которые представляют данные, находящиеся в куче, поэтому может показаться странным, что следующий код компилируется и выполняется без ошибок:

```

static void WorkWithArrayList()
{
    // Типы значений автоматически упаковываются при передаче
    // методу, который требует экземпляр типа object.
    ArrayList myInts = new ArrayList();
    myInts.Add(10);
    myInts.Add(20);
    myInts.Add(35);
}

```

Хотя здесь числовые данные напрямую передаются методам, которые требуют экземпляров типа `object`, исполняющая среда выполняет автоматическую упаковку таких основанных на стеке данных. Когда позже понадобится извлечь элемент из `ArrayList` с применением индексатора типа, находящийся в куче объект должен быть распакован в целочисленное значение, расположенное в стеке, посредством операции приведения. Не забывайте, что индексатор `ArrayList` возвращает элементы типа `System.Object`, а не `System.Int32`:


```
static void WorkWithArrayList()
{
    // Типы значений автоматически упаковываются,
    // когда передаются члену, принимающему object.
    ArrayList myInts = new ArrayList();
    myInts.Add(10);
    myInts.Add(20);
    myInts.Add(35);

    // Распаковка происходит, когда объект преобразуется
    // обратно в данные, расположенные в стеке.
    int i = (int)myInts[0];

    // Теперь значение вновь упаковывается, т.к.
    // метод WriteLine() требует типа object!
    Console.WriteLine("Value of your int: {0}", i);
}
```

Обратите внимание, что расположенное в стеке значение типа `System.Int32` перед вызовом метода `ArrayList.Add()` упаковывается, чтобы оно могло быть передано в требуемом виде `System.Object`. Кроме того, объект `System.Object` распаковывается обратно в `System.Int32` после его извлечения из `ArrayList` через операцию приведения лишь для того, чтобы снова быть упакованными при передаче методу `Console.WriteLine()`, поскольку данный метод работает с типом `System.Object`.

Упаковка и распаковка удобны с точки зрения программиста, но такой упрощенный подход к передаче данных между стеком и кучей влечет за собой проблемы, связанные с производительностью (снижение скорости выполнения и увеличение размера кода), а также приводит к утрате безопасности в отношении типов. Чтобы понять проблемы с производительностью, примите во внимание действия, которые должны произойти при упаковке и распаковке простого целочисленного значения.

1. Новый объект должен быть размещен в управляемой куче.
2. Значение данных, находящееся в стеке, должно быть передано в выделенное место в памяти.
3. При распаковке значение, которое хранится в объекте, находящемся в куче, должно быть передано обратно в стек.
4. Неиспользуемый в дальнейшем объект, расположенный в куче, будет (со временем) удален сборщиком мусора.

Несмотря на то что показанный конкретный метод `Main()` не создает значительное узкое место в плане производительности, вы определенно заметите такое влияние, если `ArrayList` будет содержать тысячи целочисленных значений, которыми программа манипулирует на регулярной основе. В идеальном мире мы могли бы обрабатывать данные, находящиеся внутри контейнера в стеке, безо всяких проблем с производительностью. Было бы замечательно иметь возможность извлекать данные из контейнера, не прибегая к конструкциям `try/catch` (именно это позволяют делать обобщения).

Проблема безопасности к типам

Мы уже затрагивали проблему безопасности к типам, когда рассматривали операции распаковки. Вспомните, что данные должны быть распакованы в тот же самый тип, с которым они объявлялись перед упаковкой. Однако существует еще один аспект безопасности к типам, который необходимо иметь в виду в мире без обобщений: тот факт, что классы из пространства имен `System.Collections` обычно могут хранить любые данные, т.к. их члены прототипированы для оперирования с типом `System.Object`.

Например, следующий метод строит список `ArrayList` с произвольными фрагментами несвязанных данных:

```
static void ArrayListOfRandomObjects()
{
    // ArrayList может хранить вообще все что угодно.
    ArrayList allMyObjects = new ArrayList();
    allMyObjects.Add(true);
    allMyObjects.Add(new OperatingSystem(PlatformID.MacOSX, new Version(10, 0)));
    allMyObjects.Add(66);
    allMyObjects.Add(3.14);
}
```

В ряде случаев вам будет требоваться исключительно гибкий контейнер, который способен хранить буквально все (как было здесь показано). Но большую часть времени вас интересует *безопасный в отношении типов* контейнер, который может работать только с определенным типом данных. Например, вы можете нуждаться в контейнере, хранящем только объекты типа подключения к базе данных, растрового изображения или класса, реализующего интерфейс `IPointy`.

До появления обобщений единственный способ решения проблемы, касающейся безопасности к типам, предусматривал создание вручную специального класса (строго типизированной) коллекции. Предположим, что вы хотите создать специальную коллекцию, которая способна содержать только объекты типа `Person`:

```
public class Person
{
    public int Age {get; set;}
    public string FirstName {get; set;}
    public string LastName {get; set;}

    public Person(){}
    public Person(string firstName, string lastName, int age)
    {
        Age = age;
        FirstName = firstName;
        LastName = lastName;
    }

    public override string ToString()
    {
        return $"Name: {FirstName} {LastName}, Age: {Age}";
    }
}
```

Чтобы построить коллекцию, которая способна хранить только объекты `Person`, можно определить переменную-член `System.Collection.ArrayList` внутри класса по имени `PeopleCollection` и сконфигурировать все члены для оперирования со строго типизированными объектами `Person`, а не с объектами типа `System.Object`. Ниже приведен простой пример (специальная коллекция производственного уровня могла бы поддерживать множество дополнительных членов и расширять абстрактный базовый класс из пространства имен `System.Collections` или `System.Collections.Specialized`):

```
public class PersonCollection : IEnumerable
{
    private ArrayList arPeople = new ArrayList();

    // Приведение для вызывающего кода.
    public Person GetPerson(int pos) => (Person)arPeople[pos];
}
```

```
// Вставка только объектов Person.
public void AddPerson(Person p)
{ arPeople.Add(p); }

public void ClearPeople()
{ arPeople.Clear(); }

public int Count => arPeople.Count;

// Поддержка перечисления с помощью foreach.
IEnumerator IEnumerable.GetEnumerator() => arPeople.GetEnumerator();
}
```

Обратите внимание, что класс `PeopleCollection` реализует интерфейс `IEnumerable`, который делает возможной итерацию в стиле `foreach` по всем элементам, содержащимся в коллекции. Кроме того, методы `GetPerson()` и `AddPerson()` прототипированы для работы только с объектами `Person`, а не растровыми изображениями, строками, подключениями к базам данных или другими элементами. Благодаря определению таких классов теперь обеспечивается безопасность к типам, учитывая, что компилятор C# будет способен выявить любую попытку вставки элемента несовместимого типа:

```
static void UsePersonCollection()
{
    Console.WriteLine("***** Custom Person Collection *****\n");
    PersonCollection myPeople = new PersonCollection();
    myPeople.AddPerson(new Person("Homer", "Simpson", 40));
    myPeople.AddPerson(new Person("Marge", "Simpson", 38));
    myPeople.AddPerson(new Person("Lisa", "Simpson", 9));
    myPeople.AddPerson(new Person("Bart", "Simpson", 7));
    myPeople.AddPerson(new Person("Maggie", "Simpson", 2));

    // Это вызовет ошибку на этапе компиляции!
    // myPeople.AddPerson(new Car());

    foreach (Person p in myPeople)
        Console.WriteLine(p);
}
```

Хотя специальные коллекции гарантируют безопасность к типам, такой подход обязывает создавать (в основном идентичные) специальные коллекции для всех уникальных типов данных, которые планируется в них помещать. Таким образом, если нужна специальная коллекция, которая могла бы оперировать только с классами, производными от базового класса `Car`, тогда придется построить очень похожий класс коллекции:

```
public class CarCollection : IEnumerable
{
    private ArrayList arCars = new ArrayList();

    // Приведение для вызывающего кода.
    public Car GetCar(int pos) => (Car)arCars[pos];

    // Вставка только объектов Car.
    public void AddCar(Car c)
    { arCars.Add(c); }

    public void ClearCars()
    { arCars.Clear(); }

    public int Count => arCars.Count;

    // Поддержка перечисления с помощью foreach.
    IEnumerator IEnumerable.GetEnumerator() => arCars.GetEnumerator();
}
```

Тем не менее, класс специальной коллекции ничего не делает для решения проблемы с накладными расходами по упаковке/распаковке. Даже если создать специальную коллекцию по имени `IntCollection`, которая предназначена для работы только с элементами `System.Int32`, то все равно придется выделять память под объект какого-то вида, хранящий данные (например, `System.Array` и `ArrayList`):

```
public class IntCollection : IEnumerable
{
    private ArrayList arInts = new ArrayList();

    // Получение int (выполняется распаковка) .
    public int GetInt(int pos) => (int)arInts[pos];

    // Вставка int (выполняется упаковка) .
    public void AddInt(int i)
    { arInts.Add(i); }

    public void ClearInts()
    { arInts.Clear(); }

    public int Count => arInts.Count;

    IEnumerator IEnumerable.GetEnumerator() => arInts.GetEnumerator();
}
```

Независимо от того, какой тип выбран для хранения целых чисел, в случае применения необобщенных контейнеров затруднительного положения с упаковкой избежать невозможно.

Первый взгляд на обобщенные коллекции

Когда используются классы обобщенных коллекций, все описанные выше проблемы исчезают, включая накладные расходы на упаковку/распаковку и отсутствие безопасности в отношении типов. К тому же необходимость в создании специального класса (обобщенной) коллекции становится довольно редкой. Вместо построения уникальных классов, которые могут хранить объекты людей, автомобилей и целые числа, можно задействовать класс обобщенной коллекции и указать тип хранимых элементов.

Взгляните на следующий метод, в котором используется класс `List<T>` (из пространства имен `System.Collection.Generic`) для хранения разнообразных видов данных в строго типизированной манере (пока что не обращайте внимания на детали синтаксиса обобщений):

```
static void UseGenericList()
{
    Console.WriteLine("***** Fun with Generics *****\n");

    // Этот объект List<> может хранить только объекты Person.
    List<Person> morePeople = new List<Person>();
    morePeople.Add(new Person ("Frank", "Black", 50));
    Console.WriteLine(morePeople[0]);

    // Этот объект List<> может хранить только целые числа.
    List<int> moreInts = new List<int>();
    moreInts.Add(10);
    moreInts.Add(2);
    int sum = moreInts[0] + moreInts[1];

    // Ошибка на этапе компиляции! Объект Person
    // не может быть добавлен в список элементов int!
    // moreInts.Add(new Person());
}
```

Первый контейнер `List<T>` может содержать только объекты `Person`. По этой причине выполнять приведение при извлечении элементов из контейнера не требуется, что делает такой подход более безопасным в отношении типов. Второй контейнер `List<T>` может хранить только целые числа, размещенные в стеке; другими словами, здесь не происходит никакой скрытой упаковки/распаковки, которая имеет место в неособоном `ArrayList`. Ниже приведен краткий перечень преимуществ обобщенных контейнеров по сравнению с их неособоными аналогами.

- Обобщения обеспечивают лучшую производительность, т.к. лишены накладных расходов по упаковке/распаковке, когда хранят типы значений.
- Обобщения безопасны к типам, потому что могут содержать только объекты указанного типа.
- Обобщения значительно сокращают потребность в специальных типах коллекций, поскольку при создании обобщенного контейнера указывается "вид типа".

Исходный код. Проект `IssuesWithNonGenericCollections` доступен в подкаталоге `Chapter_9`.

Роль параметров обобщенных типов

Обобщенные классы, интерфейсы, структуры и делегаты вы можете обнаружить повсюду в библиотеках базовых классов .NET, и они могут быть частью любого пространства имен .NET. Кроме того, имейте в виду, что применение обобщений далеко не ограничивается простым определением класса коллекции. Разумеется, в оставшихся главах книги вы встретите случаи использования многих других обобщений для самых разных целей.

На заметку! Обобщенным образом могут быть записаны только классы, структуры, интерфейсы и делегаты, но не перечисления.

Глядя на обобщенный элемент в документации .NET Framework или браузеру объектов Visual Studio, вы заметите пару угловых скобок с буквой или другой лексемой внутри. На рис. 9.1 показано окно браузера объектов Visual Studio, в котором отображается набор обобщенных элементов из пространства имен `System.Collections.Generic`, включающий выделенный класс `List<T>`.

Формально эти лексемы называются *параметрами типа*, но в более дружественных к пользователю терминах на них можно ссылаться просто как на *заполнители*. Конструкцию `<T>` можно читать как "типа *T*". Таким образом, `IEnumerable<T>` можно прочитать как "IEnumerable типа *T*" или, говоря по-другому, как "перечисление типа *T*".

На заметку! Имя параметра типа (заполнитель) к делу не относится и зависит от предпочтений разработчика, создавшего обобщенный элемент. Однако обычно имя *T* применяется для представления типов, *TKey* или *K* — для представления ключей и *TValue* или *V* — для представления значений.

Когда вы создаете обобщенный объект, реализуете обобщенный интерфейс или вызываете обобщенный член, на вас возлагается обязанность по предоставлению значения для параметра типа. Многочисленные примеры вы увидите как в этой главе, так и в остальных материалах книги. Тем не менее, для начала рассмотрим основы взаимодействия с обобщенными типами и членами.

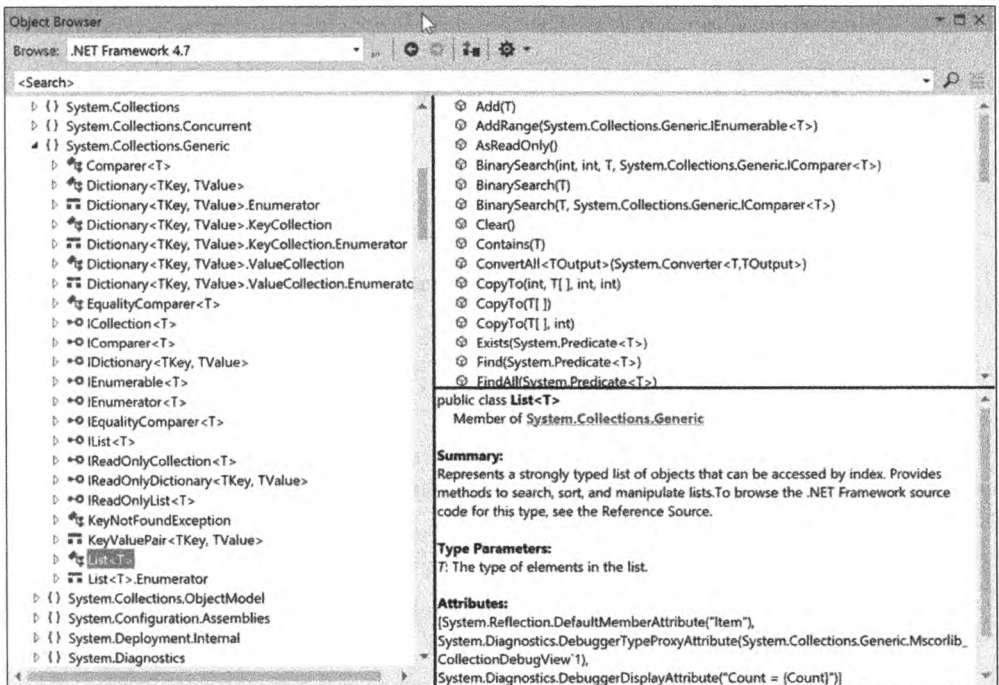


Рис. 9.1. Обобщенные элементы, поддерживающие параметры типа

Указание параметров типа для обобщенных классов и структур

При создании экземпляра обобщенного класса или структуры вы указываете параметр типа, когда объявляете переменную и когда вызываете конструктор. Как было показано в предыдущем фрагменте кода, в методе `UseGenericList()` определены два объекта `List<T>`:

```
// Этот объект List<> может хранить только объекты Person.
List<Person> morePeople = new List<Person>();
```

Данный фрагмент можно трактовать как "List<> объектов T, где T — тип Person" или более просто как "список объектов действующих лиц". После указания параметра типа обобщенного элемента изменить его нельзя (помните, что сущностью обобщений является безопасность к типам). Когда параметр типа задается для обобщенного класса или структуры, все вхождения заполнителя (заполнителей) заменяются предоставленным значением.

Если вы просмотрите полное объявление обобщенного класса `List<T>` в браузере объектов Visual Studio, то заметите, что заполнитель T используется в определении повсеместно. Ниже приведен частичный листинг (обратите внимание на элементы, выделенные полужирным):

```
// Частичное определение класса List<T>.
namespace System.Collections.Generic
{
    public class List<T> :
        IList<T>, ICollection<T>, IEnumerable<T>, IReadOnlyList<T>
        IList, ICollection, IEnumerable
```

```

{
    ...
    public void Add(T item);
    public ReadOnlyCollection<T> AsReadOnly();
    public int BinarySearch(T item);
    public bool Contains(T item);
    public void CopyTo(T[] array);
    public int FindIndex(System.Predicate<T> match);
    public T FindLast(System.Predicate<T> match);
    public bool Remove(T item);
    public int RemoveAll(System.Predicate<T> match);
    public T[] ToArray();
    public bool TrueForAll(System.Predicate<T> match);
    public T this[int index] { get; set; }
}
}

```

В случае создания `List<T>` с указанием объектов `Person` результат будет таким же, как если бы тип `List<T>` был определен следующим образом:

```

namespace System.Collections.Generic
{
    public class List<Person> :
        IList<Person>, ICollection<Person>, IEnumerable<Person>,
        IReadOnlyList<Person>
        IList, ICollection, IEnumerable
    {
        ...
        public void Add(Person item);
        public ReadOnlyCollection<Person> AsReadOnly();
        public int BinarySearch(Person item);
        public bool Contains(Person item);
        public void CopyTo(Person[] array);
        public int FindIndex(System.Predicate<Person> match);
        public Person FindLast(System.Predicate<Person> match);
        public bool Remove(Person item);
        public int RemoveAll(System.Predicate<Person> match);
        public Person[] ToArray();
        public bool TrueForAll(System.Predicate<Person> match);
        public Person this[int index] { get; set; }
    }
}

```

Несомненно, когда вы создаете в коде переменную обобщенного типа `List<T>`, компилятор вовсе не создает новую реализацию класса `List<T>`. Взамен он принимает во внимание только члены обобщенного типа, к которым вы действительно обращаетесь.

Указание параметров типа для обобщенных членов

В необобщенном классе или структуре допускается поддерживать обобщенные свойства. В таких случаях необходимо также указывать значение заполнителя во время вызова метода. Например, класс `System.Array` поддерживает набор обобщенных методов. В частности, необобщенный статический метод `Sort()` имеет обобщенный аналог по имени `Sort<T>()`. Рассмотрим представленный далее фрагмент кода, где `T` — тип `int`:

```

int[] myInts = { 10, 4, 2, 33, 93 };

// Указание заполнителя для обобщенного метода Sort<>().

```

```

Array.Sort<int>(myInts);
foreach (int i in myInts)
{
    Console.WriteLine(i);
}

```

Указание параметров типов для обобщенных интерфейсов

Обобщенные интерфейсы обычно реализуются при построении классов или структур, которые нуждаются в поддержке разнообразных аспектов поведения платформы (скажем, клонирования, сортировки и перечисления). В главе 8 вы узнали о нескольких необобщенных интерфейсах, таких как `Comparable`, `Enumerable`, `IEnumerator` и `IComparer`. Вспомните, что необобщенный интерфейс `Comparable` определен примерно так:

```

public interface Comparable
{
    int CompareTo(object obj);
}

```

В главе 8 этот интерфейс также был реализован классом `Car`, чтобы сделать возможной сортировку стандартного массива. Однако код требовал нескольких проверок времени выполнения и операций приведения, потому что параметром был общий тип `System.Object`:

```

public class Car : Comparable
{
    ...
    // Реализация Comparable.
    int Comparable.CompareTo(object obj)
    {
        Car temp = obj as Car;
        if (temp != null)
        {
            if (this.CarID > temp.CarID)
                return 1;
            if (this.CarID < temp.CarID)
                return -1;
            else
                return 0;
        }
        else
            throw new ArgumentException("Parameter is not a Car!");
        // Параметр не является объектом типа Car!
    }
}

```

Теперь представим, что применяется обобщенный аналог данного интерфейса:

```

public interface Comparable<T>
{
    int CompareTo(T obj);
}

```

В таком случае код реализации будет значительно яснее:

```

public class Car : Comparable<Car>
{
    ...
}

```



```
// Реализация IComparable<T>.
int IComparable<Car>.CompareTo(Car obj)
{
    if (this.CarID > obj.CarID)
        return 1;
    if (this.CarID < obj.CarID)
        return -1;
    else
        return 0;
}
```

Здесь уже не нужно проверять, относится ли входной параметр к типу Car, потому что он может быть *только* Car! В случае передачи несовместимого типа данных возникает ошибка на этапе компиляции. Теперь, когда вы лучше понимаете, как взаимодействовать с обобщенными элементами, а также роль параметров типа (т.е. заполнителей), вы готовы к исследованию классов и интерфейсов из пространства имен System.Collections.Generic.

Пространство имен System.Collections.Generic

Когда вы строите приложение .NET и необходим способ управления данными в памяти, классы из пространства имен System.Collections.Generic вероятно удовлетворят всем требованиям. В начале настоящей главы кратко упоминались некоторые основные необобщенные интерфейсы, реализуемые необобщенными классами коллекций. Не должен вызывать удивление тот факт, что в пространстве имен System.Collections.Generic для многих из них определены обобщенные замены.

В действительности вы сможете найти некоторое количество обобщенных интерфейсов, которые расширяют свои необобщенные аналоги, что может показаться странным. Тем не менее, за счет этого реализующие их классы будут также поддерживать унаследованную функциональность, которая имеется в их необобщенных родственниках. Например, интерфейс IEnumerable<T> расширяет IEnumerable.

В табл. 9.4 описаны основные обобщенные интерфейсы, с которыми вы столкнетесь во время работы с обобщенными классами коллекций.

Таблица 9.4. Основные интерфейсы, поддерживаемые классами из пространства имен System.Collections.Generic

Интерфейс System.Collections.Generic	Описание
ICollection<T>	Определяет общие характеристики (например, размер, перечисление и безопасность к потокам) для всех типов обобщенных коллекций
IComparer<T>	Определяет способ сравнения объектов
IDictionary<TKey, TValue>	Позволяет объекту обобщенной коллекции представлять свое содержимое посредством пар "ключ-значение"
IEnumerable<T>	Возвращает интерфейс IEnumerator<T> для заданного объекта
IEnumerator<T>	Позволяет выполнять итерацию в стиле foreach по элементам коллекции
IList<T>	Обеспечивает поведение добавления, удаления и индексации элементов в последовательном списке объектов
ISet<T>	Предоставляет базовый интерфейс для абстракции множеств

В пространстве имен `System.Collections.Generic` также определены классы, реализующие многие из указанных основных интерфейсов. В табл. 9.5 описаны часто используемые классы из этого пространства имен, реализуемые ими интерфейсы, а также их базовая функциональность.

Таблица 9.5. Классы из пространства имен `System.Collections.Generic`

Обобщенный класс	Поддерживаемые основные интерфейсы	Описание
<code>Dictionary<TKey, TValue></code>	<code>ICollection<T></code> , <code>IDictionary<TKey, TValue></code> , <code>IEnumerable<T></code>	Представляет обобщенную коллекцию ключей и значений
<code>LinkedList<T></code>	<code>ICollection<T></code> , <code>IEnumerable<T></code>	Представляет двухсвязный список
<code>List<T></code>	<code>ICollection<T></code> , <code>IEnumerable<T></code> , <code>IList<T></code>	Представляет последовательный список элементов с динамически изменяемым размером
<code>Queue<T></code>	<code>ICollection</code> (это не опечатка; именно так называется необобщенный интерфейс коллекции), <code>IEnumerable<T></code>	Обобщенная реализация списка, работающего по алгоритму "первый вошел — первый вышел" (FIFO)
<code>SortedDictionary<TKey, TValue></code>	<code>ICollection<T></code> , <code>IDictionary<TKey, TValue></code> , <code>IEnumerable<T></code>	Обобщенная реализация сортированного множества пар "ключ-значение"
<code>SortedSet<T></code>	<code>ICollection<T></code> , <code>IEnumerable<T></code> , <code>ISet<T></code>	Представляет коллекцию объектов, поддерживаемых в отсортированном порядке без дубликатов
<code>Stack<T></code>	<code>ICollection</code> (это не опечатка; именно так называется необобщенный интерфейс коллекции), <code>IEnumerable<T></code>	Обобщенная реализация списка, работающего по алгоритму "последний вошел — первый вышел" (LIFO)

В пространстве имен `System.Collections.Generic` также определены многие вспомогательные классы и структуры, которые работают в сочетании со специфическим контейнером. Например, тип `LinkedListNode<T>` представляет узел внутри обобщенного контейнера `LinkedList<T>`, исключение `KeyNotFoundException` генерируется при попытке получить элемент из коллекции с применением несуществующего ключа и т.д.

Полезно отметить, что `mscorlib.dll` и `System.dll` — не единственные сборки, которые добавляют новые типы в пространство имен `System.Collections.Generic`. Например, `System.Core.dll` добавляет класс `HashSet<T>`. Подробные сведения о пространстве имен `System.Collections.Generic` доступны в документации .NET Framework.

В любом случае следующая ваша задача состоит в том, чтобы научиться использовать некоторые из упомянутых классов обобщенных коллекций. Тем не менее, сначала полезно ознакомиться со средством языка C# (введенным в версии .NET 3.5), которое упрощает заполнение данными обобщенных (и необобщенных) коллекций.

Синтаксис инициализации коллекций

В главе 4 вы узнали о *синтаксисе инициализации массивов*, который позволяет устанавливать элементы новой переменной массива во время ее создания. С ним тесно связан *синтаксис инициализации коллекций*. Данное средство языка C# позволяет наполнять многие контейнеры (такие как `ArrayList` или `List<T>`) элементами с применением синтаксиса, похожего на тот, который используется для наполнения базовых массивов.

На заметку! Синтаксис инициализации коллекций может применяться только к классам, которые поддерживают метод `Add()`, формально определяемый интерфейсами `ICollection<T>` и `ICollection`.

Взгляните на следующие примеры:

```
// Инициализация стандартного массива.
int[] myArrayOfInts = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Инициализация обобщенного List<T> с элементами int.
List<int> myGenericList = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Инициализация ArrayList числовыми данными.
ArrayList myList = new ArrayList { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Если контейнером является коллекция классов или структур, тогда синтаксис инициализации коллекций можно смешивать с синтаксисом инициализации объектов, получая функциональный код. Вспомните класс `Point` из главы 5, в котором были определены два свойства, `X` и `Y`. Для построения обобщенного списка `List<T>` объектов `P` можно написать такой код:

```
List<Point> myListOfPoints = new List<Point>
{
    new Point { X = 2, Y = 2 },
    new Point { X = 3, Y = 3 },
    new Point(PointColor.BloodRed) { X = 4, Y = 4 }
};

foreach (var pt in myListOfPoints)
{
    Console.WriteLine(pt);
}
```

Преимущество этого синтаксиса связано с сокращением объема клавиатурного ввода. Хотя вложенные фигурные скобки могут затруднять чтение, если не позаботиться о надлежащем форматировании, вы только вообразите себе объем кода, который пришлось бы написать для наполнения следующего списка `List<T>` объектов `Rectangle` без использования синтаксиса инициализации коллекций (класс `Rectangle` создавался в главе 4 и содержит два свойства, инкапсулирующие объекты `Point`):

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
    new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
                  BottomRight = new Point { X = 200, Y = 200}},
    new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
                  BottomRight = new Point { X = 100, Y = 100}},
    new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
                  BottomRight = new Point { X = 90, Y = 75}}
};
```

```
foreach (var r in myListOfRects)
{
    Console.WriteLine(r);
}
```

Работа с классом List<T>

Создадим новый проект консольного приложения по имени FunWithGeneric Collections. Обратите внимание, что в начальном файле кода C# пространство имен System.Collections.Generic уже импортировано.

Первым мы будем исследовать обобщенный класс List<T>, который уже применялся ранее в главе. Класс List<T> используется чаще других классов из пространства имен System.Collections.Generic, т.к. он позволяет динамически изменять размер контейнера. Чтобы проиллюстрировать его особенности, добавим в класс Program метод UseGenericList(), в котором задействован класс List<T> для манипулирования набором объектов Person; вспомните, что в классе Person определены три свойства (Age, FirstName и LastName), а также специальная реализация метода ToString():

```
static void UseGenericList()
{
    // Создать список объектов Person и заполнить его с помощью
    // синтаксиса инициализации объектов и коллекции.
    List<Person> people = new List<Person>()
    {
        new Person {FirstName= "Homer", LastName="Simpson", Age=47},
        new Person {FirstName= "Marge", LastName="Simpson", Age=45},
        new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
        new Person {FirstName= "Bart", LastName="Simpson", Age=8}
    };
    // Вывести количество элементов в списке.
    Console.WriteLine("Items in list: {0}", people.Count);
    // Выполнить перечисление по списку.
    foreach (Person p in people)
        Console.WriteLine(p);
    // Вставить новый объект Person.
    Console.WriteLine("\n->Inserting new person.");
    people.Insert(2,
        new Person { FirstName = "Maggie", LastName = "Simpson", Age = 2 });
    Console.WriteLine("Items in list: {0}", people.Count);
    // Скопировать данные в новый массив.
    Person[] arrayOfPeople = people.ToArray();
    foreach (Person p in arrayOfPeople)
    {
        Console.WriteLine("First Names: {0}", p.FirstName);
    }
}
```

Здесь для наполнения списка List<T> объектами применяется синтаксис инициализации в качестве сокращенной записи *многократного* вызова метода Add(). После вывода количества элементов в коллекции (а также перечисления по всем элементам) вызывается метод Insert(). Как видите, метод Insert() позволяет вставлять новый элемент в List<T> по указанному индексу.

Наконец, обратите внимание на вызов метода ToArray(), который возвращает массив объектов Person, основанный на содержимом исходного списка List<T>. Затем осуществляется проход по всем элементам данного массива с использованием синтаксиса индексатора массива.

Вызов метода UseGenericList() в Main() приводит к получению следующего вывода:

```
***** Fun with Generic Collections *****
Items in list: 4
Name: Homer Simpson, Age: 47
Name: Marge Simpson, Age: 45
Name: Lisa Simpson, Age: 9
Name: Bart Simpson, Age: 8
->Inserting new person.
Items in list: 5
First Names: Homer
First Names: Marge
First Names: Maggie
First Names: Lisa
First Names: Bart
```

В классе `List<T>` определено множество дополнительных членов, представляющих интерес, поэтому за полным их описанием обращайтесь в документацию .NET Framework. Давайте рассмотрим еще несколько обобщенных коллекций, в частности `Stack<T>`, `Queue<T>` и `SortedSet<T>`, что должно способствовать лучшему пониманию основных вариантов хранения данных в приложении.

Работа с классом `Stack<T>`

Класс `Stack<T>` представляет коллекцию элементов, которая обслуживает элементы в стиле "последний вошел — первый вышел" (LIFO). Как и можно было ожидать, в `Stack<T>` определены члены `Push()` и `Pop()`, предназначенные для вставки и удаления элементов из стека. Приведенный ниже метод создает стек объектов `Person`:

```
static void UseGenericStack()
{
    Stack<Person> stackOfPeople = new Stack<Person>();
    stackOfPeople.Push(new Person
        { FirstName = "Homer", LastName = "Simpson", Age = 47 });
    stackOfPeople.Push(new Person
        { FirstName = "Marge", LastName = "Simpson", Age = 45 });
    stackOfPeople.Push(new Person
        { FirstName = "Lisa", LastName = "Simpson", Age = 9 });
    // Просмотреть верхний элемент, вытолкнуть его и просмотреть снова.
    Console.WriteLine("First person is: {0}", stackOfPeople.Peek());
    Console.WriteLine("Popped off {0}", stackOfPeople.Pop());

    Console.WriteLine("\nFirst person is: {0}", stackOfPeople.Peek());
    Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
    Console.WriteLine("\nFirst person item is: {0}", stackOfPeople.Peek());
    Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
    try
    {
        Console.WriteLine("\nnFirst person is: {0}", stackOfPeople.Peek());
        Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine("\nError! {0}", ex.Message); // Ошибка! Стек пуст.
    }
}
```

В коде мы строим стек, который содержит информацию о трех лицах, добавленных в порядке их имен: Homer, Marge и Lisa. Заглядывая (посредством метода `Peek()`) в стек, вы будете всегда видеть объект, находящийся на его вершине; следовательно, первый вызов `Peek()` возвращает третий объект `Person`. После серии вызовов `Pop()` и `Peek()` стек, в конце концов, опустошается, после чего дополнительные вызовы `Peek()` и `Pop()` приводят к генерации системного исключения. Вот как выглядит вывод:

```
***** Fun with Generic Collections *****
First person is: Name: Lisa Simpson, Age: 9
Popped off Name: Lisa Simpson, Age: 9

First person is: Name: Marge Simpson, Age: 45
Popped off Name: Marge Simpson, Age: 45

First person item is: Name: Homer Simpson, Age: 47
Popped off Name: Homer Simpson, Age: 47

Error! Stack empty.
```

Работа с классом `Queue<T>`

Очереди — это контейнеры, которые обеспечивают доступ к элементам в манере “первый вошел — первый вышел” (FIFO). К сожалению, людям приходится сталкиваться с очередями практически ежедневно: в банке, в супермаркете, в кафе. Когда нужно смоделировать сценарий, в котором элементы обрабатываются в режиме FIFO, класс `Queue<T>` подходит наилучшим образом. Дополнительно к функциональности, предоставляемой поддерживаемыми интерфейсами, в `Queue` определены основные члены, перечисленные в табл. 9.6.

Таблица 9.6. Члены типа `Queue<T>`

Член <code>Queue<T></code>	Описание
<code>Dequeue()</code>	Удаляет и возвращает объект из начала <code>Queue<T></code>
<code>Enqueue()</code>	Добавляет объект в конец <code>Queue<T></code>
<code>Peek()</code>	Возвращает объект из начала <code>Queue<T></code> , не удаляя его

Теперь давайте посмотрим на описанные методы в работе. Можно снова задействовать класс `Person` и построить объект `Queue<T>`, эмулирующий очередь людей, которые ожидают заказанного кофе. Первым делом предположим, что имеется следующий статический метод:

```
static void GetCoffee(Person p)
{
    Console.WriteLine("{0} got coffee!", p.FirstName);
}
```

Кроме того, есть также дополнительный вспомогательный метод, внутри которого вызывается `GetCoffee()`:

```
static void UseGenericQueue()
{
    // Создать очередь из трех человек.
    Queue<Person> peopleQ = new Queue<Person>();
    peopleQ.Enqueue(new Person {FirstName= "Homer",
        LastName="Simpson", Age=47});
    peopleQ.Enqueue(new Person {FirstName= "Marge",
        LastName="Simpson", Age=45});
```

```

peopleQ.Enqueue(new Person {FirstName= "Lisa",
    LastName="Simpson", Age=9});

// Заглянуть, кто первый в очереди.
Console.WriteLine("{0} is first in line!", peopleQ.Peek().FirstName);

// Удалить всех из очереди.
GetCoffee(peopleQ.Dequeue());
GetCoffee(peopleQ.Dequeue());
GetCoffee(peopleQ.Dequeue());

// Попробовать извлечь кого-то из очереди снова.
try
{
    GetCoffee(peopleQ.Dequeue());
}
catch(InvalidOperationException e)
{
    Console.WriteLine("Error! {0}", e.Message); // Ошибка! Очередь пуста.
}
}

```

В коде мы вставляем три элемента в `Queue<T>` с применением метода `Enqueue()`. Вызов `Peek()` позволяет просматривать (но не удалять) первый элемент, находящийся в текущий момент внутри `Queue`. Наконец, вызов `Dequeue()` удаляет элемент из очереди и передает его на обработку вспомогательной функции `GetCoffee()`. Обратите внимание, что если попробовать удалить элемент из пустой очереди, то сгенерируется исключение времени выполнения. Ниже показан вывод, полученный в результате вызова метода `UseGenericQueue()`:

```

***** Fun with Generic Collections *****

Homer is first in line!
Homer got coffee!
Marge got coffee!
Lisa got coffee!
Error! Queue empty.

```

Работа с классом `SortedSet<T>`

Класс `SortedSet<T>` полезен тем, что при вставке или удалении элементов он автоматически обеспечивает сортировку элементов в наборе. Однако классу `SortedSet<T>` необходимо сообщить, каким образом должны сортироваться объекты, путем передачи его конструктору в качестве аргумента объекта, который реализует обобщенный интерфейс `IComparer<T>`.

Начнем с создания нового класса по имени `SortPeopleByAge`, реализующего интерфейс `IComparer<T>`, где `T` — тип `Person`. Вспомните, что в этом интерфейсе определен единственный метод по имени `Compare()`, в котором можно запрограммировать логику сравнения элементов. Вот простая реализация:

```

class SortPeopleByAge : IComparer<Person>
{
    public int Compare(Person firstPerson, Person secondPerson)
    {
        if (firstPerson?.Age > secondPerson?.Age)
        {
            return 1;
        }
    }
}

```

```

    if (firstPerson?.Age < secondPerson?.Age)
    {
        return -1;
    }
    return 0;
}
}

```

Теперь добавим в класс Program следующий новый метод, который понадобится вызывать внутри Main():

```

static void UseSortedSet()
{
    // Создать несколько объектов Person с разными значениями возраста.
    SortedSet<Person> setOfPeople =
        new SortedSet<Person>(new SortPeopleByAge())
    {
        new Person {FirstName= "Homer", LastName="Simpson", Age=47},
        new Person {FirstName= "Marge", LastName="Simpson", Age=45},
        new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
        new Person {FirstName= "Bart", LastName="Simpson", Age=8}
    };

    // Обратите внимание, что элементы отсортированы по возрасту.
    foreach (Person p in setOfPeople)
    {
        Console.WriteLine(p);
    }
    Console.WriteLine();

    // Добавить еще несколько объектов Person с разными значениями возраста.
    setOfPeople.Add(new Person { FirstName = "Saku",
                                   LastName = "Jones", Age = 1 });
    setOfPeople.Add(new Person { FirstName = "Mikko",
                                   LastName = "Jones", Age = 32 });

    // Элементы по-прежнему отсортированы по возрасту.
    foreach (Person p in setOfPeople)
    {
        Console.WriteLine(p);
    }
}

```

Запустив приложение, легко заметить, что список объектов будет всегда упорядочен на основе значения свойства Age независимо от порядка вставки и удаления объектов:

```
***** Fun with Generic Collections *****
```

```

Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47

Name: Saku Jones, Age: 1
Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Mikko Jones, Age: 32
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47

```


Работа с классом Dictionary<TKey, TValue>

Еще одной удобной обобщенной коллекцией является класс Dictionary<TKey, TValue>, позволяющий хранить любое количество объектов, на которые можно ссылаться через уникальный ключ. Таким образом, вместо получения элемента из List<T> с использованием числового идентификатора (например, “извлечь второй объект”) можно применять уникальный строковый ключ (скажем, “предоставить объект с ключом Homer”).

Как и другие классы коллекций, наполнять Dictionary<TKey, TValue> можно путем вызова обобщенного метода Add() вручную. Тем не менее, заполнять Dictionary<TKey, TValue> допускается также с использованием синтаксиса инициализации коллекций. Имейте в виду, что при наполнении данного объекта коллекции ключи должны быть уникальными. Если вы по ошибке укажете один и тот же ключ несколько раз, то получите исключение времени выполнения.

Взгляните на следующий метод, который заполняет Dictionary<K, V> разнообразными объектами. Обратите внимание, что при создании объекта Dictionary<TKey, TValue> в качестве аргументов конструктора передаются тип ключа (TKey) и тип внутренних объектов (TValue). Здесь для ключа указывается тип данных string (хотя это не обязательно; ключ может относиться к любому типу), а для значения — тип Person:

```
private static void UseDictionary()
{
    // Наполнить с помощью метода Add().
    Dictionary<string, Person> peopleA = new Dictionary<string, Person>();
    peopleA.Add("Homer", new Person { FirstName = "Homer",
                                     LastName = "Simpson", Age = 47 });
    peopleA.Add("Marge", new Person { FirstName = "Marge",
                                     LastName = "Simpson", Age = 45 });
    peopleA.Add("Lisa", new Person { FirstName = "Lisa",
                                    LastName = "Simpson", Age = 9 });

    // Получить элемент с ключом Homer.
    Person homer = peopleA["Homer"];
    Console.WriteLine(homer);

    // Наполнить с помощью синтаксиса инициализации.
    Dictionary<string, Person> peopleB = new Dictionary<string, Person>()
    {
        {"Homer", new Person { FirstName = "Homer", LastName = "Simpson", Age = 47 } },
        {"Marge", new Person { FirstName = "Marge", LastName = "Simpson", Age = 45 } },
        {"Lisa", new Person { FirstName = "Lisa", LastName = "Simpson", Age = 9 } }
    };

    // Получить элемент с ключом Lisa.
    Person lisa = peopleB["Lisa"];
    Console.WriteLine(lisa);
}
```

Наполнять Dictionary<TKey, TValue> также возможно с применением связанного синтаксиса инициализации, который является специфичным для контейнера данного типа (вполне ожидаемо называемый *инициализацией словаря*). Подобно синтаксису, который использовался при наполнении объекта peopleB в предыдущем примере, для объекта коллекции определяется область инициализации; однако можно также применять индексатор, чтобы указать ключ, и присвоить ему новый объект:

```
// Наполнить с помощью синтаксиса инициализации словаря.
Dictionary<string, Person> peopleC = new Dictionary<string, Person>()
{

```

```
["Homer"] = new Person { FirstName = "Homer", LastName = "Simpson", Age = 47 },
["Marge"] = new Person { FirstName = "Marge", LastName = "Simpson", Age = 45 },
["Lisa"] = new Person { FirstName = "Lisa", LastName = "Simpson", Age = 9 }
};
```

Исходный код. Проект FunWithGenericCollections доступен в подкаталоге Chapter_9.

Пространство имен System.Collections.ObjectModel

Теперь, когда вы понимаете, как работать с основными обобщенными классами, мы можем кратко рассмотреть дополнительное пространство имен, связанное с коллекциями — System.Collections.ObjectModel. Это относительно небольшое пространство имен, содержащее совсем мало классов. В табл. 9.7 документированы два класса, о которых вы должны быть обязательно осведомлены.

Таблица 9.7. Полезные классы из пространства имен System.Collections.ObjectModel

Класс	Описание
System.Collections.ObjectModel.ObservableCollection<T>	Представляет динамическую коллекцию данных, которая обеспечивает уведомление при добавлении и удалении элементов, а также при обновлении всего списка
ReadOnlyObservableCollection<T>	Представляет версию ObservableCollection<T>, допускающую только чтение

Класс ObservableCollection<T> удобен своей возможностью информировать внешние объекты, когда его содержимое каким-то образом изменяется (как и можно было догадаться, работа с ReadOnlyObservableCollection<T> похожа, но предусматривает только чтение).

Работа с классом ObservableCollection<T>

Создадим новый проект консольного приложения по имени FunWithObservableCollections и импортируем в первоначальный файл кода C# пространство имен System.Collections.ObjectModel. Во многих отношениях работа с ObservableCollection<T> идентична работе с List<T>, учитывая, что оба класса реализуют те же самые основные интерфейсы.

Уникальным класс ObservableCollection<T> делает тот факт, что он поддерживает событие по имени CollectionChanged. Указанное событие будет инициироваться каждый раз, когда вставляется новый элемент, удаляется (или перемещается) существующий элемент либо модифицируется вся коллекция целиком.

Подобно любому другому событию событие CollectionChanged определено в терминах делегата, которым в данном случае является NotifyCollectionChangedEventHandler. Этот делегат может вызывать любой метод, который принимает object в первом параметре и NotifyCollectionChangedEventArgs — во втором. Рассмотрим следующий метод Main(), который наполняет наблюдаемую коллекцию, содержащую объекты Person, и привязывается к событию CollectionChanged:

```

class Program
{
    static void Main(string[] args)
    {
        // Сделать коллекцию наблюдаемой и добавить в нее несколько объектов Person.
        ObservableCollection<Person> people = new ObservableCollection<Person>()
        {
            new Person{ FirstName = "Peter", LastName = "Murphy", Age = 52 },
            new Person{ FirstName = "Kevin", LastName = "Key", Age = 48 },
        };
        // Привязаться к событию CollectionChanged.
        people.CollectionChanged += people_CollectionChanged;
    }
    static void people_CollectionChanged(object sender,
        System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
    {
        throw new NotImplementedException();
    }
}

```

Входной параметр `NotifyCollectionChangedEventArgs` определяет два важных свойства, `OldItems` и `NewItems`, которые выдают список элементов, имеющихся в коллекции перед генерацией события, и список новых элементов, вовлеченных в изменение. Тем не менее, такие списки будут исследоваться только в подходящих обстоятельствах. Вспомните, что событие `CollectionChanged` инициируется, когда элементы добавляются, удаляются, перемещаются или сбрасываются. Чтобы выяснить, какое из упомянутых действий запустило событие, можно использовать свойство `Action` объекта `NotifyCollectionChangedEventArgs`. Свойство `Action` допускается проверять на равенство любому из членов перечисления `NotifyCollectionChangedAction`:

```

public enum NotifyCollectionChangedAction
{
    Add = 0,
    Remove = 1,
    Replace = 2,
    Move = 3,
    Reset = 4,
}

```

Ниже показана реализация обработчика событий `CollectionChanged`, который будет обходить старый и новый наборы, когда элемент вставляется или удаляется из имеющейся коллекции:

```

static void people_CollectionChanged(object sender,
    System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
    // Выяснить действие, которое привело к генерации события.
    Console.WriteLine("Action for this event: {0}", e.Action);

    // Было что-то удалено.
    if (e.Action ==
        System.Collections.Specialized.NotifyCollectionChangedAction.Remove)
    {
        Console.WriteLine("Here are the OLD items:"); // старые элементы
        foreach (Person p in e.OldItems)
        {
            Console.WriteLine(p.ToString());
        }
    }
}

```

```

    Console.WriteLine();
}
// Было что-то добавлено.
if (e.Action ==
    System.Collections.Specialized.NotifyCollectionChangedAction.Add)
{
    // Теперь вывести новые элементы, которые были вставлены.
    Console.WriteLine("Here are the NEW items:"); // новые элементы
    foreach (Person p in e.NewItems)
    {
        Console.WriteLine(p.ToString());
    }
}
}
}

```

Предполагая, что метод `Main()` был модифицирован для добавления и удаления элемента, вывод будет выглядеть так:

```

Action for this event: Add
Here are the NEW items:
Name: Fred Smith, Age: 32

Action for this event: Remove
Here are the OLD items:
Name: Peter Murphy, Age: 52

```

На этом исследование различных пространств имен, связанных с коллекциями, в библиотеках базовых классов .NET завершено. В конце главы будет также объясняться, как и для чего строить собственные обобщенные методы и обобщенные типы.

Исходный код. Проект `FunWithObservableCollections` доступен в подкаталоге `Chapter_9`.

Создание специальных обобщенных методов

Несмотря на то что большинство разработчиков обычно применяют обобщенные типы, имеющиеся в библиотеках базовых классов, существует также возможность построения собственных обобщенных методов и специальных обобщенных типов. Давайте посмотрим, как включать обобщения в собственные проекты. Первым делом мы построим обобщенный метод обмена. Начнем с создания нового проекта консольного приложения по имени `CustomGenericMethods`.

Построение специальных обобщенных методов представляет собой более развитую версию традиционной перегрузки методов. В главе 2 вы узнали, что перегрузка представляет собой действие по определению нескольких версий одного метода, которые отличаются друг от друга количеством или типами параметров.

Хотя перегрузка является полезным средством объектно-ориентированного языка, проблема заключается в том, что при этом довольно легко получить в итоге огромное количество методов, которые по существу делают одно и то же. Например, пусть необходимо создать методы, которые позволяют менять местами два фрагмента данных посредством простой процедуры. Для начала можно написать метод, оперирующий с целочисленными значениями:

```
// Поменять местами два целочисленных значения.
static void Swap(ref int a, ref int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

Пока все идет хорошо. Но теперь предположим, что нужно менять местами также и два объекта `Person`; действие потребует написания новой версии метода `Swap()`:

```
// Поменять местами два объекта Person.
static void Swap(ref Person a, ref Person b)
{
    Person temp = a;
    a = b;
    b = temp;
}
```

Без сомнения вам должно быть ясно, чем все закончится. Если также понадобится менять местами два значения с плавающей точкой, два объекта растровых изображений, два объекта автомобилей, два объекта кнопок или что-нибудь еще, то придется писать дополнительные методы, что в итоге превратится в настоящий кошмар при сопровождении. Можно было бы построить один (необобщенный) метод, оперирующий с параметрами типа `object`, но тогда возвратятся все проблемы, которые были описаны ранее в главе, т.е. упаковка, распаковка, отсутствие безопасности к типам, явное приведение и т.д.

Наличие группы перегруженных методов, отличающихся только входными аргументами — явный признак того, что обобщения могут облегчить ситуацию. Рассмотрим следующий обобщенный метод `Swap<T>`, который способен менять местами два значения типа `T`:

```
// Этот метод будет менять местами два элемента
// типа, указанного в параметре <T>.
static void Swap<T>(ref T a, ref T b)
{
    Console.WriteLine("You sent the Swap() method a {0}", typeof(T));
    T temp = a;
    a = b;
    b = temp;
}
```

Обратите внимание, что обобщенный метод определен за счет указания параметра типа после имени метода, но перед списком параметров. Здесь заявлено, что метод `Swap<T>()` способен оперировать на любых двух параметрах типа `<T>`. Для придания некоторой пикантности имя замещаемого типа выводится на консоль с использованием операции `typeof()` языка C#. Взгляните на показанный ниже метод `Main()`, который меняет местами целочисленные и строковые значения:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Custom Generic Methods *****\n");
    // Обмен двух целочисленных значений.
    int a = 10, b = 90;
    Console.WriteLine("Before swap: {0}, {1}", a, b);
    Swap<int>(ref a, ref b);
    Console.WriteLine("After swap: {0}, {1}", a, b);
    Console.WriteLine();
}
```

```
// Обмен двух строковых значений.
string s1 = "Hello", s2 = "There";
Console.WriteLine("Before swap: {0} {1}!", s1, s2);
Swap<string>(ref s1, ref s2);
Console.WriteLine("After swap: {0} {1}!", s1, s2);

Console.ReadLine();
}
```

Вот вывод:

```
***** Fun with Custom Generic Methods *****

Before swap: 10, 90
You sent the Swap() method a System.Int32
After swap: 90, 10

Before swap: Hello There!
You sent the Swap() method a System.String
After swap: There Hello!
```

Главное преимущество такого подхода в том, что придется сопровождать только одну версию `Swap<T>()`, однако она в состоянии работать с любыми двумя элементами заданного типа в безопасной к типам манере. Еще лучше то, что находящиеся в стеке элементы остаются в стеке, а расположенные в куче — соответственно в куче.

Выведение параметров типа

При вызове обобщенных методов вроде `Swap<T>()` параметр типа можно опускать, если (и только если) обобщенный метод принимает аргументы, поскольку компилятор в состоянии вывести параметр типа на основе параметров членов. Например, добавив в метод `Main()` следующий код, можно менять местами значения `System.Boolean`:

```
// Компилятор выведет тип System.Boolean.
bool b1 = true, b2 = false;
Console.WriteLine("Before swap: {0}, {1}", b1, b2);
Swap(ref b1, ref b2);
Console.WriteLine("After swap: {0}, {1}", b1, b2);
```

Несмотря на то что компилятор может определить параметр типа на основе типа данных, который применялся в объявлениях `b1` и `b2`, вы должны выработать привычку всегда указывать параметр типа явно:

```
Swap<string>(ref b1, ref b2);
```

Такой подход позволяет другим программистам понять, что метод на самом деле является обобщенным. Кроме того, выводение типов параметров работает только в случае, если обобщенный метод принимает, по крайней мере, один параметр. Например, пусть в классе `Program` определен обобщенный метод `DisplayBaseClass<T>()`:

```
static void DisplayBaseClass<T>()
{
    // BaseType - метод, используемый в рефлексии;
    // он будет описан в главе 15.
    Console.WriteLine("Base class of {0} is: {1}.",
        typeof(T), typeof(T).BaseType);
}
```

При его вызове потребуется указать параметр типа:

```
static void Main(string[] args)
{
```

```

...
// Если метод не принимает параметров,
// то должен быть указан параметр типа.
DisplayBaseClass<int>();
DisplayBaseClass<string>();

// Ошибка на этапе компиляции! Нет параметров?
// Должен быть предоставлен заполнитель!
// DisplayBaseClass();
Console.ReadLine();
}

```

В настоящее время обобщенные методы `Swap<T>` и `DisplayBaseClass<T>` определены в классе `Program` приложения. Разумеется, как и в случае любого метода, вы можете определить эти члены в отдельном типе класса (`MyGenericMethods`), если предпочитаете поступать подобным образом:

```

public static class MyGenericMethods
{
    public static void Swap<T>(ref T a, ref T b)
    {
        Console.WriteLine("You sent the Swap() method a {0}",
            typeof(T));
        T temp = a;
        a = b;
        b = temp;
    }

    public static void DisplayBaseClass<T>()
    {
        Console.WriteLine("Base class of {0} is: {1}.",
            typeof(T), typeof(T).BaseType);
    }
}

```

Статические методы `Swap<T>` и `DisplayBaseClass<T>` находятся в области действия нового статического класса, поэтому при вызове любого члена необходимо указывать имя типа:

```
MyGenericMethods.Swap<int>(ref a, ref b);
```

Конечно, методы не обязаны быть статическими. Если бы `Swap<T>` и `DisplayBaseClass<T>` были методами уровня экземпляра (и определялись в нестатическом классе), тогда понадобилось бы просто создать экземпляр `MyGenericMethods` и вызвать их с использованием объектной переменной:

```
MyGenericMethods c = new MyGenericMethods();
c.Swap<int>(ref a, ref b);
```

Исходный код. Проект `CustomGenericMethods` доступен в подкаталоге `Chapter_9`.

Создание специальных обобщенных структур и классов

Так как вы уже знаете, каким образом определять и вызывать обобщенные методы, наступило время уделить внимание конструированию обобщенной структуры (процесс построения обобщенного класса идентичен) в новом проекте консольного приложения

по имени `GenericPoint`. Предположим, что вы построили обобщенную структуру `Point`, которая поддерживает единственный параметр типа, определяющий внутреннее представление координат (x, y). Затем вызывающий код может создавать типы `Point<T>`:

```
// Точка с координатами типа int.
Point<int> p = new Point<int>(10, 10);

// Точка с координатами типа double.
Point<double> p2 = new Point<double>(5.4, 3.3);
```

Вот полное определение структуры `Point<T>`:

```
// Обобщенная структура Point.
public struct Point<T>
{
    // Обобщенные данные состояния.
    private T xPos;
    private T yPos;

    // Обобщенный конструктор.
    public Point(T xVal, T yVal)
    {
        xPos = xVal;
        yPos = yVal;
    }

    // Обобщенные свойства.
    public T X
    {
        get { return xPos; }
        set { xPos = value; }
    }

    public T Y
    {
        get { return yPos; }
        set { yPos = value; }
    }

    public override string ToString() => $"[{xPos}, {yPos}]";

    // Сбросить поля в стандартные значения
    // для заданного параметра типа.
    public void ResetPoint()
    {
        xPos = default(T);
        yPos = default(T);
    }
}
```

Ключевое слово `default` в обобщенном коде

Как видите, структура `Point<T>` задействует параметр типа в определениях полей данных, в аргументах конструктора и в определениях свойств. Обратите внимание, что в дополнение к переопределению метода `ToString()` структура `Point<T>` определяет метод по имени `ResetPoint()`, в котором применяется не встречавшийся ранее новый синтаксис:

```
// Ключевое слово default в языке C# перегружено.
// При использовании с обобщениями оно представляет
// стандартное значение для параметра типа.
```



```
public void ResetPoint()
{
    X = default(T);
    Y = default(T);
}
```

С появлением обобщений ключевое слово `default` получило двойную идентичность. Вдобавок к использованию внутри конструкции `switch` оно также может применяться для установки параметра типа в стандартное значение. Это очень удобно, т.к. действительные типы, подставляемые вместо заполнителей, обобщенному типу заранее не известны, а потому он не может безопасно предполагать, какими будут стандартные значения. Параметры типа подчиняются следующим правилам:

- числовые типы имеют стандартное значение 0;
- ссылочные типы имеют стандартное значение `null`;
- поля структур устанавливаются в 0 (для типов значений) или в `null` (для ссылочных типов).

Для `Point<T>` устанавливать значения `X` и `Y` в 0 можно напрямую, поскольку вполне безопасно предполагать, что вызывающий код будет предоставлять только числовые данные. Однако использование синтаксиса `default(T)` в целом повышает гибкость обобщенного типа. Теперь методы типа `Point<T>` можно применять так:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Generic Structures *****\n");
    // Точка с координатами типа int.
    Point<int> p = new Point<int>(10, 10);
    Console.WriteLine("p.ToString()={0}", p.ToString());
    p.ResetPoint();
    Console.WriteLine("p.ToString()={0}", p.ToString());
    Console.WriteLine();
    // Точка с координатами типа double.
    Point<double> p2 = new Point<double>(5.4, 3.3);
    Console.WriteLine("p2.ToString()={0}", p2.ToString());
    p2.ResetPoint();
    Console.WriteLine("p2.ToString()={0}", p2.ToString());
    Console.ReadLine();
}
```

Ниже приведен вывод:

```
***** Fun with Generic Structures *****
p.ToString()=[10, 10]
p.ToString()=[0, 0]
p2.ToString()=[5.4, 3.3]
p2.ToString()=[0, 0]
```

Исходный код. Проект `GenericPoint` доступен в подкаталоге `Chapter_9`.

Ограничение параметров типа

Как объяснялось в настоящей главе, любой обобщенный элемент имеет, по крайней мере, один параметр типа, который необходимо указывать во время взаимодействия с данным обобщенным типом или его членом. Уже одно это обстоятельство позволяет

строить код, безопасный к типам; тем не менее, платформа .NET позволяет использовать ключевое слово `where` для определения особых требований к отдельному параметру типа.

С помощью ключевого слова `this` можно добавлять набор ограничений к конкретному параметру типа, которые компилятор C# проверит на этапе компиляции. В частности, параметр типа можно ограничить, как описано в табл. 9.8.

Таблица 9.8. Возможные ограничения для параметров типа в обобщениях

Ограничение	Описание
<code>where T : struct</code>	Параметр типа <code><T></code> должен иметь класс <code>System.ValueType</code> в своей цепочке наследования (т.е. <code><T></code> должен быть структурой)
<code>where T : class</code>	Параметр типа <code><T></code> не должен иметь класс <code>System.ValueType</code> в своей цепочке наследования (т.е. <code><T></code> должен быть ссылочным типом)
<code>where T : new()</code>	Параметр типа <code><T></code> должен иметь стандартный конструктор. Это полезно, если обобщенный тип должен создавать экземпляры параметра типа, т.к. предугадать формат специальных конструкторов невозможно. Обратите внимание, что в типе с множеством ограничений данное ограничение должно указываться последним
<code>where T : ИмяБазовогоКласса</code>	Параметр типа <code><T></code> должен быть производным от класса, указанного как <i>ИмяБазовогоКласса</i>
<code>where T : ИмяИнтерфейса</code>	Параметр типа <code><T></code> должен реализовывать интерфейс, указанный как <i>ИмяИнтерфейса</i> . Можно задавать список из нескольких интерфейсов, разделяя их запятыми

Возможно, применять ключевое слово `where` в проектах C# вам никогда и не придется, если только не требуется строить какие-то исключительно безопасные к типам специальные коллекции. Невзирая на сказанное, в следующих нескольких примерах (частичного) кода демонстрируется работа с ключевым словом `where`.

Примеры использования ключевого слова `where`

Начнем с предположения о том, что создан специальный обобщенный класс, и необходимо гарантировать наличие в параметре типа стандартного конструктора. Это может быть полезно, когда специальный обобщенный класс должен создавать экземпляры типа `T`, потому что стандартный конструктор является единственным конструктором, потенциально общим для всех типов. Кроме того, подобное ограничение `T` позволяет получить проверку на этапе компиляции; если `T` — ссылочный тип, то программист будет помнить о повторном определении стандартного конструктора в объявлении класса (как вам уже известно, в случае определения собственного конструктора класса стандартный конструктор из него удаляется).

```
// Класс MyGenericClass является производным от object, в то время как
// содержащиеся в нем элементы должны иметь стандартный конструктор.
public class MyGenericClass<T> where T : new()
{
    ...
}
```

Обратите внимание, что конструкция `where` указывает параметр типа, подлежащий ограничению, за которым следует операция двоеточия. После операции двоеточия перечисляются все возможные ограничения (в данном случае — стандартный конструктор). Вот еще один пример:

```
// Класс MyGenericClass является производным от object, в то время как
// содержащиеся в нем элементы должны относиться к классу, реализующему
// интерфейс IDrawable, и поддерживать стандартный конструктор.
public class MyGenericClass<T> where T : class, IDrawable, new()
{
    ...
}
```

Здесь к типу `T` предъявляются три требования. Во-первых, он должен быть ссылочным типом (не структурой), как помечено лексемой `class`. Во-вторых, `T` должен реализовывать интерфейс `IDrawable`. В-третьих, тип `T` также должен иметь стандартный конструктор. Множество ограничений перечисляются в виде списка с разделителями-запятыми, но имейте в виду, что ограничение `new()` должно указываться последним! Таким образом, представленный далее код не скомпилируется:

```
// Ошибка! Ограничение new() должно быть последним в списке!
public class MyGenericClass<T> where T : new(), class, IDrawable
{
    ...
}
```

При создании класса обобщенной коллекции с несколькими параметрами типа можно указывать уникальный набор ограничений для каждого параметра, применяя отдельные конструкции `where`:

```
// Тип <K> должен расширять SomeBaseClass и иметь стандартный конструктор,
// в то время как тип <T> должен быть структурой и реализовывать
// обобщенный интерфейс IComparable.
public class MyGenericClass<K, T> where K : SomeBaseClass, new()
    where T : struct, IComparable<T>
{
    ...
}
```

Необходимость построения полного специального обобщенного класса коллекции возникает редко; однако ключевое слово `where` допускается использовать также в обобщенных методах. Например, если нужно гарантировать, что метод `Swap<T>()` может работать только со структурами, измените его код следующим образом:

```
// Этот метод меняет местами любые структуры, но не классы.
static void Swap<T>(ref T a, ref T b) where T : struct
{
    ...
}
```

Обратите внимание, что если ограничить метод `Swap<T>()` в подобной манере, то менять местами объекты `string` (как было показано в коде примера) больше не удастся, т.к. `string` является ссылочным типом.

Отсутствие ограничений операций

В завершение главы следует упомянуть об еще одном факте, связанном с обобщенными методами и ограничениями. При создании обобщенных методов может оказаться неожиданным получение ошибки на этапе компиляции в случае применения к параметрам типа любых операций C# (+, -, *, == и т.д.). Например, только вообразите, насколько полезным оказался бы класс, способный выполнять сложение, вычитание, умножение и деление с обобщенными типами:

```
// Ошибка на этапе компиляции! Невозможно
// применять операции к параметрам типа!
public class BasicMath<T>
{
    public T Add(T arg1, T arg2)
    { return arg1 + arg2; }
    public T Subtract(T arg1, T arg2)
    { return arg1 - arg2; }
    public T Multiply(T arg1, T arg2)
    { return arg1 * arg2; }
    public T Divide(T arg1, T arg2)
    { return arg1 / arg2; }
}
```

К сожалению, приведенный выше класс `BasicMath<T>` не скомпилируется. Хотя это может показаться крупным недостатком, следует вспомнить, что обобщения имеют общий характер. Конечно, числовые данные прекрасно работают с двоичными операциями C#. Тем не менее, если аргумент `<T>` является специальным классом или структурой, то компилятор мог бы предположить, что он поддерживает операции +, -, * и /. В идеале язык C# позволял бы ограничивать обобщенный тип поддерживаемыми операциями, как показано ниже:

```
// Только в целях иллюстрации!
public class BasicMath<T> where T : operator +, operator -,
    operator *, operator /
{
    public T Add(T arg1, T arg2)
    { return arg1 + arg2; }
    public T Subtract(T arg1, T arg2)
    { return arg1 - arg2; }
    public T Multiply(T arg1, T arg2)
    { return arg1 * arg2; }
    public T Divide(T arg1, T arg2)
    { return arg1 / arg2; }
}
```

Увы, ограничения операций в текущей версии C# не поддерживаются. Однако достичь желаемого результата можно (хотя и с дополнительными усилиями) путем определения интерфейса, который поддерживает такие операции (интерфейсы C# могут определять операции!), и указания ограничения интерфейса для обобщенного класса. В любом случае первоначальный обзор построения специальных обобщенных типов завершен. В главе 10 мы вновь обратимся к теме обобщений, когда будем исследовать тип делегата .NET.

Резюме

Глава начиналась с рассмотрения необобщенных типов коллекций в пространствах имен `System.Collections` и `System.Collections.Specialized`, включая разнообразные проблемы, которые связаны со многими необобщенными контейнерами, в том числе отсутствие безопасности к типам и накладные расходы времени выполнения в форме операций упаковки и распаковки. Как упоминалось, именно по этим причинам в современных приложениях .NET будут использоваться классы обобщенных коллекций из пространств имен `System.Collections.Generic` и `System.Collections.ObjectModel`.

Вы видели, что обобщенный элемент позволяет указывать заполнители (параметры типа), которые задаются во время создания объекта (или вызова в случае обобщенных методов). Хотя чаще всего вы будете просто применять обобщенные типы, предоставляемые библиотеками базовых классов .NET, также имеется возможность создавать собственные обобщенные типы (и обобщенные методы). При этом допускается указывать любое количество ограничений (с использованием ключевого слова `where`) для повышения уровня безопасности к типам и гарантии того, что операции выполняются над типами *известного размера*, демонстрируя наличие определенных базовых возможностей.

В качестве финального замечания: не забывайте, что обобщения можно обнаружить во многих местах внутри библиотек базовых классов .NET. Здесь мы сосредоточились конкретно на обобщенных коллекциях. Тем не менее, по мере проработки материала оставшихся глав (и освоения платформы) вы наверняка найдете обобщенные классы, структуры и делегаты в том или ином пространстве имен. Кроме того, будьте готовы столкнуться с обобщенными членами в необобщенном классе!

глава 10

Делегаты, события и лямбда-выражения

Вплоть до настоящего момента в большинстве разработанных приложений к методу `Main()` добавлялись разнообразные порции кода, тем или иным способом отправляющие запросы к заданному объекту. Однако многие приложения требуют, чтобы объект имел возможность обращаться *обратно* к сущности, которая его создала, используя механизм обратного вызова. Хотя механизмы обратного вызова могут применяться в любом приложении, они особенно важны в графических пользовательских интерфейсах, где элементы управления (такие как кнопки) нуждаются в вызове внешних методов при надлежащих обстоятельствах (когда произведен щелчок на кнопке, курсор мыши наведен на поверхность кнопки и т.д.).

В рамках платформы .NET предпочтительным средством определения и реагирования на обратные вызовы в приложении является тип *делегата*. По существу тип делегата .NET — это безопасный к типам объект, “указывающий” на метод или список методов, которые могут быть вызваны позднее. Тем не менее, в отличие от традиционного указателя на функцию C++ делегаты .NET представляют собой классы, которые обладают встроенной поддержкой группового и асинхронного вызова методов.

В главе вы узнаете, каким образом создавать и манипулировать типами делегатов, а также использовать ключевое слово `event` языка C#, которое облегчает работу с типами делегатов. По ходу дела вы также изучите несколько языковых средств C#, ориентированных на делегаты и события, в том числе анонимные методы и групповые преобразования методов.

Глава завершается исследованием *лямбда-выражений*. С помощью лямбда-операции C# (`=>`) можно указывать блок операторов кода (и подлежащие передаче им параметры) везде, где требуется строго типизированный делегат. Как будет показано, лямбда-выражение — не более чем замаскированный анонимный метод и является упрощенным подходом к работе с делегатами. Вдобавок та же самая операция (начиная с версии .NET 4.6) может применяться для реализации метода или свойства, содержащего единственный оператор, посредством лаконичного синтаксиса.

Понятие типа делегата .NET

Прежде чем формально определить делегаты .NET, давайте оглянемся немного назад. Исторически сложилось так, что в API-интерфейсе Windows часто использовались указатели на функции в стиле C для создания сущностей под названием *функции обратного вызова* или просто *обратные вызовы*. С помощью обратных вызовов программисты могли конфигурировать одну функцию так, чтобы она обращалась к другой функции в

приложении (т.е. делала обратный вызов). С применением такого подхода разработчики Windows-приложений имели возможность обрабатывать щелчки на кнопках, перемещение курсора мыши, выбор пунктов меню и общие двусторонние коммуникации между двумя сущностями в памяти.

В .NET Framework обратные вызовы выполняются в безопасной к типам объектно-ориентированной манере с использованием *делегатов*. В сущности, делегат — это безопасный в отношении типов объект, указывающий на другой метод или возможно на список методов приложения, которые могут быть вызваны в более позднее время. В частности, делегат поддерживает три важных порции информации:

- адрес метода, к которому он делает вызовы;
- аргументы (если есть) вызываемого метода;
- возвращаемое значение (если есть) вызываемого метода.

На заметку! Делегаты .NET могут указывать либо на статические методы, либо на методы экземпляра.

После того как делегат создан и снабжен необходимой информацией, он может во время выполнения динамически вызывать метод или методы, на которые указывает. Каждый делегат в .NET Framework (включая ваши специальные делегаты) автоматически наделяется способностью вызывать свои методы *синхронно* или *асинхронно*. Данный факт значительно упрощает задачи программирования, поскольку метод можно вызывать во вторичном потоке выполнения без ручного создания и управления объектом Thread.

На заметку! Вы ознакомитесь с асинхронным поведением типов делегатов во время исследования многопоточности и асинхронных вызовов в главе 19. Здесь же мы будем касаться только синхронных аспектов типа делегата.

Определение типа делегата в C#

Для определения типа делегата в языке C# применяется ключевое слово `delegate`. Имя типа делегата может быть любым желаемым. Однако сигнатура определяемого делегата должна совпадать с сигнатурой метода или методов, на которые он будет указывать. Например, приведенный ниже тип делегата (по имени `BinaryOp`) может указывать на любой метод, который возвращает целое число и принимает два целых числа в качестве входных параметров (позже в главе вы самостоятельно постройте такой делегат, а пока он показан лишь кратко):

```
// Этот делегат может указывать на любой метод, который принимает
// два целочисленных значения и возвращает целочисленное значение.
public delegate int BinaryOp(int x, int y);
```

Когда компилятор C# обрабатывает тип делегата, он автоматически генерирует запечатанный (sealed) класс, производный от `System.MulticastDelegate`. Этот класс (в сочетании со своим базовым классом `System.Delegate`) предоставляет необходимую инфраструктуру для делегата, которая позволяет хранить список методов, подлежащих вызову в будущем. Например, если вы изучите делегат `BinaryOp` с помощью утилиты `ildasm.exe`, то обнаружите класс, показанный на рис. 10.1 (если хотите, можете построить полный пример).

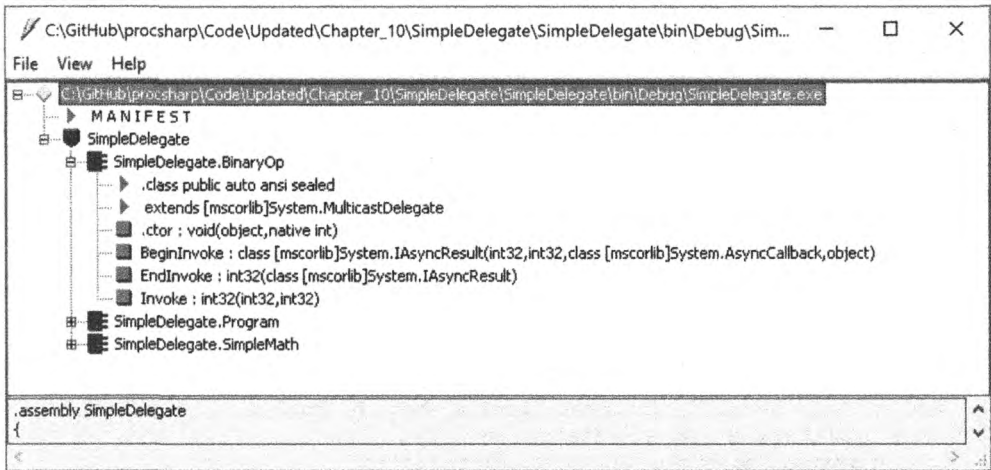


Рис. 10.1. Ключевое слово `delegate` языка C# представляет запечатанный класс, производный от `System.MulticastDelegate`

Как видите, сгенерированный компилятором класс `BinaryOp` определяет три открытых метода. Вероятно, главным из них является `Invoke()`, т.к. он используется для вызова каждого метода, поддерживаемого объектом делегата, в синхронной манере; это означает, что вызывающий код должен ожидать завершения вызова, прежде чем продолжить свою работу. Довольно странно, но синхронный метод `Invoke()` может не нуждаться в явном вызове внутри вашего кода C#. Вскоре будет показано, что `Invoke()` вызывается «за кулисами», когда вы применяете соответствующий синтаксис C#.

Методы `BeginInvoke()` и `EndInvoke()` обеспечивают возможность вызова текущего метода асинхронным образом в отдельном потоке выполнения. Если у вас есть опыт многопоточной разработки, тогда вам известно, что одна из наиболее распространенных причин, вынуждающих разработчиков создавать вторичные потоки выполнения, связана с необходимостью вызова методов, которые требуют определенного времени на свое завершение. Хотя в библиотеках базовых классов .NET поставляется несколько пространств имен, предназначенных для многопоточного и параллельного программирования, делегаты предлагают такую функциональность в готовом виде.

Так благодаря чему же компилятор знает, как определять методы `Invoke()`, `BeginInvoke()` и `EndInvoke()`? Для понимания процесса ниже приведен код сгенерированного компилятором класса `BinaryOp` (полужирным курсивом выделены элементы, указанные в определении типа делегата):

```
sealed class BinaryOp : System.MulticastDelegate
{
    public int Invoke(int x, int y);
    public IAsyncResult BeginInvoke(int x, int y,
        AsyncCallback cb, object state);
    public int EndInvoke(IAsyncResult result);
}
```

Первым делом обратите внимание, что параметры и возвращаемый тип для метода `Invoke()` в точности соответствуют определению делегата `BinaryOp`. Начальные параметры для членов `BeginInvoke()` (в данном случае два целых числа) также основаны на делегате `BinaryOp`; тем не менее, `BeginInvoke()` всегда будет предоставлять

два финальных параметра (типа `AsyncCallback` и `object`), которые используются для облегчения асинхронного вызова методов. Наконец, возвращаемый тип `EndInvoke()` идентичен исходному объявлению делегата и будет всегда принимать единственный параметр — объект, реализующий интерфейс `IAsyncResult`.

Давайте рассмотрим еще один пример. Предположим, что определен тип делегата, который может указывать на любой метод, возвращающий значение `string` и принимающий три входных параметра типа `System.Boolean`:

```
public delegate string MyDelegate(bool a, bool b, bool c);
```

На этот раз сгенерированный компилятором класс можно представить так:

```
sealed class MyDelegate : System.MulticastDelegate
{
    public string Invoke(bool a, bool b, bool c);
    public IAsyncResult BeginInvoke(bool a, bool b, bool c,
        AsyncCallback cb, object state);
    public string EndInvoke(IAsyncResult result);
}
```

Делегаты могут также “указывать” на методы, которые содержат любое количество параметров `out` и `ref` (а также параметры типа массивов, помеченные с помощью ключевого слова `params`). Например, пусть имеется следующий тип делегата:

```
public delegate string MyOtherDelegate(out bool a, ref bool b, int c);
```

Сигнатуры методов `Invoke()` и `BeginInvoke()` выглядят вполне ожидаемо; однако взгляните на метод `EndInvoke()`, который теперь включает набор аргументов `out/ref`, определенных типом делегата:

```
public sealed class MyOtherDelegate : System.MulticastDelegate
{
    public string Invoke(out bool a, ref bool b, int c);
    public IAsyncResult BeginInvoke(out bool a, ref bool b, int c,
        AsyncCallback cb, object state);
    public string EndInvoke(out bool a, ref bool b, IAsyncResult result);
}
```

Чтобы подвести итог: определение типа делегата C# дает в результате запечатанный класс с тремя сгенерированными компилятором методами, в которых типы параметров и возвращаемые типы основаны на объявлении делегата. Базовый шаблон может быть приближенно описан с помощью следующего псевдокода:

```
// Это всего лишь псевдокод!
public sealed class ИмяДелегата : System.MulticastDelegate
{
    public возвращаемоеЗначениеДелегата
        Invoke(всеВходныеRefиOutПараметрыДелегата);

    public IAsyncResult BeginInvoke(всеВходныеRefиOutПараметрыДелегата,
        AsyncCallback cb, object state);

    public возвращаемоеЗначениеДелегата EndInvoke(всеRefиOutПараметрыДелегата,
        IAsyncResult result);
}
```

Базовые классы `System.MulticastDelegate` и `System.Delegate`

Итак, когда вы строите тип с применением ключевого слова `delegate`, то неявно объявляете тип класса, производного от `System.MulticastDelegate`. Данный класс предоставляет своим наследникам доступ к списку, который содержит адреса методов, поддерживаемых типом делегата, а также несколько дополнительных методов (и перегруженных операций) для взаимодействия со списком вызовов. Ниже приведены избранные методы класса `System.MulticastDelegate`:

```
public abstract class MulticastDelegate : Delegate
{
    // Возвращает список методов, на которые "указывает" делегат.
    public sealed override Delegate[] GetInvocationList();

    // Перегруженные операции.
    public static bool operator ==(MulticastDelegate d1, MulticastDelegate d2);
    public static bool operator !=(MulticastDelegate d1, MulticastDelegate d2);

    // Используются внутренне для управления списком методов,
    // поддерживаемых делегатом.
    private IntPtr _invocationCount;
    private object _invocationList;
}
```

Класс `System.MulticastDelegate` получает дополнительную функциональность от своего родительского класса `System.Delegate`. Вот фрагмент его определения:

```
public abstract class Delegate : ICloneable, ISerializable
{
    // Методы для взаимодействия со списком функций.
    public static Delegate Combine(params Delegate[] delegates);
    public static Delegate Combine(Delegate a, Delegate b);
    public static Delegate Remove(Delegate source, Delegate value);
    public static Delegate RemoveAll(Delegate source, Delegate value);

    // Перегруженные операции.
    public static bool operator ==(Delegate d1, Delegate d2);
    public static bool operator !=(Delegate d1, Delegate d2);

    // Свойства, открывающие доступ к цели делегата.
    public MethodInfo Method { get; }
    public object Target { get; }
}
```

Имейте в виду, что вы никогда не сможете напрямую наследовать от таких базовых классов в своем коде (попытка наследования приводит к ошибке на этапе компиляции). Тем не менее, когда вы используете ключевое слово `delegate`, то тем самым неявно создаете класс, который "является" `MulticastDelegate`. В табл. 10.1 описаны основные члены, общие для всех типов делегатов.

**Таблица 10.1. Избранные члены классов `System.MulticastDelegate/`
`System.Delegate`**

Член	Назначение
Method	Это свойство возвращает объект <code>System.Reflection.Method</code> , который представляет детали статического метода, поддерживаемого делегатом
Target	Если подлежащий вызову метод определен на уровне объектов (т.е. он нестатический), тогда это свойство возвращает объект, который представляет метод, поддерживаемый делегатом. Если возвращенное <code>Target</code> значение равно <code>null</code> , то подлежащий вызову метод является статическим
Combine()	Этот статический метод добавляет метод в список, поддерживаемый делегатом. В языке C# данный метод вызывается с применением перегруженной операции <code>+=</code> в качестве сокращенной записи
GetInvokationList()	Этот метод возвращает массив объектов <code>System.Delegate</code> , каждый из которых представляет определенный метод, доступный для вызова
Remove() RemoveAll()	Эти статические методы удаляют метод (или все методы) из списка вызовов делегата. В языке C# метод <code>Remove()</code> может быть вызван косвенно с использованием перегруженной операции <code>--</code>

Пример простейшего делегата

На первый взгляд делегаты могут показаться несколько запутанными. Рассмотрим для начала простой проект консольного приложения (по имени `SimpleDelegate`), в котором применяется определенный ранее тип делегата `BinaryOp`. Ниже показан полный код с последующим анализом:

```
namespace SimpleDelegate
{
    // Этот делегат может указывать на любой метод, который принимает
    // два целочисленных значения и возвращает целочисленное значение.
    public delegate int BinaryOp(int x, int y);

    // Этот класс содержит методы, на которые
    // будет указывать BinaryOp.
    public class SimpleMath
    {
        public static int Add(int x, int y) => x + y;
        public static int Subtract(int x, int y) => x - y;
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Simple Delegate Example *****\n");
            // Создать объект делегата BinaryOp, который
            // "указывает" на SimpleMath.Add().
            BinaryOp b = new BinaryOp(SimpleMath.Add);
            // Вызвать метод Add() косвенно с использованием объекта делегата.
            Console.WriteLine("10 + 10 is {0}", b(10, 10));
            Console.ReadLine();
        }
    }
}
```

И снова обратите внимание на формат объявления типа делегата `BinaryOp`: он определяет, что объекты делегата `BinaryOp` могут указывать на любой метод, принимающий два целочисленных значения и возвращающий целочисленное значение (действительное имя метода, на который он указывает, к делу не относится). Здесь мы создали класс по имени `SimpleMath`, определяющий два статических метода, которые соответствуют шаблону, определяемому делегатом `BinaryOp`.

Когда вы хотите присвоить целевой метод заданному объекту делегата, просто передайте имя нужного метода конструктору делегата:

```
// Создать объект делегата BinaryOp, который
// "указывает" на SimpleMath.Add().
BinaryOp b = new BinaryOp(SimpleMath.Add);
```

На данной стадии метод, на который указывает делегат, можно вызывать с использованием синтаксиса, выглядящего подобным прямому вызову функции:

```
// В действительности здесь вызывается метод Invoke()!
Console.WriteLine("10 + 10 is {0}", b(10, 10));
```

“За кулисами” исполняющая среда на самом деле вызывает сгенерированный компилятором метод `Invoke()` на вашем производном от `MulticastDelegate` классе. В этом можно удостовериться, открыв сборку в утилите `ildasm.exe` и просмотрев код CIL внутри метода `Main()`:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    ...
    callvirt instance int32 SimpleDelegate.BinaryOp::Invoke(int32, int32)
}
```

Язык C# вовсе не требует явного вызова метода `Invoke()` внутри вашего кода. Поскольку `BinaryOp` может указывать на методы, которые принимают два аргумента, следующий оператор кода также допустим:

```
Console.WriteLine("10 + 10 is {0}", b.Invoke(10, 10));
```

Вспомните, что делегаты .NET безопасны в отношении типов. Следовательно, если вы попытаетесь передать делегату метод, который не соответствует его шаблону, то получите ошибку на этапе компиляции. В целях иллюстрации предположим, что в классе `SimpleMath` теперь определен дополнительный метод по имени `SquareNumber()`, принимающий единственный целочисленный аргумент:

```
public class SimpleMath
{
    public static int SquareNumber(int a) => a * a;
}
```

Учитывая, что делегат `BinaryOp` может указывать *только* на методы, которые принимают два целочисленных значения и возвращают целочисленное значение, представленный ниже код некорректен и приведет к ошибке на этапе компиляции:

```
// Ошибка на этапе компиляции! Метод не соответствует шаблону делегата!
BinaryOp b2 = new BinaryOp(SimpleMath.SquareNumber);
```

Исследование объекта делегата

Давайте усложним текущий пример, создав в классе `Program` статический метод (по имени `DisplayDelegateInfo()`). Он будет выводить на консоль имена методов, поддерживаемых объектом делегата, а также имя класса, определяющего метод. Для

этого организуется итерация по массиву `System.Delegate`, возвращенному методом `GetInvocationList()`, с обращением к свойствам `Target` и `Method` каждого объекта:

```
static void DisplayDelegateInfo(Delgate delObj)
{
    // Вывести имена всех членов в списке вызовов делегата.
    foreach (Delegate d in delObj.GetInvocationList())
    {
        Console.WriteLine("Method Name: {0}", d.Method); // имя метода
        Console.WriteLine("Type Name: {0}", d.Target);    // имя типа
    }
}
```

Предполагая, что в метод `Main()` добавлен вызов нового вспомогательного метода:

```
BinaryOp b = new BinaryOp(SimpleMath.Add);
DisplayDelegateInfo(b);
```

вывод приложения будет таким:

```
***** Simple Delegate Example *****
Method Name: Int32 Add(Int32, Int32)
Type Name:
10 + 10 is 20
```

Обратите внимание, что при обращении к свойству `Target` имя целевого класса (`SimpleMath`) в настоящий момент не отображается. Причина в том, что делегат `BinaryOp` указывает на *статический метод*, и потому объект для ссылки попросту отсутствует! Однако если сделать методы `Add()` и `Subtract()` нестатическими (удалив ключевое слово `static` из их объявлений), тогда можно будет создавать экземпляр класса `SimpleMath` и указывать методы для вызова с применением ссылки на объект:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Simple Delegate Example *****\n");
    // Делегаты .NET могут также указывать на методы экземпляра.
    SimpleMath m = new SimpleMath();
    BinaryOp b = new BinaryOp(m.Add);
    // Вывести сведения об объекте.
    DisplayDelegateInfo(b);

    Console.WriteLine("10 + 10 is {0}", b(10, 10));
    Console.ReadLine();
}
```

В данном случае вывод будет выглядеть следующим образом:

```
***** Simple Delegate Example *****
Method Name: Int32 Add(Int32, Int32)
Type Name: SimpleDelegate.SimpleMath
10 + 10 is 20
```

Отправка уведомлений о состоянии объекта с использованием делегатов

Очевидно, что предыдущий пример SimpleDelegate был чисто иллюстративным по своей природе, т.к. нет особых причин создавать делегат просто для того, чтобы сложить два числа. Рассмотрим более реалистичный пример, в котором делегаты применяются для определения класса Car, обладающего способностью информировать внешние сущности о текущем состоянии двигателя. В таком случае нужно выполнить перечисленные ниже действия.

1. Определить новый тип делегата, который будет использоваться для отправки уведомлений вызывающему коду.
2. Объявить переменную-член этого типа делегата в классе Car.
3. Создать в классе Car вспомогательную функцию, которая позволяет вызывающему коду указывать метод для обратного вызова.
4. Реализовать метод Accelerate() для обращения к списку вызовов делегата в подходящих обстоятельствах.

Для начала создадим новый проект консольного приложения по имени CarDelegate. Определим в нем новый класс Car, начальный код которого показан ниже:

```
public class Car
{
    // Данные состояния.
    public int CurrentSpeed { get; set; }
    public int MaxSpeed { get; set; } = 100;
    public string PetName { get; set; }

    // Исправен ли автомобиль?
    private bool carIsDead;

    // Конструкторы класса.
    public Car() {}
    public Car(string name, int maxSp, int currSp)
    {
        CurrentSpeed = currSp;
        MaxSpeed = maxSp;
        PetName = name;
    }
}
```

А теперь модифицируем его, выполнив первые три действия из числа указанных выше:

```
public class Car
{
    ...
    // 1. Определить тип делегата.
    public delegate void CarEngineHandler(string msgForCaller);
    // 2. Определить переменную-член этого типа делегата.
    private CarEngineHandler listOfHandlers;
    // 3. Добавить регистрационную функцию для вызывающего кода.
    public void RegisterWithCarEngine(CarEngineHandler methodToCall)
    {
        listOfHandlers = methodToCall;
    }
}
```

В приведенном примере обратите внимание на то, что типы делегатов определяются прямо внутри области действия класса `Car`; безусловно, это необязательно, но помогает закрепить идею о том, что делегат естественным образом работает с таким отдельным классом. Тип делегата `CarEngineHandler` может указывать на любой метод, который принимает значение `string` как параметр и имеет `void` в качестве возвращаемого типа.

Кроме того, была объявлена закрытая переменная-член делегата (`listOfHandlers`) и вспомогательная функция (`RegisterWithCarEngine()`), которая позволяет вызывающему коду добавлять метод в список вызовов делегата.

На заметку! Строго говоря, переменную-член типа делегата можно было бы определить как `public`, избежав тем самым необходимости в создании дополнительных методов регистрации. Тем не менее, за счет определения этой переменной-члена типа делегата как `private` усиливается инкапсуляция и обеспечивается решение, более безопасное в отношении типов. Позже в главе при рассмотрении ключевого слова `event` языка C# мы еще вернемся к анализу рисков объявления переменных-членов с типами делегатов как `public`.

Теперь необходимо создать метод `Accelerate()`. Вспомните, что цель в том, чтобы позволить объекту `Car` отправлять связанные с двигателем сообщения любому подписавшемуся прослушивателю. Вот необходимое обновление:

```
// 4. Реализовать метод Accelerate() для обращения к списку
// вызовов делегата в подходящих обстоятельствах.
public void Accelerate(int delta)
{
    // Если этот автомобиль сломан, то отправить сообщение об этом.
    if (carIsDead)
    {
        if (listOfHandlers != null)
            listOfHandlers("Sorry, this car is dead...");
    }
    else
    {
        CurrentSpeed += delta;
        // Автомобиль почти сломан?
        if (10 == (MaxSpeed - CurrentSpeed)
            && listOfHandlers != null)
        {
            listOfHandlers("Careful buddy! Gonna blow!");
        }
        if (CurrentSpeed >= MaxSpeed)
            carIsDead = true;
        else
            Console.WriteLine("CurrentSpeed = {0}", CurrentSpeed);
    }
}
```

Обратите внимание, что прежде чем вызывать методы, поддерживаемые переменной-членом `listOfHandlers`, ее значение проверяется на равенство `null`. Причина в том, что создание таких объектов посредством вызова вспомогательного метода `RegisterWithCarEngine()` является задачей вызывающего кода. Если вызывающий код не вызывал `RegisterWithCarEngine()`, а мы попытаемся обратиться к списку вызовов делегата, то получим исключение `NullReferenceException` во время выполнения. При наличии готовой инфраструктуры делегатов займемся модификацией класса `Program`:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Delegates as event enablers *****\n");

        // Создать объект Car.
        Car c1 = new Car("SlugBug", 100, 10);

        // Сообщить объекту Car, какой метод вызывать,
        // когда он пожелает отправить сообщение.
        c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

        // Увеличить скорость (это инициирует события).
        Console.WriteLine("***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);
        Console.ReadLine();
    }

    // Цель для входящих сообщений.

    {
        Console.WriteLine("\n***** Message From Car Object *****");
        Console.WriteLine("=> {0}", msg);
        Console.WriteLine("*****\n");
    }
}

```

Метод `Main()` начинается с создания нового объекта `Car`. Поскольку мы заинтересованы в событиях, связанных с двигателем, следующий шаг заключается в вызове специальной регистрационной функции `RegisterWithCarEngine()`. Вспомните, что метод `RegisterWithCarEngine()` ожидает получения экземпляра вложенного делегата `CarEngineHandler`, и как в случае любого делегата, в параметре конструктора передается метод, на который он должен указывать. Трюк здесь в том, что интересующий метод находится в классе `Program`! Обратите также внимание, что метод `OnCarEngineEvent()` полностью соответствует связанному делегату, потому что принимает `string` и возвращает `void`. Ниже показан вывод приведенного примера:

```

***** Delegates as event enablers *****
***** Speeding up *****
CurrentSpeed = 30
CurrentSpeed = 50
CurrentSpeed = 70

***** Message From Car Object *****
=> Careful buddy! Gonna blow!
*****
CurrentSpeed = 90
***** Message From Car Object *****
=> Sorry, this car is dead...
*****

```

Включение группового вызова

Вспомните, что делегаты .NET обладают встроенной возможностью *группового вызова*. Другими словами, объект делегата может поддерживать целый список методов для вызова, а не просто единственный метод. Для добавления нескольких методов к объекту делегата вместо прямого присваивания применяется перегруженная операция `+=`.

Чтобы включить групповой вызов в классе Car, можно модифицировать метод RegisterWithCarEngine():

```
public class Car
{
    // Добавление поддержки группового вызова.
    // Обратите внимание на использование операции +=,
    // а не обычной операции присваивания (=).
    public void RegisterWithCarEngine(CarEngineHandler methodToCall)
    {
        listOfHandlers += methodToCall;
    }
    ...
}
```

Когда операция += используется с объектом делегата, компилятор преобразует ее в вызов статического метода Delegate.Combine(). На самом деле можно было бы вызывать Delegate.Combine() напрямую, однако операция += предлагает более простую альтернативу. Хотя нет никакой необходимости в модификации текущего метода RegisterWithCarEngine(), ниже представлен пример применения Delegate.Combine() вместо операции +=:

```
public void RegisterWithCarEngine( CarEngineHandler methodToCall )
{
    if (listOfHandlers == null)
        listOfHandlers = methodToCall;
    else
        Delegate.Combine(listOfHandlers, methodToCall);
}
```

В любом случае вызывающий код теперь может регистрировать множественные цели для одного и того же обратного вызова. Второй обработчик выводит входное сообщение в верхнем регистре просто ради отображения:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Delegates as event enablers *****\n");
        // Создать объект Car.
        Car c1 = new Car("SlugBug", 100, 10);
        // Зарегистрировать несколько обработчиков событий.
        c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));
        c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent2));
        // Увеличить скорость (это инициирует события).
        Console.WriteLine("***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);
        Console.ReadLine();
    }
    // Теперь есть ДВА метода, которые будут
    // вызываться Car при отправке уведомлений.
    public static void OnCarEngineEvent(string msg)
    {
        Console.WriteLine("\n***** Message From Car Object *****");
        Console.WriteLine("=> {0}", msg);
    }
}
```

```

        Console.WriteLine("*****\n");
    }

    public static void OnCarEngineEvent2(string msg)
    {
        Console.WriteLine("=> {0}", msg.ToUpper());
    }
}

```

Удаление целей из списка вызовов делегата

В классе `Delegate` также определен статический метод `Remove()`, который позволяет вызывающему коду динамически удалять отдельные методы из списка вызовов объекта делегата. В итоге у вызывающего кода появляется возможность легко “отменять подписку” на заданное уведомление во время выполнения. Хотя метод `Delegate.Remove()` допускается вызывать в коде напрямую, разработчики C# могут использовать операцию `--` в качестве удобного сокращения. Давайте добавим в класс `Car` новый метод, который позволяет вызывающему коду исключать метод из списка вызовов:

```

public class Car
{
    ...
    public void UnRegisterWithCarEngine(CarEngineHandler methodToCall)
    {
        listOfHandlers -= methodToCall;
    }
}

```

При таком обновлении класса `Car` прекратить получение уведомлений от второго обработчика можно за счет изменения метода `Main()` следующим образом:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Delegates as event enablers *****\n");

    // Создать объект Car.
    Car c1 = new Car("SlugBug", 100, 10);
    c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

    // На этот раз сохранить объект делегата, чтобы позже можно было
    // отменить регистрацию.
    Car.CarEngineHandler handler2 = new Car.CarEngineHandler(OnCarEngineEvent2);
    c1.RegisterWithCarEngine(handler2);

    // Увеличить скорость (это инициирует события).
    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
        c1.Accelerate(20);

    // Отменить регистрацию второго обработчика.
    c1.UnRegisterWithCarEngine(handler2);

    // Сообщения в верхнем регистре больше не выводятся.
    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
        c1.Accelerate(20);

    Console.ReadLine();
}

```

Метод `Main()` отличается тем, что в нем создается объект `Car.CarEngineHandler`, который сохраняется в локальной переменной, чтобы впоследствии можно было отменить подписку на получение уведомлений. Таким образом, при увеличении скорости объекта `Car` во второй раз версия входного сообщения в верхнем регистре больше выводится не будет, поскольку данная цель исключена из списка вызовов делегата.

Исходный код. Проект `CarDelegate` доступен в подкаталоге `Chapter_10`.

Синтаксис групповых преобразований методов

В предыдущем примере `CarDelegate` явно создавались экземпляры класса делегата `Car.CarEngineHandler` для регистрации и отмены регистрации на получение уведомлений:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Delegates as event enablers *****\n");
    Car c1 = new Car("SlugBug", 100, 10);
    c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

    Car.CarEngineHandler handler2 =
        new Car.CarEngineHandler(OnCarEngineEvent2);
    c1.RegisterWithCarEngine(handler2);
    ...
}
```

Конечно, если необходимо вызывать любые унаследованные члены класса `MulticastDelegate` или `Delegate`, то проще всего вручную создать переменную делегата. Однако в большинстве случаев иметь дело с внутренним устройством объекта делегата не требуется. Объект делегата обычно придется применять только для передачи имени метода в параметре конструктора.

Для простоты в языке C# предлагается сокращение, называемое *групповым преобразованием методов*. Это средство позволяет указывать вместо объекта делегата прямое имя метода, когда вызываются методы, которые принимают делегаты в качестве аргументов.

На заметку! Позже в главе вы увидите, что синтаксис группового преобразования методов можно также использовать для упрощения регистрации событий C#.

В целях иллюстрации создадим новый проект консольного приложения по имени `CarDelegateMethodGroupConversion` и добавим к нему файл, содержащий класс `Car`, который был определен в проекте `CarDelegate` (а также скорректируем название пространства имен в файле `Car.cs`). В показанном ниже классе `Program` для регистрации и отмены регистрации подписки на уведомления применяется групповое преобразование методов:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Method Group Conversion *****\n");
        Car c1 = new Car();

        // Зарегистрировать простое имя метода.
        c1.RegisterWithCarEngine(CallMeHere);
    }
}
```

```

Console.WriteLine("***** Speeding up *****");
for (int i = 0; i < 6; i++)
    cl.Accelerate(20);

// Отменить регистрацию простого имени метода.
cl.UnRegisterWithCarEngine(CallMeHere);

// Уведомления больше не поступают!
for (int i = 0; i < 6; i++)
    cl.Accelerate(20);

Console.ReadLine();
}

static void CallMeHere(string msg)
{
    Console.WriteLine("=> Message from Car: {0}", msg);
}
}

```

Обратите внимание, что мы не создаем напрямую ассоциированный объект делегата, а просто указываем метод, который соответствует ожидаемой сигнатуре делегата (в данном случае метод, возвращающий `void` и принимающий единственный аргумент `string`). Имейте в виду, что компилятор C# по-прежнему обеспечивает безопасность к типам. Таким образом, если метод `CallMeHere()` не принимает `string` и не возвращает `void`, тогда возникнет ошибка на этапе компиляции.

Исходный код. Проект `CarDelegateMethodGroupConversion` доступен в подкаталоге `Chapter_10`.

Понятие обобщенных делегатов

В предыдущей главе упоминалось о том, что язык C# позволяет определять обобщенные типы делегатов. Например, предположим, что необходимо определить тип делегата, который может вызывать любой метод, возвращающий `void` и принимающий единственный параметр. Если передаваемый аргумент может изменяться, то это легко смоделировать с использованием параметра типа. Взгляните на следующий код внутри нового проекта консольного приложения по имени `GenericDelegate`:

```

namespace GenericDelegate
{
    // Этот обобщенный делегат может вызывать любой метод, который
    // возвращает void и принимает единственный параметр типа T.
    public delegate void MyGenericDelegate<T>(T arg);

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Generic Delegates *****\n");

            // Зарегистрировать цели.
            MyGenericDelegate<string> strTarget =
                new MyGenericDelegate<string>(StringTarget);
            strTarget("Some string data");

            MyGenericDelegate<int> intTarget =
                new MyGenericDelegate<int>(IntTarget);
            intTarget(9);
        }
    }
}

```

```

        Console.ReadLine();
    }
    static void StringTarget(string arg)
    {
        Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
    }
    static void IntTarget(int arg)
    {
        Console.WriteLine("++arg is: {0}", ++arg);
    }
}

```

Как видите, в типе делегата `MyGenericDelegate<T>` определен единственный параметр, представляющий аргумент для передачи цели делегата. При создании экземпляра этого типа должно быть указано значение параметра типа наряду с именем метода, который делегат может вызывать. Таким образом, если указать тип `string`, тогда целевому методу будет отправляться строковое значение:

```

// Создать экземпляр MyGenericDelegate<T>
// с указанием string в качестве параметра типа.
MyGenericDelegate<string> strTarget =
    new MyGenericDelegate<string>(StringTarget);
strTarget("Some string data");

```

С учетом формата объекта `strTarget` метод `StringTarget` теперь должен принимать в качестве параметра единственную строку:

```

static void StringTarget(string arg)
{
    Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}

```

Исходный код. Проект `GenericDelegate` доступен в подкаталоге `Chapter_10`.

Обобщенные делегаты `Action<>` и `Func<>`

В настоящей главе вы уже видели, что когда нужно применять делегаты для обратных вызовов в приложениях, обычно должны быть выполнены следующие шаги:

- определение специального делегата, соответствующего формату метода, на который он указывает;
- создание экземпляра специального делегата с передачей имени метода в качестве аргумента конструктора;
- косвенное обращение к методу через вызов `Invoke()` на объекте делегата.

В случае принятия такого подхода в итоге, как правило, получается несколько специальных делегатов, которые могут никогда не использоваться за рамками текущей задачи (например, `MyGenericDelegate<T>`, `CarEngineHandler` и т.д.). Хотя вполне может быть и так, что для проекта требуется специальный уникально именованный делегат, в других ситуациях точное имя типа делегата роли не играет. Во многих случаях просто необходим "какой-нибудь делегат", который принимает набор аргументов и возможно возвращает значение, отличное от `void`. В таких ситуациях можно применять встроенные в инфраструктуру делегаты `Action<>` и `Func<>`. Чтобы продемонстрировать их полезность, создадим новый проект консольного приложения по имени `ActionAndFuncDelegates`.

Обобщенный делегат `Action<>` определен в пространствах имен `System` внутри сборок `mscorlib.dll` и `System.Core.dll`. Его можно использовать для “указания” на метод, который принимает вплоть до 16 аргументов (чего должно быть вполне достаточно!) и возвращает `void`. Вспомните, что поскольку `Action<>` является обобщенным делегатом, понадобится также указывать типы всех параметров.

Модифицируем класс `Program`, определив в нем новый статический метод, который принимает (скажем) три уникальных параметра:

```
// Это цель для делегата Action<>.
static void DisplayMessage(string msg, ConsoleColor txtColor, int printCount)
{
    // Установить цвет текста консоли.
    ConsoleColor previous = Console.ForegroundColor;
    Console.ForegroundColor = txtColor;

    for (int i = 0; i < printCount; i++)
    {
        Console.WriteLine(msg);
    }

    // Восстановить цвет.
    Console.ForegroundColor = previous;
}
```

Теперь вместо построения специального делегата вручную для передачи потока программы методу `DisplayMessage()` мы можем применять готовый делегат `Action<>`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Action and Func *****");

    // Использовать делегат Action<> для указания на DisplayMessage.
    Action<string, ConsoleColor, int> actionTarget =
        new Action<string, ConsoleColor, int>(DisplayMessage);
    actionTarget("Action Message!", ConsoleColor.Yellow, 5);

    Console.ReadLine();
}
```

Как видите, при использовании делегата `Action<>` не нужно беспокоиться об определении специального типа делегата. Тем не менее, как уже упоминалось, тип делегата `Action<>` позволяет указывать только на методы, возвращающие `void`. Если необходимо указывать на метод, имеющий возвращаемое значение (и нет желания заниматься написанием собственного типа делегата), тогда можно применять тип делегата `Func<>`.

Обобщенный делегат `Func<>` способен указывать на методы, которые (подобно `Action<>`) принимают вплоть до 16 параметров и имеют специальное возвращаемое значение. В целях иллюстрации добавим в класс `Program` новый метод:

```
// Цель для делегата Func<>.
static int Add(int x, int y)
{
    return x + y;
}
```

Ранее в главе был построен специальный делегат `BinaryOp` для “указания” на методы сложения и вычитания. Теперь задачу можно упростить за счет использования версии `Func<>`, которая принимает всего три параметра типа. Учтите, что последний параметр в `Func<>` всегда представляет возвращаемое значение метода. Чтобы закрепить данный момент, предположим, что в классе `Program` также определен следующий метод:

```
static string SumToString(int x, int y)
{
    return (x + y).ToString();
}
```

Вызовем эти методы внутри Main():

```
Func<int, int, int> funcTarget = new Func<int, int, int>(Add);
int result = funcTarget.Invoke(40, 40);
Console.WriteLine("40 + 40 = {0}", result);

Func<int, int, string> funcTarget2 = new Func<int, int,
string>(SumToString);
string sum = funcTarget2(90, 300);
Console.WriteLine(sum);
```

Синтаксис групповых преобразований методов позволяет упростить предшествующий код:

```
Func<int, int, int> funcTarget = Add;
int result = funcTarget.Invoke(40, 40);
Console.WriteLine("40 + 40 = {0}", result);

Func<int, int, string> funcTarget2 = SumToString;
string sum = funcTarget2(90, 300);
Console.WriteLine(sum);
```

С учетом того, что делегаты `Action<>` и `Func<>` могут устранить шаг по ручному определению специального делегата, вас может интересовать, должны ли они применяться всегда. Подобно большинству аспектов программирования ответ таков: в зависимости от ситуации. Во многих случаях `Action<>` и `Func<>` будут предпочтительным вариантом. Однако если необходим делегат со специальным именем, которое, как вам кажется, помогает лучше отразить предметную область, то построение специального делегата сводится к единственному оператору кода. В оставшихся материалах книги вы увидите оба подхода.

На заметку! Делегаты `Action<>` и `Func<>` интенсивно используются во многих важных API-интерфейсах .NET, включая инфраструктуру параллельного программирования и LINQ (помимо прочих).

Итак, первоначальный экскурс в типы делегатов .NET завершен. Мы обратимся к ряду дополнительных деталей работы с делегатами в конце настоящей главы и еще раз в главе 19, когда будем рассматривать многопоточность и асинхронные вызовы. А теперь давайте перейдем к обсуждению связанной темы — ключевого слова `event` языка C#.

Исходный код. Проект `ActionAndFuncDelegates` доступен в подкаталоге `Chapter_10`.

Понятие событий C#

Делегаты — довольно интересные конструкции в том плане, что позволяют объектам, находящимся в памяти, участвовать в двустороннем взаимодействии. Тем не менее, прямая работа с делегатами может приводить к написанию стереотипного кода (определение делегата, определение необходимых переменных-членов, создание специальных методов регистрации и отмены регистрации для предохранения инкапсуляции и т.д.).

Более того, во время применения делегатов непосредственным образом как механизма обратного вызова в приложениях, если вы не определите переменную-член типа делегата в классе как закрытую, тогда вызывающий код будет иметь прямой доступ к объектам делегатов. В таком случае вызывающий код может присвоить переменной-члену новый объект делегата (фактически удаляя текущий список функций, которые подлежат вызову) и, что даже хуже, вызывающий код сможет напрямую обращаться к списку вызовов делегата. Чтобы проиллюстрировать проблему, рассмотрим следующую переделанную (и упрощенную) версию класса `Car` из предыдущего примера `CarDelegate`:

```
public class Car
{
    public delegate void CarEngineHandler(string msgForCaller);
    // Теперь это член public!
    public CarEngineHandler listOfHandlers;
    // Просто вызвать уведомление Exploded.
    public void Accelerate(int delta)
    {
        if (listOfHandlers != null)
            listOfHandlers("Sorry, this car is dead...");
    }
}
```

Обратите внимание, что теперь больше нет закрытых переменных-членов с типами делегатов, инкапсулированных с помощью специальных методов регистрации. Поскольку эти члены на самом деле открытые, вызывающий код может получить доступ прямо к переменной-члену `listOfHandlers`, присвоить ей новые объекты `CarEngineHandler` и вызвать делегат по своему желанию:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Agh! No Encapsulation! *****\n");
        // Создать объект Car.
        Car myCar = new Car();
        // Есть прямой доступ к делегату!
        myCar.listOfHandlers = new Car.CarEngineHandler(CallWhenExploded);
        myCar.Accelerate(10);

        // Теперь можно присвоить полностью новый объект...
        // что в лучшем случае сбивает с толку.
        myCar.listOfHandlers = new Car.CarEngineHandler(CallHereToo);
        myCar.Accelerate(10);

        // Вызывающий код может также напрямую вызывать делегат!
        myCar.listOfHandlers.Invoke("hee, hee, hee...");
        Console.ReadLine();
    }
    static void CallWhenExploded(string msg)
    { Console.WriteLine(msg); }
    static void CallHereToo(string msg)
    { Console.WriteLine(msg); }
}
```

Открытие доступа к членам типа делегатов нарушает инкапсуляцию, что не только затруднит сопровождение кода (и отладку), но также сделает приложение уязвимым в плане безопасности! Ниже показан вывод текущего примера:


```
***** Agh! No Encapsulation! *****
Sorry, this car is dead...
Sorry, this car is dead...
hee, hee, hee...
```

Очевидно, что вы не захотите предоставлять другим приложениям возможность изменять то, на что указывает делегат, или вызывать его члены без вашего разрешения. С учетом сказанного общепринятая практика предусматривает объявление переменных-членов, имеющих типы делегатов, как закрытых.

Исходный код. Проект `PublicDelegateProblem` доступен в подкаталоге `Chapter_10`.

Ключевое слово `event`

В качестве сокращения, избавляющего от необходимости создавать специальные методы для добавления и удаления методов из списка вызовов делегата, в языке C# предлагается ключевое слово `event`. В результате обработки компилятором ключевого слова `event` вы автоматически получаете методы регистрации и отмены регистрации, а также все необходимые переменные-члены для типов делегатов. Такие переменные-члены с типами делегатов *всегда* объявляются как закрытые и потому они не доступны напрямую из объекта, инициирующего событие. В итоге ключевое слово `event` может использоваться для упрощения отправки специальным классом уведомлений внешним объектам.

Определение события представляет собой двухэтапный процесс. Во-первых, понадобится определить тип делегата (или задействовать существующий тип), который будет хранить список методов, подлежащих вызову при возникновении события. Во-вторых, необходимо объявить событие (с применением ключевого слова `event`) в терминах связанного типа делегата.

Чтобы продемонстрировать использование ключевого слова `event`, создадим новый проект консольного приложения по имени `CarEvents`. В этой версии класса `Car` будут определены два события под названиями `AboutToBlow` и `Exploded`, которые ассоциированы с единственным типом делегата по имени `CarEngineHandler`. Ниже показаны начальные изменения, внесенные в класс `Car`:

```
public class Car
{
    // Этот делегат работает в сочетании с событиями Car.
    public delegate void CarEngineHandler(string msg);

    // Car может отправлять следующие события:
    public event CarEngineHandler Exploded;
    public event CarEngineHandler AboutToBlow;
    ...
}
```

Отправка события вызывающему коду сводится просто к указанию события по имени наряду со всеми обязательными параметрами, как определено ассоциированным делегатом. Чтобы удостовериться в том, что вызывающий код действительно зарегистрировал событие, перед вызовом набора методов делегата событие следует проверить на равенство `null`. Ниже приведена новая версия метода `Accelerate()` класса `Car`:

```
public void Accelerate(int delta)
{
    // Если автомобиль сломан, то инициировать событие Exploded.
    if (carIsDead)
```

```

{
    if (Exploded != null)
        Exploded("Sorry, this car is dead...");
}
else
{
    CurrentSpeed += delta;
    // Почти сломан?
    if (10 == MaxSpeed - CurrentSpeed
        && AboutToBlow != null)
    {
        AboutToBlow("Careful buddy! Gonna blow!");
    }
    // Все еще в порядке!
    if (CurrentSpeed >= MaxSpeed)
        carIsDead = true;
    else
        Console.WriteLine("CurrentSpeed = {0}", CurrentSpeed);
}
}

```

Итак, класс `Car` был сконфигурирован для отправки двух специальных событий без необходимости в определении специальных функций регистрации или в объявлении переменных-членов, имеющих типы делегатов. Применение нового объекта вы увидите очень скоро, но сначала давайте чуть подробнее рассмотрим архитектуру событий.

“За кулисами” событий

Когда компилятор C# обрабатывает ключевое слово `event`, он генерирует два скрытых метода, один с префиксом `add_`, а другой с префиксом `remove_`. За префиксом следует имя события C#. Например, событие `Exploded` дает в результате два скрытых метода с именами `add_Exploded()` и `remove_Exploded()`. Если заглянуть в код CIL метода `add_AboutToBlow()`, то можно обнаружить вызов метода `Delegate.Combine()`. Взгляните на частичный код CIL:

```

.method public hideby sig specialname instance void
    add_AboutToBlow(class CarEvents.Car/CarEngineHandler 'value') cil managed
{
    ...
    call class [mscorlib]System.Delegate
        [mscorlib]System.Delegate::Combine(
            class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
    ...
}

```

Как и можно было ожидать, метод `remove_AboutToBlow()` будет вызывать `Delegate.Remove()`:

```

.method public hideby sig specialname instance void
    remove_AboutToBlow(class CarEvents.Car/CarEngineHandler 'value') cil managed
{
    ...
    call class [mscorlib]System.Delegate
        [mscorlib]System.Delegate::Remove(
            class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
    ...
}

```

Наконец, в коде CIL, представляющем само событие, используются директивы `.addon` и `.removeon` для отображения на имена корректных методов `add_XXX()` и `remove_XXX()`, подлежащих вызову:

```
.event CarEvents.Car/EngineHandler AboutToBlow
{
    .addon instance void CarEvents.Car::add_AboutToBlow
        (class CarEvents.Car/CarEngineHandler)
    .removeon instance void CarEvents.Car::remove_AboutToBlow
        (class CarEvents.Car/CarEngineHandler)
}
```

Теперь, когда вы понимаете, каким образом строить класс, способный отправлять события C# (и знаете, что события — всего лишь способ сэкономить время на наборе кода), следующий крупный вопрос связан с организацией прослушивания входящих событий на стороне вызывающего кода.

Прослушивание входящих событий

События C# также упрощают действие по регистрации обработчиков событий на стороне вызывающего кода. Вместо того чтобы указывать специальные вспомогательные методы, вызывающий код просто применяет операции `+=` и `-=` напрямую (что приводит к внутренним вызовам методов `add_XXX()` или `remove_XXX()`). При регистрации события руководствуйтесь показанным ниже шаблоном:

```
// ИмяОбъекта.ИмяСобытия += new СвязанныйДелегат(функцияДляВызова);
//
Car.CarEngineHandler d = new Car.CarEngineHandler(CarExplodedEventHandler);
myCar.Exploded += d;
```

Отключить от источника событий можно с помощью операции `-=` в соответствии со следующим шаблоном:

```
// ИмяОбъекта.ИмяСобытия -= СвязанныйДелегат(функцияДляВызова);
//
myCar.Exploded -= d;
```

Имея такие весьма предсказуемые шаблоны, переделаем метод `Main()`, используя на этот раз синтаксис регистрации методов C#:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Events *****\n");
        Car c1 = new Car("SlugBug", 100, 10);

        // Зарегистрировать обработчики событий.
        c1.AboutToBlow += new Car.CarEngineHandler(CarIsAlmostDoomed);
        c1.AboutToBlow += new Car.CarEngineHandler(CarAboutToBlow);

        Car.CarEngineHandler d = new Car.CarEngineHandler(CarExploded);
        c1.Exploded += d;

        Console.WriteLine("***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);

        // Удалить метод CarExploded из списка вызовов.
        c1.Exploded -= d;
```

```

    Console.WriteLine("\n***** Speeding up *****");
    for (int i = 0; i < 6; i++)
        cl.Accelerate(20);
    Console.ReadLine();
}

public static void CarAboutToBlow(string msg)
{ Console.WriteLine(msg); }

public static void CarIsAlmostDoomed(string msg)
{ Console.WriteLine("=> Critical Message from Car: {0}", msg); }

public static void CarExploded(string msg)
{ Console.WriteLine(msg); }
}

```

Чтобы еще больше упростить регистрацию событий, можно применять групповое преобразование методов. Вот очередная модификация метода Main():

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Events *****\n");
    Car cl = new Car("SlugBug", 100, 10);
    // Зарегистрировать обработчики событий.
    cl.AboutToBlow += CarIsAlmostDoomed;
    cl.AboutToBlow += CarAboutToBlow;
    cl.Exploded += CarExploded;

    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
        cl.Accelerate(20);

    cl.Exploded -= CarExploded;

    Console.WriteLine("\n***** Speeding up *****");
    for (int i = 0; i < 6; i++)
        cl.Accelerate(20);

    Console.ReadLine();
}

```

Упрощение регистрации событий с использованием Visual Studio

Среда Visual Studio предлагает помощь в процессе регистрации обработчиков событий. В случае применения синтаксиса += при регистрации событий открывается окно IntelliSense, приглашающее нажать клавишу <Tab> для автоматического завершения связанного экземпляра делегата (рис. 10.2), что достигается с использованием синтаксиса *групповых преобразований методов*.

После нажатия клавиши <Tab> будет сгенерирован новый метод, как показано на рис. 10.3.

Обратите внимание, что код заглушки имеет корректный формат цели делегата (кроме того, метод объявлен как static, т.к. событие было зарегистрировано внутри статического метода):

```

static void NewCar_AboutToBlow(string msg)
{
    // Удалите следующую строку и добавьте свой код!
    throw new NotImplementedException();
}

```

Средство IntelliSense доступно для всех событий .NET из библиотек базовых классов. Такая возможность IDE-среды значительно экономит время, избавляя от необходимости выяснять с помощью справочной системы .NET подходящий тип делегата для применения с заданным событием и формат целевого метода делегата.



Рис. 10.2. Выбор делегата с помощью средства IntelliSense

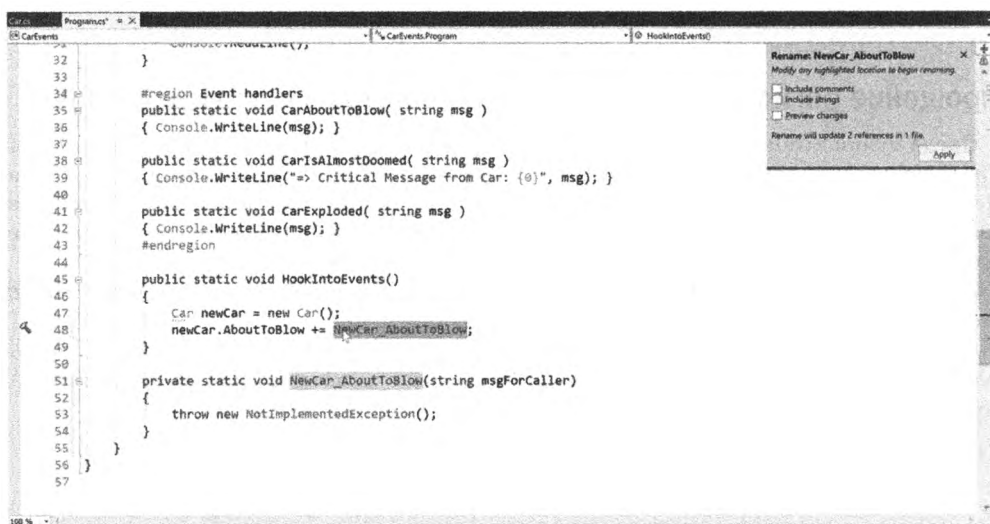


Рис. 10.3. Форматирование цели делегата средством IntelliSense

Приведение в порядок кода обращения к событиям с использованием null-условной операции C# 6.0

В рассматриваемом примере вы наверняка заметили, что перед отправкой события любому прослушивателю производится проверка на равенство `null`. Это важно, т.к. если событие никто не прослушивает, но вы в любом случае его иницилируете, то во время выполнения будет получено исключение из-за ссылки `null`. Одновременно с важностью такого подхода вы согласитесь с тем, что код с многочисленными проверками на предмет `null` выглядит несколько неуклюжим.

К счастью, начиная с версии C# 6, можно задействовать null-условную операцию (`?`), которая по существу выполняет проверку подобного рода автоматически. Имейте в виду, что при использовании нового упрощенного синтаксиса потребуются вручную вызывать метод `Invoke()` лежащего в основе делегата. Например, вместо кода:

```
// Если автомобиль сломан, то инициировать событие Exploded.
if (carIsDead)
{
    if (Exploded != null)
        Exploded("Sorry, this car is dead...");
}
```

теперь можно применять следующий код:

```
// Если автомобиль сломан, то инициировать событие Exploded.
if (carIsDead)
{
    Exploded?.Invoke("Sorry, this car is dead...");
}
```

Кроме того, аналогичным образом можно также обновить код, иницирующий событие `AboutToBlow` (обратите внимание, что здесь убрана проверка на равенство `null`, которая присутствовала в первоначальном операторе `if`):

```
// Почти сломан?
if (10 == MaxSpeed - CurrentSpeed)
{
    AboutToBlow?.Invoke("Careful buddy! Gonna blow!");
}
```

По причине более простого синтаксиса при инициировании событий вероятнее всего вы отдадите предпочтение null-условной операции. Однако реализация проверки на предмет `null` вручную, когда это необходимо, по-прежнему считается совершенно приемлемой.

Исходный код. Проект `CarEvents` доступен в подкаталоге `Chapter_10`.

Создание специальных аргументов событий

По правде говоря, в текущую итерацию класса `Car` можно было бы внести последнее усовершенствование, которое отражает рекомендованный Microsoft шаблон событий. Если вы начнете исследовать события, отправляемые определенным типом из библиотек базовых классов, то обнаружите, что первый параметр лежащего в основе делегата имеет тип `System.Object`, в то время как второй — тип, производный от `System.EventArgs`.

Параметр `System.Object` представляет ссылку на объект, который отправляет событие (такой как `Car`), а второй параметр — информацию, относящуюся к обрабатываемому событию. Базовый класс `System.EventArgs` представляет событие, которое не сопровождается какой-либо специальной информацией:

```
public class EventArgs
{
    public static readonly EventArgs Empty;
    public EventArgs();
}
```

Для простых событий экземпляр `EventArgs` можно передать напрямую. Тем не менее, когда нужно передавать специальные данные, вы должны построить подходящий класс, производный от `EventArgs`. В этом примере предположим, что есть класс по имени `CarEventArgs`, который поддерживает строковое представление сообщения, отправленного получателю:

```
public class CarEventArgs : EventArgs
{
    public readonly string msg;
    public CarEventArgs(string message)
    {
        msg = message;
    }
}
```

Теперь можно модифицировать тип делегата `CarEventHandler`, как показано ниже (события изменяться не будут):

```
public class Car
{
    public delegate void CarEventHandler(object sender, CarEventArgs e);
    ...
}
```

Здесь при инициировании событий внутри метода `Accelerate()` необходимо использовать ссылку на текущий объект `Car` (посредством ключевого слова `this`) и экземпляр типа `CarEventArgs`. Например, рассмотрим следующее обновление:

```
public void Accelerate(int delta)
{
    // Если этот автомобиль сломан, то инициировать событие Exploded.
    if (carIsDead)
    {
        Exploded?.Invoke(this, new CarEventArgs("Sorry, this car is dead..."));
    }
    ...
}
```

На вызывающей стороне понадобится лишь модифицировать обработчики событий для приема входных параметров и получения сообщения через поле, доступное только для чтения. Вот пример:

```
public static void CarAboutToBlow(object sender, CarEventArgs e)
{
    Console.WriteLine("{0} says: {1}", sender, e.msg);
}
```

Если получатель желает взаимодействовать с объектом, отправившим событие, тогда можно выполнить явное приведение `System.Object`. Такая ссылка позволит вызывать любой открытый метод объекта, который отправил уведомление:

```
public static void CarAboutToBlow(object sender, CarEventArgs e)
{
    // Просто для подстраховки перед приведением
    // произвести проверку во время выполнения.
    if (sender is Car c)
    {
        Car c = (Car)sender;
        Console.WriteLine("Critical Message from {0}: {1}", c.PetName, e.msg);
    }
}
```

Исходный код. Проект `CarEventArgs` доступен в подкаталоге `Chapter_10`.

Обобщенный делегат `EventHandler<T>`

Учитывая, что очень многие специальные делегаты принимают экземпляр `object` в первом параметре и экземпляр производного от `EventArgs` класса во втором, предыдущий пример можно дополнительно упростить за счет применения обобщенного типа `EventHandler<T>`, где `T` — специальный тип, производный от `EventArgs`. Рассмотрим следующую модификацию типа `Car` (обратите внимание, что определять специальный тип делегата больше не нужно):

```
public class Car
{
    public event EventHandler<CarEventArgs> Exploded;
    public event EventHandler<CarEventArgs> AboutToBlow;
    ...
}
```

Затем в методе `Main()` можно использовать тип `EventHandler<CarEventArgs>` везде, где ранее указывался `CarEngineHandler` (или снова применять групповое преобразование методов):

```
static void Main(string[] args)
{
    Console.WriteLine("***** Prim and Proper Events *****\n");
    // Создать объект Car обычным образом.
    Car c1 = new Car("SlugBug", 100, 10);
    // Зарегистрировать обработчики событий.
    c1.AboutToBlow += CarIsAlmostDoomed;
    c1.AboutToBlow += CarAboutToBlow;
    EventHandler<CarEventArgs> d = new EventHandler<CarEventArgs>(CarExploded);
    c1.Exploded += d;
    ...
}
```

К настоящему моменту вы видели основные аспекты работы с делегатами и событиями в C#. Хотя этого вполне достаточно для решения практически любых задач, связанных с обратными вызовами, в завершение главы мы рассмотрим несколько финальных упрощений, в частности анонимные методы и лямбда-выражения.

Понятие анонимных методов C#

Как было показано ранее, когда вызывающий код желает прослушивать входящие события, он должен определить специальный метод в классе (или структуре), который соответствует сигнатуре ассоциированного делегата. Ниже приведен пример:

```
class Program
{
    static void Main(string[] args)
    {
        SomeType t = new SomeType();

        // Предположим, что SomeDeletage может указывать на методы,
        // которые не принимают аргументов и возвращают void.
        t.SomeEvent += new SomeDelegate(MyEventHandler);
    }

    // Обычно вызывается только объектом SomeDelegate.
    public static void MyEventHandler()
    {
        // Что-то делать при возникновении события.
    }
}
```

Однако если подумать, то такие методы, как `MyEventHandler()`, редко предназначены для вызова из любой другой части программы кроме делегата. С точки зрения продуктивности вручную определять отдельный метод для вызова объектом делегата несколько хлопотно (хотя и вполне допустимо).

Для решения указанной проблемы событие можно ассоциировать прямо с блоком операторов кода во время регистрации события. Формально такой код называется *анонимным методом*. Чтобы проиллюстрировать синтаксис, давайте создадим метод `Main()`, который обрабатывает события, отправленные из класса `Car`, с использованием анонимных методов вместо специальных именованных обработчиков событий:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Anonymous Methods *****\n");
        Car c1 = new Car("SlugBug", 100, 10);

        // Зарегистрировать обработчики событий как анонимные методы.
        c1.AboutToBlow += delegate
        {
            Console.WriteLine("Eek! Going too fast!");
        };

        c1.AboutToBlow += delegate(object sender, CarEventArgs e)
        {
            Console.WriteLine("Message from Car: {0}", e.msg);
        };

        c1.Exploded += delegate(object sender, CarEventArgs e)
        {
            Console.WriteLine("Fatal Message from Car: {0}", e.msg);
        };
    }
}
```

```
// В конце концов, этот код будет инициировать события.
for (int i = 0; i < 6; i++)
    cl.Accelerate(20);
Console.ReadLine();
}
}
```

На заметку! После последней фигурной скобки в анонимном методе должна быть помещена точка с запятой, иначе возникнет ошибка на этапе компиляции.

И снова легко заметить, что специальные статические обработчики событий вроде `CarAboutToBlow()` или `CarExploded()` в классе `Program` больше не определяются. Взамен с помощью синтаксиса `+=` определяются встроенные неименованные (т.е. анонимные) методы, к которым вызывающий код будет обращаться во время обработки события. Базовый синтаксис анонимного метода представлен следующим псевдокодом:

```
class Program
{
    static void Main(string[] args)
    {
        НекоторыйТип t = new НекоторыйТип();
        t.НекотороеСобытие +=
            delegate (необязательноУказываемыеАргументыДелегата)
            { /* операторы */ };
    }
}
```

Обратите внимание, что при обработке первого события `AboutToBlow` внутри предыдущего метода `Main()` аргументы, передаваемые из делегата, не указывались:

```
cl.AboutToBlow += delegate
{
    Console.WriteLine("Eek! Going too fast!");
};
```

Строго говоря, вы не обязаны принимать входные аргументы, отправленные специфическим событием. Но если вы хотите задействовать эти входные аргументы, тогда понадобится указать параметры, прототипированные типом делегата (как показано во второй обработке событий `AboutToBlow` и `Exploded`). Например:

```
cl.AboutToBlow += delegate(object sender, CarEventArgs e)
{
    Console.WriteLine("Critical Message from Car: {0}", e.msg);
};
```

Доступ к локальным переменным

Анонимные методы интересны тем, что способны обращаться к локальным переменным метода, где они определены. Формально такие переменные называются *внешними переменными* анонимного метода. Ниже перечислены важные моменты, касающиеся взаимодействия между областью действия анонимного метода и областью действия метода, в котором он определен.

- Анонимный метод не имеет доступа к параметрам `ref` и `out` определяющего метода.
- Анонимный метод не может иметь локальную переменную, имя которой совпадает с именем локальной переменной внешнего метода.

- Анонимный метод может обращаться к переменным экземпляра (или статическим переменным) из области действия внешнего класса.
- Анонимный метод может объявлять локальную переменную с тем же именем, что и у переменной-члена внешнего класса (локальные переменные имеют отдельную область действия и скрывают переменные-члены из внешнего класса).

Предположим, что в методе `Main()` определена локальная переменная по имени `aboutToBlowCounter` типа `int`. Внутри анонимных методов, которые обрабатывают событие `AboutToBlow`, мы увеличим значение `aboutToBlowCounter` на 1 и выведем результат на консоль перед завершением `Main()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Anonymous Methods *****\n");
    int aboutToBlowCounter = 0;

    // Создать объект Car обычным образом.
    Car c1 = new Car("SlugBug", 100, 10);

    // Зарегистрировать обработчики событий как анонимные методы.
    c1.AboutToBlow += delegate
    {
        aboutToBlowCounter++;
        Console.WriteLine("Eek! Going too fast!");
    };

    c1.AboutToBlow += delegate(object sender, CarEventArgs e)
    {
        aboutToBlowCounter++;
        Console.WriteLine("Critical Message from Car: {0}", e.msg);
    };

    // В конце концов, это будет инициировать события.
    for (int i = 0; i < 6; i++)
        c1.Accelerate(20);

    Console.WriteLine("AboutToBlow event was fired {0} times.",
        aboutToBlowCounter);
    Console.ReadLine();
}
```

После запуска модифицированного метода `Main()` вы обнаружите, что финальный вывод `Console.WriteLine()` сообщает о двукратном инициировании события `AboutToBlow`.

Исходный код. Проект `AnonymousMethods` доступен в подкаталоге `Chapter_10`.

Понятие лямбда-выражений

Чтобы завершить знакомство с архитектурой событий .NET, необходимо исследовать лямбда-выражения. Как объяснялось ранее в главе, язык C# поддерживает возможность обработки событий "встраиваемым образом", позволяя назначать блок операторов кода прямо событию с применением анонимных методов вместо построения отдельного метода, который должен вызываться делегатом. Лямбда-выражения — всего лишь лаконичный способ записи анонимных методов, который в конечном итоге упрощает работу с типами делегатов .NET.

В целях подготовки фундамента для изучения лямбда-выражений создадим новый проект консольного приложения по имени SimpleLambdaExpressions. Для начала взгляните на метод FindAll() обобщенного класса List<T>. Данный метод можно вызывать, когда нужно извлечь подмножество элементов из коллекции; вот его прототип:

```
// Метод класса System.Collections.Generic.List<T>.
public List<T> FindAll(Predicate<T> match)
```

Как видите, метод FindAll() возвращает новый объект List<T>, который представляет подмножество данных. Также обратите внимание, что единственным параметром FindAll() является обобщенный делегат типа System.Predicate<T>, способный указывать на любой метод, который возвращает bool и принимает единственный параметр:

```
// Этот делегат используется методом FindAll()
// для извлечения подмножества.
public delegate bool Predicate<T>(T obj);
```

Когда вызывается FindAll(), каждый элемент в List<T> передается методу, указанному объектом Predicate<T>. Реализация упомянутого метода будет выполнять некоторые вычисления для проверки соответствия элемента данных заданному критерию, возвращая в результате true или false. Если метод возвращает true, то текущий элемент будет добавлен в новый объект List<T>, представляющий интересующее подмножество.

Прежде чем мы посмотрим, как лямбда-выражения могут упростить работу с методом FindAll(), давайте решим задачу длинным способом, используя объекты делегатов непосредственно. Добавим в класс Program метод (TraditionalDelegateSyntax()), который взаимодействует с типом System.Predicate<T> для обнаружения четных чисел в списке List<T> целочисленных значений:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Lambdas *****\n");
        TraditionalDelegateSyntax();
        Console.ReadLine();
    }
    static void TraditionalDelegateSyntax()
    {
        // Создать список целочисленных значений.
        List<int> list = new List<int>();
        list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
        // Вызвать FindAll() с применением традиционного синтаксиса делегатов.
        Predicate<int> callback = IsEvenNumber;
        List<int> evenNumbers = list.FindAll(callback);
        Console.WriteLine("Here are your even numbers:");
        foreach (int evenNumber in evenNumbers)
        {
            Console.Write("{0}\t", evenNumber);
        }
        Console.WriteLine();
    }
    // Цель для делегата Predicate<>.
    static bool IsEvenNumber(int i)
    {
        // Это четное число?
        return (i % 2) == 0;
    }
}
```

Здесь мы имеем метод (`IsEvenNumber()`), который отвечает за проверку входного целочисленного параметра на предмет четности или нечетности с применением операции получения остатка от деления (%) языка C#. Запуск приложения приводит к выводу на консоль чисел 20, 4, 8 и 44.

Хотя такой традиционный подход к работе с делегатами ведет себя ожидаемым образом, `IsEvenNumber()` вызывается только при ограниченных обстоятельствах — в частности, когда вызывается метод `FindAll()`, который возлагает на нас обязанность по полному определению метода. Если бы взамен использовался анонимный метод, то код стал бы значительно чище. Рассмотрим следующий новый метод в классе `Program`:

```
static void AnonymousMethodSyntax()
{
    // Создать список целочисленных значений.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

    // Теперь использовать анонимный метод.
    List<int> evenNumbers = list.FindAll(delegate(int i)
    { return (i % 2) == 0; }));

    // Вывести четные числа.
    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}
```

В этом случае вместо прямого создания объекта делегата `Predicate<T>` и последующего написания отдельного метода есть возможность встроить метод как анонимный. Несмотря на шаг в правильном направлении, вам по-прежнему придется применять ключевое слово `delegate` (или строго типизированный класс `Predicate<T>`) и обеспечивать точное соответствие списка параметров:

```
List<int> evenNumbers = list.FindAll(
    delegate(int i)
    {
        return (i % 2) == 0;
    }
);
```

Для еще большего упрощения вызова метода `FindAll()` могут использоваться *лямбда-выражения*. Во время применения синтаксиса лямбда-выражений вообще не приходится иметь дело с лежащим в основе объектом делегата. Взгляните на показанный далее новый метод в классе `Program`:

```
static void LambdaExpressionSyntax()
{
    // Создать список целочисленных значений.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

    // Теперь использовать лямбда-выражение C#.
    List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);

    // Вывести четные числа.
    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
```

```

    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}

```

Обратите внимание на довольно странный оператор кода, передаваемый методу `FindAll()`, который на самом деле и представляет собой лямбда-выражение. В такой версии примера нет вообще никаких следов делегата `Predicate<T>` (или ключевого слова `delegate`, если на то пошло). Должно указываться только лямбда-выражение:

```
1 => (1 % 2) == 0
```

Перед разбором синтаксиса запомните, что лямбда-выражения могут использоваться везде, где должен применяться анонимный метод или строго типизированный делегат (обычно с клавиатурным набором гораздо меньшего объема). “За кулисами” компилятор C# транслирует лямбда-выражение в стандартный анонимный метод, использующий тип делегата `Predicate<T>` (в чем можно удостовериться с помощью утилиты `ildasm.exe` или `reflector.exe`). Скажем, следующий оператор кода:

```
// Это лямбда-выражение...
List<int> evenNumbers = list.FindAll(1 => (i % 2) == 0);
```

компилируется в приблизительно такой код C#:

```
// ...становится следующим анонимным методом.
List<int> evenNumbers = list.FindAll(delegate (int i)
{
    return (i % 2) == 0;
});
```

Анализ лямбда-выражения

Лямбда-выражение начинается со списка параметров, за которым следует лексема `=>` (лексема C# для лямбда-операции позаимствована из области лямбда-вычислений), а за ней — набор операторов (или одиночный оператор), который будет обрабатывать передаваемые аргументы. На самом высоком уровне лямбда-выражение можно представить следующим образом:

```
АргументыДляОбработки => ОбработывающиеОператоры
```

То, что находится внутри метода `LambdaExpressionSyntax()`, понимается так:

```
// i - список параметров.
// (1 % 2) == 0 - набор операторов для обработки 1.
List<int> evenNumbers = list.FindAll(1 => (i % 2) == 0);
```

Параметры лямбда-выражения могут быть явно или неявно типизированными. В настоящий момент тип данных, представляющий параметр `i` (целочисленное значение), определяется неявно. Компилятор в состоянии понять, что `i` является целочисленным значением, на основе области действия всего лямбда-выражения и лежащего в основе делегата. Тем не менее, определять тип каждого параметра в лямбда-выражении можно также и явно, помещая тип данных и имя переменной в пару круглых скобок, как показано ниже:

```
// Теперь установим тип параметров явно.
List<int> evenNumbers = list.FindAll((int i) => (i % 2) == 0);
```

Как вы уже видели, если лямбда-выражение имеет одиночный неявно типизированный параметр, то круглые скобки в списке параметров могут быть опущены. Если вы

желаете соблюдать согласованность относительно применения параметров лямбда-выражений, тогда можете *всегда* заключать в скобки список параметров:

```
List<int> evenNumbers = list.FindAll((i) => (i % 2) == 0);
```

Наконец, обратите внимание, что в текущий момент выражение не заключено в круглые скобки (естественно, вычисление остатка от деления помещено в скобки, чтобы гарантировать его выполнение перед проверкой на равенство). В лямбда-выражениях разрешено заключать оператор в круглые скобки:

```
// Поместить в скобки и выражение.
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

После ознакомления с разными способами построения лямбда-выражения давайте выясним, как его можно читать в понятных человеку терминах. Оставив чистую математику в стороне, можно привести следующее объяснение:

```
// Список параметров (в данном случае единственное целочисленное
// значение по имени i) будет обработан выражением (i % 2) == 0.
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

Обработка аргументов внутри множества операторов

Первое рассмотренное лямбда-выражение включало единственный оператор, который в итоге вычислялся в булевское значение. Однако, как вы знаете, многие цели делегатов должны выполнять несколько операторов кода. По этой причине язык C# позволяет строить лямбда-выражения, состоящие из множества блоков операторов. Когда выражение должно обрабатывать параметры, используя несколько строк кода, для операторов понадобится обозначить область действия с помощью фигурных скобок. Взгляните на приведенную далее модификацию метода `LambdaExpressionSyntax()`:

```
static void LambdaExpressionSyntax()
{
    // Создать список целочисленных значений.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

    // Обработать каждый аргумент внутри группы операторов кода.
    List<int> evenNumbers = list.FindAll((i) =>
    {
        Console.WriteLine("value of i is currently: {0}", i); // текущее значение i
        bool isEven = ((i % 2) == 0);
        return isEven;
    });

    // Вывести четные числа.
    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}
```

В данном случае список параметров (опять состоящий из единственного целочисленного значения `i`) обрабатывается набором операторов кода. Помимо вызова метода `Console.WriteLine()` оператор вычисления остатка от деления разбит на два оператора ради повышения читабельности. Предположим, что каждый из рассмотренных выше методов вызывается внутри `Main()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Lambdas *****\n");
    TraditionalDelegateSyntax();
    AnonymousMethodSyntax();
    Console.WriteLine();
    LambdaExpressionSyntax();
    Console.ReadLine();
}
```

Запуск приложения дает следующий вывод:

```
***** Fun with Lambdas *****
Here are your even numbers:
20      4      8      44
Here are your even numbers:
20      4      8      44
value of i is currently: 20
value of i is currently: 1
value of i is currently: 4
value of i is currently: 8
value of i is currently: 9
value of i is currently: 44
Here are your even numbers:
20      4      8      44
```

Исходный код. Проект SimpleLambdaExpressions доступен в подкаталоге Chapter_10.

Лямбда-выражения с несколькими параметрами и без параметров

Показанные ранее лямбда-выражения обрабатывали единственный параметр. Тем не менее, это вовсе не обязательно, т.к. лямбда-выражения могут обрабатывать множество аргументов (или ни одного). Для демонстрации первого сценария создадим проект консольного приложения по имени LambdaExpressionsMultipleParams с показанной ниже версией класса SimpleMath:

```
public class SimpleMath
{
    public delegate void MathMessage(string msg, int result);
    private MathMessage mmDelegate;

    public void SetMathHandler(MathMessage target)
    {mmDelegate = target; }
    public void Add(int x, int y)
    {
        mmDelegate?.Invoke("Adding has completed!", x + y);
    }
}
```

Обратите внимание, что делегат MathMessage ожидает два параметра. Чтобы представить их в виде лямбда-выражения, метод Main() может быть реализован так:

```
static void Main(string[] args)
{
    // Зарегистрировать делегат как лямбда-выражение.
    SimpleMath m = new SimpleMath();
    m.SetMathHandler((msg, result) =>
        {Console.WriteLine("Message: {0}, Result: {1}", msg, result)});
}
```



```
// Это приведет к выполнению лямбда-выражения.
m.Add(10, 10);
Console.ReadLine();
}
```

Здесь задействовано выведение типа, поскольку для простоты два параметра не были строго типизированы. Однако метод `SetMathHandler()` можно было бы вызвать следующим образом:

```
m.SetMathHandler((string msg, int result) =>
    {Console.WriteLine("Message: {0}, Result: {1}", msg, result);});
```

Наконец, если лямбда-выражение применяется для взаимодействия с делегатом, который вообще не принимает параметров, то можно указать в качестве параметра пару пустых круглых скобок. Таким образом, предполагая, что определен приведенный далее тип делегата:

```
public delegate string VerySimpleDelegate();
```

вот как можно было бы обработать результат вызова:

```
// Выводит на консоль строку "Enjoy your string!".
VerySimpleDelegate d = new VerySimpleDelegate( () => {return "Enjoy your string!";} );
Console.WriteLine(d());
```

Исходный код. Проект `LambdaExpressionsMultipleParams` доступен в подкаталоге `Chapter_10`.

Модернизация примера `CarEvents` с использованием лямбда-выражений

С учетом того, что основной целью лямбда-выражений является предоставление способа ясного и компактного определения анонимных методов (косвенно упрощая работу с делегатами), давайте модернизируем проект `CarEvents`, созданный ранее в главе. Ниже приведена упрощенная версия метода `Main()`, в которой для перехвата всех событий, поступающих от объекта `Car`, применяется синтаксис лямбда-выражений (вместо простых делегатов):

```
static void Main(string[] args)
{
    Console.WriteLine("***** More Fun with Lambdas *****\n");
    // Создать объект Car обычным образом.
    Car c1 = new Car("SlugBug", 100, 10);
    // Привязаться к событиям с помощью лямбда-выражений.
    c1.AboutToBlow += (sender, e) => { Console.WriteLine(e.msg); };
    c1.Exploded += (sender, e) => { Console.WriteLine(e.msg); };
    // Увеличить скорость (это инициирует события).
    Console.WriteLine("\n***** Speeding up *****");
    for (int i = 0; i < 6; i++)
        c1.Accelerate(20);
    Console.ReadLine();
}
```

Лямбда-выражения и члены, сжатые до выражений (обновление)

Понимая лямбда-выражения и зная, как они работают, вам должно стать намного яснее, каким образом внутренне функционируют члены, сжатые до выражений. В главе 4 упоминалось, что в версии C# 6 появилась возможность использовать операцию `=>` для упрощения некоторых (но не всех) реализаций членов. В частности, если есть метод или свойство (в дополнение к специальной операции или процедуре преобразования, как показано в главе 11), реализация которого содержит единственную строку кода, тогда определять область действия посредством фигурных скобок необязательно. Взамен можно задействовать лямбда-операцию и написать член, сжатый до выражения. В версии C# 7 такой синтаксис можно применять для конструкторов и финализаторов классов (раскрываемых в главе 13), а также для средств доступа `get` и `set` к свойствам.

Рассмотрим предыдущий пример кода, где производится привязка обработчиков к событиям `AboutToBlow` и `Exploded`. Обратите внимание на определение с помощью фигурных скобок области действия, которая внутри содержит вызов метода `Console.WriteLine()`. При желании можно было бы поступить так:

```
cl.AboutToBlow += (sender, e) => Console.WriteLine(e.msg);
cl.Exploded += (sender, e) => Console.WriteLine(e.msg);
```

Тем не менее, имейте в виду, что новый сокращенный синтаксис может применяться где угодно, даже когда код не имеет никакого отношения к делегатам или событиям. Например, если вы строите элементарный класс для сложения двух чисел, то могли бы написать следующий код:

```
class SimpleMath
{
    public int Add(int x, int y)
    {
        return x + y;
    }

    public void PrintSum(int x, int y)
    {
        Console.WriteLine(x + y);
    }
}
```

В качестве альтернативы теперь код может выглядеть так:

```
class SimpleMath
{
    public int Add(int x, int y) => x + y;
    public void PrintSum(int x, int y) => Console.WriteLine(x + y);
}
```

В идеале к этому моменту вы должны уловить суть лямбда-выражений и понимать, что они предлагают “функциональный способ” работы с анонимными методами и типами делегатов. Хотя привыкание к лямбда-операции (`=>`) может занять некоторое время, просто запомните, что лямбда-выражение сводится к следующей форме:

АргументыДляОбработки => ОбрабатывающиеОператоры

Или, если операция `=>` используется для реализации члена типа с единственным оператором, то это будет выглядеть так:

ЧленТипа => ЕдинственныйОператорКода

Полезно отметить, что лямбда-выражения широко задействованы также в модели программирования LINQ, помогая упростить кодирование. Исследование LINQ начинается в главе 12.

Исходный код. Проект CarEventsWithLambdas доступен в подкаталоге Chapter_10.

Резюме

В настоящей главе вы получили представление о нескольких способах организации двустороннего взаимодействия для множества объектов. Во-первых, было рассмотрено ключевое слово `delegate`, которое применяется для косвенного конструирования класса, производного от `System.MulticastDelegate`. Вы узнали, что объект делегата поддерживает список методов для вызова тогда, когда ему об этом будет указано. Такие вызовы могут выполняться синхронно (с использованием метода `Invoke()`) или асинхронно (посредством методов `BeginInvoke()` и `EndInvoke()`). Асинхронная природа типов делегатов .NET будет обсуждаться в главе 19.

Во-вторых, вы ознакомились с ключевым словом `event`, которое в сочетании с типом делегата может упростить процесс отправки уведомлений ожидающим объектам. Как можно заметить в результирующем коде CIL, модель событий .NET отображается на скрытые обращения к типам `System.Delegate/System.MulticastDelegate`. В данном отношении ключевое слово `event` является совершенно необязательным, т.к. оно просто позволяет сэкономить на наборе кода. Кроме того, вы видели, что `null`-условная операция C# 6.0 упрощает безопасное инициирование событий для любой заинтересованной стороны.

В-третьих, в главе также рассматривалось средство языка C#, которое называется *анонимными методами*. С помощью такой синтаксической конструкции можно явно ассоциировать блок операторов кода с заданным событием. Было показано, что анонимные методы вполне могут игнорировать параметры, переданные событием, и имеют доступ к “внешним переменным” определяющего их метода. Вы также освоили упрощенный подход к регистрации событий с применением *групповых преобразований методов*.

Наконец, в-четвертых, мы взглянули на лямбда-операцию (`=>`) языка C#. Как было показано, этот синтаксис представляет собой сокращенный способ для записи анонимных методов, когда набор аргументов может быть передан на обработку группе операторов. Любой метод внутри платформы .NET, который принимает объект делегата в качестве аргумента, может быть заменен связанным лямбда-выражением, что обычно несколько упрощает кодовую базу.

глава 11

Расширенные средства языка C#

В настоящей главе ваше понимание языка программирования C# будет углублено за счет исследования нескольких более сложных тем. Сначала вы узнаете, как реализовывать и применять *индексаторный метод*. Такой механизм C# позволяет строить специальные типы, которые предоставляют доступ к внутренним элементам с использованием синтаксиса, подобного синтаксису массивов. Вы научитесь перегружать разнообразные операции (+, -, <, > и т.д.) и создавать для своих типов специальные процедуры явного и неявного преобразования (а также ознакомитесь с причинами, по которым они могут понадобиться).

Затем будут обсуждаться темы, которые особенно полезны при работе с API-интерфейсами LINQ (хотя они применимы и за рамками контекста LINQ): расширяющие методы и анонимные типы.

В завершение главы вы узнаете, каким образом создавать контекст “небезопасного” кода, чтобы напрямую манипулировать неуправляемыми указателями. Хотя использование указателей в приложениях C# — довольно редкое явление, понимание того, как это делается, может пригодиться в определенных обстоятельствах, связанных со сложными сценариями взаимодействия.

Понятие индексаторных методов

Программистам хорошо знаком процесс доступа к индивидуальным элементам, содержащимся внутри простого массива, с применением операции индекса ([]). Вот пример:

```
static void Main(string[] args)
{
    // Организовать цикл по аргументам командной строки
    // с использованием операции индекса.
    for(int i = 0; i < args.Length; i++)
        Console.WriteLine("Args: {0}", args[i]);

    // Объявить локальный массив целочисленных значений.
    int[] myInts = { 10, 9, 100, 432, 9874};

    // Применить операцию индекса для доступа к каждому элементу.
    for(int j = 0; j < myInts.Length; j++)
        Console.WriteLine("Index {0} = {1} ", j, myInts[j]);
    Console.ReadLine();
}
```

Приведенный код никоим образом не должен выглядеть как нечто совершенно новое. Но в языке C# предлагается возможность проектирования специальных классов и структур, которые могут индексироваться подобно стандартному массиву, за счет определения *индексаторного метода*. Такое языковое средство наиболее полезно при создании специальных классов коллекций (обобщенных или необобщенных).

Прежде чем выяснять, каким образом реализуется специальный индексатор, давайте начнем с того, что продемонстрируем его в действии. Пусть к специальному типу `PersonCollection`, разработанному в главе 9 (в проекте `IssuesWithNonGeneric Collections`), добавлена поддержка индексаторного метода. Хотя сам индексатор пока не добавлен, давайте посмотрим, как он используется внутри нового проекта консольного приложения по имени `SimpleIndexer`:

```
// Индексаторы позволяют обращаться к элементам в стиле массива.
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Indexers *****\n");
        PersonCollection myPeople = new PersonCollection();
        // Добавить объекты с применением синтаксиса индексатора.
        myPeople[0] = new Person("Homer", "Simpson", 40);
        myPeople[1] = new Person("Marge", "Simpson", 38);
        myPeople[2] = new Person("Lisa", "Simpson", 9);
        myPeople[3] = new Person("Bart", "Simpson", 7);
        myPeople[4] = new Person("Maggie", "Simpson", 2);
        // Получить и отобразить элементы, используя индексатор.
        for (int i = 0; i < myPeople.Count; i++)
        {
            Console.WriteLine("Person number: {0}", i);    // номер лица
            Console.WriteLine("Name: {0} {1}",
                myPeople[i].FirstName, myPeople[i].LastName); // имя и фамилия
            Console.WriteLine("Age: {0}", myPeople[i].Age); // возраст
            Console.WriteLine();
        }
    }
}
```

Как видите, индексаторы позволяют манипулировать внутренней коллекцией объектов подобно стандартному массиву. Но тут возникает серьезный вопрос: каким образом сконфигурировать класс `PersonCollection` (или любой другой класс либо структуру) для поддержки такой функциональности? Индексатор представлен как слегка видоизмененное определение свойства C#. В своей простейшей форме индексатор создается с применением синтаксиса `this[]`. Ниже показано необходимое обновление класса `PersonCollection`:

```
// Добавим индексатор к существующему определению класса.
public class PersonCollection : IEnumerable
{
    private ArrayList arPeople = new ArrayList();
    // Специальный индексатор для этого класса.
    public Person this[int index]
    {
        get => (Person)arPeople[index];
        set => arPeople.Insert(index, value);
    }
}
```

На заметку! Средства доступа `get` и `set` используют дополнение C# 7 к членам, сжатым до выражений, которое было представлено в главе 4 и подробно объяснялось в главе 10.

Если не считать факт использования ключевого слова `this`, то индексатор похож на объявление любого другого свойства C#. Например, роль области `get` заключается в возвращении корректного объекта вызывающему коду. Здесь мы достигаем цели делегированием запроса к индексатору объекта `ArrayList`, т.к. данный класс также поддерживает индексатор. Область `set` отвечает за добавление новых объектов `Person`, что достигается вызовом метода `Insert()` объекта `ArrayList`.

Индексаторы являются еще одной формой "синтаксического сахара", учитывая, что ту же самую функциональность можно получить с применением "нормальных" открытых методов, таких как `AddPerson()` или `GetPerson()`. Тем не менее, поддержка методов индексаторов в специальных типах коллекций обеспечивает хорошую интеграцию с инфраструктурой библиотек базовых классов .NET.

Несмотря на то что создание методов индексаторов является вполне обычным явлением при построении специальных коллекций, не забывайте, что обобщенные типы предлагают такую функциональность в готовом виде. В следующем методе используется обобщенный список `List<T>` объектов `Person`. Обратите внимание, что индексатор `List<T>` можно просто применять непосредственно:

```
static void UseGenericListOfPeople()
{
    List<Person> myPeople = new List<Person>();
    myPeople.Add(new Person("Lisa", "Simpson", 9));
    myPeople.Add(new Person("Bart", "Simpson", 7));

    // Изменить первый объект лица с помощью индексатора.
    myPeople[0] = new Person("Maggie", "Simpson", 2);

    // Получить и отобразить каждый элемент, используя индексатор.
    for (int i = 0; i < myPeople.Count; i++)
    {
        Console.WriteLine("Person number: {0}", i);
        Console.WriteLine("Name: {0} {1}", myPeople[i].FirstName,
            myPeople[i].LastName);
        Console.WriteLine("Age: {0}", myPeople[i].Age);
        Console.WriteLine();
    }
}
```

Исходный код. Проект `SimpleIndexer` доступен в подкаталоге `Chapter_11`.

Индексация данных с использованием строковых значений

В текущей версии класса `PersonCollection` определен индексатор, позволяющий вызывающему коду идентифицировать элементы с применением числовых значений. Однако вы должны понимать, что это не требование индексаторного метода. Предположим, что вы предпочитаете хранить объекты `Person`, используя тип `System.Collections.Generic.Dictionary<TKey, TValue>`, а не `ArrayList`. Поскольку типы `Dictionary` разрешают доступ к содержащимся внутри них элементам с применением ключа (такого как фамилия лица), индексатор можно было бы определить следующим образом:

```

public class PersonCollection : IEnumerable
{
    private Dictionary<string, Person> listPeople =
        new Dictionary<string, Person>();

    // Этот индексатор возвращает объект лица на основе строкового индекса.
    public Person this[string name]
    {
        get => (Person)listPeople[name];
        set => listPeople[name] = value;
    }

    public void ClearPeople()
    { listPeople.Clear(); }

    public int Count => listPeople.Count;

    IEnumerator IEnumerable.GetEnumerator() => listPeople.GetEnumerator();
}

```

Теперь вызывающий код может взаимодействовать с содержащимися внутри объектами `Person`, как показано ниже:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Indexers *****\n");
    PersonCollection myPeople = new PersonCollection();
    myPeople["Homer"] = new Person("Homer", "Simpson", 40);
    myPeople["Marge"] = new Person("Marge", "Simpson", 38);

    // Получить объект лица Homer и вывести данные.
    Person homer = myPeople["Homer"];
    Console.WriteLine(homer.ToString());

    Console.ReadLine();
}

```

И снова, если бы обобщенный тип `Dictionary<TKey, TValue>` использовался напрямую, то функциональность индексаторного метода была бы получена в готовом виде без построения специального необобщенного класса, поддерживающего строковый индексатор. Тем не менее, имейте в виду, что тип данных любого индексатора будет основан на том, как поддерживающий тип коллекции позволяет вызывающему коду извлекать элементы.

Исходный код. Проект `StringIndexer` доступен в подкаталоге `Chapter_11`.

Перегрузка методов индексаторов

Методы индексаторов могут быть перегружены в отдельном классе или структуре. Таким образом, если имеет смысл предоставить вызывающему коду возможность доступа к элементам с применением числового индекса или строкового значения, то в одном типе можно определить несколько индексаторов. Например, в ADO.NET (встроенный API-интерфейс .NET для доступа к базам данных) класс `DataSet` поддерживает свойство по имени `Tables`, которое возвращает строго типизированную коллекцию `DataTableCollection`. В свою очередь тип `DataTableCollection` определяет три индексатора для получения и установки объектов `DataTable` — по порядковой позиции, по дружественному строковому имени и по строковому имени с дополнительным пространством имен:

```
public sealed class DataTableCollection : InternalDataCollectionBase
{
    ...
    // Перегруженные индексаторы.
    public DataTable this[int index] { get; }
    public DataTable this[string name] { get; }
    public DataTable this[string name, string tableNamespace] { get; }
}
```

Поддержка индексаторных методов обычна для типов в библиотеках базовых классов. Поэтому даже если текущий проект не требует построения специальных индексаторов для классов и структур, помните, что многие типы уже поддерживают такой синтаксис.

Многомерные индексаторы

Допускается также создавать индексаторный метод, который принимает несколько параметров. Предположим, что есть специальная коллекция, хранящая элементы в двумерном массиве. В таком случае индексаторный метод можно определить следующим образом:

```
public class SomeContainer
{
    private int[,] my2DintArray = new int[10, 10];

    public int this[int row, int column]
    { /* получить или установить значение в двумерном массиве */ }
}
```

Если только вы не строите высокоспециализированный класс коллекций, то вряд ли будете особо нуждаться в создании многомерного индексатора. Пример ADO.NET еще раз демонстрирует, насколько полезной может оказаться такая конструкция. Класс `DataTable` в ADO.NET по существу представляет собой коллекцию строк и столбцов, похожую на миллиметровку или на общую структуру электронной таблицы Microsoft Excel.

Хотя объекты `DataTable` обычно наполняются без вашего участия с использованием связанного “адаптера данных”, в приведенном ниже коде показано, как вручную создать находящийся в памяти объект `DataTable`, который содержит три столбца (для имени, фамилии и возраста каждой записи). Обратите внимание на то, как после добавления одной строки в `DataTable` с помощью многомерного индексатора производится обращение ко всем столбцам первой (и единственной) строки. (Если вы собираетесь следовать примеру, тогда импортируйте в файл кода пространство имен `System.Data`.)

```
static void MultiIndexerWithDataTable()
{
    // Создать простой объект DataTable с тремя столбцами.
    DataTable myTable = new DataTable();
    myTable.Columns.Add(new DataColumn("FirstName"));
    myTable.Columns.Add(new DataColumn("LastName"));
    myTable.Columns.Add(new DataColumn("Age"));
    // Добавить строку в таблицу.
    myTable.Rows.Add("Mel", "Appleby", 60);
    // Использовать многомерный индексатор для вывода деталей первой строки.
    Console.WriteLine("First Name: {0}", myTable.Rows[0][0]);
    Console.WriteLine("Last Name: {0}", myTable.Rows[0][1]);
    Console.WriteLine("Age : {0}", myTable.Rows[0][2]);
}
```


Начиная с главы 21, мы продолжим рассмотрение ADO.NET, так что не пугайтесь, если что-то в приведенном выше коде выглядит незнакомым. Пример просто иллюстрирует, что методы индексаторов способны поддерживать множество измерений, а при правильном применении могут упростить взаимодействие с объектами, содержащимися в специальных коллекциях.

Определения индексаторов в интерфейсных типах

Индексаторы могут определяться в выбранном типе интерфейса .NET, чтобы позволить поддерживающим типам предоставлять специальные реализации. Ниже показан простой пример интерфейса, который задает протокол для получения строковых объектов с использованием числового индексатора:

```
public interface IStringContainer
{
    string this[int index] { get; set; }
}
```

При таком определении интерфейса любой класс или структура, которые его реализуют, должны поддерживать индексатор с чтением/записью, манипулирующий элементами с применением числового значения. Вот частичная реализация примера класса:

```
class SomeClass : IStringContainer
{
    private List<string> myStrings = new List<string>();
    public string this[int index]
    {
        get => myStrings[index];
        set => myStrings.Insert(index, value);
    }
}
```

На этом первая крупная тема настоящей главы завершена. А теперь давайте перейдем к исследованиям языкового средства, которое позволяет строить специальные классы и структуры, уникальным образом реагирующие на внутренние операции C#. Итак, займемся концепцией *перегрузки операций*.

Понятие перегрузки операций

Как и любой язык программирования, C# имеет заготовленный набор лексем, используемых для выполнения базовых операций над встроенными типами. Например, вы знаете, что операция + может применяться к двум целым числам, чтобы получить большее целое число:

```
// Операция + с целыми числами.
int a = 100;
int b = 240;
int c = a + b; // c теперь имеет значение 340
```

Опять-таки, здесь нет ничего нового, но задумывались ли вы когда-нибудь о том, что одну и ту же операцию + разрешено использовать с большинством встроенных типов данных C#? Скажем, взгляните на следующий код:

```
// Операция + со строками.
string s1 = "Hello";
string s2 = " World!";
string s3 = s1 + s2; // s3 теперь имеет значение "Hello World!"
```

В сущности, операция `+` функционирует специфическим образом на основе типа предоставленных данных (в рассматриваемом случае строкового или целочисленного). Когда операция `+` применяется к числовым типам, в результате выполняется суммирование операндов, а когда к строковым типам — то конкатенация строк.

Язык C# дает возможность строить специальные классы и структуры, которые также уникально реагируют на один и тот же набор базовых лексем (вроде операции `+`). Хотя не каждая операция C# может быть перегружена, перегрузку допускают многие операции (табл. 11.1).

Таблица 11.1. Возможность перегрузки операций C#

Операция C#	Возможность перегрузки
<code>+, -, !, ~, ++, --, true, false</code>	Эти унарные операции могут быть перегружены
<code>+, -, *, /, %, &, , ^, <<, >></code>	Эти бинарные операции могут быть перегружены
<code>==, !=, <, >, <=, >=</code>	Эти операции сравнения могут быть перегружены. Язык C# требует совместной перегрузки “похожих” операций (т.е. <code>< и ></code> , <code><= и >=</code> , <code>== и !=</code>)
<code>[]</code>	Операция <code>[]</code> не может быть перегружена. Однако, как было показано ранее в главе, ту же самую функциональность обеспечивает индексатор
<code>()</code>	Операция <code>()</code> не может быть перегружена. Тем не менее, как вы увидите далее в главе, ту же самую функциональность предоставляют специальные методы преобразования
<code>+=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=</code>	Сокращенные операции присваивания не могут быть перегружены; однако вы получаете их автоматически, когда перегружаете соответствующие бинарные операции

Перегрузка бинарных операций

Чтобы проиллюстрировать процесс перегрузки бинарных операций, рассмотрим приведенный ниже простой класс `Point`, который определен в новом проекте консольного приложения по имени `OverloadedOps`:

```
// Простой будничный класс C#.
public class Point
{
    public int X {get; set;}
    public int Y {get; set;}

    public Point(int xPos, int yPos)
    {
        X = xPos;
        Y = yPos;
    }

    public override string ToString() => $"[{this.X}, {this.Y}]";
}
```

Рассуждая логически, суммирование объектов `Point` имеет смысл. Например, сложение двух переменных `Point` должно давать новый объект `Point` с просуммированными значениями свойств `X` и `Y`. Конечно, полезно также и вычитать один объект `Point` из другого. В идеале желательно иметь возможность записи примерно такого кода:

```
// Сложение и вычитание двух точек?
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Overloaded Operators *****\n");

    // Создать две точки.
    Point ptOne = new Point(100, 100);
    Point ptTwo = new Point(40, 40);
    Console.WriteLine("ptOne = {0}", ptOne);
    Console.WriteLine("ptTwo = {0}", ptTwo);

    // Сложить две точки, чтобы получить большую?
    Console.WriteLine("ptOne + ptTwo: {0} ", ptOne + ptTwo);

    // Вычесть одну точку из другой, чтобы получить меньшую?
    Console.WriteLine("ptOne - ptTwo: {0} ", ptOne - ptTwo);
    Console.ReadLine();
}
```

Тем не менее, с существующим видом класса `Point` вы получите ошибки на этапе компиляции, потому что типу `Point` не известно, как реагировать на операции `+` и `-`. Для оснащения специального типа способностью уникально реагировать на встроенные операции язык C# предлагает ключевое слово `operator`, которое может использоваться только в сочетании с ключевым словом `static`. При перегрузке бинарной операции (такой как `+` и `-`) вы чаще всего будете передавать два аргумента того же типа, что и класс, определяющий операцию (`Point` в этом примере):

```
// Более интеллектуальный тип Point.
public class Point
{
    ...
    // Перегруженная операция +.
    public static Point operator + (Point p1, Point p2) =>
        new Point(p1.X + p2.X, p1.Y + p2.Y);

    // Перегруженная операция -.
    public static Point operator - (Point p1, Point p2) =>
        new Point(p1.X - p2.X, p1.Y - p2.Y);
}
```

Логика, положенная в основу операции `+`, предусматривает просто возвращение нового объекта `Point`, основанного на сложении соответствующих полей входных параметров `Point`. Таким образом, когда вы пишете `p1 + p2`, "за кулисами" происходит следующий скрытый вызов статического метода `operator +`:

```
// Псевдокод: Point p3 = Point.operator+ (p1, p2)
Point p3 = p1 + p2;
```

Аналогично выражение `p1 - p2` отображается так:

```
// Псевдокод: Point p4 = Point.operator- (p1, p2)
Point p4 = p1 - p2;
```

После произведенной модификации типа `Point` программа скомпилируется и появится возможность сложения и вычитания объектов `Point`, что подтверждает представленный далее вывод:

```
ptOne = [100, 100]
ptTwo = [40, 40]
ptOne + ptTwo: [140, 140]
ptOne - ptTwo: [60, 60]
```

При перегрузке бинарной операции передавать ей два параметра того же самого типа не обязательно. Если это имеет смысл, тогда один из аргументов может относиться к другому типу. Например, ниже показана перегруженная операция `+`, которая позволяет вызывающему коду получить новый объект `Point` на основе числовой коррекции:

```
public class Point
{
    ...
    public static Point operator + (Point p1, int change) =>
        new Point(p1.X + change, p1.Y + change);
    public static Point operator + (int change, Point p1) =>
        new Point(p1.X + change, p1.Y + change);
}
```

Обратите внимание, что если вы хотите передавать аргументы в любом порядке, то потребуется реализовать обе версии метода (т.е. нельзя просто определить один из методов и рассчитывать, что компилятор автоматически будет поддерживать другой). Теперь новые версии операции `+` можно применять следующим образом:

```
// Выводит [110, 110].
Point biggerPoint = ptOne + 10;
Console.WriteLine("ptOne + 10 = {0}", biggerPoint);

// Выводит [120, 120].
Console.WriteLine("10 + biggerPoint = {0}", 10 + biggerPoint);
Console.WriteLine();
```

А как насчет операций `+=` и `--`?

Если до перехода на C# вы имели дело с языком C++, тогда вас может удивить отсутствие возможности перегрузки операций сокращенного присваивания (`+=`, `--` и т.д.). Не беспокойтесь. В C# операции сокращенного присваивания автоматически эмулируются в случае перегрузки связанных бинарных операций. Таким образом, если в классе `Point` уже перегружены операции `+` и `-`, то можно написать приведенный далее код:

```
// Перегрузка бинарных операций автоматически обеспечивает
// перегрузку сокращенных операций.
static void Main(string[] args)
{
    ...
    // Операция += перегружена автоматически.
    Point ptThree = new Point(90, 5);
    Console.WriteLine("ptThree = {0}", ptThree);
    Console.WriteLine("ptThree += ptTwo: {0}", ptThree += ptTwo);

    // Операция -= перегружена автоматически.
    Point ptFour = new Point(0, 500);
    Console.WriteLine("ptFour = {0}", ptFour);
    Console.WriteLine("ptFour -= ptThree: {0}", ptFour -= ptThree);
    Console.ReadLine();
}
```

Перегрузка унарных операций

В C# также разрешено перегружать унарные операции, такие как `++` и `--`. При перегрузке унарной операции также должно использоваться ключевое слово `static` с ключевым словом `operator`, но в этом случае просто передается единственный параметр того же типа, что и класс или структура, где операция определена.

Например, дополним реализацию `Point` следующими перегруженными операциями:

```
public class Point
{
    ...
    // Добавить 1 к значениям X/Y входного объекта Point.
    public static Point operator ++(Point p1) => new Point(p1.X+1, p1.Y+1);
    // Вычесть 1 из значений X/Y входного объекта Point.
    public static Point operator --(Point p1) => new Point(p1.X-1, p1.Y-1);
}
```

В результате появляется возможность инкрементировать и декрементировать значения `X` и `Y` класса `Point`:

```
static void Main(string[] args)
{
    ...
    // Применение унарных операций ++ и -- к объекту Point.
    Point ptFive = new Point(1, 1);
    Console.WriteLine("++ptFive = {0}", ++ptFive);    // [2, 2]
    Console.WriteLine("--ptFive = {0}", --ptFive);    // [1, 1]

    // Применение тех же операций в виде постфиксного инкремента/декремента.
    Point ptSix = new Point(20, 20);
    Console.WriteLine("ptSix++ = {0}", ptSix++);      // [20, 20]
    Console.WriteLine("ptSix-- = {0}", ptSix--);      // [21, 21]
    Console.ReadLine();
}
```

В предыдущем примере кода специальные операции `++` и `--` применяются двумя разными способами. В языке C++ допускается перегружать операции префиксного и постфиксного инкремента/декремента по отдельности. В C# это невозможно. Однако возвращаемое значение инкремента/декремента автоматически обрабатывается корректно (т.е. для перегруженной операции `++` выражение `pt++` дает значение неизмененного объекта, в то время как `++pt` — новое значение, устанавливаемое перед использованием в выражении).

Перегрузка операций эквивалентности

Как упоминалось в главе 6, метод `System.Object.Equals()` может быть перегружен для выполнения сравнений на основе значений (а не ссылок) между ссылочными типами. Если вы решили переопределить `Equals()` (часто вместе со связанным методом `System.Object.GetHashCode()`), то легко переопределите и операции проверки эквивалентности (`==` и `!=`). Взгляните на обновленный тип `Point`:

```
// В данной версии типа Point также перегружены операции == и !=.
public class Point
{
    ...
    public override bool Equals(object o) => o.ToString() == this.ToString();
    public override int GetHashCode() => this.ToString().GetHashCode();
    // Теперь перегрузить операции == и !=.
    public static bool operator ==(Point p1, Point p2) => p1.Equals(p2);
    public static bool operator !=(Point p1, Point p2) => !p1.Equals(p2);
}
```

Обратите внимание, что для выполнения всей работы в реализациях операций `==` и `!=` просто вызывается перегруженный метод `Equals()`. Вот как теперь может применяться класс `Point`:

```
// Использование перегруженных операций эквивалентности.
static void Main(string[] args)
{
    ...
    Console.WriteLine("ptOne == ptTwo : {0}", ptOne == ptTwo);
    Console.WriteLine("ptOne != ptTwo : {0}", ptOne != ptTwo);
    Console.ReadLine();
}
```

Как видите, сравнение двух объектов с использованием хорошо знакомых операций `==` и `!=` выглядит намного интуитивно понятнее, чем вызов метода `Object.Equals()`. При перегрузке операций эквивалентности для определенного класса имейте в виду, что C# требует, чтобы в случае перегрузки операции `==` *обязательно* перегружалась также и операция `!=` (компилятор напомним, если вы забудете это сделать).

Перегрузка операций сравнения

В главе 8 было показано, каким образом реализовывать интерфейс `Comparable` для сравнения двух похожих объектов. В действительности для того же самого класса можно также перегрузить операции сравнения (`<`, `>`, `<=` и `>=`). Как и в случае операций эквивалентности, язык C# требует, чтобы при перегрузке операции `<` *обязательно* перегружалась также операция `>`. Если класс `Point` перегружает указанные операции сравнения, тогда пользователь объекта может сравнивать объекты `Point`:

```
// Использование перегруженных операций < и >.
static void Main(string[] args)
{
    ...
    Console.WriteLine("ptOne < ptTwo : {0}", ptOne < ptTwo);
    Console.WriteLine("ptOne > ptTwo : {0}", ptOne > ptTwo);
    Console.ReadLine();
}
```

Когда интерфейс `Comparable` (или, что еще лучше, его обобщенный эквивалент) реализован, перегрузка операций сравнения становится тривиальной. Вот модифицированное определение класса:

```
// Объекты Point также можно сравнивать посредством операций сравнения.
public class Point : Comparable<Point>
{
    ...
    public int CompareTo(Point other)
    {
        if (this.X > other.X && this.Y > other.Y)
            return 1;
        if (this.X < other.X && this.Y < other.Y)
            return -1;
        else
            return 0;
    }
    public static bool operator <(Point p1, Point p2) => p1.CompareTo(p2) < 0;
    public static bool operator >(Point p1, Point p2) => p1.CompareTo(p2) > 0;
    public static bool operator <=(Point p1, Point p2) => p1.CompareTo(p2) <= 0;
    public static bool operator >=(Point p1, Point p2) => p1.CompareTo(p2) >= 0;
}
```

Финальные соображения относительно перегрузки операций

Как уже упоминалось, язык C# предоставляет возможность построения типов, которые могут уникальным образом реагировать на разнообразные встроенные хорошо известные операции. Перед добавлением поддержки такого поведения в классы вы должны удостовериться в том, что операции, которые планируется перегружать, имеют какой-нибудь смысл в реальности.

Например, пусть перегружена операция умножения для класса `MiniVan`, представляющего минивэн. Что по своей сути будет означать перемножение двух объектов `MiniVan`? В нем нет особого смысла. На самом деле коллеги по команде даже могут быть озадачены, когда увидят следующее применение класса `MiniVan`:

```
// Что?! Понять это непросто...
MiniVan newVan = myVan * yourVan;
```

Перегрузка операций обычно полезна только при построении атомарных типов данных. Текст, точки, прямоугольники, функции и шестиугольники — подходящие кандидаты на перегрузку операций, но люди, менеджеры, автомобили, подключения к базе данных и веб-страницы — нет. В качестве эмпирического правила запомните, что если перегруженная операция *затрудняет* понимание пользователем функциональности типа, то не перегружайте ее. Используйте такую возможность с умом.

Исходный код. Проект `OverloadedOps` доступен в подкаталоге `Chapter_11`.

Понятие специальных преобразований типов

Давайте теперь обратимся к теме, тесно связанной с перегрузкой операций — специальные преобразования типов. Чтобы заложить фундамент для последующего обсуждения, кратко вспомним понятие явных и неявных преобразований между числовыми данными и связанными типами классов.

Повторение: числовые преобразования

В терминах встроенных числовых типов (`sbyte`, `int`, `float` и т.д.) *явное преобразование* требуется, когда вы пытаетесь сохранить большее значение в контейнере меньшего размера, т.к. подобное действие может привести к утере данных. По существу тем самым вы сообщаете компилятору, что отдаете себе отчет в том, что делаете. И наоборот — *неявное преобразование* происходит автоматически, когда вы пытаетесь поместить меньший тип в больший целевой тип, что не должно вызвать потерю данных:

```
static void Main()
{
    int a = 123;
    long b = a;           // Неявное преобразование из int в long.
    int c = (int) b;      // Явное преобразование из long в int.
}
```

Повторение: преобразования между связанными типами классов

В главе 6 было показано, что типы классов могут быть связаны классическим наследованием (отношение “является”). В таком случае процесс преобразования C# позволяет осуществлять приведение вверх и вниз по иерархии классов. Например, производный класс всегда может быть неявно приведен к базовому классу. Тем не менее, если вы хотите сохранить объект базового класса в переменной производного класса, то должны выполнить явное приведение:

```
// Два связанных типа классов.
class Base{}
class Derived : Base{}

class Program
{
    static void Main(string[] args)
    {
        // Неявное приведение производного класса к базовому.
        Base myBaseType;
        myBaseType = new Derived();
        // Для сохранения ссылки на базовый класс в переменной
        // производного класса требуется явное преобразование.
        Derived myDerivedType = (Derived)myBaseType;
    }
}
```

Продemonстрированное явное приведение работает из-за того, что классы Base и Derived связаны классическим наследованием. Однако что если есть два типа классов в *разных иерархиях* без общего предка (кроме System.Object), которые требуют преобразований? Учитывая, что они не связаны классическим наследованием, типичные операции приведения здесь не помогут (и вдобавок компилятор сообщит об ошибке).

В качестве связанного замечания обратимся к типам значений (структурам). Предположим, что имеются две структуры .NET с именами Square и Rectangle. Поскольку они не могут задействовать классическое наследование (т.к. запечатаны), не существует естественного способа выполнить приведение между этими по внешнему виду связанными типами.

Несмотря на то что в структурах можно было бы создать вспомогательные методы (наподобие Rectangle.ToSquare()), язык C# позволяет строить специальные процедуры преобразования, которые дают типам возможность реагировать на операцию приведения (). Следовательно, если корректно сконфигурировать структуры, тогда для явного преобразования между ними можно будет применять такой синтаксис:

```
// Преобразовать Rectangle в Square!
Rectangle rect = new Rectangle
{
    Width = 3;
    Height = 10;
}
Square sq = (Square)rect;
```

Создание специальных процедур преобразования

Начнем с создания нового проекта консольного приложения по имени CustomConversions. В языке C# предусмотрены два ключевых слова, explicit и implicit, которые можно использовать для управления тем, как типы должны реагировать на попытку преобразования. Предположим, что есть следующие определения структур:

```
public struct Rectangle
{
    public int Width {get; set;}
    public int Height {get; set;}
    public Rectangle(int w, int h) : this()
    {
        Width = w; Height = h;
    }
}
```



```

public void Draw()
{
    for (int i = 0; i < Height; i++)
    {
        for (int j = 0; j < Width; j++)
        {
            Console.Write("*");
        }
        Console.WriteLine();
    }
}

public override string ToString() => $"[Width = {Width}; Height = {Height}]";
}

public struct Square
{
    public int Length {get; set;}
    public Square(int l) : this()
    {
        Length = l;
    }
    public void Draw()
    {
        for (int i = 0; i < Length; i++)
        {
            for (int j = 0; j < Length; j++)
            {
                Console.Write("*");
            }
            Console.WriteLine();
        }
    }
    public override string ToString() => $"[Length = {Length}]", Length);
    // Rectangle можно явно преобразовывать в Square.
    public static explicit operator Square(Rectangle r)
    {
        Square s = new Square {Length = r.Height};
        return s;
    }
}

```

На заметку! Вы заметите, что в конструкторах `Square` и `Rectangle` производится явное связывание в цепочку со стандартным конструктором. Дело в том, что когда в структуре применяется синтаксис автоматических свойств (как в рассматриваемом случае), внутри всех специальных конструкторов должен явно вызываться стандартный конструктор для инициализации закрытых поддерживающих полей. (Например, если структуры имеют любые дополнительные поля/свойства, то данный стандартный конструктор будет их инициализировать стандартными значениями.) Да, это необычное правило C#, но ведь настоящая глава посвящена сложным темам.

Обратите внимание, что в текущей версии типа `Square` определена явная операция преобразования. Подобно перегрузке операций процедуры преобразования используют ключевое слово `operator` в сочетании с ключевым словом `explicit` или `implicit` и должны быть определены как `static`. Входным параметром является сущность, из которой выполняется преобразование, а типом операции — сущность, в которую производится преобразование.

В данном случае предположение заключается в том, что квадрат (будучи геометрической фигурой с четырьмя сторонами равной длины) может быть получен из высоты прямоугольника. Таким образом, вот как преобразовать `Rectangle` в `Square`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Conversions *****\n");
    // Создать экземпляр Rectangle.
    Rectangle r = new Rectangle(15, 4);
    Console.WriteLine(r.ToString());
    r.Draw();

    Console.WriteLine();
    // Преобразовать r в Square на основе высоты Rectangle.
    Square s = (Square)r;
    Console.WriteLine(s.ToString());
    s.Draw();
    Console.ReadLine();
}
```

Ниже показан вывод:

```
***** Fun with Conversions *****
[Width = 15; Height = 4]
*****
*****
*****
*****

[Length = 4]
****
****
****
****
```

Хотя преобразование `Rectangle` в `Square` в пределах той же самой области действия может быть не особенно полезным, взгляните на следующий метод, который спроектирован для приема параметров типа `Square`:

```
// Этот метод требует параметр типа Square.
static void DrawSquare(Square sq)
{
    Console.WriteLine(sq.ToString());
    sq.Draw();
}
```

Благодаря наличию операции явного преобразования в типе `Square` методу `DrawSquare()` на обработку можно передавать типы `Rectangle`, применяя явное приведение:

```
static void Main(string[] args)
{
    ...
    // Преобразовать Rectangle в Square для вызова метода.
    Rectangle rect = new Rectangle(10, 5);
    DrawSquare((Square)rect);
    Console.ReadLine();
}
```

Дополнительные явные преобразования для типа Square

Теперь, когда экземпляры `Rectangle` можно явно преобразовывать в экземпляры `Square`, давайте рассмотрим несколько дополнительных явных преобразований. Учитывая, что квадрат симметричен по всем сторонам, полезно предусмотреть процедуру преобразования, которая позволит вызывающему коду привести целочисленный тип к типу `Square` (который, естественно, будет иметь длину стороны, равную переданному целочисленному значению). А что если вы захотите модифицировать еще и `Square` так, чтобы вызывающий код мог выполнять приведение из `Square` в `int`? Вот как выглядит логика вызова:

```
static void Main(string[] args)
{
    ...
    // Преобразование int в Square.
    Square sq2 = (Square)90;
    Console.WriteLine("sq2 = {0}", sq2);
    // Преобразование Square в int.
    int side = (int)sq2;
    Console.WriteLine("Side length of sq2 = {0}", side);
    Console.ReadLine();
}
```

Ниже показаны изменения, внесенные в структуру `Square`:

```
public struct Square
{
    ...
    public static explicit operator Square(int sideLength)
    {
        Square newSq = new Square {Length = sideLength};
        return newSq;
    }

    public static explicit operator int (Square s) => s.Length;
}
```

По правде говоря, преобразование `Square` в `int` может показаться не слишком интуитивно понятной (или полезной) операцией (скорее всего, вы просто будете передавать нужные значения конструктору). Тем не менее, оно указывает на важный факт, касающийся процедур специальных преобразований: компилятор не беспокоится о том, из чего и во что происходит преобразование, до тех пор, пока вы пишете синтаксически корректный код.

Таким образом, как и с перегрузкой операций, возможность создания операции явного приведения для заданного типа вовсе не означает *необходимость* ее создания. Обычно этот прием будет наиболее полезным при создании типов структур .NET, учитывая, что они не могут принимать участие в классическом наследовании (где приведение обеспечивается автоматически).

Определение процедур неявного преобразования

До сих пор мы создавали различные специальные операции *явного* преобразования. Но что насчет следующего *неявного* преобразования?

```
static void Main(string[] args)
{
    ...
    Square s3 = new Square {Length = 83};
```

```
// Попытка сделать неявное приведение?
Rectangle rect2 = s3;

Console.ReadLine();
}
```

Данный код не скомпилируется, т.к. вы не предоставили процедуру неявного преобразования для типа `Rectangle`. Ловушка здесь вот в чем: определять одновременно функции явного и неявного преобразования не разрешено, если они не различаются по типу возвращаемого значения или по списку параметров. Это может показаться ограничением; однако вторая ловушка связана с тем, что когда тип определяет процедуру неявного преобразования, то вызывающий код вполне законно может использовать синтаксис *явного* приведения!

Запутались? Чтобы прояснить ситуацию, давайте добавим к структуре `Rectangle` процедуру неявного преобразования с применением ключевого слова `implicit` (обратите внимание, что в показанном ниже коде предполагается, что ширина результирующего прямоугольника вычисляется умножением стороны квадрата на 2):

```
public struct Rectangle
{
    ...
    public static implicit operator Rectangle(Square s)
    {
        Rectangle r = new Rectangle
        {
            Height = s.Length,
            Width = s.Length * 2      // Предположим, что ширина нового квадрата
                                     // будет равна (Length x 2).
        };
        return r;
    }
}
```

После такой модификации можно выполнять преобразование между типами:

```
static void Main(string[] args)
{
    ...
    // Неявное преобразование работает!
    Square s3 = new Square { Length= 7 };
    Rectangle rect2 = s3;
    Console.WriteLine("rect2 = {0}", rect2);

    // Синтаксис явного приведения также работает!
    Square s4 = new Square { Length = 3 };
    Rectangle rect3 = (Rectangle)s4;

    Console.WriteLine("rect3 = {0}", rect3);
    Console.ReadLine();
}
```

Обзор определения операций специального преобразования завершен. Как и с перегруженными операциями, помните о том, что данный фрагмент синтаксиса представляет собой просто сокращенное обозначение для "нормальных" функций-членов и потому всегда необязателен. Тем не менее, в случае правильного использования специальные структуры могут применяться более естественным образом, поскольку будут трактоваться как настоящие типы классов, связанные наследованием.

Понятие расширяющих методов

В версии .NET 3.5 появилась концепция *расширяющих методов*, которая позволила добавлять новые методы или свойства к классу либо структуре, не модифицируя исходный тип непосредственно. Когда такой прием может оказаться полезным? Рассмотрим следующие ситуации.

Пусть есть класс, находящийся в производстве. Со временем выясняется, что имеющийся класс должен поддерживать несколько новых членов. Изменение текущего определения класса напрямую сопряжено с риском нарушения обратной совместимости со старыми кодовыми базами, использующими его, т.к. они могут не скомпилироваться с последним улучшенным определением класса. Один из способов обеспечения обратной совместимости предусматривает создание нового класса, производного от существующего, но тогда придется сопровождать два класса. Как все мы знаем, сопровождение кода является самой скучной частью деятельности разработчика программного обеспечения.

Представим другую ситуацию. Предположим, что имеется структура (или, может быть, запечатанный класс), и необходимо добавить новые члены, чтобы получить полиморфное поведение в рамках системы. Поскольку структуры и запечатанные классы не могут быть расширены, единственный выбор заключается в том, чтобы добавить желаемые члены к типу, снова рискуя нарушить обратную совместимость!

За счет применения расширяющих методов появляется возможность модифицировать типы, не создавая подклассов и не изменяя код типа напрямую. Загвоздка в том, что новая функциональность предлагается типом, только если в текущем проекте будут присутствовать ссылки на расширяющие методы.

Определение расширяющих методов

Первое ограничение, связанное с расширяющими методами, состоит в том, что они должны быть определены внутри статического класса (см. главу 5), а потому каждый расширяющий метод должен объявляться с ключевым словом `static`. Вторая проблема в том, что все расширяющие методы помечаются как таковые посредством ключевого слова `this` в качестве модификатора первого (и только первого) параметра заданного метода. Параметр, помеченный с помощью `this`, представляет расширяемый элемент.

В целях иллюстрации создадим новый проект консольного приложения под названием `ExtensionMethods`. Предположим, что создается класс по имени `MyExtensions`, в котором определены два расширяющих метода. Первый расширяющий метод позволяет объекту любого типа взаимодействовать с новым методом `DisplayDefiningAssembly()`, который использует типы из пространства имен `System.Reflection` для отображения имени сборки, содержащей данный тип.

На заметку! API-интерфейс рефлексии формально рассматривается в главе 15. Если эта тема для вас нова, тогда просто запомните, что рефлексия позволяет исследовать структуру сборок, типов и членов типов во время выполнения.

Второй расширяющий метод по имени `ReverseDigits()` позволяет любому значению типа `int` получить новую версию самого себя с обратным порядком следования цифр. Например, если целочисленное значение 1234 вызывает `ReverseDigits()`, то в результате возвратится 4321. Взгляните на следующую реализацию класса (не забудьте импортировать пространство имен `System.Reflection`):

```
static class MyExtensions
{
    // Этот метод позволяет объекту любого типа
    // отобразить сборку, в которой он определен.
```

```

public static void DisplayDefiningAssembly(this object obj)
{
    Console.WriteLine("{0} lives here: => {1}\n", obj.GetType().Name,
        Assembly.GetAssembly(obj.GetType()).GetName().Name);
}

// Этот метод позволяет любому целочисленному значению изменить порядок
// следования десятичных цифр на обратный. Например, для 56 возвратится 65.
public static int ReverseDigits(this int i)
{
    // Транслировать int в string и затем получить все его символы.
    char[] digits = i.ToString().ToCharArray();

    // Изменить порядок следования элементов массива.
    Array.Reverse(digits);

    // Поместить обратно в строку.
    string newDigits = new string(digits);

    // Возвратить модифицированную строку как int.
    return int.Parse(newDigits);
}
}

```

Снова обратите внимание на то, что первый параметр каждого расширяющего метода снабжен ключевым словом `this`, находящимся перед определением типа параметра. Первый параметр расширяющего метода всегда представляет расширяемый тип. Учитывая, что метод `DisplayDefiningAssembly()` был прототипирован для расширения `System.Object`, этот новый член теперь присутствует в каждом типе, поскольку `Object` является родительским для всех типов платформы .NET. Однако метод `ReverseDigits()` прототипирован для расширения только целочисленных типов, и потому если к нему обращается какое-то другое значение, то возникнет ошибка на этапе компиляции.

На заметку! Запомните, что каждый расширяющий метод может иметь множество параметров, но только первый параметр разрешено пометить посредством `this`. Дополнительные параметры будут трактоваться как нормальные входные параметры, применяемые методом.

Вызов расширяющих методов

Располагая созданными расширяющими методами, рассмотрим следующий метод `Main()`, который их использует с разнообразными типами из библиотек базовых классов:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Extension Methods *****\n");

    // В int появилась новая отличительная черта!
    int myInt = 12345678;
    myInt.DisplayDefiningAssembly();

    // То же и в DataSet!
    System.Data.DataSet d = new System.Data.DataSet();
    d.DisplayDefiningAssembly();

    // И в SoundPlayer!
    System.Media.SoundPlayer sp = new System.Media.SoundPlayer();
    sp.DisplayDefiningAssembly();

    // Использовать новую функциональность int.
}

```

```

Console.WriteLine("Value of myInt: {0}", myInt);
Console.WriteLine("Reversed digits of myInt: {0}", myInt.ReverseDigits());
Console.ReadLine();
}

```

Ниже показан вывод:

```

***** Fun with Extension Methods *****
Int32 lives here: => mscorlib
DataSet lives here: => System.Data
SoundPlayer lives here: => System
Value of myInt: 12345678
Reversed digits of myInt: 87654321

```

Импортирование расширяющих методов

Когда определяется класс, содержащий расширяющие методы, он вне всяких сомнений будет принадлежать какому-то пространству имен .NET. Если это пространство имен отличается от пространства имен, где расширяющие методы применяются, тогда придется использовать ключевое слово `using` языка C#, которое позволит файлу кода иметь доступ ко всем расширяющим методам интересующего типа. Об этом важно помнить, потому что если не импортировать явно корректное пространство имен, то в таком файле кода C# расширяющие методы окажутся недоступными.

Хотя на первый взгляд может показаться, что расширяющие методы глобальны по своей природе, на самом деле они ограничены пространствами имен, где определены, или пространствами имен, которые их импортируют. Таким образом, если вы поместите класс `MyExtensions` в пространство имен `MyExtensionMethods`, как показано ниже:

```

namespace MyExtensionMethods
{
    static class MyExtensions
    {
        ...
    }
}

```

то другим пространствам имен в проекте придется явно импортировать `MyExtensionMethods` для получения расширяющих методов, определенных вашим классом.

На заметку! Обычная практика предусматривает изоляцию расширяющих методов не только в отдельном пространстве имен .NET, но также в отдельной библиотеке классов. В таком случае новые приложения могут обращаться к расширяющим методам путем явной ссылки на подходящую библиотеку и импортирования пространства имен. В главе 14 будут представлены детали построения и применения специальных библиотек классов .NET.

Поддержка расширяющих методов средством IntelliSense

Учитывая то, что расширяющие методы не определены буквально в расширяемом типе, при чтении кода есть шанс запутаться. Например, предположим, что вы импортировали пространство имен, в котором определено несколько расширяющих методов, написанных кем-то из команды разработчиков. При написании своего кода вы можете создать переменную расширенного типа, применить операцию точки и обнаружить десятки новых методов, которые не являются членами исходного определения класса!

К счастью, средство IntelliSense в Visual Studio помечает все расширяющие методы, как показано на рис. 11.1.

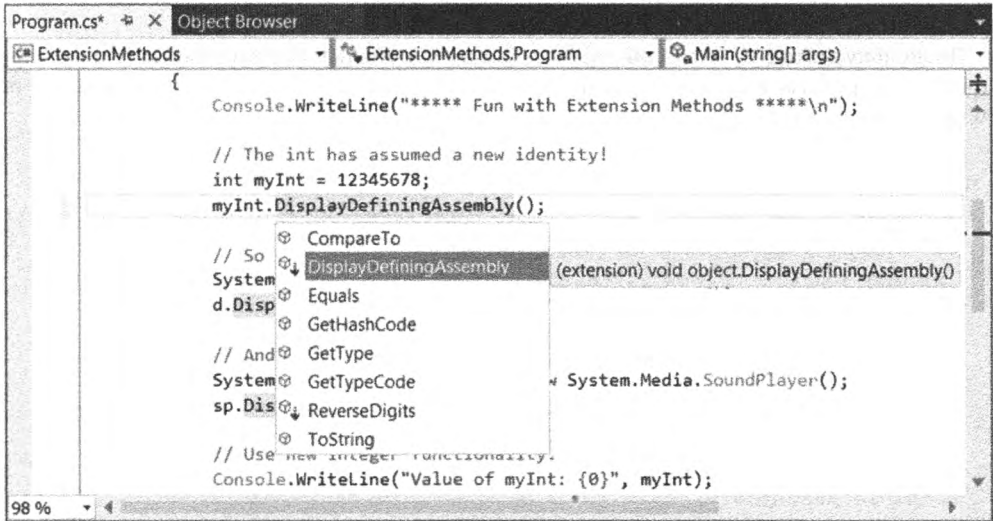


Рис. 11.1. Отображение расширяющих методов в IntelliSense

Любой метод, помеченный подобным образом, определен за пределами исходного определения класса в виде расширяющего метода.

Исходный код. Проект ExtensionMethods доступен в подкаталоге Chapter_11.

Расширение типов, реализующих специфичные интерфейсы

К настоящему моменту вы видели, как расширять классы (и косвенно структуры, которые следуют тому же синтаксису) новой функциональностью через расширяющие методы. Также есть возможность определить расширяющий метод, который способен расширять только класс или структуру, реализующую корректный интерфейс. Например, можно было бы заявить следующее: если класс или структура реализует интерфейс `IEnumerable<T>`, тогда этот тип получит новые члены. Разумеется, вполне допустимо требовать, чтобы тип поддерживал вообще любой интерфейс, включая ваши специальные интерфейсы.

Для примера создадим новый проект консольного приложения по имени `InterfaceExtensions`. Цель здесь заключается в том, чтобы добавить новый метод к любому типу, который реализует интерфейс `IEnumerable`, что охватывает все массивы и многие классы необобщенных коллекций (вспомните из главы 8, что обобщенный интерфейс `IEnumerable<T>` расширяет необобщенный интерфейс `IEnumerable`). Добавьте в проект следующий расширяющий класс:

```
static class AnnoyingExtensions
{
    public static void PrintDataAndBeep(this System.Collections.IEnumerable iterator)
    {
        foreach (var item in iterator)
        {
            Console.WriteLine(item);
        }
    }
}
```



```

        Console.Beep();
    }
}

```

Поскольку метод `PrintDataAndBeep()` может использоваться любым классом или структурой, реализующей интерфейс `IEnumerable`, мы можем протестировать его с помощью такого метода `Main()`:

```

static void Main( string[] args )
{
    Console.WriteLine("***** Extending Interface Compatible Types *****\n");
    // System.Array реализует IEnumerable!
    string[] data = { "Wow", "this", "is", "sort", "of", "annoying",
        "but", "in", "a", "weird", "way", "fun!" };
    data.PrintDataAndBeep();
    Console.WriteLine();
    // List<T> реализует IEnumerable!
    List<int> myInts = new List<int>() {10, 15, 20};
    myInts.PrintDataAndBeep();
    Console.ReadLine();
}

```

Итак, исследование расширяющих методов C# завершено. Помните, что это конкретное языковое средство полезно, когда необходимо расширить функциональность типа, но вы не хотите создавать подклассы (или не можете, если тип запечатан) в целях обеспечения полиморфизма. Как вы увидите позже, расширяющие методы играют ключевую роль в API-интерфейсах LINQ. На самом деле вы узнаете, что в API-интерфейсах LINQ одним из самых часто расширяемых элементов является класс или структура, реализующая обобщенную версию интерфейса `IEnumerable`.

Исходный код. Проект `InterfaceExtensions` доступен в подкаталоге `Chapter_11`.

Понятие анонимных типов

Программистам на объектно-ориентированных языках хорошо известны преимущества определения классов для представления состояния и функциональности заданного элемента, который требуется моделировать. Всякий раз, когда необходимо определить класс, который предназначен для многократного применения и предоставляет обширную функциональность через набор методов, событий, свойств и специальных конструкторов, устоявшаяся практика предусматривает создание нового класса C#.

Тем не менее, возникают и другие ситуации, когда желательно определять класс просто в целях моделирования набора инкапсулированных (и каким-то образом связанных) элементов данных безо всяких ассоциированных методов, событий или другой специализированной функциональности. Кроме того, что если такой тип должен использоваться только небольшим набором методов внутри программы? Было бы довольно утомительно строить полное определение класса вроде показанного ниже, если хорошо известно, что класс будет применяться только в нескольких местах. Чтобы подчеркнуть данный момент, вот примерный план того, что может понадобиться делать, когда нужно создать “простой” тип данных, который следует обычной семантике на основе значений:

```

class SomeClass
{
    // Определить набор закрытых переменных-членов...
    // Создать свойство для каждой закрытой переменной...
    // Переопределить метод ToString() для учета основных переменных-членов...
    // Переопределить методы GetHashCode() и Equals()
    // для работы с эквивалентностью на основе значений...
}

```

Как видите, задача не обязательно оказывается настолько простой. Вам потребуется не только написать большой объем кода, но еще и сопровождать дополнительный класс в системе. Для временных данных подобного рода было бы удобно формировать специальный тип на лету. Например, пусть необходимо построить специальный метод, который принимает какой-то набор входных параметров. Такие параметры нужно использовать для создания нового типа данных, который будет применяться внутри области действия метода. Вдобавок желательно иметь возможность быстрого вывода данных с помощью метода ToString() и работы с другими членами System.Object. Все сказанного можно достичь с помощью синтаксиса анонимных типов.

Определение анонимного типа

Анонимный тип определяется с использованием ключевого слова var (глава 3) в сочетании с синтаксисом инициализации объектов (глава 5). Ключевое слово var должно применяться из-за того, что компилятор будет автоматически генерировать новое определение класса на этапе компиляции (причем имя этого класса никогда не встретится в коде C#). Синтаксис инициализации применяется для сообщения компилятору о необходимости создания в новом типе закрытых поддерживающих полей и (допускающих только чтение) свойств.

В целях иллюстрации создадим новый проект консольного приложения по имени AnonymousTypes. Затем добавим в класс Program показанный ниже метод, который формирует новый тип на лету, используя данные входных параметров:

```

static void BuildAnonType( string make, string color, int currSp )
{
    // Построить анонимный тип с применением входных аргументов.
    var car = new { Make = make, Color = color, Speed = currSp };
    // Обратите внимание, что теперь этот тип можно
    // использовать для получения данных свойств!
    Console.WriteLine("You have a {0} {1} going {2} MPH",
        car.Color, car.Make, car.Speed);
    // Анонимные типы имеют специальные реализации каждого
    // виртуального метода System.Object. Например:
    Console.WriteLine("ToString() == {0}", car.ToString());
}

```

Метод BuildAnonType() можно вызвать в Main() ожидаемым образом. Однако обратите внимание, что анонимный тип можно также создать с применением жестко закодированных значений:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Anonymous Types *****\n");
    // Создать анонимный тип, представляющий автомобиль.
    var myCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };
}

```

```
// Вывести на консоль цвет и производителя.
Console.WriteLine("My car is a {0} {1}.", myCar.Color, myCar.Make);

// Теперь вызвать вспомогательный метод для построения
// анонимного типа с указанием аргументов.
BuildAnonType("BMW", "Black", 90);

Console.ReadLine();
}
```

В данный момент достаточно понимать, что анонимные типы позволяют быстро моделировать “форму” данных с очень малыми накладными расходами. Они всего лишь являются способом построения на лету нового типа данных, который поддерживает базовую инкапсуляцию через свойства и действует в соответствии с семантикой на основе значений. Чтобы уловить суть последнего утверждения, давайте посмотрим, каким образом компилятор C# строит анонимные типы на этапе компиляции, и в особенности — как он переопределяет члены `System.Object`.

Внутреннее представление анонимных типов

Все анонимные типы автоматически наследуются от `System.Object` и потому поддерживают все члены, предоставленные этим базовым классом. В результате можно вызывать метод `ToString()`, `GetHashCode()`, `Equals()` или `GetType()` на неявно типизированном объекте `myCar`. Предположим, что в классе `Program` определен следующий статический вспомогательный метод:

```
static void ReflectOverAnonymousType(object obj)
{
    Console.WriteLine("obj is an instance of: {0}", obj.GetType().Name);
    Console.WriteLine("Base class of {0} is {1}",
        obj.GetType().Name,
        obj.GetType().BaseType);
    Console.WriteLine("obj.ToString() == {0}", obj.ToString());
    Console.WriteLine("obj.GetHashCode() == {0}", obj.GetHashCode());
    Console.WriteLine();
}
```

Теперь вызовем метод `ReflectOverAnonymousType()` в `Main()`, передав ему объект `myCar` в качестве параметра:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Anonymous Types *****\n");

    // Создать анонимный тип, представляющий автомобиль.
    var myCar = new {Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55};

    // Выполнить рефлексии того, что сгенерировал компилятор.
    ReflectOverAnonymousType(myCar);

    ...

    Console.ReadLine();
}
```

Вывод будет выглядеть приблизительно так:

```
***** Fun with Anonymous Types *****
obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() = { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() = -439083487
```

Прежде всего, обратите внимание в примере на то, что объект `myCar` имеет тип `<>f__AnonymousType0`3` (в вашем выводе имя типа может быть другим). Помните, что имя, назначаемое типу, полностью определяется компилятором и не доступно в коде C# напрямую.

Пожалуй, наиболее важно здесь то, что каждая пара “имя-значение”, определенная с использованием синтаксиса инициализации объектов, отображается на идентично именованное свойство, доступное только для чтения, и соответствующее закрытое поддерживающее поле, которое также допускает только чтение. Приведенный ниже код C# примерно отражает сгенерированный компилятором класс, применяемый для представления объекта `myCar` (его можно просмотреть посредством утилиты `ildasm.exe`):

```
internal sealed class <>f__AnonymousType0<<Color>j__TPar,
    <Make>j__TPar, <CurrentSpeed>j__TPar>
{
    // Поля только для чтения.
    private readonly <Color>j__TPar <Color>i__Field;
    private readonly <CurrentSpeed>j__TPar <CurrentSpeed>i__Field;
    private readonly <Make>j__TPar <Make>i__Field;

    // Стандартный конструктор.
    public <>f__AnonymousType0(<Color>j__TPar Color,
        <Make>j__TPar Make, <CurrentSpeed>j__TPar CurrentSpeed);

    // Переопределенные методы.
    public override bool Equals(object value);
    public override int GetHashCode();
    public override string ToString();

    // Свойства только для чтения.
    public <Color>j__TPar Color { get; }
    public <CurrentSpeed>j__TPar CurrentSpeed { get; }
    public <Make>j__TPar Make { get; }
}
```

Реализация методов `ToString()` и `GetHashCode()`

Все анонимные типы автоматически являются производными от `System.Object` и предоставляют переопределенные версии методов `Equals()`, `GetHashCode()` и `ToString()`. Реализация `ToString()` просто строит строку из пар “имя-значение”. Вот пример:

```
public override string ToString()
{
    StringBuilder builder = new StringBuilder();
    builder.Append("{ Color = ");
    builder.Append(this.<Color>i__Field);
    builder.Append(", Make = ");
    builder.Append(this.<Make>i__Field);
    builder.Append(", CurrentSpeed = ");
    builder.Append(this.<CurrentSpeed>i__Field);
    builder.Append(" }");
    return builder.ToString();
}
```

Реализация `GetHashCode()` вычисляет хеш-значение, используя каждую переменную-член анонимного типа в качестве входных данных для типа `System.Collections.Generic.EqualityComparer<T>`. С такой реализацией `GetHashCode()` два анонимных типа будут выдавать одинаковые хеш-значения тогда (и только тогда), когда они об-

ладают одним и тем же набором свойств, которым присвоены те же самые значения. Благодаря такой реализации анонимные типы хорошо подходят для помещения внутрь контейнера Hashtable.

Семантика эквивалентности анонимных типов

Наряду с тем, что реализация переопределенных методов ToString() и GetHashCode() достаточно проста, вас может интересовать, как был реализован метод Equals(). Например, если определены две переменные "анонимных автомобилей" с одинаковыми наборами пар "имя-значение", то должны ли эти переменные считаться эквивалентными? Чтобы увидеть результат такого сравнения, дополним класс Program следующим новым методом:

```
static void EqualityTest()
{
    // Создать два анонимных класса с идентичными наборами пар "имя-значение".
    var firstCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };
    var secondCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };

    // Считаются ли они эквивалентными, когда используется Equals()?
    if (firstCar.Equals(secondCar))
        Console.WriteLine("Same anonymous object!");           // Тот же самый
                                                                // анонимный объект
    else
        Console.WriteLine("Not the same anonymous object!");    // Не тот же самый
                                                                // анонимный объект

    // Можно ли проверить их эквивалентность с помощью операции ==?
    if (firstCar == secondCar)
        Console.WriteLine("Same anonymous object!");
    else
        Console.WriteLine("Not the same anonymous object!");

    // Имеют ли эти объекты в основе один и тот же тип?
    if (firstCar.GetType().Name == secondCar.GetType().Name)
        Console.WriteLine("We are both the same type!");       // Оба объекта имеют
                                                                // тот же самый тип
    else
        Console.WriteLine("We are different types!");          // Объекты относятся
                                                                // к разным типам

    // Отобразить все детали.
    Console.WriteLine();
    ReflectOverAnonymousType(firstCar);
    ReflectOverAnonymousType(secondCar);
}
```

В результате вызова метода EqualityTest() внутри Main() получается (несколько неожиданный) вывод:

```
My car is a Bright Pink Saab.
You have a Black BMW going 90 MPH
ToString() == { Make = BMW, Color = Black, Speed = 90 }

Same anonymous object!
Not the same anonymous object!
We are both the same type!

obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() == { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() == -439083487
```

```
obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() == { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() == -439083487
```

Как видите, первая проверка, где вызывается `Equals()`, возвращает `true`, и потому на консоль выводится сообщение `Same anonymous object!` (Тот же самый анонимный объект). Причина в том, что сгенерированный компилятором метод `Equals()` при проверке эквивалентности применяет семантику на основе значений (т.е. проверяет значения каждого поля сравниваемых объектов).

Тем не менее, вторая проверка, в которой используется операция `==`, приводит к выводу на консоль строки `Not the same anonymous object!` (Не тот же самый анонимный объект), что на первый взгляд выглядит несколько нелогично. Такой результат обусловлен тем, что анонимные типы не получают перегруженных версий операций проверки равенства (`==` и `!=`), поэтому при проверке эквивалентности объектов анонимных типов с применением операций равенства C# (вместо метода `Equals()`) проверяются ссылки, а не значения, поддерживаемые объектами.

Последнее, но не менее важное замечание: финальная проверка (где исследуется имя лежащего в основе типа) показывает, что объекты анонимных типов являются экземплярами одного и того же типа класса, сгенерированного компилятором (`<>f__AnonymousType0`3` в данном примере), т.к. `firstCar` и `secondCar` имеют одинаковые наборы свойств (`Color`, `Make` и `CurrentSpeed`).

Это иллюстрирует важный, но тонкий аспект: компилятор будет генерировать новое определение класса, только когда анонимный тип содержит уникальные имена свойств. Таким образом, если вы объявляете идентичные анонимные типы (в смысле имеющие те же самые имена свойств) внутри сборки, то компилятор генерирует единственное определение анонимного типа.

Анонимные типы, содержащие другие анонимные типы

Разрешено создавать анонимные типы, которые состоят из других анонимных типов. Предположим для примера, что требуется смоделировать заказ на приобретение, который хранит метку времени, цену и сведения о приобретаемом автомобиле. Вот новый (чуть более сложный) анонимный тип, представляющий такую сущность:

```
// Создать анонимный тип, состоящий из еще одного анонимного типа.
var purchaseItem = new {
    TimeBought = DateTime.Now,
    ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},
    Price = 34.000};
ReflectOverAnonymousType(purchaseItem);
```

Сейчас вы уже должны понимать синтаксис, используемый для определения анонимных типов, но возможно все еще интересуетесь, где (и когда) применять такое языковое средство. Выражаясь кратко, объявления анонимных типов следует использовать умеренно, обычно только в случае применения набора технологий LINQ (глава 12). С учетом описанных ниже многочисленных ограничений анонимных типов вы никогда не должны отказываться от использования строго типизированных классов и структур просто из-за того, что это возможно.

- Контроль над именами анонимных типов отсутствует.
- Анонимные типы всегда расширяют `System.Object`.
- Поля и свойства анонимного типа всегда допускают только чтение.
- Анонимные типы не могут поддерживать события, специальные методы, специальные операции или специальные переопределения.

- Анонимные типы всегда неявно запечатаны.
- Экземпляры анонимных типов всегда создаются с применением стандартных конструкторов.

Однако при программировании с использованием набора технологий LINQ вы обнаружите, что во многих случаях такой синтаксис оказывается удобным, когда нужно быстро смоделировать общую *форму* сущности, а не ее функциональность.

Исходный код. Проект `AnonymousTypes` доступен в подкаталоге `Chapter_11`.

Работа с типами указателей

Последняя тема главы касается средства C#, которое в подавляющем большинстве проектов .NET применяется реже всех остальных.

На заметку! В последующих примерах предполагается наличие у вас определенных навыков манипулирования указателями в C++. Если это не так, тогда можете спокойно пропустить данную тему. В подавляющем большинстве приложений C# указатели не используются.

В главе 4 вы узнали, что в рамках платформы .NET определены две крупные категории данных: типы значений и ссылочные типы. По правде говоря, на самом деле есть еще и третья категория: *типы указателей*. Для работы с типами указателей доступны специфичные операции и ключевые слова (табл. 11.2), которые позволяют обойти схему управления памятью CLR и взять дело в свои руки.

Таблица 11.2. Операции и ключевые слова C#, связанные с указателями

Операция/ ключевое слово	Назначение
*	Эта операция применяется для создания переменной указателя (т.е. переменной, которая представляет непосредственное местоположение в памяти). Как и в языке C++, та же самая операция используется для разыменования указателя
&	Эта операция применяется для получения адреса переменной в памяти
->	Эта операция используется для доступа к полям типа, представленного указателем (небезопасная версия операции точки в C#)
[]	Эта операция (в небезопасном контексте) позволяет индексировать область памяти, на которую указывает переменная указателя (если вы программировали на C++, то вспомните о взаимодействии между переменной указателя и операцией [])
++, --	В небезопасном контексте операции инкремента и декремента могут применяться к типам указателей
+, -	В небезопасном контексте операции сложения и вычитания могут применяться к типам указателей
==, !=, <, >, <=, >=	В небезопасном контексте операции сравнения и эквивалентности могут применяться к типам указателей
stackalloc	В небезопасном контексте ключевое слово <code>stackalloc</code> может использоваться для размещения массивов C# прямо в стеке
fixed	В небезопасном контексте ключевое слово <code>fixed</code> может применяться для временного закрепления переменной, чтобы впоследствии удалось найти ее адрес

Перед погружением в детали следует еще раз подчеркнуть, что вам очень *редко*, если вообще когда-нибудь, понадобится использовать типы указателей. Хотя C# позволяет опуститься на уровень манипуляций указателями, помните, что исполняющая среда .NET не имеет абсолютно никакого понятия о ваших намерениях. Соответственно, если вы неправильно управляете указателем, то сами и будете отвечать за последствия. С учетом этих предупреждений возникает вопрос: когда в принципе может возникнуть необходимость работы с типами указателей? Существуют две распространенные ситуации.

- Нужно оптимизировать избранные части приложения, напрямую манипулируя памятью за рамками ее управления со стороны CLR.
- Необходимо вызывать методы из DLL-библиотеки, написанной на C, или сервера COM, которые требуют передачи типов указателей в качестве параметров. Но даже в таком случае часто можно обойтись без применения типов указателей, отдав предпочтение типу `System.IntPtr` и членам типа `System.Runtime.InteropServices.Marshal`.

В случае если вы все же решили задействовать данное средство языка C#, тогда придется информировать компилятор C# (`csc.exe`) о своих намерениях, разрешив проекту поддерживать “небезопасный код”. Чтобы сделать это в командной строке, при запуске компилятора укажите флаг `/unsafe`:

```
csc /unsafe *.cs
```

В среде Visual Studio перейдите на страницу свойств проекта и на вкладке Build (Сборка) отметьте флажок Allow unsafe code (Разрешить небезопасный код), как показано на рис. 11.2. Чтобы поэкспериментировать с типами указателей, создадим новый проект консольного приложения по имени `UnsafeCode` и разрешим небезопасный код.

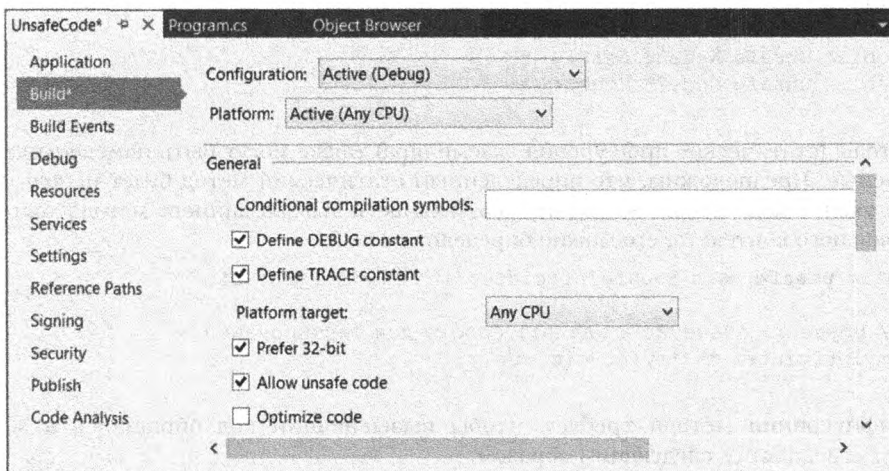


Рис. 11.2. Включение поддержки небезопасного кода в Visual Studio

Ключевое слово `unsafe`

Для работы с указателями в C# должен быть специально объявлен блок “небезопасного кода” с использованием ключевого слова `unsafe` (любой код, который не помечен ключевым словом `unsafe`, автоматически считается “безопасным”). Например, в следующем классе `Program` объявляется область небезопасного кода внутри метода `Main()`:


```
class Program
{
    static void Main(string[] args)
    {
        unsafe
        {
            // Здесь работаем с указателями!
        }
        // Здесь работа с указателями невозможна!
    }
}
```

В дополнение к объявлению области небезопасного кода внутри метода можно строить “небезопасные” структуры, классы, члены типов и параметры. Ниже приведено несколько примеров (типы `Node` и `Node2` в текущем проекте определять не нужно):

```
// Эта структура целиком является небезопасной и может
// использоваться только в небезопасном контексте.
unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}

// Эта структура безопасна, но члены Node2* – нет.
// Формально извне небезопасного контекста можно
// обращаться к Value, но не к Left и Right.
public struct Node2
{
    public int Value;
    // Эти члены доступны только в небезопасном контексте!
    public unsafe Node2* Left;
    public unsafe Node2* Right;
}
```

Методы (статические либо уровня экземпляра) также могут быть помечены как небезопасные. Предположим, что определенный статический метод будет использовать логику указателей. Чтобы обеспечить возможность вызова данного метода только из небезопасного контекста, его можно определить так:

```
static unsafe void SquareIntPtr(int* myIntPtr)
{
    // Возвести значение в квадрат просто для тестирования.
    *myIntPtr *= *myIntPtr;
}
```

Конфигурация метода требует, чтобы вызывающий код обращался к методу `SquareIntPtr()` следующим образом:

```
static void Main(string[] args)
{
    unsafe
    {
        int myInt = 10;
        // Нормально, мы находимся в небезопасном контексте.
        SquareIntPtr(&myInt);
        Console.WriteLine("myInt: {0}", myInt);
    }
}
```

```
int myInt2 = 5;
// Ошибка на этапе компиляции! Это должно делаться в небезопасном контексте!
SquareIntPointer(&myInt2);
Console.WriteLine("myInt: {0}", myInt2);
}
```

Если вы не хотите вынуждать вызывающий код помещать такой вызов внутрь небезопасного контекста, то можете пометить весь метод Main() ключевым словом unsafe. В таком случае приведенный ниже код скомпилируется:

```
static unsafe void Main(string[] args)
{
    int myInt2 = 5;
    SquareIntPointer(&myInt2);
    Console.WriteLine("myInt: {0}", myInt2);
}
```

Залучив на выполнение метод Main(), вы получите следующий вывод:

```
myInt: 25
```

Работа с операциями * и &

После установления небезопасного контекста можно строить указатели и типы данных с помощью операции *, а также получать адрес указываемых данных посредством операции &. В отличие от C или C++ в языке C# операция * применяется только к лежащему в основе типу, а не является префиксом имени каждой переменной указателя. Например, взгляните на показанный далее код, демонстрирующий правильный и неправильный способы объявления указателей на целочисленные переменные:

```
// Нет! В C# это некорректно!
int *pi, *pj;

// Да! Так поступают в C#.
int* p1, p2;
```

Рассмотрим следующий небезопасный метод:

```
static unsafe void PrintValueAndAddress()
{
    int myInt;

    // Определить указатель на int и присвоить ему адрес myInt.
    int* ptrToMyInt = &myInt;

    // Присвоить значение myInt, используя обращение через указатель.
    *ptrToMyInt = 123;

    // Вывести некоторые значения.
    Console.WriteLine("Value of myInt {0}", myInt); // значение myInt
    Console.WriteLine("Address of myInt {0:X}", (int)&ptrToMyInt); //адрес myInt
}
```

Небезопасная (и безопасная) функция обмена

Разумеется, объявлять указатели на локальные переменные, чтобы просто присваивать им значения (как в предыдущем примере), никогда не понадобится и к тому же неудобно. В качестве более практичного примера небезопасного кода предположим, что необходимо построить функцию обмена с использованием арифметики указателей:

```
public unsafe static void UnsafeSwap(int* i, int* j)
{
    int temp = *i;
    *i = *j;
    *j = temp;
}
```

Очень похоже на язык C, не так ли? Тем не менее, учитывая предшествующую работу, вы должны знать, что можно было бы написать безопасную версию алгоритма обмена с применением ключевого слова `ref` языка C#:

```
public static void SafeSwap(ref int i, ref int j)
{
    int temp = i;
    i = j;
    j = temp;
}
```

Функциональность обеих версий метода идентична, доказывая тем самым, что прямые манипуляции указателями в C# не являются обязательными. Ниже показана логика вызова, использующая безопасный метод `Main()`, но с небезопасным контекстом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Calling method with unsafe code *****");
    // Значения, подлежащие обмену.
    int i = 10, j = 20;

    // "Безопасный" обмен значений.
    Console.WriteLine("\n***** Safe swap *****");
    Console.WriteLine("Values before safe swap: i = {0}, j = {1}", i, j);
    SafeSwap(ref i, ref j);
    Console.WriteLine("Values after safe swap: i = {0}, j = {1}", i, j);

    // "Небезопасный" обмен значений.
    Console.WriteLine("\n***** Unsafe swap *****");
    Console.WriteLine("Values before unsafe swap: i = {0}, j = {1}", i, j);
    unsafe { UnsafeSwap(&i, &j); }

    Console.WriteLine("Values after unsafe swap: i = {0}, j = {1}", i, j);
    Console.ReadLine();
}
```

Доступ к полям через указатели (операция `->`)

Теперь предположим, что определена простая безопасная структура `Point`:

```
struct Point
{
    public int x;
    public int y;
    public override string ToString() => $"({x}, {y})";
}
```

В случае объявления указателя на тип `Point` для доступа к открытым членам структуры понадобится применять операцию доступа к полям (имеющую вид `->`). Как упоминалось в табл. 11.2, она представляет собой небезопасную версию стандартной (безопасной) операции точки (`.`). В сущности, используя операцию обращения к указателю (`*`), можно разменовывать указатель для применения операции точки. Взгляните на следующий небезопасный метод:

```
static unsafe void UsePointerToPoint()
{
    // Доступ к членам через указатель.
    Point point;
    Point* p = &point;
    p->x = 100;
    p->y = 200;
    Console.WriteLine(p->ToString());

    // Доступ к членам через разыменованный указатель.
    Point point2;
    Point* p2 = &point2;
    (*p2).x = 100;
    (*p2).y = 200;
    Console.WriteLine((*p2).ToString());
}
```

Ключевое слово **stackalloc**

В небезопасном контексте может возникнуть необходимость в объявлении локальной переменной, для которой память выделяется непосредственно в стеке вызовов (и потому она не обрабатывается сборщиком мусора .NET). Для этого в языке C# предусмотрено ключевое слово **stackalloc**, которое является эквивалентом функции `_alloca` библиотеки времени выполнения C. Вот простой пример:

```
static unsafe void UnsafeStackAlloc()
{
    char* p = stackalloc char[256];
    for (int k = 0; k < 256; k++)
        p[k] = (char)k;
}
```

Закрепление типа посредством ключевого слова **fixed**

В предыдущем примере вы видели, что выделение фрагмента памяти внутри небезопасного контекста может делаться с помощью ключевого слова **stackalloc**. Из-за природы операции **stackalloc** выделенная память очищается, как только выделяющий ее метод возвращает управление (т.к. память распределена в стеке). Однако рассмотрим более сложный пример. Во время исследования операции `->` создавался тип значения по имени **Point**. Как и все типы значений, выделяемая его экземплярам память исчезает из стека по окончании выполнения. Предположим, что тип **Point** взамен определен как **ссылочный**:

```
class PointRef // <= Переименован и обновлен.
{
    public int x;
    public int y;
    public override string ToString() => $"({x}, {y})";
}
```

Как вам известно, если в вызывающем коде объявляется переменная типа **Point**, то память для нее выделяется в куче, поддерживающей сборку мусора. И тут возникает животрепещущий вопрос: а что если небезопасный контекст пожелает взаимодействовать с этим объектом (или любым другим объектом из кучи)? Учитывая, что сборка мусора может произойти в любой момент, вы только вообразите, какие проблемы возникнут при обращении к членам **Point** именно в тот момент, когда происходит реорганизация кучи! Теоретически может случиться так, что небезопасный контекст попытается взаи-

действовать с членом, который больше недоступен или был перемещен в другое место кучи после ее очистки с учетом поколений (что является очевидной проблемой).

Для фиксации переменной ссылочного типа в памяти из небезопасного контекста язык C# предлагает ключевое слово `fixed`. Оператор `fixed` устанавливает указатель на управляемый тип и "закрепляет" такую переменную на время выполнения кода. Без `fixed` от указателей на управляемые переменные было бы мало толку, поскольку сборщик мусора может перемещать переменные в памяти непредсказуемым образом. (На самом деле компилятор C# даже не позволит установить указатель на управляемую переменную, если оператор `fixed` отсутствует.)

Таким образом, если вы создали объект `Point` и хотите взаимодействовать с его членами, тогда должны написать следующий код (либо иначе получить ошибку на этапе компиляции):

```
public unsafe static void UseAndPinPoint()
{
    PointRef pt = new PointRef
    {
        x = 5,
        y = 6
    };

    // Закрепить указатель pt на месте, чтобы он не мог
    // быть перемещен или уничтожен сборщиком мусора.
    fixed (int* p = &pt.x)
    {
        // Использовать здесь переменную int*!
    }

    // Указатель pt теперь не закреплен и готов
    // к сборке мусора после завершения метода.
    Console.WriteLine("Point is: {0}", pt);
}
```

Выражаясь кратко, ключевое слово `fixed` позволяет строить оператор, который фиксирует ссылочную переменную в памяти, чтобы ее адрес оставался постоянным на протяжении работы оператора (или блока операторов). Каждый раз, когда вы взаимодействуете со ссылочным типом из контекста небезопасного кода, закрепление ссылки обязательно.

Ключевое слово `sizeof`

Последнее ключевое слово C#, связанное с небезопасным кодом — `sizeof`. Как и в C++, ключевое слово `sizeof` в C# используется для получения размера в байтах *встроенного типа данных*, но не специального типа, разве только в небезопасном контексте. Например, показанный ниже метод не нуждается в объявлении "небезопасным", т.к. все аргументы ключевого слова `sizeof` относятся к встроенным типам:

```
static void UseSizeOfOperator()
{
    Console.WriteLine("The size of short is {0}.", sizeof(short)); //размер short
    Console.WriteLine("The size of int is {0}.", sizeof(int)); //размер int
    Console.WriteLine("The size of long is {0}.", sizeof(long)); //размер long
}
```

Тем не менее, если вы хотите получить размер специальной структуры `Point`, то метод `UseSizeOfOperator()` придется модифицировать (обратите внимание на добавление ключевого слова `unsafe`):

```
static unsafe void UseSizeOfOperator()
{
    ...
    Console.WriteLine("The size of Point is {0}.", sizeof(Point));
    // размер Point
}
```

Исходный код. Проект `UnsafeCode` доступен в подкаталоге `Chapter_11`.

Итак, обзор нескольких более сложных средств языка программирования C# завершен. Напоследок снова необходимо отметить, что в большинстве проектов .NET эти средства могут вообще не понадобиться (особенно указатели). Тем не менее, как будет показано в последующих главах, некоторые средства действительно полезны (и даже обязательны) при работе с API-интерфейсами LINQ, в частности расширяющие методы и анонимные типы.

Резюме

Целью главы было углубление знаний языка программирования C#. Первым делом мы исследовали разнообразные более сложные конструкции в типах (индексаторные методы, перегруженные операции и специальные процедуры преобразования).

Затем мы рассмотрели роль расширяющих методов и анонимных типов. Как вы увидите в следующей главе, эти средства удобны при работе с API-интерфейсами LINQ (хотя при желании их можно применять в коде повсеместно). Вспомните, что анонимные методы позволяют быстро моделировать “форму” типа, в то время как расширяющие методы дают возможность добавлять новую функциональность к типам без необходимости в определении подклассов.

Финальная часть главы была посвящена небольшому набору менее известных ключевых слов (`sizeof`, `unsafe` и т.п.); наряду с ними рассматривалась работа с низкоуровневыми типами указателей. Как было установлено в процессе исследования типов указателей, в подавляющем большинстве приложений C# их никогда не придется использовать.

ГЛАВА 12

LINQ to Objects

Независимо от типа приложения, которое вы создаете с использованием платформы .NET, ваша программа во время выполнения определенно нуждается в доступе к данным какой-нибудь формы. Разумеется, данные могут находиться в многочисленных местах, включая файлы XML, реляционные базы данных, коллекции в памяти и элементарные массивы. Исторически сложилось так, что в зависимости от места хранения данных программистам приходилось применять разные и несвязанные друг с другом API-интерфейсы. Набор технологий LINQ (Language Integrated Query — язык интегрированных запросов), появившийся в версии .NET 3.5, предоставил краткий, симметричный и строго типизированный способ доступа к широкому разнообразию хранилищ данных. В настоящей главе изучение LINQ начинается с исследования LINQ to Objects.

Прежде чем погрузиться в LINQ to Objects, в первой части главы предлагается обзор основных программных конструкций языка C#, которые делают возможным существование LINQ. По мере чтения главы вы убедитесь, насколько полезны (а иногда и обязательны) такие средства, как неявно типизированные переменные, синтаксис инициализации объектов, лямбда-выражения, расширяющие методы и анонимные типы.

После пересмотра поддерживающей инфраструктуры в оставшемся материале главы будет представлена модель программирования LINQ и объяснена ее роль в рамках платформы .NET. Вы узнаете предназначение операций и выражений запросов, позволяющих определять операторы, которые будут опрашивать источник данных для выдачи требуемого результирующего набора. Попутно будут строиться многочисленные примеры LINQ, взаимодействующие с данными в массивах и коллекциях различного типа (обобщенных и необобщенных), а также исследоваться сборки, пространства имен и типы, которые представляют API-интерфейс LINQ to Objects.

На заметку! Информация, приведенная в главе, послужит фундаментом для освоения материала последующих глав книги, в которых описаны дополнительные технологии LINQ, включая Parallel LINQ (глава 19), Entity Framework (глава 22) и Entity Framework Core (глава 30).

Программные конструкции, специфичные для LINQ

С высокоуровневой точки зрения LINQ можно трактовать как строго типизированный язык запросов, встроенный непосредственно в грамматику самого языка C#. Используя LINQ, можно создавать любое количество выражений, которые выглядят и ведут себя подобно SQL-запросам к базе данных. Однако запрос LINQ может применяться к любым хранилищам данных, включая хранилища, которые не имеют ничего общего с настоящими реляционными базами данных.

На заметку! Хотя запросы LINQ внешне похожи на запросы SQL, их синтаксис *не* идентичен. В действительности многие запросы LINQ имеют формат, прямо противоположный формату подобного запроса к базе данных! Если вы попытаетесь отобразить LINQ непосредственно на SQL, то определенно запутаетесь. Чтобы подобного не произошло, рекомендуется воспринимать запросы LINQ как уникальные операторы, которые просто случайно оказались похожими на SQL.

Когда LINQ впервые был представлен в составе платформы .NET 3.5, языки C# и VB уже были расширены огромным количеством программных конструкций для поддержки набора технологий LINQ. В частности, язык C# использует следующие связанные с LINQ средства:

- неявно типизированные локальные переменные;
- синтаксис инициализации объектов и коллекций;
- лямбда-выражения;
- расширяющие методы;
- анонимные типы.

Перечисленные средства уже детально рассматривались в других главах книги. Тем не менее, чтобы освежить все в памяти, давайте быстро вспомним о каждом средстве по очереди, удостоверившись в правильном их понимании.

На заметку! Из-за того, что в последующих разделах приводится обзор материала, рассматриваемого где-то в других местах книги, проект кода C# здесь не предусмотрен.

Неявная типизация локальных переменных

В главе 3 вы узнали о ключевом слове `var` языка C#. Оно позволяет определять локальную переменную без явного указания типа данных. Однако такая переменная будет строго типизированной, потому что компилятор определит ее корректный тип данных на основе начального присваивания. Вспомните показанный ниже код примера из главы 3:

```
static void DeclareImplicitVars()
{
    // Неявно типизированные локальные переменные.
    var myInt = 0;
    var myBool = true;
    var myString = "Time, marches on...";

    // Вывести имена лежащих в основе типов.
    Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
    Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
    Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

Это языковое средство удобно и зачастую обязательно, когда применяется LINQ. Как вы увидите на протяжении главы, многие запросы LINQ возвращают последовательность типов данных, которые не будут известны вплоть до этапа компиляции. Учитывая, что лежащий в основе тип данных не известен до того, как приложение скомпилируется, вполне очевидно, что явно объявить такую переменную невозможно!

Синтаксис инициализации объектов и коллекций

В главе 5 объяснялась роль синтаксиса инициализации объектов, который позволяет создавать переменную типа класса или структуры и устанавливать любое количество ее открытых свойств за один прием. В результате получается компактный (и по-прежнему легко читаемый) синтаксис, который может использоваться для подготовки объектов к потреблению. Также вспомните из главы 9, что язык C# поддерживает похожий синтаксис инициализации коллекций объектов. Взгляните на следующий фрагмент кода, где синтаксис инициализации коллекций применяется для наполнения `List<T>` объектами `Rectangle`, каждый из которых состоит из пары объектов `Point`, представляющих точку с координатами (x, y):

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
    new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
                  BottomRight = new Point { X = 200, Y = 200}},
    new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
                  BottomRight = new Point { X = 100, Y = 100}},
    new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
                  BottomRight = new Point { X = 90, Y = 75}}
};
```

Несмотря на то что использовать синтаксис инициализации коллекций или объектов совершенно не обязательно, с его помощью можно получить более компактную кодовую базу. Кроме того, этот синтаксис в сочетании с неявной типизацией локальных переменных позволяет объявлять анонимный тип, что удобно при создании проекций LINQ. О проекциях LINQ речь пойдет позже в главе.

Лямбда-выражения

Лямбда-операция C# (`=>`) была полностью описана в главе 10. Вспомните, что данная операция позволяет строить лямбда-выражение, которое может применяться в любой момент при вызове метода, требующего строго типизированный делегат в качестве аргумента. Лямбда-выражения значительно упрощают работу с делегатами .NET, т.к. сокращают объем кода, который должен быть написан вручную. Лямбда-выражения могут быть представлены следующим образом:

```
( АргументыДляОбработки ) => { ОбработывающиеОператоры }
```

В главе 10 было показано, как взаимодействовать с методом `FindAll()` обобщенного класса `List<T>` с использованием трех разных подходов. После работы с низкоуровневым делегатом `Predicate<T>` и анонимным методом C# мы пришли к приведенной ниже (исключительно компактной) версии, в которой использовалось лямбда-выражение:

```
static void LambdaExpressionSyntax()
{
    // Создать список целочисленных значений.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
    // Теперь использовать лямбда-выражение C#.
    List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
    // Вывести четные числа.
    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
    {
        Console.WriteLine("{0}\t", evenNumber);
    }
    Console.WriteLine();
}
```

Лямбда-выражения будут удобны при работе с объектной моделью, лежащей в основе LINQ. Как вы вскоре выясните, операции запросов LINQ в C# — просто сокращенная запись для вызова методов класса по имени `System.Linq.Enumerable`. Эти методы обычно всегда требуют передачи в качестве параметров делегатов (в частности, делегата `Func<>`), которые применяются для обработки данных с целью выдачи корректного результирующего набора. За счет использования лямбда-выражений можно упростить код и позволить компилятору вывести нужный делегат.

Расширяющие методы

Расширяющие методы C# позволяют оснащать существующие классы новой функциональностью без необходимости в создании подклассов. Кроме того, расширяющие методы дают возможность добавлять новую функциональность к запечатанным классам и структурам, которые в принципе не допускают построения подклассов. Вспомните из главы 11, что когда создается расширяющий метод, первый его параметр снабжается ключевым словом `this` и помечает расширяемый тип. Также вспомните, что расширяющие методы должны всегда определяться внутри статического класса, а потому объявляться с применением ключевого слова `static`. Вот пример:

```
namespace MyExtensions
{
    static class ObjectExtensions
    {
        // Определить расширяющий метод для System.Object.
        public static void DisplayDefiningAssembly(this object obj)
        {
            Console.WriteLine("{0} lives here:\n\t->{1}\n", obj.GetType().Name,
                Assembly.GetAssembly(obj.GetType()));
        }
    }
}
```

Чтобы использовать такое расширение, приложение должно импортировать пространство имен, определяющее расширение (и возможно добавить ссылку на внешнюю сборку). Затем можно приступить к написанию кода:

```
static void Main(string[] args)
{
    // Поскольку все типы расширяют System.Object, все
    // классы и структуры могут использовать это расширение.
    int myInt = 12345678;
    myInt.DisplayDefiningAssembly();

    System.Data.DataSet d = new System.Data.DataSet();
    d.DisplayDefiningAssembly();
    Console.ReadLine();
}
```

При работе с LINQ вам редко (если вообще когда-либо) потребуется вручную строить собственные расширяющие методы. Тем не менее, создавая выражения запросов LINQ, вы на самом деле будете применять многочисленные расширяющие методы, уже определенные Microsoft. Фактически каждая операция запроса LINQ в C# представляет собой сокращенную запись для ручного вызова лежащего в основе расширяющего метода, который обычно определен в служебном классе `System.Linq.Enumerable`.

Анонимные типы

Последним средством языка C#, описание которого мы здесь кратко повторим, являются анонимные типы, рассмотренные в главе 11. Такое средство может использоваться для быстрого моделирования “формы” данных, разрешая компилятору генерировать на этапе компиляции новое определение класса, которое основано на предоставленном наборе пар “имя-значение”.

Вспомните, что результирующий тип будет составлен с применением семантики на основе значений, а каждый виртуальный метод `System.Object` будет соответствующим образом переопределен. Чтобы определить анонимный тип, понадобится объявить неявно типизированную переменную и указать форму данных с использованием синтаксиса инициализации объектов:

```
// Создать анонимный тип, состоящий из еще одного анонимного типа.
var purchaseItem = new {
    TimeBought = DateTime.Now,
    ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},
    Price = 34.000};
```

Анонимные типы часто применяются в LINQ, когда необходимо проецировать новые формы данных на лету. Например, предположим, что есть коллекция объектов `Person`, и вы хотите использовать LINQ для получения информации о возрасте и номере карточки социального страхования в каждом объекте. Применяя проекцию LINQ, можно предоставить компилятору возможность генерации нового анонимного типа, который содержит интересующую информацию.

Роль LINQ

На этом краткий обзор средств языка C#, которые позволяют LINQ делать свою работу, завершен. Однако важно понимать, зачем вообще нужен язык LINQ. Любой разработчик программного обеспечения согласится с утверждением, что подавляющее большинство времени при программировании тратится на получение и манипулирование данными.

Когда говорят о “данных”, на ум немедленно приходит информация, хранящаяся внутри реляционных баз данных. Тем не менее, другими популярными местоположениями для данных являются документы XML или простые текстовые файлы.

Данные могут находиться в многочисленных местах помимо указанных двух распространенных хранилищ информации. Например, пусть имеется массив или обобщенный тип `List<T>`, содержащий 300 целых чисел, и требуется получить подмножество, которое удовлетворяет заданному критерию (например, только четные или нечетные числа, только простые числа, только неповторяющиеся числа больше 50). Или, возможно, при использовании API-интерфейсов рефлексии необходимо получить в массиве элементов `Type` метаданные только для классов, производных от определенного родительского класса. На самом деле данные находятся *повсюду*.

До появления версии .NET 3.5 взаимодействие с отдельной разновидностью данных требовало от программистов применения совершенно несходных API-интерфейсов.

В табл. 12.1 описаны некоторые популярные API-интерфейсы, используемые для доступа к разнообразным типам данных (наверняка вы в состоянии привести и другие примеры).

Таблица 12.1. Способы манипулирования различными типами данных

Интересующие данные	Способ получения
Реляционные данные	System.Data.dll, System.Data.SqlClient.dll и т.д.
Данные документов XML	System.Xml.dll
Таблицы метаданных	Пространство имен System.Reflection
Коллекции объектов	Пространства имен System.Array и System.Collections/System.Collections.Generic

Разумеется, с такими подходами к манипулированию данными не связано ничего плохого. В сущности, вы можете (и будете) работать напрямую с ADO.NET, пространствами имен XML, службами рефлексии и разнообразными типами коллекций. Однако основная проблема заключается в том, что каждый из API-интерфейсов подобного рода является “самостоятельным островком”, трудно интегрируемым с другими. Правда, можно (например) сохранить объект DataSet из ADO.NET в документ XML и затем манипулировать им посредством пространств имен System.Xml, но все равно манипуляции данными остаются довольно асимметричными.

В рамках API-интерфейса LINQ была предпринята попытка предложить программистам согласованный, симметричный способ получения и манипулирования “данными” в широком смысле этого понятия. Применяя LINQ, можно создавать прямо внутри языка программирования C# конструкции, которые называются *выражениями запросов*. Такие выражения запросов основаны на многочисленных операциях запросов, которые намеренно сделаны похожими внешне и по поведению (но не идентичными) на выражения SQL.

Тем не менее, трюк в том, что выражение запроса может использоваться для взаимодействия с разнообразными типами данных — даже с теми, которые не имеют ничего общего с реляционными базами данных. Строго говоря, LINQ представляет собой термин, в целом описывающий сам подход доступа к данным. Однако в зависимости от того, где применяются запросы LINQ, вы встретите разные обозначения вроде перечисленных ниже.

- LINQ to Objects. Этот термин относится к действию по применению запросов LINQ к массивам и коллекциям.
- LINQ to XML. Этот термин относится к действию по использованию LINQ для манипулирования и запрашивания документов XML.
- LINQ to DataSet. Этот термин относится к действию по применению запросов LINQ к объектам DataSet из ADO.NET.
- LINQ to Entities. Этот аспект LINQ позволяет использовать запросы LINQ внутри API-интерфейса ADO.NET Entity Framework (EF).
- Parallel LINQ (он же PLINQ). Этот аспект делает возможной параллельную обработку данных, возвращаемых из запроса LINQ.

В настоящее время LINQ является неотъемлемой частью библиотек базовых классов .NET, управляемых языков и самой среды Visual Studio.

Выражения LINQ строго типизированы

Важно также отметить, что выражение запроса LINQ (в отличие от традиционного оператора SQL) *строго типизировано*. Следовательно, компилятор C# следит за этим и гарантирует, что выражения оформлены корректно с точки зрения синтаксиса. Инструменты вроде Visual Studio могут применять метаданные для поддержки удобных средств, таких как IntelliSense, автозавершение и т.д.

Основные сборки LINQ

Как упоминалось в главе 2, в диалоговом окне New Project (Новый проект) среды Visual Studio доступна возможность выбора версии платформы .NET, для которой должна проводиться компиляция. Когда компиляция осуществляется для .NET 3.5 или последующей версии, каждый шаблон проекта автоматически ссылается на основные сборки LINQ, что можно просмотреть в окне Solution Explorer. Основные сборки LINQ описаны в табл. 12.2. В оставшихся главах вы столкнетесь с дополнительными библиотеками LINQ.

Таблица 12.2. Основные сборки, связанные с LINQ

Сборка	Описание
System.Core.dll	Определяет типы, представляющие основной API-интерфейс LINQ. Это единственная сборка, к которой вы должны иметь доступ, если хотите использовать любые API-интерфейсы LINQ, включая LINQ to Objects
System.Data.DataSetExtensions.dll	Определяет набор типов для интеграции типов ADO.NET с парадигмой программирования LINQ (LINQ to DataSet)
System.Xml.Linq.dll	Предоставляет функциональность для применения LINQ с данными документов XML (LINQ to XML)

Для работы с LINQ to Objects вы должны обеспечить, чтобы в каждом файле кода C#, который содержит запросы LINQ, было импортировано пространство имен System.Linq (определенное главным образом внутри сборки System.Core.dll). В противном случае возникнут проблемы. Если на этапе компиляции вы получили сообщение об ошибке следующего вида:

```
Error 1 Could not find an implementation of the query pattern for source
type 'int[]'. 'Where' not found. Are you missing a reference
to 'System.Core.dll' or a using directive for 'System.Linq'?
```

Ошибка 1 Не удается обнаружить реализацию шаблона запросов для исходного типа int[]. Where не найдено. Возможно, пропущена ссылка на System.Core.dll или директива using для System.Linq?

то очень высока вероятность, что в файле кода C# отсутствует нужная директива using:

```
using System.Linq;
```

Применение запросов LINQ к элементарным массивам

Чтобы начать исследование LINQ to Objects, давайте построим приложение, которое будет применять запросы LINQ к разнообразным объектам типа массива. Создадим проект консольного приложения под названием LinqOverArray и определим внутри класса Program статический вспомогательный метод по имени QueryOverStrings(). Внутри метода создадим массив типа string, содержащий несколько произвольных элементов (скажем, названий видеоигр). Необходимо обеспечить, чтобы хотя бы два элемента имели числовые значения, а несколько элементов включали внутренние пробелы:

```
static void QueryOverStrings()
{
    // Предположим, что есть массив строк.
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                   "Fallout 3", "Daxter", "System Shock 2"};
}
```

Теперь модифицируем метод Main() для вызова QueryOverStrings():

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with LINQ to Objects *****\n");
    QueryOverStrings();
    Console.ReadLine();
}
```

При работе с любым массивом данных часто приходится извлекать из него подмножество элементов на основе определенного критерия. Возможно, требуется получить только элементы, которые содержат число (например, "System Shock 2", "Uncharted 2" и "Fallout 3"), имеют длину больше или меньше заданного количества символов, не содержат встроенных пробелов (скажем, "Morrowind" или "Daxter") и т.п. В то время как такие задачи определенно можно решать с использованием членов типа System.Array, прикладывая приличные усилия, выражения запросов LINQ значительно упрощают процесс.

Исходя из предположения, что из массива нужно получить только элементы, содержащие внутри себя пробел, и представить их в алфавитном порядке, можно построить следующее выражение запроса LINQ:

```
static void QueryOverStrings()
{
    // Предположим, что имеется массив строк.
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                   "Fallout 3", "Daxter", "System Shock 2"};

    // Построить выражение запроса для нахождения
    // элементов массива, которые содержат пробелы.
    IEnumerable<string> subset = from g in currentVideoGames
                                where g.Contains(" ") orderby g select g;

    // Вывести результаты.
    foreach (string s in subset)
        Console.WriteLine("Item: {0}", s);
}
```

Обратите внимание, что в созданном здесь выражении запроса применяются операции from, in, where, orderby и select языка LINQ. Формальности синтаксиса выражений запросов будут подробно излагаться далее в главе. Тем не менее, даже сейчас вы в состоянии прочесть данный оператор примерно так: "предоставить мне элементы из currentVideoGames, содержащие пробелы, в алфавитном порядке".

Каждому элементу, который соответствует критерию поиска, будет назначено имя g (от "game"); однако подошло бы любое допустимое имя переменной C#:

```
IEnumerable<string> subset = from game in currentVideoGames
                              where game.Contains(" ") orderby
                              game select game;
```

Возвращенная последовательность сохраняется в переменной по имени subset, которая имеет тип, реализующий обобщенную версию интерфейса IEnumerable<T>, где T — тип System.String (в конце концов, вы запрашиваете массив элементов string).

После получения результирующего набора его элементы затем просто выводятся на консоль с использованием стандартной конструкции `foreach`. Запустив приложение, вы получите следующий вывод:

```
***** Fun with LINQ to Objects *****
Item: Fallout 3
Item: System Shock 2
Item: Uncharted 2
```

Решение с использованием расширяющих методов

Применяемый ранее (и далее в главе) синтаксис LINQ называется *выражениями запросов LINQ*, которые представляют собой формат, похожий на SQL, но (отчасти досадно) отличающийся. Существует еще один синтаксис, который использует расширяющие методы. Большинство операторов LINQ можно записывать с применением любого из двух форматов; тем не менее, некоторые более сложные запросы будут требовать использования выражений запросов.

Создадим новый метод по имени `QueryOverStringsWithExtensionMethods()` и поместим в него такой код:

```
static void QueryOverStringsWithExtensionMethods()
{
    // Пусть имеется массив строк.
    string[] currentVideoGames = {"Morrowind", "Uncharted 2", "Fallout 3",
                                   "Daxter", "System Shock 2"};

    // Построить выражение запроса для поиска
    // в массиве элементов, содержащих пробелы.
    IEnumerable<string> subset =
        currentVideoGames.Where(g => g.Contains(" ")).OrderBy(g => g).Select(g => g);

    // Вывести результаты.
    foreach (string s in subset)
        Console.WriteLine("Item: {0}", s);
}
```

Код здесь тот же, что и в предыдущем методе, кроме выделенных полужирным строк. В них демонстрируется применение синтаксиса расширяющих методов, в котором для определения операций внутри каждого метода используются лямбда-выражения. Например, лямбда-выражение в методе `Where()` определяет условие (содержит ли значение пробел). Как и в синтаксисе выражений запросов, используемая для идентификации значения буква произвольна; в примере применяется *v* для видеоигр (video game).

Хотя результаты аналогичны (метод дает такой же вывод, как и предыдущий метод, использующий выражение запроса), вскоре вы увидите, что *тип* результирующего набора несколько отличается. В большинстве (если фактически не во всех) сценариях такое отличие не приводит к каким-либо проблемам и форматы могут применяться взаимозаменяемо.

Решение без использования LINQ

Конечно, применение LINQ никогда не бывает обязательным. При желании идентичный результирующий набор можно получить без участия LINQ с помощью таких программных конструкций, как операторы `if` и циклы `for`. Ниже приведен метод, который выдает тот же самый результат, что и `QueryOverStrings()`, но в намного более многословной манере:

```

static void QueryOverStringsLongHand()
{
    // Предположим, что имеется массив строк.
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};
    string[] gamesWithSpaces = new string[5];
    for (int i = 0; i < currentVideoGames.Length; i++)
    {
        if (currentVideoGames[i].Contains(" "))
            gamesWithSpaces[i] = currentVideoGames[i];
    }

    // Отсортировать набор.
    Array.Sort(gamesWithSpaces);

    // Вывести результаты.
    foreach (string s in gamesWithSpaces)
    {
        if (s != null)
            Console.WriteLine("Item: {0}", s);
    }
    Console.WriteLine();
}

```

Несмотря на возможные пути улучшения метода `QueryOverStringsLongHand()`, факт остается фактом — запросы LINQ способны радикально упростить процесс извлечения новых подмножеств данных из источника. Вместо построения вложенных циклов, сложной логики `if/else`, временных типов данных и т.п. компилятор C# сделает всю черновую работу, как только вы создадите подходящий запрос LINQ.

Выполнение рефлексии результирующего набора LINQ

А теперь определим в классе `Program` дополнительный вспомогательный метод по имени `ReflectOverQueryResults()`, который выводит на консоль разнообразные детали о результирующем наборе LINQ (обратите внимание на параметр типа `System.Object`, позволяющий учитывать множество типов результирующих наборов):

```

static void ReflectOverQueryResults(object resultSet,
                                    string queryType = "Query Expressions")
{
    Console.WriteLine($"***** Info about your query using {queryType} *****");
    // Вывести тип результирующего набора.
    Console.WriteLine("resultSet is of type: {0}", resultSet.GetType().Name);
    // Вывести местоположение результирующего набора.
    Console.WriteLine("resultSet location: {0}",
        resultSet.GetType().Assembly.GetName().Name);
}

```

Модифицируем код метода `QueryOverStrings()` следующим образом:

```

// Построить выражение запроса для поиска
// в массиве элементов, содержащих пробел.
IEnumerable<string> subset = from g in currentVideoGames
    where g.Contains(" ") orderby g select g;

```

ReflectOverQueryResults(subset);

```

// Вывести результаты.
foreach (string s in subset)
    Console.WriteLine("Item: {0}", s);

```


Запустив приложение, легко заметить, что переменная `subset` в действительности представляет собой экземпляр обобщенного типа `OrderedEnumerable<TElement, TKey>` (представленного в коде CIL как `OrderedEnumerable`2`), который является внутренним абстрактным типом, находящимся в сборке `System.Core.dll`:

```
***** Info about your query using Query Expressions*****
resultSet is of type: OrderedEnumerable`2
resultSet location: System.Core
```

Внесем такое же изменение в код метода `QueryOverStringsWithExtensionMethods()`, но с передачей во втором параметре строки `"Extension Methods"`:

```
// Построить выражение запроса для поиска
// в массиве элементов, содержащих пробел.
IEnumerable<string> subset =
    currentVideoGames.Where(g => g.Contains(" ")).OrderBy(g => g).Select(g => g);
ReflectOverQueryResults(subset, "Extension Methods");

// Вывести результаты.
foreach (string s in subset)
    Console.WriteLine("Item: {0}", s);
```

После запуска приложения выяснится, что переменная `subset` является экземпляром типа `System.Linq.WhereSelectEnumerableIterator`. Но если удалить из запроса конструкцию `Select(g=>g)`, то `subset` снова станет экземпляром типа `OrderedEnumerable<TElement, TKey>`. Что все это значит? Для подавляющего большинства разработчиков немного (если вообще что-либо). Оба типа являются производными от `IEnumerable<T>`, проход по ним осуществляется одинаковым образом и они оба способны создавать список или массив своих значений.

```
***** Info about your query using Extension Methods *****
resultSet is of type: WhereSelectEnumerableIterator`2
resultSet location: System.Core
```

На заметку! Многие типы, представляющие результат LINQ, в браузере объектов Visual Studio скрыты. Эти низкоуровневые типы не предназначены для прямого использования в приложениях.

LINQ и неявно типизированные локальные переменные

Хотя в приведенной программе относительно легко выяснить, что результирующий набор может быть интерпретирован как перечисление объектов `string` (например, `IEnumerable<string>`), тот факт, что подмножество на самом деле имеет тип `OrderedEnumerable<TElement, TKey>`, не настолько ясен.

Поскольку результирующие наборы LINQ могут быть представлены с применением порядочного количества типов из разнообразных пространств имен LINQ, было бы утомительно определять подходящий тип для хранения результирующего набора. Причина в том, что во многих случаях лежащий в основе тип не очевиден и даже напрямую не доступен в коде (и как вы увидите, в ряде ситуаций тип генерируется на этапе компиляции).

Чтобы еще более подчеркнуть данное обстоятельство, ниже показан дополнительный вспомогательный метод, определенный внутри класса `Program` (который должен быть вызван из метода `Main()`):

```
static void QueryOverInts()
{
    int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};
```

```
// Вывести только элементы меньше 10.
IEnumerable<int> subset = from i in numbers where i < 10 select i;

foreach (int i in subset)
    Console.WriteLine("Item: {0}", i);
ReflectOverQueryResults(subset);
}
```

В рассматриваемом случае переменная `subset` имеет совершенно другой внутренний тип. На этот раз тип, реализующий интерфейс `IEnumerable<int>`, представляет собой низкоуровневый класс по имени `WhereArrayIterator<T>`:

```
Item: 1
Item: 2
Item: 3
Item: 8

***** Info about your query *****
resultSet is of type: WhereArrayIterator`1
resultSet location: System.Core
```

Учитывая, что точный тип запроса LINQ определенно не очевиден, в первых примерах результаты запросов были представлены как переменная `IEnumerable<T>`, где `T` — тип данных в возвращенной последовательности (`string`, `int` и т.д.). Тем не менее, ситуация по-прежнему довольно запутана. Чтобы еще больше все усложнить, стоит упомянуть, что поскольку интерфейс `IEnumerable<T>` расширяет необобщенный `IEnumerable`, получать результат запроса LINQ допускается и так:

```
System.Collections.IEnumerable subset = from i in numbers where i < 10 select i;
```

К счастью, неявная типизация при работе с запросами LINQ значительно проясняет картину:

```
static void QueryOverInts()
{
    int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};

    // Здесь используется неявная типизация...
    var subset = from i in numbers where i < 10 select i;

    // ...и здесь тоже.
    foreach (var i in subset)
        Console.WriteLine("Item: {0} ", i);
    ReflectOverQueryResults(subset);
}
```

В качестве эмпирического правила: при захвате результатов запроса LINQ всегда необходимо использовать неявную типизацию. Однако помните, что (в подавляющем большинстве случаев) *действительное* возвращаемое значение имеет тип, реализующий интерфейс `IEnumerable<T>`.

Какой точно тип кроется за ним (`OrderedEnumerable<TElement, TKey>`, `WhereArrayIterator<T>` и т.п.), к делу не относится, и определять его вовсе не обязательно. Как было показано в предыдущем примере кода, для прохода по извлеченным данным можно просто применить ключевое слово `var` внутри конструкции `foreach`.

LINQ и расширяющие методы

Несмотря на то что в текущем примере совершенно не требуется напрямую писать какие-то расширяющие методы, на самом деле они благополучно используются на заднем плане. Выражения запросов LINQ могут применяться для прохода по содержимому

контейнеров данных, которые реализуют обобщенный интерфейс `IEnumerable<T>`. Тем не менее, класс `System.Array` в .NET (используемый для представления массива строк и массива целых чисел) не реализует этот контракт:

```
// Похоже, что тип System.Array не реализует
// корректную инфраструктуру для выражений запросов!
public abstract class Array : ICloneable, IList, ICollection,
    IEnumerable, IStructuralComparable, IStructuralEquatable
{
    ...
}
```

Хотя класс `System.Array` не реализует напрямую интерфейс `IEnumerable<T>`, он косвенно получает необходимую функциональность данного типа (а также многие другие члены, связанные с LINQ) через статический тип класса `System.Linq.Enumerable`.

В служебном классе `System.Linq.Enumerable` определено множество обобщенных расширяющих методов (таких как `Aggregate<T>()`, `First<T>()`, `Max<T>()` и т.д.), которые класс `System.Array` (и другие типы) получают в свое распоряжение на заднем плане. Таким образом, если вы примените операцию точки к локальной переменной `currentVideoGames`, то обнаружите большое количество членов, которые *отсутствуют* в формальном определении `System.Array` (рис. 12.1).

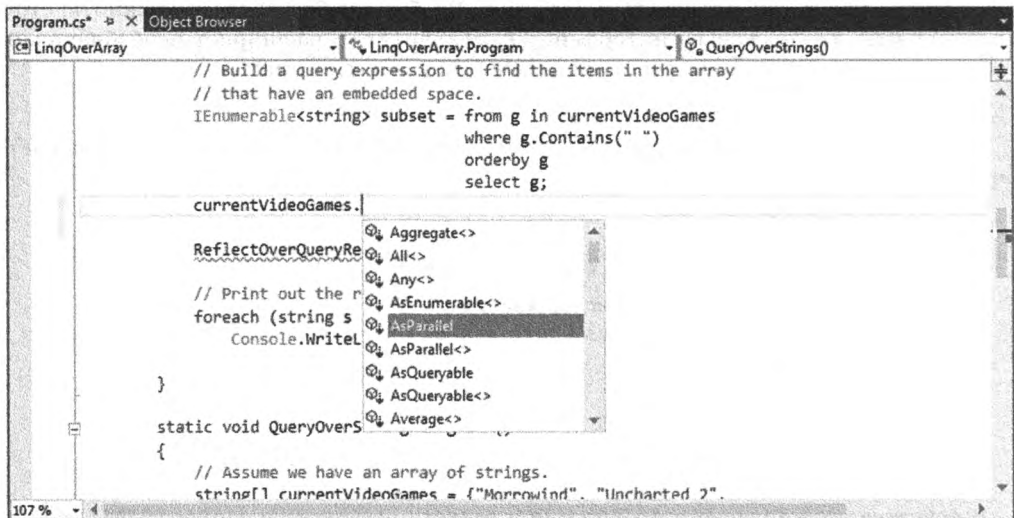


Рис. 12.1. Тип `System.Array` был расширен членами типа `System.Linq.Enumerable`

Роль отложенного выполнения

Еще один важный момент, касающийся выражений запросов LINQ, заключается в том, что в действительности они не оцениваются до тех пор, пока не начнется итерация по последовательности. Формально это называется *отложенным выполнением*. Преимущество такого подхода связано с возможностью применения одного и того же запроса LINQ многократно к тому же самому контейнеру и полной гарантией получения актуальных результатов. Взгляните на следующее обновление метода `QueryOverInts()`:

```
static void QueryOverInts()
{
    int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
}
```

```
// Получить числа меньше 10.
var subset = from i in numbers where i < 10 select i;

// Оператор LINQ здесь оценивается!
foreach (var i in subset)
    Console.WriteLine("{0} < 10", i);
Console.WriteLine();
// Изменить некоторые данные в массиве.
numbers[0] = 4;

// Снова производится оценка!
foreach (var j in subset)
    Console.WriteLine("{0} < 10", j);
Console.WriteLine();
ReflectOverQueryResults(subset);
}
```

Ниже показан вывод, полученный в результате запуска программы. Обратите внимание, что во второй итерации по запрошенной последовательности появился дополнительный член, т.к. для первого элемента массива было установлено значение меньше 10:

```
1 < 10
2 < 10
3 < 10
8 < 10

4 < 10
1 < 10
2 < 10
3 < 10
8 < 10
```

Среда Visual Studio обладает одним полезным аспектом: если вы поместите точку останова перед оценкой запроса LINQ, то получите возможность просматривать содержимое во время сеанса отладки. Просто наведите курсор мыши на переменную результирующего набора LINQ (subset на рис. 12.2) и вам будет предложено выполнить запрос, развернув узел Results View (Представление результатов).

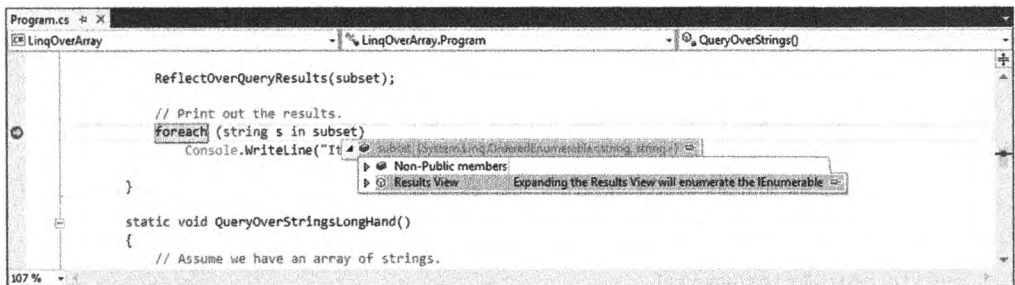


Рис. 12.2. Отладка выражений LINQ

Роль немедленного выполнения

Когда требуется оценить выражение LINQ за пределами логики foreach, можно вызывать любое количество расширяющих методов, определенных в типе Enumerable, таких как ToArray<T>, ToDictionary<TSource, TKey>() и ToList<T>(). Все методы приводят к выполнению запроса LINQ в момент их вызова для получения снимка данных. Затем полученным снимком данных можно манипулировать независимым образом.

```
static void ImmediateExecution()
{
    int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
    // Получить данные НЕМЕДЛЕННО как int[].
    int[] subsetAsIntArray =
        (from i in numbers where i < 10 select i).ToArray<int>();
    // Получить данные НЕМЕДЛЕННО как List<int>.
    List<int> subsetAsListOfInts =
        (from i in numbers where i < 10 select i).ToList<int>();
}
```

Обратите внимание, что для вызова методов `Enumerable` выражение LINQ целиком помещено в круглые скобки с целью приведения к корректному внутреннему типу (каким бы он ни был).

Вспомните из главы 9, что если компилятор C# в состоянии однозначно определить параметр типа обобщенного элемента, то вы не обязаны указывать этот параметр типа. Следовательно, `ToArray<T>` (или `ToList<T>`) можно было бы вызвать так:

```
int[] subsetAsIntArray =
    (from i in numbers where i < 10 select i).ToArray();
```

Полезность немедленного выполнения очевидна, когда нужно вернуть результаты запроса LINQ внешнему вызывающему коду, что и будет темой следующего раздела главы.

Исходный код. Проект `LinqOverArray` доступен в подкаталоге `Chapter_12`.

Возвращение результатов запроса LINQ

Внутри класса (или структуры) можно определить поле, значением которого будет результат запроса LINQ. Однако для этого нельзя использовать неявную типизацию (т.к. ключевое слово `var` не может применяться к полям), и целью запроса LINQ не могут быть данные уровня экземпляра, а потому он должен быть статическим. С учетом указанных ограничений необходимость в написании кода следующего вида будет возникать редко:

```
class LINQBasedFieldsAreClunky
{
    private static string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};

    // Здесь нельзя использовать неявную типизацию! Тип subset должен быть известен!
    private IEnumerable<string> subset = from g in currentVideoGames
        where g.Contains(" ") orderby g select g;

    public void PrintGames()
    {
        foreach (var item in subset)
        {
            Console.WriteLine(item);
        }
    }
}
```

Запросы LINQ очень часто определяются внутри области действия метода или свойства. Кроме того, для упрощения программирования результирующий набор будет

храниться в неявно типизированной локальной переменной, использующей ключевое слово `var`. Вспомните из главы 3, что неявно типизированные переменные не могут применяться для определения параметров, возвращаемых значений, а также полей класса или структуры.

Итак, вас наверняка интересует, каким образом вернуть результат запроса внешнему коду. Смотря по обстоятельствам. Если у вас есть результирующий набор, состоящий из строго типизированных данных, такой как массив строк или список `List<T>` объектов `Car`, тогда вы могли бы отказаться от использования ключевого слова `var` и указать подходящий тип `IEnumerable<T>` либо `IEnumerable` (т.к. `IEnumerable<T>` расширяет `IEnumerable`). Ниже приведен пример класса `Program` в новом проекте консольного приложения по имени `LinqRetValues`:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** LINQ Return Values *****\n");
        IEnumerable<string> subset = GetStringSubset();

        foreach (string item in subset)
        {
            Console.WriteLine(item);
        }

        Console.ReadLine();
    }
    static IEnumerable<string> GetStringSubset()
    {
        string[] colors = {"Light Red", "Green",
            "Yellow", "Dark Red", "Red", "Purple"};

        // Обратите внимание, что subset является
        // совместимым с IEnumerable<string> объектом.
        IEnumerable<string> theRedColors = from c in colors
            where c.Contains("Red") select c;

        return theRedColors;
    }
}
```

Результат выглядит вполне ожидаемо:

```
Light Red
Dark Red
Red
```

Возвращение результатов LINQ посредством немедленного выполнения

Рассмотренный пример работает ожидаемым образом только потому, что возвращаемое значение `GetStringSubset()` и запрос LINQ внутри этого метода были строго типизированными. Если применить ключевое слово `var` для определения переменной `subset`, то возвращать значение будет разрешено, *только* если метод по-прежнему прототипирован с возвращаемым типом `IEnumerable<string>` (и если неявно типизированная локальная переменная на самом деле совместима с указанным возвращаемым типом).

Поскольку оперировать с типом `IEnumerable<T>` несколько неудобно, можно задействовать немедленное выполнение. Скажем, вместо возвращения `IEnumerable<string>` можно было бы вернуть просто `string[]` при условии трансформации последовательности в строго типизированный массив. Именно такое действие выполняет новый метод класса `Program`:

```
static string[] GetStringSubsetAsArray()
{
    string[] colors = {"Light Red", "Green",
        "Yellow", "Dark Red", "Red", "Purple"};

    var theRedColors = from c in colors
        where c.Contains("Red") select c;

    // Отобразить результаты в массив.
    return theRedColors.ToArray();
}
```

В таком случае вызывающий код совершенно не знает, что полученный им результат поступил от запроса LINQ, и он просто работает с массивом строк вполне ожидаемым образом. Вот пример:

```
foreach (string item in GetStringSubsetAsArray())
{
    Console.WriteLine(item);
}
```

Немедленное выполнение также важно при попытке вернуть вызывающему коду результаты проекции LINQ. Мы исследуем эту тему чуть позже в главе. А сейчас давайте посмотрим, как применять запросы LINQ к обобщенным и необобщенным объектам коллекций.

Исходный код. Проект `LinqRetValues` доступен в подкаталоге `Chapter_12`.

Применение запросов LINQ к объектам коллекций

Помимо извлечения результатов из простого массива данных выражения запросов LINQ могут также манипулировать данными внутри классов из пространства имен `System.Collections.Generic`, таких как `List<T>`. Создадим новый проект консольного приложения по имени `ListOverCollections` и определим базовый класс `Car`, который поддерживает текущую скорость, цвет, производителя и дружественное имя:

```
class Car
{
    public string PetName {get; set;} = "";
    public string Color {get; set;} = "";
    public int Speed {get; set;}
    public string Make {get; set;} = "";
}
```

Теперь определим в методе `Main()` локальную переменную типа `List<T>` для хранения элементов типа `Car` и с помощью синтаксиса инициализации объектов заполним список несколькими новыми объектами `Car`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** LINQ over Generic Collections *****\n");
```

```
// Создать список List<> объектов Car.
List<Car> myCars = new List<Car>() {
    new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
    new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
    new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
    new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
    new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
};

Console.ReadLine();
}
```

Доступ к содержащимся в контейнере подобъектам

Применение запроса LINQ к обобщенному контейнеру ничем не отличается от такого же действия в отношении простого массива, потому что LINQ to Objects может использоваться с любым типом, реализующим интерфейс `IEnumerable<T>`. На этот раз цель заключается в построении выражения запроса для выборки из списка `myCars` только тех объектов `Car`, у которых значение скорости больше 55.

После получения подмножества на консоль будет выведено имя каждого объекта `Car` за счет обращения к его свойству `PetName`. Предположим, что определен следующий вспомогательный метод (принимаящий параметр `List<Car>`), который вызывается внутри `Main()`:

```
static void GetFastCars(List<Car> myCars)
{
    // Найти в List<> все объекты Car, у которых значение Speed больше 55.
    var fastCars = from c in myCars where c.Speed > 55 select c;
    foreach (var car in fastCars)
    {
        Console.WriteLine("{0} is going too fast!", car.PetName);
    }
}
```

Обратите внимание, что выражение запроса захватывает из `List<T>` только те элементы, у которых значение `Speed` больше 55. Запустив приложение, вы увидите, что критерию поиска отвечают только два элемента — Henry и Daisy.

Чтобы построить более сложный запрос, можно искать только автомобили марки BMW со значением `Speed` больше 90. Для этого нужно просто создать составной булевский оператор с применением операции `&&` языка C#:

```
static void GetFastBMWs(List<Car> myCars)
{
    // Найти быстрые автомобили BMW!
    var fastCars =
        from c in myCars where c.Speed > 90 && c.Make == "BMW" select c;
    foreach (var car in fastCars)
    {
        Console.WriteLine("{0} is going too fast!", car.PetName);
    }
}
```

Теперь выводится только одно имя Henry.

Применение запросов LINQ к необобщенным коллекциям

Вспомните, что операции запросов LINQ спроектированы для работы с любым типом, реализующим интерфейс `IEnumerable<T>` (как напрямую, так и через расширяющие методы). Учитывая то, что класс `System.Array` оснащен всей необходимой инфраструктурой, может оказаться сюрпризом, что унаследованные (необобщенные) контейнеры в пространстве имен `System.Collections` такой поддержкой не обладают. К счастью, итерация по данным, содержащимся внутри необобщенных коллекций, по-прежнему возможна с использованием обобщенного расширяющего метода `Enumerable.OfType<T>()`.

При вызове метода `OfType<T>()` на объекте необобщенной коллекции (наподобие `ArrayList`) нужно просто указать тип элемента внутри контейнера, чтобы извлечь совместимый с `IEnumerable<T>` объект. Сохранить этот элемент данных в коде можно посредством неявно типизированной переменной.

Взгляните на показанный ниже новый метод, который заполняет `ArrayList` набором объектов `Car` (не забудьте импортировать пространство имен `System.Collections` в файл `Program.cs`):

```
static void LINQOverArrayList()
{
    Console.WriteLine("***** LINQ over ArrayList *****");
    // Необобщенная коллекция объектов Car.
    ArrayList myCars = new ArrayList() {
        new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
        new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
        new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
        new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
        new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
    };
    // Трансформировать ArrayList в тип, совместимый с IEnumerable<T>.
    var myCarsEnum = myCars.OfType<Car>();
    // Создать выражение запроса, нацеленное на совместимый с IEnumerable<T> тип.
    var fastCars = from c in myCarsEnum where c.Speed > 55 select c;
    foreach (var car in fastCars)
    {
        Console.WriteLine("{0} is going too fast!", car.PetName);
    }
}
```

Аналогично предыдущим примерам этот метод, вызванный в `Main()`, отобразит только имена `Henry` и `Daisy`, основываясь на формате запроса LINQ.

Фильтрация данных с использованием метода `OfType<T>()`

Как вы уже знаете, необобщенные типы способны содержать любые комбинации элементов, поскольку члены этих контейнеров (вроде `ArrayList`) прототипированы для приема `System.Object`. Например, предположим, что `ArrayList` содержит разные элементы, часть которых являются числовыми. Получить подмножество, состоящее только из числовых данных, можно с помощью метода `OfType<T>()`, т.к. во время итерации он отфильтрует элементы, тип которых отличается от заданного:

```
static void OfTypeAsFilter()
{
    // Извлечь из ArrayList целочисленные значения.
    ArrayList myStuff = new ArrayList();
```

```

myStuff.AddRange(new object[] { 10, 400, 8, false, new Car(), "string data" });
var myInts = myStuff.OfType<int>();

// Выводит 10, 400 и 8.
foreach (int i in myInts)
{
    Console.WriteLine("Int value: {0}", i);
}
}

```

К настоящему моменту вы уже умеете применять запросы LINQ к массивам, обобщенным и необобщенным коллекциям. Контейнеры подобного рода содержат элементарные типы C# (целочисленные и строковые данные), а также специальные классы. Следующей задачей будет изучение многочисленных дополнительных операций LINQ, которые могут использоваться для построения более сложных и полезных запросов.

Исходный код. Проект `LinqOverCollectons` доступен в подкаталоге `Chapter_12`.

Исследование операций запросов LINQ

В языке C# предопределено порядочное число операций запросов. Некоторые часто применяемые из них перечислены в табл. 12.3.

На заметку! Документация .NET Framework SDK содержит подробные сведения по каждой операции LINQ языка C# (ищите раздел “LINQ General Programming Guide” (“Общее руководство по программированию на LINQ”)).

Таблица 12.3. Распространенные операции запросов LINQ

Операции запросов	Описание
<code>from, in</code>	Используются для определения основы любого выражения LINQ, позволяющей извлекать подмножество данных из подходящего контейнера
<code>where</code>	Применяется для определения ограничений относительно того, какие элементы должны извлекаться из контейнера
<code>select</code>	Используется для выборки последовательности из контейнера
<code>join, on, equals, into</code>	Выполняют соединения на основе указанного ключа. Помните, что эти “соединения” ничего не обязаны делать с данными в реляционной базе данных
<code>orderby, ascending, descending</code>	Позволяют упорядочить результирующий набор по возрастанию или убыванию
<code>group, by</code>	Выдают подмножество с данными, сгруппированными по указанному значению

В дополнение к неполному списку операций, приведенному в табл. 12.3, класс `System.Linq.Enumerable` предлагает набор методов, которые не имеют прямого сокращенного обозначения в виде операций запросов C#, а доступны как расширяющие методы. Эти обобщенные методы можно вызывать для трансформации результирующего набора разными способами (`Reverse<>()`, `ToArray<>()`, `ToList<>()` и т.д.). Некоторые из них применяются для извлечения одиночных элементов из результирующего набора, другие выполняют разнообразные операции над множествами (`Distinct<>()`,

Union<>(), Intersect<>() и т.п.), а еще одни агрегируют результаты (Count<>(), Sum<>(), Min<>(), Max<>() и т.д.).

Чтобы приступить к исследованию более замысловатых запросов LINQ, создадим новый проект консольного приложения по имени FunWithLinqExpressions и затем определим массив или коллекцию некоторых выборочных данных. В проекте FunWithLinqExpressions мы создадим массив объектов типа ProductInfo, определенного следующим образом:

```
class ProductInfo
{
    public string Name {get; set;} = "";
    public string Description {get; set;} = "";
    public int NumberInStock {get; set;} = 0;
    public override string ToString()
        => $"Name={Name}, Description={Description},
           Number in Stock={NumberInStock}";
}
```

Теперь заполним массив объектами ProductInfo внутри метода Main():

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Query Expressions *****\n");
    // Этот массив будет основой для тестирования...
    ProductInfo[] itemsInStock = new[] {
        new ProductInfo{ Name = "Mac's Coffee",
            Description = "Coffee with TEETH",
            NumberInStock = 24},
        new ProductInfo{ Name = "Milk Maid Milk",
            Description = "Milk cow's love",
            NumberInStock = 100},
        new ProductInfo{ Name = "Pure Silk Tofu",
            Description = "Bland as Possible",
            NumberInStock = 120},
        new ProductInfo{ Name = "Crunchy Pops",
            Description = "Cheezy, peppery goodness",
            NumberInStock = 2},
        new ProductInfo{ Name = "RipOff Water",
            Description = "From the tap to your wallet",
            NumberInStock = 100},
        new ProductInfo{ Name = "Classic Valpo Pizza",
            Description = "Everyone loves pizza'",
            NumberInStock = 73}
    };
    // Здесь мы будем вызывать разнообразные методы!
    Console.ReadLine();
}
```

Базовый синтаксис выборки

Поскольку синтаксическая корректность выражения запроса LINQ проверяется на этапе компиляции, вы должны помнить, что порядок следования операций критически важен. В простейшем виде каждый запрос LINQ строится с использованием операций from, in и select. Вот базовый шаблон, который нужно соблюдать:

```
var результат =
    from сопоставляемыйЭлемент in контейнер select сопоставляемыйЭлемент;
```

Элемент после операции `from` представляет элемент, соответствующий критерию запроса LINQ; именовать его можно по своему усмотрению. Элемент после операции `in` представляет контейнер данных, в котором производится поиск (массив, коллекция, документ XML и т.д.).

Рассмотрим простой запрос, не делающий ничего кроме извлечения каждого элемента контейнера (по поведению похожий на SQL-оператор `SELECT *` в базе данных):

```
static void SelectEverything(ProductInfo[] products)
{
    // Получить все!
    Console.WriteLine("All product details:");
    var allProducts = from p in products select p;
    foreach (var prod in allProducts)
    {
        Console.WriteLine(prod.ToString());
    }
}
```

По правде говоря, это выражение запроса не особенно полезно, т.к. выдаст подмножество, идентичное содержимому входного параметра. При желании можно извлечь только значения `Name` каждого товара, применив следующий синтаксис выборки:

```
static void ListProductNames(ProductInfo[] products)
{
    // Теперь получить только наименования товаров.
    Console.WriteLine("Only product names:");
    var names = from p in products select p.Name;
    foreach (var n in names)
    {
        Console.WriteLine("Name: {0}", n);
    }
}
```

Получение подмножества данных

Чтобы получить определенное подмножество из контейнера, можно использовать операцию `where`. Общий шаблон запроса становится таким:

```
var результат =
    from элемент in контейнер where булевскоеВыражение select элемент;
```

Обратите внимание, что операция `where` ожидает выражение, результатом вычисления которого является булевское значение. Например, чтобы извлечь из аргумента `ProductInfo[]` только товарные позиции, складские запасы которых составляют более 25 единиц, можно написать следующий код:

```
static void GetOverstock(ProductInfo[] products)
{
    Console.WriteLine("The overstock items!");
    // Получить только товары со складским запасом более 25 единиц.
    var overstock = from p in products where p.NumberInStock > 25 select p;
    foreach (ProductInfo c in overstock)
    {
        Console.WriteLine(c.ToString());
    }
}
```

Как демонстрировалось ранее в главе, при указании конструкции `where` разрешено применять любые операции C# для построения сложных выражений. Например, вспом-

ните запрос, который извлекал только автомобили марки BMW, движущиеся со скоростью минимум 100 миль в час:

```
// Получить автомобили BMW, движущиеся со скоростью минимум 100 миль в час.
var onlyFastBMWws = from c in myCars
                    where c.Make == "BMW" && c.Speed >= 100 select c;
foreach (Car c in onlyFastBMWws)
{
    Console.WriteLine("{0} is going {1} MPH", c.PetName, c.Speed);
}
```

Проецирование новых типов данных

Новые формы данных также можно проецировать из существующего источника данных. Давайте предположим, что необходимо принять входной параметр `ProductInfo[]` и получить результирующий набор, который учитывает только имя и описание каждого товара. Для этого понадобится определить оператор `select`, динамически выдающий новый анонимный тип:

```
static void GetNamesAndDescriptions(ProductInfo[] products)
{
    Console.WriteLine("Names and Descriptions:");
    var nameDesc = from p in products select new { p.Name, p.Description };
    foreach (var item in nameDesc)
    {
        // Можно было бы также использовать свойства Name и Description напрямую.
        Console.WriteLine(item.ToString());
    }
}
```

Не забывайте, что когда запрос LINQ использует проекцию, нет никакого способа узнать лежащий в ее основе тип данных, т.к. он определяется на этапе компиляции. В подобных случаях ключевое слово `var` является обязательным. Кроме того, вспомните о невозможности создания методов с неявно типизированными возвращаемыми значениями. Таким образом, следующий метод не скомпилируется:

```
static var GetProjectedSubset(ProductInfo[] products)
{
    var nameDesc = from p in products select new { p.Name, p.Description };
    return nameDesc; // Так поступать нельзя!
}
```

В случае необходимости возвращения спроецированных данных вызывающему коду один из подходов предусматривает трансформацию результата запроса в объект `System.Array` из .NET с применением расширяющего метода `ToArray()`. Следовательно, модифицировав выражение запроса, как показано ниже:

```
// Теперь возвращаемым значением является объект Array.
static Array GetProjectedSubset(ProductInfo[] products)
{
    var nameDesc = from p in products select new { p.Name, p.Description };
    // Отобразить набор анонимных объектов на объект Array.
    return nameDesc.ToArray();
}
```

метод `GetProjectedSubset()` можно вызвать в методе `Main()` и обработать возвращенные им данные:

```
Array objs = GetProjectedSubset(itemsInStock);
foreach (object o in objs)
{
    Console.WriteLine(o); //Вызывает метод ToString() на каждом анонимном объекте.
}
```

Как видите, здесь должен использоваться буквальный объект `System.Array`, а изменять синтаксис объявления массива `C#` невозможно, учитывая, что лежащий в основе проекции тип неизвестен, поскольку речь идет об анонимном классе, который сгенерирован компилятором. Кроме того, параметр типа для обобщенного метода `ToArry<T>()` не указывается, потому что он тоже не известен вплоть до этапа компиляции.

Очевидная проблема связана с утратой строгой типизации, т.к. каждый элемент в объекте `Array` считается относящимся к типу `Object`. Тем не менее, когда нужно вернуть результирующий набор LINQ, полученный посредством операции проецирования, то трансформация данных в тип `Array` (или другой подходящий контейнер через другие члены типа `Enumerable`) обязательна.

Подсчет количества с использованием класса `Enumerable`

При проецировании новых пакетов данных у вас может возникнуть необходимость высчитать количество элементов, возвращаемых внутри последовательности. Для определения числа элементов, которые возвращаются из выражения запроса LINQ, можно применять расширяющий метод `Count()` класса `Enumerable`. Например, следующий метод будет искать в локальном массиве все объекты `string`, которые имеют длину, превышающую шесть символов, и выводить их количество:

```
static void GetCountFromQuery()
{
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                   "Fallout 3", "Daxter", "System Shock 2"};
    // Получить количество элементов из запроса.
    int numb = (from g in currentVideoGames where g.Length > 6 select g).Count();
    // Вывести количество элементов.
    Console.WriteLine("{0} items honor the LINQ query.", numb);
}
```

Изменение порядка следования элементов в результирующих наборах на противоположный

Изменить порядок следования элементов в результирующем наборе на противоположный довольно легко с помощью расширяющего метода `Reverse<T>()` класса `Enumerable`. Например, в показанном далее методе выбираются все элементы из входного параметра `ProductInfo[]` в обратном порядке:

```
static void ReverseEverything(ProductInfo[] products)
{
    Console.WriteLine("Product in reverse:");
    var allProducts = from p in products select p;
    foreach (var prod in allProducts.Reverse())
    {
        Console.WriteLine(prod.ToString());
    }
}
```

Выражения сортировки

В начальных примерах настоящей главы вы видели, что в выражении запроса может использоваться операция `orderby` для сортировки элементов в подмножестве по заданному значению. По умолчанию принят порядок по возрастанию, поэтому строки сортируются в алфавитном порядке, числовые значения — от меньшего к большему и т.д. Если вы хотите просматривать результаты в порядке по убыванию, просто включите в выражение запроса операцию `descending`. Взгляните на следующий метод:

```
static void AlphabetizeProductNames(ProductInfo[] products)
{
    // Получить названия товаров в алфавитном порядке.
    var subset = from p in products orderby p.Name select p;
    Console.WriteLine("Ordered by Name:");
    foreach (var p in subset)
    {
        Console.WriteLine(p.ToString());
    }
}
```

Хотя порядок по возрастанию является стандартным, свои намерения можно прояснить, явно указав операцию `ascending`:

```
var subset = from p in products orderby p.Name ascending select p;
```

Для получения элементов в порядке убывания служит операция `descending`:

```
var subset = from p in products orderby p.Name descending select p;
```

LINQ как лучшее средство построения диаграмм Венна

Класс `Enumerable` поддерживает набор расширяющих методов, которые позволяют применять два (или более) запроса LINQ в качестве основы для нахождения объединений, разностей, конкатенаций и пересечений данных. Первым мы рассмотрим расширяющий метод `Except()`. Он возвращает результирующий набор LINQ, содержащий разность между двумя контейнерами, которой в этом случае является значение `Yugo`:

```
static void DisplayDiff()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
    var carDiff = (from c in myCars select c).Except(from c2 in yourCars select c2);
    Console.WriteLine("Here is what you don't have, but I do:");
    foreach (string s in carDiff)
        Console.WriteLine(s); // Выводит Yugo.
}
```

Метод `Intersect()` возвращает результирующий набор, который содержит общие элементы данных в наборе контейнеров. Например, следующий метод возвращает последовательность из `Aztec` и `BMW`:

```
static void DisplayIntersection()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
    // Получить общие члены.
    var carIntersect = (from c in myCars select c)
        .Intersect(from c2 in yourCars select c2);
}
```

```

Console.WriteLine("Here is what we have in common:");
foreach (string s in carIntersect)
    Console.WriteLine(s); // Выводит Aztec и BMW.
}

```

Метод `Union()` возвращает результирующий набор, который включает все члены множества запросов LINQ. Подобно любому объединению, даже если общий член встречается более одного раза, повторяющихся значений в результирующем наборе не будет. Следовательно, показанный ниже метод выведет на консоль значения Yugo, Aztec, BMW и Saab:

```

static void DisplayUnion()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

    // Получить объединение двух контейнеров.
    var carUnion = (from c in myCars select c).Union(from c2 in yourCars select c2);

    Console.WriteLine("Here is everything:");
    foreach (string s in carUnion)
        Console.WriteLine(s); // Выводит все общие члены.
}

```

Наконец, расширяющий метод `Concat()` возвращает результирующий набор, который является прямой конкатенацией результирующих наборов LINQ. Например, следующий метод выводит на консоль результаты Yugo, Aztec, BMW, BMW, Saab и Aztec:

```

static void DisplayConcat()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

    var carConcat = (from c in myCars select c)
        .Concat(from c2 in yourCars select c2);

    // Выводит:
    // Yugo Aztec BMW BMW Saab Aztec.
    foreach (string s in carConcat)
        Console.WriteLine(s);
}

```

Устранение дубликатов

При вызове расширяющего метода `Concat()` в результате очень легко получить избыточные элементы, и зачастую это может быть именно тем, что нужно. Однако в других случаях может понадобиться удалить дублированные элементы данных. Для этого необходимо просто вызвать расширяющий метод `Distinct()`:

```

static void DisplayConcatNoDups()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

    var carConcat = (from c in myCars select c).Concat(from c2 in yourCars select c2);

    // Выводит:
    // Yugo Aztec BMW Saab.
    foreach (string s in carConcat.Distinct())
        Console.WriteLine(s);
}

```


Операции агрегирования LINQ

Запросы LINQ могут также проектироваться для выполнения над результирующим набором разнообразных операций агрегирования. Одним из примеров может служить расширяющий метод `Count()`. Другие возможности включают получение среднего, максимального, минимального или суммы значений с использованием членов `Average()`, `Max()`, `Min()` либо `Sum()` класса `Enumerable`. Вот простой пример:

```
static void AggregateOps()
{
    double[] winterTemps = { 2.0, -21.3, 8, -4, 0, 8.2 };
    // Разнообразные примеры агрегации. Выводит максимальную температуру:
    Console.WriteLine("Max temp: {0}", (from t in winterTemps select t).Max());
    // Выводит минимальную температуру:
    Console.WriteLine("Min temp: {0}", (from t in winterTemps select t).Min());
    // Выводит среднюю температуру:
    Console.WriteLine("Average temp: {0}", (from t in winterTemps select t).Average());
    // Выводит сумму всех температур:
    Console.WriteLine("Sum of all temps: {0}", (from t in winterTemps select t).Sum());
}
```

Приведенные примеры должны предоставить достаточный объем сведений, чтобы вы освоились с процессом построения выражений запросов LINQ. Хотя существуют дополнительные операции, которые пока еще не рассматривались, вы увидите примеры позже в книге, когда речь пойдет о связанных технологиях LINQ. В завершение вводного экскурса в LINQ оставшиеся материалы главы посвящены подробностям отношений между операциями запросов LINQ и лежащей в основе объектной моделью.

Исходный код. Проект `FunWithLinqExpressions` доступен в подкаталоге `Chapter_12`.

Внутреннее представление операторов запросов LINQ

К настоящему моменту вы уже знакомы с процессом построения выражений запросов с применением разнообразных операций запросов C# (таких как `from`, `in`, `where`, `orderby` и `select`). Вдобавок вы узнали, что определенная функциональность API-интерфейса LINQ to Objects доступна только через вызов расширяющих методов класса `Enumerable`. В действительности компилятор C# транслирует все операции запросов LINQ в вызовы методов класса `Enumerable`.

Огромное количество методов класса `Enumerable` прототипированы для приема делегатов в качестве аргументов. В частности, многие методы требуют обобщенный делегат по имени `Func<>`, который был описан во время рассмотрения обобщенных делегатов в главе 9. Взгляните на метод `Where()` класса `Enumerable`, вызываемый автоматически в случае использования операции `where`:

```
// Перегруженные версии метода Enumerable.Where<T>().
// Обратите внимание, что второй параметр имеет тип System.Func<>.
public static IEnumerable<TSource>
    Where<TSource>(this IEnumerable<TSource> source,
        System.Func<TSource, int, bool> predicate)

public static IEnumerable<TSource>
    Where<TSource>(this IEnumerable<TSource> source,
        System.Func<TSource, bool> predicate)
```

Делегат `Func<>` представляет шаблон фиксированной функции с набором до 16 аргументов и возвращаемым значением. Если вы исследуете этот тип в браузере объектов Visual Studio, то заметите разнообразные формы делегата `Func<>`. Например:

```
// Различные формы делегата Func<>.
public delegate TResult Func<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
public delegate TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)
public delegate TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2)
public delegate TResult Func<T1,TResult>(T1 arg1)
public delegate TResult Func<TResult>()
```

Учитывая, что многие члены класса `System.Linq.Enumerable` при вызове ожидают получить делегат, можно вручную создать новый тип делегата и написать для него необходимые целевые методы, применить анонимный метод C# или определить подходящее лямбда-выражение. Независимо от выбранного подхода конечный результат будет одним и тем же.

Хотя использование операций запросов LINQ, несомненно, является самым простым способом построения запросов LINQ, давайте взглянем на все возможные подходы, чтобы увидеть связь между операциями запросов C# и лежащим в основе типом `Enumerable`.

Построение выражений запросов с применением операций запросов

Для начала создадим новый проект консольного приложения по имени `LinqUsingEnumerable`. В классе `Program` будут определены статические вспомогательные методы (вызываемые внутри `Main()`) для иллюстрации разнообразных подходов к построению выражений запросов LINQ.

Первый метод, `QueryStringsWithOperators()`, предлагает наиболее прямой способ создания выражений запросов и идентичен коду примера `LinqOverArray`, который приводился ранее в главе:

```
static void QueryStringWithOperators()
{
    Console.WriteLine("***** Using Query Operators *****");
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};
    var subset = from game in currentVideoGames
        where game.Contains(" ") orderby game select game;
    foreach (string s in subset)
        Console.WriteLine("Item:{0}", s);
}
```

Очевидное преимущество использования операций запросов C# при построении выражений запросов заключается в том, что делегаты `Func<>` и вызовы методов `Enumerable` остаются вне поля зрения и внимания, т.к. выполнение необходимой трансляции возлагается на компилятор C#. Бесспорно, создание выражений LINQ с применением различных операций запросов (`from`, `in`, `where` или `orderby`) является наиболее распространенным и простым подходом.

Построение выражений запросов с использованием типа `Enumerable` и лямбда-выражений

Имейте в виду, что применяемые здесь операции запросов LINQ представляют собой сокращенные версии вызова расширяющих методов, определенных в типе `Enumerable`. Рассмотрим показанный ниже метод `QueryStringsWithEnumerableAndLambdas()`, который обрабатывает локальный массив строк, но на этот раз в нем напрямую используются расширяющие методы `Enumerable`:

```
static void QueryStringsWithEnumerableAndLambdas()
{
    Console.WriteLine("***** Using Enumerable / Lambda Expressions *****");
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};
    // Построить выражение запроса с использованием расширяющих методов,
    // предоставленных типу Array через тип Enumerable.
    var subset = currentVideoGames.Where(game => game.Contains(" "))
        .OrderBy(game => game).Select(game => game);
    // Вывести результаты.
    foreach (var game in subset)
        Console.WriteLine("Item: {0}", game);
    Console.WriteLine();
}
```

Здесь сначала вызывается расширяющий метод `Where()` на строковом массиве `currentVideoGames`. Вспомните, что класс `Array` получает данный метод от класса `Enumerable`. Метод `Enumerable.Where()` требует параметра типа делегата `System.Func<T1, TResult>`. Первый параметр типа упомянутого делегата представляет совместимые с интерфейсом `IEnumerable<T>` данные для обработки (массив строк в рассматриваемом случае), а второй — результирующие данные метода, которые получаются от единственного оператора, вставленного в лямбда-выражение.

Возвращаемое значение метода `Where()` в приведенном примере кода скрыто от глаз, но “за кулисами” работа происходит с типом `OrderedEnumerable`. На объекте указанного типа вызывается обобщенный метод `OrderBy()`, который также принимает параметр типа делегата `Func<>`. Теперь производится передача всех элементов по очереди посредством подходящего лямбда-выражения. Результатом вызова `OrderBy()` является новая упорядоченная последовательность первоначальных данных.

И, наконец, осуществляется вызов метода `Select()` на последовательности, возвращенной `OrderBy()`, который в итоге дает окончательный набор данных, сохраняемый в неявно типизированной переменной по имени `subset`. Конечно, такой “длинный” запрос LINQ несколько сложнее для восприятия, чем предыдущий пример с операциями запросов LINQ. Без сомнения, часть сложности связана с объединением в цепочку вызовов посредством операции точки. Вот тот же самый запрос с выделением каждого шага в отдельный фрагмент (разбивать запрос на части можно разными способами):

```
static void QueryStringsWithEnumerableAndLambdas2()
{
    Console.WriteLine("***** Using Enumerable / Lambda Expressions *****");
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};
    // Разбить на части.
    var gamesWithSpaces = currentVideoGames.Where(game => game.Contains(" "));
    var orderedGames = gamesWithSpaces.OrderBy(game => game);
    var subset = orderedGames.Select(game => game);
    foreach (var game in subset)
        Console.WriteLine("Item: {0}", game);
    Console.WriteLine();
}
```

Как видите, построение выражения запроса LINQ с применением методов класса `Enumerable` напрямую приводит к намного более многословному запросу, чем в случае использования операций запросов C#. Кроме того, поскольку методы `Enumerable` требуют передачи делегатов в качестве параметров, обычно необходимо писать лямбда-выражения, чтобы обеспечить обработку входных данных внутренней целью делегата.

Построение выражений запросов с использованием типа `Enumerable` и анонимных методов

Учитывая, что лямбда-выражения C# — это просто сокращенный способ работы с анонимными методами, рассмотрим третье выражение запроса внутри вспомогательного метода `QueryStringsWithAnonymousMethods()`:

```
static void QueryStringsWithAnonymousMethods()
{
    Console.WriteLine("***** Using Anonymous Methods *****");
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};
    // Построить необходимые делегаты Func<> с использованием анонимных методов.
    Func<string, bool> searchFilter =
        delegate(string game) { return game.Contains(" "); };
    Func<string, string> itemToProcess = delegate(string s) { return s; };
    // Передать делегаты в методы класса Enumerable.
    var subset = currentVideoGames.Where(searchFilter).OrderBy(itemToProcess)
        .Select(itemToProcess);

    // Вывести результаты.
    foreach (var game in subset)
        Console.WriteLine("Item: {0}", game);
    Console.WriteLine();
}
```

Такой вариант выражения запроса оказывается еще более многословным из-за создания вручную делегатов `Func<>`, применяемых методами `Where()`, `OrderBy()` и `Select()` класса `Enumerable`. Положительная сторона данного подхода связана с тем, что синтаксис анонимных методов позволяет заключить всю обработку, выполняемую делегатами, в единственное определение метода. Тем не менее, этот метод функционально эквивалентен методам `QueryStringsWithEnumerableAndLambdas()` и `QueryStringsWithOperators()`, созданным в предшествующих разделах.

Построение выражений запросов с использованием типа `Enumerable` и низкоуровневых делегатов

Наконец, если вы хотите строить выражение запроса с применением *по-настоящему многословного подхода*, то можете отказаться от использования синтаксиса лямбда-выражений и анонимных методов и напрямую создавать цели делегатов для каждого типа `Func<>`. Ниже показана финальная версия выражения запроса, смоделированная внутри нового типа класса по имени `VeryComplexQueryExpression`:

```
class VeryComplexQueryExpression
{
    public static void QueryStringsWithRawDelegates()
    {
        Console.WriteLine("***** Using Raw Delegates *****");
        string[] currentVideoGames = {"Morrowind", "Uncharted 2",
            "Fallout 3", "Daxter", "System Shock 2"};
        // Построить необходимые делегаты Func<>.
        Func<string, bool> searchFilter = new Func<string, bool>(Filter);
        Func<string, string> itemToProcess = new Func<string, string>(ProcessItem);
        // Передать делегаты в методы класса Enumerable.
        var subset =
            currentVideoGames.Where(searchFilter).OrderBy(itemToProcess)
                .Select(itemToProcess);
    }
}
```

```
// Вывести результаты.
foreach (var game in subset)
    Console.WriteLine("Item: {0}", game);
    Console.WriteLine();
}
// Цели делегатов.
public static bool Filter(string game) {return game.Contains(" ");}
public static string ProcessItem(string game) { return game; }
}
```

Чтобы протестировать такую версию логики обработки строк, метод `QueryStrings` `WithRawDelegates()` понадобится вызвать внутри метода `Main()` класса `Program`:

```
VeryComplexQueryExpression.QueryStringsWithRawDelegates();
```

Если теперь запустить приложение, чтобы опробовать все возможные подходы, вывод окажется идентичным независимо от выбранного пути. Запомните перечисленные ниже моменты относительно выражений запросов и их внутреннего представления.

- Выражения запросов создаются с применением разнообразных операций запросов C#.
- Операции запросов — это просто сокращенное обозначение для вызова расширяющих методов, определенных в типе `System.Linq.Enumerable`.
- Многие методы класса `Enumerable` требуют передачи делегатов (в частности, `Func<>`) в качестве параметров.
- Любой метод, ожидающий параметра типа делегата, может принимать вместо него лямбда-выражение.
- Лямбда-выражения являются всего лишь замаскированными анонимными методами (и значительно улучшают читаемость).
- Анонимные методы представляют собой сокращенные обозначения для размещения экземпляра низкоуровневого делегата и ручного построения целевого метода делегата.

Хотя здесь мы погрузились в детали чуть глубже, чем возможно хотелось, приведенное обсуждение должно было способствовать пониманию того, что фактически делают "за кулисами" дружественные к пользователю операции запросов C#.

Исходный код. Проект `LinqUsingEnumerable` доступен в подкаталоге `Chapter_12`.

Резюме

LINQ — это набор взаимосвязанных технологий, которые были разработаны для предоставления единого и симметричного стиля взаимодействия с данными несходных форм. Как объяснялось в главе, **LINQ** может взаимодействовать с любым типом, реализующим интерфейс `IEnumerable<T>`, в том числе с простыми массивами, а также с обобщенными и необобщенными коллекциями данных.

Было показано, что работа с технологиями **LINQ** обеспечивается несколькими средствами языка C#. Например, учитывая тот факт, что выражения запросов **LINQ** могут возвращать любое количество результирующих наборов, для представления лежащего в основе типа данных принято использовать ключевое слово `var`. Кроме того, для построения функциональных и компактных запросов **LINQ** могут применяться лямбда-выражения, синтаксис инициализации объектов и анонимные типы.

Более важно то, что операции запросов C# **LINQ** в действительности являются просто сокращенными обозначениями для обращения к статическим членам типа `System.Linq.Enumerable`. Вы узнали, что большинство членов класса `Enumerable` оперируют с типами делегатов `Func<T>` и для выполнения запроса могут принимать на входе адреса существующих методов, анонимные методы или лямбда-выражения.

ГЛАВА 13

Время существования объектов

К настоящему моменту вы уже умеете создавать специальные типы классов в C#. Теперь вы узнаете, каким образом среда CLR управляет размещенными экземплярами классов (т.е. объектами) посредством *сборки мусора*. Программистам на C# никогда не приходится непосредственно удалять управляемый объект из памяти (вспомните, что в языке C# даже нет ключевого слова вроде `delete`). Взамен объекты .NET размещаются в области памяти, которая называется *управляемой кучей*, где они автоматически уничтожаются сборщиком мусора “когда-нибудь в будущем”.

После изложения основных деталей, касающихся процесса сборки мусора, будет показано, каким образом программно взаимодействовать со сборщиком мусора, используя класс `System.GC` (что в большинстве проектов .NET обычно не требуется). Мы рассмотрим, как с применением виртуального метода `System.Object.Finalize()` и интерфейса `IDisposable` строить классы, которые своевременно и предсказуемо освобождают внутренние *неуправляемые ресурсы*.

Кроме того, будут описаны некоторые функциональные возможности сборщика мусора, появившиеся в версии .NET 4.0, включая фоновую сборку мусора и ленивое (отложенное) создание объектов с использованием обобщенного класса `System.Lazy<>`. После освоения материалов данной главы вы должны хорошо понимать, каким образом среда CLR управляет объектами .NET.

Классы, объекты и ссылки

Прежде чем приступить к исследованию основных тем главы, важно дополнительно прояснить отличие между классами, объектами и ссылочными переменными. Вспомните, что класс — всего лишь модель, которая описывает то, как экземпляр такого типа будет выглядеть и вести себя в памяти. Разумеется, классы определяются внутри файлов кода (которым по соглашению назначается расширение `*.cs`). Взгляните на следующий простой класс `Car`, определенный в новом проекте консольного приложения C# по имени `SimpleGC`:

```
// Car.cs
public class Car
{
    public int CurrentSpeed {get; set;}
    public string PetName {get; set;}

    public Car() {}
}
```

```

public Car(string name, int speed)
{
    PetName = name;
    CurrentSpeed = speed;
}
public override string ToString() => $"{PetName} is going {CurrentSpeed} MPH";
}

```

После того как класс определен, в памяти можно размещать любое количество его объектов, применяя ключевое слово `new` языка C#. Однако следует иметь в виду, что ключевое слово `new` возвращает ссылку на объект в куче, а не действительный объект. Если ссылочная переменная объявляется как локальная переменная в области действия метода, то она сохраняется в стеке для дальнейшего использования внутри приложения. Для доступа к членам объекта в отношении сохраненной ссылки необходимо применять операцию точки C#:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** GC Basics *****");
        // Создать новый объект Car в управляемой куче.
        // Возвращается ссылка на этот объект (refToMyCar).
        Car refToMyCar = new Car("Zippy", 50);
        // Операция точки (.) используется для обращения к членам
        // объекта с применением ссылочной переменной.
        Console.WriteLine(refToMyCar.ToString());
        Console.ReadLine();
    }
}

```

На рис. 13.1 показаны отношения между классами, объектами и ссылками.

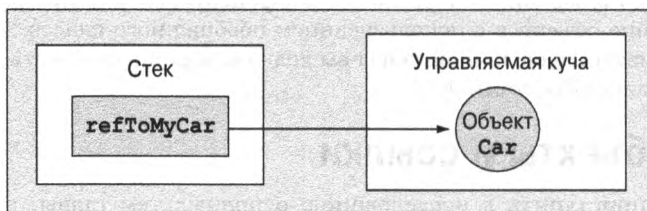


Рис. 13.1. Ссылки на объекты в управляемой куче

На заметку! Вспомните из главы 4, что структуры являются *типами значений*, которые всегда размещаются прямо в стеке и никогда не попадают в управляемую кучу .NET. Размещение в куче происходит только при создании экземпляров классов.

Базовые сведения о времени жизни объектов

При создании приложений C# корректно допускать, что исполняющая среда .NET (также известная как CLR) позаботится об управляемой куче без вашего прямого вмешательства. В действительности "золотое правило" по управлению памятью в .NET выглядит просто.

Правило. Используя ключевое слово `new`, разместите экземпляр класса в управляемой куче и забудьте о нем.

После создания объект будет автоматически удален сборщиком мусора, когда в нем отпадет необходимость. Конечно, возникает вполне закономерный вопрос о том, каким образом сборщик мусора выясняет, что объект больше не нужен? Краткий (т.е. неполный) ответ можно сформулировать так: сборщик мусора удаляет объект из кучи, только когда он становится *недостижимым* для любой части кодовой базы. Добавим в класс `Program` метод, который размещает в памяти локальный объект `Car`:

```
static void MakeACar()
{
    // Если myCar - единственная ссылка на объект Car, то после
    // завершения этого метода объект Car *может* быть уничтожен.
    Car myCar = new Car();
}
```

Обратите внимание, что ссылка на объект `Car` (`myCar`) была создана непосредственно внутри метода `MakeACar()` и не передавалась за пределы определяющей области действия (через возвращаемое значение или параметр `ref/out`). Таким образом, после завершения данного метода ссылка `myCar` оказывается *недостижимой*, и объект `Car` теперь является кандидатом на удаление сборщиком мусора. Тем не менее, важно помнить, что восстановление занимаемой этим объектом памяти немедленно после завершения метода `MakeACar()` гарантировать нельзя. В данный момент можно предполагать лишь то, что когда среда CLR выполнит следующую сборку мусора, объект `myCar` может быть безопасно уничтожен.

Как вы наверняка обнаружите, программирование в среде со сборкой мусора значительно облегчает разработку приложений. И напротив, программистам на языке C++ хорошо известно, что если они не позаботятся о ручном удалении размещенных в куче объектов, тогда утечки памяти не заставят себя долго ждать. На самом деле отслеживание утечек памяти — один из требующих самых больших затрат времени (и утомительных) аспектов программирования в неуправляемых средах. За счет того, что сборщику мусора разрешено взять на себя заботу об уничтожении объектов, обязанности по управлению памятью перекладываются с программистов на среду CLR.

Код CIL для ключевого слова `new`

Когда компилятор C# сталкивается с ключевым словом `new`, он вставляет в реализацию метода инструкцию `newobj` языка CIL. Если вы скомпилируете текущий пример кода и заглянете в полученную сборку с помощью утилиты `ildasm.exe`, то найдете внутри метода `MakeACar()` следующие операторы CIL:

```
.method private hidebysig static void MakeACar() cil managed
{
    // Code size 8 (0x8)
    .maxstack 1
    .locals init ([0] class SimpleGC.Car myCar)
    IL_0000: nop
    IL_0001: newobj instance void SimpleGC.Car::.ctor()
    IL_0006: stloc.0
    IL_0007: ret
} // end of method Program::MakeACar
```

Прежде чем ознакомиться с точными правилами, которые определяют момент, когда объект должен удаляться из управляемой кучи, давайте более подробно рассмотрим

роль CIL-инструкции `newobj`. Первым делом важно понимать, что управляемая куча представляет собой нечто большее, чем просто произвольную область памяти, к которой CLR имеет доступ. Сборщик мусора .NET "убирает" кучу довольно тщательно, при необходимости даже сжимая пустые блоки памяти в целях оптимизации.

Для содействия его усилиям в управляемой куче поддерживается указатель (обычно называемый *указателем на следующий объект* или *указателем на новый объект*), который идентифицирует точное местоположение, куда будет помещен следующий объект. Таким образом, инструкция `newobj` заставляет среду CLR выполнить перечисленные ниже основные операции.

1. Подсчитать общий объем памяти, требуемой для размещения объекта (включая память, необходимую для членов данных и базовых классов).
2. Выяснить, действительно ли в управляемой куче имеется достаточно пространства для сохранения размещаемого объекта. Если места хватает, то указанный конструктор вызывается, и вызывающий код в конечном итоге получает ссылку на новый объект в памяти, адрес которого совпадает с последней позицией указателя на следующий объект.
3. Наконец, перед возвращением ссылки вызывающему коду переместить указатель на следующий объект, чтобы он указывал на следующую доступную область в управляемой куче.

Описанный процесс проиллюстрирован на рис. 13.2.

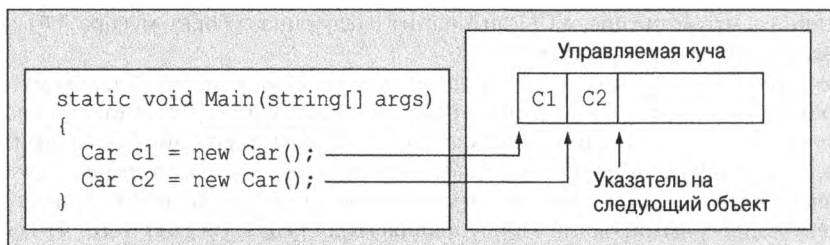


Рис. 13.2. Детали размещения объектов в управляемой куче

По мере интенсивного размещения объектов приложением пространство в управляемой куче может со временем заполниться. Если при обработке инструкции `newobj` среда CLR определяет, что в управляемой куче недостаточно места для размещения объекта запрашиваемого типа, тогда она выполнит сборку мусора, пытаясь освободить память. Соответственно, следующее правило сборки мусора также выглядит довольно простым.

Правило. Если в управляемой куче не хватает пространства для размещения требуемого объекта, то произойдет сборка мусора.

Однако то, как конкретно происходит сборка мусора, зависит от версии платформы .NET, под управлением которой выполняется приложение. Различия будут описаны далее в главе.

Установка объектных ссылок в `null`

Программисты на C/C++ часто устанавливают переменные указателей в `null`, гарантируя тем самым, что они больше не ссылаются на какие-то местоположения в неуправляемой памяти. Учитывая такой факт, вас может интересовать, что происходит

в результате установки в `null` ссылок на объекты в C#. Для примера изменим метод `MakeACar()` следующим образом:

```
static void MakeACar()
{
    Car myCar = new Car();
    myCar = null;
}
```

Когда ссылке на объект присваивается `null`, компилятор C# генерирует код CIL, который гарантирует, что ссылка (`myCar` в данном примере) больше не указывает на какой-либо объект. Если теперь снова с помощью утилиты `ildasm.exe` просмотреть код CIL модифицированного метода `MakeACar()`, то можно обнаружить в нем код операции `ldnull` (заталкивает значение `null` в виртуальный стек выполнения), за которым следует код операции `stloc.0` (устанавливает для переменной ссылку `null`):

```
.method private hidebysig static void MakeACar() cil managed
{
    // Code size 10 (0xa)
    .maxstack 1
    .locals init ([0] class SimpleGC.Car myCar)
    IL_0000: nop
    IL_0001: newobj instance void SimpleGC.Car::.ctor()
    IL_0006: stloc.0
    IL_0007: ldnull
    IL_0008: stloc.0
    IL_0009: ret
} // end of method Program::MakeACar
```

Тем не менее, вы должны понимать, что присваивание ссылке значения `null` ни в коей мере не вынуждает сборщик мусора немедленно запуститься и удалить объект из кучи. Единственное, что при этом достигается — явный разрыв связи между ссылкой и объектом, на который она ранее указывала. Таким образом, установка ссылок в `null` в C# имеет гораздо меньше последствий, чем в других языках, основанных на C; однако никакого вреда она определенно не причиняет.

Роль корневых элементов приложения

Теперь вернемся к вопросу о том, как сборщик мусора определяет момент, когда объект больше не нужен. Чтобы вникнуть во все детали, вы должны ознакомиться с понятием *корневых элементов приложения*. Выражаясь просто, *корневой элемент* — это местоположение в памяти, содержащее ссылку на объект в управляемой куче. Строго говоря, корневой элемент может относиться к любой из следующих категорий:

- ссылки на глобальные объекты (хотя в C# они не допускаются, код CIL разрешает размещать глобальные объекты);
- ссылки на любые статические объекты и статические поля;
- ссылки на локальные объекты внутри кодовой базы приложения;
- ссылки на объектные параметры, передаваемые методу;
- ссылки на объекты, ожидающие финализации (обсуждаются далее в главе);
- любой регистр центрального процессора, который ссылается на объект.

Во время процесса сборки мусора исполняющая среда будет исследовать объекты в управляемой куче с целью выяснения, являются ли они по-прежнему достижимыми (т.е. корневыми) для приложения. Для такой цели CLR будет строить *граф объектов*, кото-

рый представляет каждый достижимый объект в куче. Более подробно графы объектов объясняются во время рассмотрения сериализации объектов в главе 20. Пока достаточно знать, что графы объектов применяются для документирования всех достижимых объектов. Кроме того, имейте в виду, что сборщик мусора никогда не будет создавать граф для того же самого объекта дважды, избегая необходимости в выполнении дополнительного подсчета циклических ссылок, который характерен при программировании COM.

Предположим, что в управляемой куче находится набор объектов с именами A, B, C, D, E, F и G. Во время сборки мусора эти объекты (а также любые внутренние объектные ссылки, которые они могут содержать) будут проверяться на предмет активных корневых элементов. После построения графа недостижимые объекты (пусть ими будут объекты C и F) помечаются как являющиеся мусором. На рис. 13.3 показан возможный граф объектов для только что описанного сценария (линии со стрелками можно читать как “зависит от” или “требуется”, т.е. E зависит от G и B, A не зависит ни от чего и т.д.).

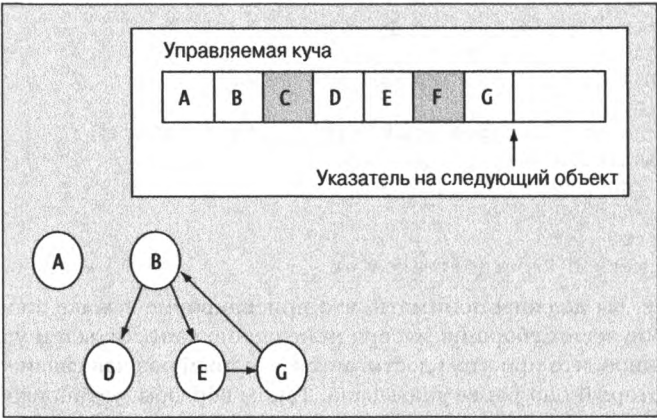


Рис. 13.3. Графы объектов строятся с целью определения объектов, достижимых для корневых элементов приложения

После того как объекты помечены для уничтожения (в данном случае C и F, т.к. они не учтены в графе объектов), они удаляются из памяти. Оставшееся пространство в куче сжимается, что в свою очередь вынуждает среду CLR модифицировать набор активных корневых элементов приложения (и лежащих в основе указателей), чтобы они ссылались на корректные местоположения в памяти (это делается автоматически и прозрачно). И последнее, но не менее важное действие — указатель на следующий объект перенастраивается так, чтобы указывать на следующую доступную область памяти. Конечный результат описанных изменений представлен на рис. 13.4.

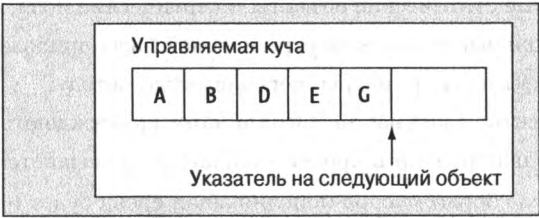


Рис. 13.4. Очищенная и сжатая куча

На заметку! Строго говоря, сборщик мусора использует две отдельные кучи, одна из которых предназначена специально для хранения крупных объектов. Во время сборки мусора обращение к данной куче производится менее часто из-за возможного снижения производительности, связанного с перемещением больших объектов. Невзирая на упомянутый факт, управляемую кучу вполне безопасно считать единственной областью памяти.

Понятие поколений объектов

Когда среда CLR пытается найти недостижимые объекты, она не проверяет буквально каждый объект, помещенный в управляемую кучу. Очевидно, это потребовало бы значительного времени, особенно в крупных (т.е. реальных) приложениях.

Для содействия оптимизации процесса каждому объекту в куче назначается специфичное "поколение". Лежащая в основе поколений идея проста: чем дольше объект существует в куче, тем выше вероятность того, что он там будет оставаться. Например, класс, который определяет главное окно настольного приложения, будет находиться в памяти вплоть до завершения приложения. С другой стороны объекты, которые были помещены в кучу только недавно (такие как объект, размещенный внутри области действия метода), по всей видимости, довольно быстро станут недостижимыми. Исходя из таких предположений, каждый объект в куче относится к одному из перечисленных ниже поколений.

- Поколение 0. Идентифицирует новый размещенный в памяти объект, который еще никогда не помечался как подлежащий сборке мусора.
- Поколение 1. Идентифицирует объект, который уже пережил одну сборку мусора (т.е. он был помечен как подлежащий сборке мусора, но не удался из-за наличия достаточного пространства в куче).
- Поколение 2. Идентифицирует объект, которому удалось пережить более одной очистки сборщиком мусора.

На заметку! Поколения 0 и 1 называются *эфемерными* (недолговечными). В следующем разделе будет показано, что процесс сборки мусора трактует эфемерные поколения по-разному.

Сначала сборщик мусора исследует все объекты, относящиеся к поколению 0. Если пометка и удаление (или освобождение) таких объектов в результате обеспечивают требуемый объем свободной памяти, то любые уцелевшие объекты повышаются до поколения 1. Чтобы увидеть, каким образом поколение объекта влияет на процесс сборки мусора, взгляните на рис. 13.5, где схематически показано, как набор уцелевших объектов поколения 0 (A, B и E) повышается до следующего поколения после восстановления требуемого объема памяти.

Если все объекты поколения 0 проверены, но по-прежнему требуется дополнительная память, тогда начинают исследоваться на предмет достижимости и подвергаться сборке мусора объекты поколения 1. Уцелевшие объекты поколения 1 повышаются до поколения 2. Если же сборщику мусора *все еще* требуется дополнительная память, то начинают проверяться объекты поколения 2. На этом этапе объекты поколения 2, которым удастся пережить сборку мусора, остаются объектами того же поколения 2, учитывая заранее определенный верхний предел поколений объектов.

В заключение следует отметить, что за счет назначения объектам в куче определенного поколения более новые объекты (такие как локальные переменные) будут удаляться быстрее, в то время как более старые (наподобие главного окна приложения) будут существовать дольше.



Рис. 13.5. Объекты поколения 0, которые уцелели после сборки мусора, повышаются до поколения 1

Параллельная сборка мусора до версии .NET 4.0

До выхода версии .NET 4.0 исполняющая среда производила очистку неиспользуемых объектов с применением приема, который назывался *параллельной сборкой мусора*. Согласно такой модели при проведении сборки мусора для любых объектов поколения 0 или поколения 1 (как вы помните, они являются *эфемерными поколениями*) сборщик мусора временно приостанавливал все активные *потоки* внутри текущего процесса, гарантируя невозможность доступа к управляемой куче со стороны приложения во время сборки.

Тема потоков раскрывается в главе 19, а пока рассматривайте поток просто как путь выполнения внутри функционирующей программы. После завершения цикла сборки мусора приостановленным потокам разрешалось продолжить свою работу. К счастью, сборщик мусора в .NET 3.5 (и предшествующих версиях) был хорошо оптимизирован и потому такие короткие перерывы в работе приложения были заметны редко (если вообще когда-либо).

В качестве оптимизации параллельная сборка мусора позволяла производить очистку объектов, которые не были обнаружены в одном из эфемерных поколений, в отдельном потоке, что сокращало (но не устраняло) необходимость в приостановке активных потоков исполняющей средой .NET. Более того, параллельная сборка мусора разрешала программам продолжать размещение объектов в куче во время проведения сборки мусора для объектов неэфемерных поколений.

Фоновая сборка мусора в .NET 4.0 и последующих версиях

Начиная с версии .NET 4.0, сборщик мусора способен решать вопрос с приостановкой потоков при очистке объектов в управляемой куче, используя *фоновую сборку мусора*. Несмотря на название приема, это вовсе не означает, что вся сборка мусора теперь происходит в дополнительных фоновых потоках выполнения. На самом деле, если фоновая сборка мусора производится для объектов, принадлежащих к неэфемерному поколению, то исполняющая среда .NET может выполнять сборку мусора в отношении объектов эфемерных поколений внутри отдельного фонового потока.

В качестве связанного замечания: механизм сборки мусора в .NET 4.0 и последующих версиях был усовершенствован с целью дальнейшего сокращения времени приостановки заданного потока, которая связана со сборкой мусора. Конечным результатом

таких изменений стало то, что процесс очистки неиспользуемых объектов поколения 0 или поколения 1 был оптимизирован и позволяет обеспечить более высокую производительность приложений (что действительно важно для систем реального времени, которые требуют небольших и предсказуемых перерывов на сборку мусора).

Тем не менее, важно понимать, что ввод новой модели сборки мусора совершенно не повлиял на способ построения приложений .NET. С практической точки зрения вы можете просто разрешить сборщику мусора .NET выполнять свою работу без непосредственного вмешательства с вашей стороны (и радоваться тому, что разработчики в Microsoft продолжают улучшать процесс сборки мусора в прозрачной манере).

Тип System.GC

Сборка `mscorlib.dll` предлагает класс по имени `System.GC`, который позволяет программно взаимодействовать со сборщиком мусора, применяя набор статических членов. Имейте в виду, что необходимость в прямом взаимодействии с классом `System.GC` внутри разрабатываемого кода возникает редко (если вообще возникает). Обычно единственной ситуацией, когда будут использоваться члены `System.GC`, является создание классов, которые внутренне работают с *неуправляемыми ресурсами*. Например, может строиться класс, в котором присутствуют вызовы API-интерфейса Windows, основанного на C, с применением протокола обращения к платформе .NET, или какая-то низкоуровневая и сложная логика взаимодействия с COM. В табл. 13.1 приведено краткое описание некоторых наиболее интересных членов класса `System.GC` (полные сведения можно найти в документации .NET Framework SDK).

Таблица 13.1. Избранные члены типа System.GC

Члены System.GC	Описание
<code>AddMemoryPressure()</code> , <code>RemoveMemoryPressure()</code>	Позволяют указывать числовое значение, которое представляет "уровень срочности" (или давление) вызывающего объекта относительно процесса сборки мусора. Имейте в виду, что эти методы должны изменять уровень давления <i>в тандеме</i> ; следовательно, нельзя удалять более высокий показатель давления, чем тот, который был добавлен
<code>Collect()</code>	Заставляет сборщик мусора выполнить сборку мусора. Этот метод перегружен для указания поколения, подлежащего сборке, а также режима сборки (посредством перечисления <code>GC.CollectMode</code>)
<code>CollectionCount()</code>	Возвращает числовое значение, которое показывает, сколько раз производилась сборка мусора для заданного поколения
<code>GetGeneration()</code>	Возвращает поколение, к которому относится объект в текущий момент
<code>GetTotalMemory()</code>	Возвращает оценочный объем памяти (в байтах), выделенной в управляемой куче в текущий момент. Булевский параметр указывает, должен ли вызов дожидаться выполнения сборки мусора перед возвращением
<code>MaxGeneration</code>	Возвращает максимальное количество поколений, поддерживаемое целевой системой. Начиная с версии .NET 4.0 есть три возможных поколения: 0, 1 и 2
<code>SuppressFinalize()</code>	Устанавливает флаг, который указывает, что заданный объект не должен вызывать свой метод <code>Finalize()</code>
<code>WaitForPendingFinalizers()</code>	Приостанавливает текущий поток до тех пор, пока не будут финализированы все финализируемые объекты. Обычно вызывается сразу после вызова метода <code>GC.Collect()</code>

Чтобы проиллюстрировать использование `System.GC` для получения разнообразных деталей, связанных со сборкой мусора, создадим следующий метод `Main()`, в котором применяются некоторые члены `System.GC`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with System.GC *****");
    // Вывести оценочное количество байтов, выделенных в куче.
    Console.WriteLine("Estimated bytes on heap: {0}",
        GC.GetTotalMemory(false));
    // Значения MaxGeneration начинаются с 0, поэтому при выводе добавить 1.
    Console.WriteLine("This OS has {0} object generations.\n",
        (GC.MaxGeneration + 1));
    Car refToMyCar = new Car("Zippy", 100);
    Console.WriteLine(refToMyCar.ToString());
    // Вывести поколение объекта refToMyCar.
    Console.WriteLine("Generation of refToMyCar is: {0}",
        GC.GetGeneration(refToMyCar));
    Console.ReadLine();
}
```

Принудительный запуск сборщика мусора

Повторим еще раз: основное предназначение сборщика мусора .NET заключается в управлении памятью вместо программистов. Однако в некоторых редких обстоятельствах может оказаться полезным принудительный запуск в коде сборщика мусора, используя метод `GC.Collect()`. Взаимодействие с процессом сборки мусора требуется в двух ситуациях.

- Приложение собирается войти в блок кода, который не должен прерываться возможной сборкой мусора.
- Приложение только что закончило размещение исключительно большого количества объектов и нужно насколько возможно скоро освободить большой объем выделенной памяти.

Если выясняется, что принудительная проверка сборщиком мусора наличия недостижимых объектов может принести пользу, тогда следует явно инициировать процесс сборки мусора:

```
static void Main(string[] args)
{
    ...
    // Принудительно запустить сборку мусора
    // и ожидать финализации каждого объекта.
    GC.Collect();
    GC.WaitForPendingFinalizers();
    ...
}
```

Когда сборка мусора запускается вручную, всегда должен вызываться метод `GC.WaitForPendingFinalizers()`. Благодаря такому подходу можно иметь уверенность в том, что все *финализируемые объекты* (описанные в следующем разделе) получат шанс выполнить любую необходимую очистку перед продолжением работы программы. “За кулисами” метод `GC.WaitForPendingFinalizers()` приостановит вызывающий поток на время прохождения сборки мусора. Это очень хорошо, т.к. гарантирует невозможность обращения в коде к методам объекта, который в текущий момент уничтожается.

Методу `GC.Collect()` можно также предоставить числовое значение, идентифицирующее самое старое поколение, для которого будет выполняться сборка мусора. Например, чтобы проинструктировать среду CLR о необходимости исследования только объектов поколения 0, можно написать следующий код:

```
static void Main(string[] args)
{
    ...
    // Исследовать только объекты поколения 0.
    GC.Collect(0);
    GC.WaitForPendingFinalizers();
    ...
}
```

Кроме того, методу `Collect()` можно передать во втором параметре значение перечисления `GCCollectionMode` для точной настройки способа, которым исполняющая среда должна принудительно инициализировать сборку мусора. Ниже показаны значения, определенные этим перечислением:

```
public enum GCCollectionMode
{
    Default,    // Текущим стандартным значением является Forced.
    Forced,     // Указывает исполняющей среде начать сборку мусора немедленно.
    Optimized   // Позволяет исполняющей среде выяснить, оптимален
                // ли текущий момент для удаления объектов.
}
```

Как и при любой сборке мусора, в результате вызова `GC.Collect()` уцелевшие объекты переводятся в более высокие поколения. В целях иллюстрации предположим, что метод `Main()` модифицирован следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with System.GC *****");
    // Вывести оценочное количество байтов, выделенных в куче.
    Console.WriteLine("Estimated bytes on heap: {0}",
        GC.GetTotalMemory(false));
    // Значения MaxGeneration начинаются с 0.
    Console.WriteLine("This OS has {0} object generations.\n",
        (GC.MaxGeneration + 1));
    Car refToMyCar = new Car("Zippy", 100);
    Console.WriteLine(refToMyCar.ToString());
    // Вывести поколение refToMyCar.
    Console.WriteLine("\nGeneration of refToMyCar is: {0}",
        GC.GetGeneration(refToMyCar));
    // Создать большое количество объектов в целях тестирования.
    object[] tonsOfObjects = new object[50000];
    for (int i = 0; i < 50000; i++)
        tonsOfObjects[i] = new object();
    // Выполнить сборку мусора только для объектов поколения 0.
    GC.Collect(0, GCCollectionMode.Forced);
    GC.WaitForPendingFinalizers();
    // Вывести поколение refToMyCar.
    Console.WriteLine("Generation of refToMyCar is: {0}",
        GC.GetGeneration(refToMyCar));
}
```



```
// Посмотреть, существует ли еще tonsOfObjects[9000].
if (tonsOfObjects[9000] != null)
{
    Console.WriteLine("Generation of tonsOfObjects[9000] is: {0}",
        GC.GetGeneration(tonsOfObjects[9000]));
}
else
    Console.WriteLine("tonsOfObjects[9000] is no longer alive.");
// Вывести количество проведенных сборок мусора для разных поколений.
Console.WriteLine("\nGen 0 has been swept {0} times",
    GC.CollectionCount(0));
Console.WriteLine("Gen 1 has been swept {0} times",
    GC.CollectionCount(1));
Console.WriteLine("Gen 2 has been swept {0} times",
    GC.CollectionCount(2));
Console.ReadLine();
}
```

Здесь в целях тестирования преднамеренно был создан большой массив типа `object` (состоящий из 50 000 элементов). В показанном ниже выводе видно, что хотя в методе `Main()` был сделан только один явный запрос на проведение сборки мусора (посредством метода `GC.Collect()`), среда CLR в фоновом режиме выполнила несколько сборок:

```
***** Fun with System.GC *****
Estimated bytes on heap: 70240
This OS has 3 object generations.

Zippy is going 100 MPH

Generation of refToMyCar is: 0
Generation of refToMyCar is: 1
Generation of tonsOfObjects[9000] is: 1

Gen 0 has been swept 1 times
Gen 1 has been swept 0 times
Gen 2 has been swept 0 times
```

К этому моменту вы должны лучше понимать детали жизненного цикла объектов. В следующем разделе мы продолжим изучение процесса сборки мусора, обратившись к теме создания *финализируемых объектов* и *освобождаемых объектов*. Имейте в виду, что описываемые далее приемы обычно необходимы только при построении классов C#, которые поддерживают внутренние неуправляемые ресурсы.

Исходный код. Проект `SimpleGC` доступен в подкаталоге `Chapter_13`.

Построение финализируемых объектов

В главе 6 вы узнали, что в самом главном базовом классе .NET, `System.Object`, определен виртуальный метод по имени `Finalize()`. В своей стандартной реализации он ничего не делает:

```
// System.Object
public class Object
{
    ...
    protected virtual void Finalize() {}
}
```

За счет переопределения метода `Finalize()` в специальных классах устанавливается специфическое место для выполнения любой логики очистки, необходимой данному типу. Учитывая, что метод `Finalize()` определен как защищенный, вызывать его напрямую из экземпляра класса через операцию точки нельзя. Взамен метод `Finalize()`, если он поддерживается, будет вызываться *сборщиком мусора* перед удалением объекта из памяти.

На заметку! Переопределять метод `Finalize()` в типах структур не допускается. Подобное ограничение вполне логично, поскольку структуры являются типами значений, которые изначально никогда не размещаются в куче и, следовательно, никогда не подвергаются сборке мусора. Тем не менее, при создании структуры, которая содержит неуправляемые ресурсы, нуждающиеся в очистке, можно реализовать интерфейс `IDisposable` (вскоре он будет описан).

Разумеется, вызов метода `Finalize()` будет происходить (в итоге) во время “естественной” сборки мусора или в случае ее принудительного запуска внутри кода с помощью `GC.Collect()`. Вдобавок метод финализатора типа будет автоматически вызываться, когда домен, в котором обслуживается приложение, выгружается из памяти. В зависимости от опыта работы с .NET вам уже может быть известно, что домены приложений применяются для обслуживания исполняемой сборки и любых необходимых внешних библиотек кода. Если вы еще не знакомы с такой концепцией .NET, то необходимые сведения будут даны в главе 17. Пока запомните, что при выгрузке домена приложения из памяти среда CLR автоматически вызывает финализаторы всех финализируемых объектов, созданных за время существования домена приложения.

О чем бы ни говорили ваши инстинкты разработчика, подавляющее большинство классов C# не требует написания явной логики очистки или специального финализатора. Причина проста: если в классах используются лишь другие управляемые объекты, то все они в конечном итоге будут подвергнуты сборке мусора. Единственная ситуация, когда может возникнуть потребность спроектировать класс, способный выполнять после себя очистку, предусматривает работу с *неуправляемыми* ресурсами (такими как низкоуровневые файловые дескрипторы операционной системы, низкоуровневые неуправляемые подключения к базам данных, фрагменты неуправляемой памяти и т.д.). В рамках платформы .NET неуправляемые ресурсы получают путем прямого обращения к API-интерфейсу операционной системы с применением служб вызова платформы (Platform Invocation Services — P/Invoke) или в сложных сценариях взаимодействия с COM. С учетом сказанного можно сформулировать еще одно правило сборки мусора.

Правило. Единственная серьезная причина для переопределения метода `Finalize()` связана с использованием в классе C# неуправляемых ресурсов через P/Invoke или сложные задачи взаимодействия с COM (обычно посредством разнообразных членов типа `System.Runtime.InteropServices.Marshal`). Это объясняется тем, что в таких сценариях производится манипулирование памятью, которой среда CLR управлять не может.

Переопределение метода `System.Object.Finalize()`

В том редком случае, когда строится класс C#, в котором применяются неуправляемые ресурсы, вы вполне очевидно захотите обеспечить предсказуемое освобождение занимаемой памяти. Для примера создадим новый проект консольного приложения C# по имени `SimpleFinalize` и вставим в него класс `MyResourceWrapper`, в котором используется неуправляемый ресурс (каким бы он ни был). Теперь необходимо переопределить метод `Finalize()`. Как ни странно, для этого нельзя применять ключевое слово `override` языка C#:

```
class MyResourceWrapper
{
    // Ошибка на этапе компиляции!
    protected override void Finalize() { }
}
```

На самом деле для достижения того же самого эффекта используется синтаксис деструктора (подобный C++). Причина такой альтернативной формы переопределения виртуального метода заключается в том, что при обработке синтаксиса финализатора компилятор автоматически добавляет внутрь неявно переопределяемого метода `Finalize()` много обязательных элементов инфраструктуры (как вскоре будет показано).

Финализаторы C# выглядят похожими на конструкторы тем, что именуются идентично классу, в котором определены. Вдобавок они снабжаются префиксом в виде тильды (~). Однако в отличие от конструкторов финализаторы никогда не получают модификатор доступа (они всегда неявно защищенные), не принимают параметров и не могут быть перегружены (в каждом классе допускается наличие только одного финализатора).

Ниже приведен специальный финализатор для класса `MyResourceWrapper`, который при вызове выдает звуковой сигнал. Очевидно, такой пример предназначен только для демонстрационных целей. В реальном приложении финализатор только освобождает любые неуправляемые ресурсы и не взаимодействует с другими управляемыми объектами, даже с теми, на которые ссылается текущий объект, т.к. нельзя предполагать, что они все еще существуют на момент вызова этого метода `Finalize()` сборщиком мусора.

```
//Переопределить System.Object.Finalize() посредством синтаксиса финализатора
class MyResourceWrapper
{
    // Очистить неуправляемые ресурсы.
    // Выдать звуковой сигнал при уничтожении
    // (только в целях тестирования).
    ~MyResourceWrapper() => Console.Beep();
}
```

Если теперь просмотреть код CIL данного деструктора с помощью утилиты `ildasm.exe`, то обнаружится, что компилятор добавил необходимый код для проверки ошибок. Первым делом операторы внутри области действия метода `Finalize()` помещены в блок `try` (см. главу 7). Связанный с ним блок `finally` гарантирует, что методы `Finalize()` базовых классов будут всегда выполняться независимо от любых исключений, возникших в области `try`.

```
.method family hidebysig virtual instance void
    Finalize() cil managed
{
    // Code size      13 (0xd)
    .maxstack 1
    .try
    {
        IL_0000: ldc.i4 0x4e20
        IL_0005: ldc.i4 0x3e8
        IL_000a: call
            void [mscorlib]System.Console::Beep(int32, int32)
        IL_000f: nop
        IL_0010: nop
        IL_0011: leave.s IL_001b
    } // end .try
}
```

```

finally
{
    IL_0013: ldarg.0
    IL_0014:
        call instance void [mscorlib]System.Object::Finalize()
    IL_0019: nop
    IL_001a: endfinally
} // end handler
IL_001b: nop
IL_001c: ret
} // end of method MyResourceWrapper::Finalize

```

Тестирование класса `MyResourceWrapper` показывает, что звуковой сигнал выдается во время завершения приложения, потому что среда CLR автоматически вызывает финализаторы при выгрузке домена приложения:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Finalizers *****\n");
    Console.WriteLine("Hit the return key to shut down this app");
    Console.WriteLine("and force the GC to invoke Finalize()");
    Console.WriteLine("for finalizable objects created in this AppDomain.");
    // Нажмите клавишу <Enter>, чтобы завершить приложение
    // и заставить сборщик мусора вызвать метод Finalize()
    // для всех финализируемых объектов, которые
    // были созданы в домене этого приложения.
    Console.ReadLine();
    MyResourceWrapper rw = new MyResourceWrapper();
}

```

Исходный код. Проект `SimpleFinalize` доступен в подкаталоге `Chapter_13`.

Подробности процесса финализации

Чтобы не стараться понапрасну, всегда помните о том, что роль метода `Finalize()` состоит в обеспечении того, что объект .NET сумеет освободить неуправляемые ресурсы, когда он подвергается сборке мусора. Таким образом, если вы строите класс, в котором неуправляемая память не применяется (общепризнанно самый распространенный случай), то финализация малопригодна. На самом деле по возможности вы должны проектировать свои типы так, чтобы избегать в них поддержки метода `Finalize()` по той простой причине, что финализация занимает время.

При размещении объекта в управляемой куче исполняющая среда автоматически определяет, поддерживает ли он специальный метод `Finalize()`. Если да, тогда объект помечается как *финализируемый*, а указатель на него сохраняется во внутренней очереди, называемой *очередью финализации*. Очередь финализации — это таблица, обслуживаемая сборщиком мусора, в которой содержатся указатели на все объекты, подлежащие финализации перед удалением из кучи.

Когда сборщик мусора решает, что наступило время высвободить объект из памяти, он просматривает каждую запись в очереди финализации и копирует объект из кучи в еще одну управляемую структуру под названием *таблица объектов, доступных для финализации*. На этой стадии порождается отдельный поток для вызова метода `Finalize()` на каждом объекте из упомянутой таблицы *при следующей сборке мусора*. Итак, действительная финализация объекта требует, по меньшей мере, двух сборок мусора.

Подводя итоги, следует отметить, что хотя финализация объекта гарантирует ему возможность освобождения неуправляемых ресурсов, она все равно остается недетерминированной по своей природе, а из-за незаметной дополнительной обработки протекает значительно медленнее.

Построение освобождаемых объектов

Как вы уже видели, финализаторы могут использоваться для освобождения неуправляемых ресурсов при запуске сборщика мусора. Тем не менее, учитывая тот факт, что многие неуправляемые объекты являются “ценными элементами” (вроде низкоуровневых дескрипторов для файлов или подключений к базам данных), зачастую полезно их освобождать как можно раньше, не дожидаясь наступления сборки мусора. В качестве альтернативы переопределению метода `Finalize()` класс может реализовать интерфейс `IDisposable`, в котором определен единственный метод по имени `Dispose()`:

```
public interface IDisposable
{
    void Dispose();
}
```

При реализации интерфейса `IDisposable` предполагается, что когда пользователь объекта завершает работу с ним, он вручную вызывает метод `Dispose()` перед тем, как позволить объектной ссылке покинуть область действия. Таким способом объект может производить любую необходимую очистку неуправляемых ресурсов без помещения в очередь финализации и ожидания, пока сборщик мусора запустит логику финализации класса.

На заметку! Интерфейс `IDisposable` может быть реализован и структурами, и классами (в отличие от переопределения метода `Finalize()`, что допускается только для классов), т.к. метод `Dispose()` вызывает пользователь объекта, а не сборщик мусора.

Чтобы проиллюстрировать применение интерфейса `IDisposable`, создадим новый проект консольного приложения C# по имени `SimpleDispose`. Ниже приведен модифицированный класс `MyResourceWrapper`, который вместо переопределения метода `System.Object.Finalize()` теперь реализует интерфейс `IDisposable`:

```
// Реализация интерфейса IDisposable.
class MyResourceWrapper : IDisposable
{
    // После окончания работы с объектом пользователь
    // объекта должен вызывать этот метод.
    public void Dispose()
    {
        // Очистить неуправляемые ресурсы...
        // Освободить другие освобождаемые объекты, содержащиеся внутри.
        // Только для целей тестирования.
        Console.WriteLine("***** In Dispose! *****");
    }
}
```

Обратите внимание, что метод `Dispose()` отвечает не только за освобождение неуправляемых ресурсов самого типа, но может также вызывать методы `Dispose()` для любых других освобождаемых объектов, которые содержатся внутри типа. В отличие от `Finalize()` в методе `Dispose()` вполне безопасно взаимодействовать с другими управляемыми объектами. Причина проста: сборщик мусора не имеет понятия об интерфейсе

се `IDisposable`, а потому никогда не будет вызывать метод `Dispose()`. Следовательно, когда пользователь объекта вызывает данный метод, объект все еще существует в управляемой куче и имеет доступ ко всем остальным находящимся там объектам. Логика вызова метода `Dispose()` прямолинейна:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Dispose *****\n");
        // Создать освобождаемый объект и вызвать метод Dispose()
        // для освобождения любых внутренних ресурсов.
        MyResourceWrapper rw = new MyResourceWrapper();
        rw.Dispose();
        Console.ReadLine();
    }
}
```

Конечно, перед попыткой вызова метода `Dispose()` на объекте понадобится проверить, поддерживает ли тип интерфейс `IDisposable`. Хотя всегда можно выяснить, какие типы в библиотеках базовых классов реализуют `IDisposable`, заглянув в документацию .NET Framework 4.7 SDK, программная проверка производится с помощью ключевого слова `is` или `as` (см. главу 6):

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Dispose *****\n");
        MyResourceWrapper rw = new MyResourceWrapper();
        if (rw is IDisposable)
            rw.Dispose();
        Console.ReadLine();
    }
}
```

Приведенный пример раскрывает очередное правило, касающееся управления памятью.

Правило. Неплохо вызывать метод `Dispose()` на любом создаваемом напрямую объекте, если он поддерживает интерфейс `IDisposable`. Предположение заключается в том, что когда проектировщик типа решил реализовать метод `Dispose()`, тогда тип должен выполнять какую-то очистку. Если вы забудете вызвать `Dispose()`, то память в конечном итоге будет очищена (так что можно не переживать), но это может занять больше времени, чем необходимо.

С предыдущим правилом связано одно предостережение. Несколько типов в библиотеках базовых классов, которые реализуют интерфейс `IDisposable`, предоставляют (кое в чем сбивающий с толку) псевдоним для метода `Dispose()` в попытке сделать имя метода очистки более естественным для определяющего его типа. В качестве примера можно взять класс `System.IO.FileStream`, который реализует интерфейс `IDisposable` (и потому поддерживает метод `Dispose()`), но также определяет следующий метод `Close()`, предназначенный для той же цели:

```
// Предполагается, что было импортировано пространство имен System.IO.
static void DisposeFileStream()
{
```

```

FileStream fs = new FileStream("myFile.txt", FileMode.OpenOrCreate);
// Мягко выражаясь, сбивает с толку!
// Вызовы этих методов делают одно и то же!
fs.Close();
fs.Dispose();
}

```

В то время как "заккрытие" (close) файла выглядит более естественным, чем его "освобождение" (dispose), подобное дублирование методов очистки может запутывать. При работе с типами, предлагающими псевдонимы, просто помните о том, что если тип реализует интерфейс `IDisposable`, то вызов метода `Dispose()` всегда является безопасным способом действия.

Повторное использование ключевого слова `using` в C#

Имея дело с управляемым объектом, который реализует интерфейс `IDisposable`, довольно часто приходится применять структурированную обработку исключений, гарантируя тем самым, что метод `Dispose()` типа будет вызываться даже в случае генерации исключения во время выполнения:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Dispose *****\n");
    MyResourceWrapper rw = new MyResourceWrapper ();
    try
    {
        // Использовать члены rw.
    }
    finally
    {
        // Всегда вызывать Dispose(), возникла ошибка или нет.
        rw.Dispose();
    }
}

```

Хотя это является хорошим примером защитного программирования, в действительности лишь немногих разработчиков привлекает перспектива помещения каждого освобождаемого типа внутрь блока `try/finally`, просто чтобы гарантировать вызов метода `Dispose()`. Того же самого результата можно достичь гораздо менее навязчивым способом, используя специальный фрагмент синтаксиса C#, который выглядит следующим образом:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Dispose *****\n");
    // Метод Dispose() вызывается автоматически
    // при выходе за пределы области действия using.
    using (MyResourceWrapper rw = new MyResourceWrapper ())
    {
        // Использовать объект rw.
    }
}

```

Если вы просмотрите код CIL метода `Main()` посредством `ildasm.exe`, то обнаружите, что синтаксис `using` на самом деле расширяется до логики `try/finally` с вполне ожидаемым вызовом `Dispose()`:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    ...
    .try
    {
        ...
    } // end .try
    finally
    {
        ...
        IL_0012: callvirt instance void
            SimpleFinalize.MyResourceWrapper::Dispose()
        } // end handler
        ...
    } // end of method Program::Main
```

На заметку! Попытка применения `using` к объекту, который не реализует интерфейс `IDisposable`, приводит к ошибке на этапе компиляции.

Несмотря на то что такой синтаксис устраняет необходимость вручную помещать освобождаемые объекты внутрь блоков `try/finally`, к сожалению, теперь ключевое слово `using` в C# имеет двойной смысл (импортирование пространств имен и вызов метода `Dispose()`). Однако при работе с типами .NET, которые поддерживают интерфейс `IDisposable`, такая синтаксическая конструкция будет гарантировать, что используемый объект автоматически вызовет свой метод `Dispose()` по завершении блока `using`.

Кроме того, имейте в виду, что внутри `using` допускается объявлять несколько объектов *одного и того же типа*. Как и можно было ожидать, компилятор вставит код для вызова `Dispose()` на каждом объявленном объекте.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Dispose *****\n");
    // Использовать список с разделителями-запятymi для объявления
    // нескольких объектов, подлежащих освобождению.
    using(MyResourceWrapper rw = new MyResourceWrapper(),
        rw2 = new MyResourceWrapper())
    {
        // Работать с объектами rw и rw2.
    }
}
```

Исходный код. Проект `SimpleDispose` доступен в подкаталоге `Chapter_13`.

Создание финализируемых и освобождаемых типов

К настоящему моменту вы видели два разных подхода к конструированию класса, который очищает внутренние неуправляемые ресурсы. С одной стороны, можно применять финализатор. Использование такого подхода дает уверенность в том, что объект будет очищать себя сам во время сборки мусора (когда бы она ни произошла) без

вмешательства со стороны пользователя. С другой стороны, можно реализовать интерфейс `IDisposable` и предоставить пользователю объекта способ очистки объекта по окончании работы с ним. Тем не менее, если пользователь объекта забудет вызвать метод `Dispose()`, то неуправляемые ресурсы могут оставаться в памяти неопределенно долго.

Нетрудно догадаться, что в одном определении класса можно смешивать оба подхода, извлекая лучшее от обеих моделей. Если пользователь объекта не забыл вызвать метод `Dispose()`, тогда можно проинформировать сборщик мусора о пропуске процесса финализации, вызвав метод `GC.SuppressFinalize()`. Если же пользователь объекта забыл вызвать `Dispose()`, то объект со временем будет финализирован и получит шанс освободить внутренние ресурсы. Преимущество здесь в том, что внутренние неуправляемые ресурсы будут тем или иным способом освобождены.

Ниже представлена очередная версия класса `MyResourceWrapper`, который теперь является финализируемым и освобождаемым; она определена в проекте консольного приложения C# по имени `FinalizableDisposableClass`:

```
// Усовершенствованная оболочка для ресурсов.
public class MyResourceWrapper : IDisposable
{
    // Сборщик мусора будет вызывать этот метод, если
    // пользователь объекта забыл вызвать Dispose().
    ~MyResourceWrapper()
    {
        // Очистить любые внутренние неуправляемые ресурсы.
        // **Не** вызывать Dispose() на управляемых объектах.
    }

    // Пользователь объекта будет вызывать этот метод
    // для как можно более скорой очистки ресурсов.
    public void Dispose()
    {
        // Очистить неуправляемые ресурсы.
        // Вызвать Dispose() для других освобождаемых объектов,
        // содержащихся внутри.
        // Если пользователь вызвал Dispose(), то финализация
        // не нужна, поэтому подавить ее.
        GC.SuppressFinalize(this);
    }
}
```

Метод `Dispose()` был модифицирован для вызова метода `GC.SuppressFinalize()`, который информирует среду CLR о том, что вызывать деструктор при обработке данного объекта сборщиком мусора больше не обязательно, т.к. неуправляемые ресурсы уже освобождены посредством логики `Dispose()`.

Формализованный шаблон освобождения

Текущая реализация класса `MyResourceWrapper` работает довольно хорошо, но осталось еще несколько небольших недостатков. Во-первых, методы `Finalize()` и `Dispose()` должны освобождать те же самые неуправляемые ресурсы. Это может привести к появлению дублированного кода, что существенно усложнит сопровождение. В идеале следовало бы определить закрытый вспомогательный метод и вызывать его внутри указанных методов.

Во-вторых, желательно удостовериться в том, что метод `Finalize()` не пытается освободить любые управляемые объекты, когда такие действия должен делать метод `Dispose()`. В-третьих, имеет смысл также позаботиться о том, чтобы пользователь объекта мог безопасно вызывать метод `Dispose()` много раз без возникновения ошибки. В настоящий момент защита подобного рода в методе `Dispose()` отсутствует.

Для решения таких проектных задач в Microsoft определили формальный шаблон освобождения, который соблюдает баланс между надежностью, возможностью сопровождения и производительностью. Вот окончательная версия класса `MyResourceWrapper`, в которой применяется официальный шаблон:

```
class MyResourceWrapper : IDisposable
{
    // Используется для выяснения, вызывался ли метод Dispose().
    private bool disposed = false;

    public void Dispose()
    {
        // Вызвать вспомогательный метод.
        // Указание true означает, что очистку
        // запустил пользователь объекта.
        Cleanup(true);

        // Подавить финализацию.
        GC.SuppressFinalize(this);
    }

    private void Cleanup(bool disposing)
    {
        // Удостовериться, не выполнялось ли уже освобождение.
        if (!this.disposed)
        {
            // Если disposing равно true, тогда
            // освободить все управляемые ресурсы.
            if (disposing)
            {
                // Освободить управляемые ресурсы.
            }
            // Очистить неуправляемые ресурсы.
        }
        disposed = true;
    }

    ~MyResourceWrapper()
    {
        // Вызвать вспомогательный метод.
        // Указание false означает, что
        // очистку запустил сборщик мусора.
        Cleanup(false);
    }
}
```

Обратите внимание, что в `MyResourceWrapper` теперь определен закрытый вспомогательный метод по имени `Cleanup()`. Передавая ему `true` в качестве аргумента, мы указываем, что очистку инициировал пользователь объекта, поэтому должны быть очищены все управляемые и неуправляемые ресурсы. Однако когда очистка инициируется сборщиком мусора, при вызове метода `Cleanup()` передается значение `false`, чтобы внутренние освобождаемые объекты не освобождались (поскольку нельзя рассчитывать на то, что они все еще присутствуют в памяти). И, наконец, перед выходом

из `CleanUp()` переменная-член `disposed` типа `bool` устанавливается в `true`, что дает возможность вызывать метод `Dispose()` много раз без возникновения ошибки.

На заметку! После того как объект был "освобожден", клиент по-прежнему может обращаться к его членам, т.к. объект пока еще находится в памяти. Следовательно, в надежном классе оболочки для ресурсов каждый член также необходимо снабдить дополнительной логикой, которая бы сообщала: "если объект освобожден, то ничего не делать, а просто вернуть управление".

Чтобы протестировать финальную версию класса `MyResourceWrapper`, добавим внутрь финализатора вызов `Console.Beep()`:

```
~MyResourceWrapper()
{
    Console.Beep();
    // Вызвать вспомогательный метод.
    // Указание false означает, что
    // очистку запустил сборщик мусора.
    CleanUp(false);
}
```

Далее обновим метод `Main()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Dispose() / Destructor Combo Platter *****");
    // Вызвать метод Dispose() вручную. Это не приведет к вызову финализатора.
    MyResourceWrapper rw = new MyResourceWrapper();
    rw.Dispose();

    // Не вызывать метод Dispose(). Это приведет к вызову финализатора
    // и выдаче звукового сигнала.
    MyResourceWrapper rw2 = new MyResourceWrapper();
}
```

Обратите внимание, что мы явно вызываем метод `Dispose()` на объекте `rw`, поэтому вызов деструктора подавляется. Тем не менее, мы "забыли" вызвать метод `Dispose()` на объекте `rw2`, так что по окончании выполнения приложения возникает звуковой сигнал. Если закомментировать вызов `Dispose()` на объекте `rw`, то звуковых сигналов будет два.

Исходный код. Проект `FinalizableDisposableClass` доступен в подкаталоге `Chapter_13`.

На этом исследование особенностей управления объектами со стороны среды CLR через сборку мусора завершено. Хотя дополнительные (довольно экзотические) детали, касающиеся процесса сборки мусора (такие как слабые ссылки и восстановление объектов), здесь не рассматривались, полученных сведений должно быть вполне достаточно, чтобы продолжить изучение самостоятельно. В завершение главы мы взглянем на программное средство под названием *ленивое (отложенное) создание объектов*.

Ленивое создание объектов

При создании классов иногда приходится учитывать, что отдельная переменная-член на самом деле может никогда не понадобиться из-за того, что пользователь объекта не будет обращаться к методу (или свойству), в котором она используется. Действительно,

подобное происходит нередко. Однако проблема может возникнуть, если создание такой переменной-члена сопряжено с выделением большого объема памяти.

Для примера предположим, что строится класс, который инкапсулирует операции цифрового музыкального проигрывателя. В дополнение к ожидаемым методам вроде `Play()`, `Pause()` и `Stop()` вы также хотите обеспечить возможность возвращения коллекции объектов `Song` (посредством класса по имени `AllTracks`), которая представляет все имеющиеся на устройстве цифровые музыкальные файлы.

Давайте создадим новый проект консольного приложения по имени `LazyObjectInstantiation` и определим в нем следующие типы классов:

```
// Представляет одиночную композицию.
class Song
{
    public string Artist { get; set; }
    public string TrackName { get; set; }
    public double TrackLength { get; set; }
}

// Представляет все композиции в проигрывателе.
class AllTracks
{
    // Наш проигрыватель может содержать
    // максимум 10 000 композиций.
    private Song[] allSongs = new Song[10000];

    public AllTracks()
    {
        // Предположим, что здесь производится
        // заполнение массива объектов Song.
        Console.WriteLine("Filling up the songs!");
    }
}

// Объект MediaPlayer имеет объекты AllTracks.
class MediaPlayer
{
    // Предположим, что эти методы делают что-то полезное.
    public void Play() { /* Воспроизведение композиции */ }
    public void Pause() { /* Пауза в воспроизведении */ }
    public void Stop() { /* Останов воспроизведения */ }
    private AllTracks allSongs = new AllTracks();

    public AllTracks GetAllTracks()
    {
        // Возвратить все композиции.
        return allSongs;
    }
}
```

В текущей реализации `MediaPlayer` предполагается, что пользователь объекта пожелает получать список объектов с помощью метода `GetAllTracks()`. Хорошо, а что если пользователю объекта не нужен такой список? В этой реализации память под переменную-член `AllTracks` по-прежнему будет выделяться, приводя тем самым к созданию 10 000 объектов `Song` в памяти:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Lazy Instantiation *****\n");
```

```
// В этом вызывающем коде получение всех композиций не производится,
// но косвенно все равно создаются 10 000 объектов!
MediaPlayer myPlayer = new MediaPlayer();
myPlayer.Play();

Console.ReadLine();
}
```

Безусловно, лучше не создавать 10 000 объектов, с которыми никто не будет работать, потому что в результате нагрузка на сборщик мусора .NET намного увеличится. В то время как можно вручную добавить код, который обеспечит создание объекта `allSongs` только в случае, если он применяется (скажем, используя шаблон фабричного метода), есть более простой путь.

Библиотеки базовых классов предоставляют удобный обобщенный класс по имени `Lazy<>`, который определен в пространстве имен `System` внутри сборки `mscorlib.dll`. Он позволяет определять данные, которые не будут создаваться до тех пор, пока действительно не начнут применяться в коде. Поскольку класс является обобщенным, при первом его использовании вы должны явно указать тип создаваемого элемента, которым может быть любой тип из библиотек базовых классов .NET или специальный тип, построенный вами самостоятельно. Чтобы включить отложенную инициализацию переменной-члена `AllTracks`, просто замените показанный ниже код:

```
// Объект MediaPlayer имеет объект AllTracks.
class MediaPlayer
{
    ...
    private AllTracks allSongs = new AllTracks();
    public AllTracks GetAllTracks()
    {
        // Возвратить все композиции.
        return allSongs;
    }
}
```

следующим кодом:

```
// Объект MediaPlayer имеет объект Lazy<AllTracks>.
class MediaPlayer
{
    ...
    private Lazy<AllTracks> allSongs = new Lazy<AllTracks>();
    public AllTracks GetAllTracks()
    {
        // Возвратить все композиции.
        return allSongs.Value;
    }
}
```

Помимо того факта, что переменная-член `AllTracks` теперь имеет тип `Lazy<>`, важно обратить внимание на изменение также и реализации показанного выше метода `GetAllTracks()`. В частности, для получения актуальных сохраненных данных (в этом случае объекта `AllTracks`, поддерживающего 10 000 объектов `Song`) должно применяться доступное только для чтения свойство `Value` класса `Lazy<>`.

Взгляните, как благодаря такому простому изменению приведенный далее модифицированный метод `Main()` будет косвенно размещать объекты `Song` в памяти, только если метод `GetAllTracks()` действительно вызывается:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Lazy Instantiation *****\n");
    // Память под объект AllTracks здесь не выделяется!
    MediaPlayer myPlayer = new MediaPlayer();
    myPlayer.Play();

    // Размещение объекта AllTracks происходит
    // только в случае вызова метода GetAllTracks().
    MediaPlayer yourPlayer = new MediaPlayer();
    AllTracks yourMusic = yourPlayer.GetAllTracks();

    Console.ReadLine();
}
```

На заметку! Ленивое создание объектов полезно не только для уменьшения количества выделений памяти под ненужные объекты. Этот прием можно также использовать в ситуации, когда для создания члена применяется затратный в плане ресурсов код, такой как вызов удаленного метода, взаимодействие с реляционной базой данных и т.п.

Настройка процесса создания данных Lazy<>

При объявлении переменной Lazy<> действительный внутренний тип данных создается с использованием стандартного конструктора:

```
// При использовании переменной Lazy<> вызывается
// стандартный конструктор класса AllTracks.
private Lazy<AllTracks> allSongs = new Lazy<AllTracks>();
```

В некоторых случаях приведенный код может оказаться приемлемым, но что если класс AllTracks имеет дополнительные конструкторы, и нужно обеспечить вызов подходящего из них? Более того, что если при создании переменной Lazy() должна выполняться какая-то специальная работа (кроме простого создания объекта AllTracks)? К счастью, класс Lazy() позволяет указывать в качестве необязательного параметра обобщенный делегат, который задает метод для вызова во время создания находящегося внутри типа.

Таким обобщенным делегатом является тип System.Func<>, который может указывать на метод, возвращающий тот же самый тип данных, что и создаваемый связанной переменной Lazy<>, и способный принимать вплоть до 16 аргументов (типизированных с применением обобщенных параметров типа). В большинстве случаев никаких параметров для передачи методу, на который указывает Func<>, задавать не придется. Вдобавок, чтобы значительно упростить работу с типом Func<>, рекомендуется использовать лямбда-выражения (отношения между делегатами и лямбда-выражениями были подробно освещены в главе 10).

Ниже показана окончательная версия класса MediaPlayer, в которой добавлен небольшой специальный код, выполняемый при создании внутреннего объекта AllTracks. Не забывайте, что перед завершением метод должен вернуть новый экземпляр типа, помещенного в Lazy<>, причем применять можно любой конструктор по своему выбору (здесь по-прежнему вызывается стандартный конструктор AllTracks).

```
class MediaPlayer
{
    ...
    // Использовать лямбда-выражение для добавления дополнительного
    // кода, который выполняется при создании объекта AllTracks.
```

```
private Lazy<AllTracks> allSongs = new Lazy<AllTracks>(() =>
{
    Console.WriteLine("Creating AllTracks object!");
    return new AllTracks();
});
public AllTracks GetAllTracks()
{
    // Возвратить все композиции.
    return allSongs.Value;
}
```

Итак, вы наверняка смогли оценить полезность класса `Lazy<>`. По существу этот обобщенный класс позволяет гарантировать, что затратные в плане ресурсов объекты будут размещены в памяти, только когда они требуются их пользователю. Если данная тема вас заинтересовала, тогда загляните в раздел документации .NET Framework 4.7 SDK, посвященный классу `System.Lazy<>`, и ознакомьтесь с дополнительными примерами реализации ленивого создания.

Исходный код. Проект `LazyObjectInstantiation` доступен в подкаталоге `Chapter_13`.

Резюме

Целью настоящей главы было прояснение процесса сборки мусора. Вы видели, что сборщик мусора запускается, только если не удастся получить необходимый объем памяти из управляемой кучи (либо когда происходит выгрузка из памяти соответствующего домена приложения). Помните, что разработанный в Microsoft алгоритм сборки мусора хорошо оптимизирован и предусматривает использование поколений объектов, дополнительных потоков для финализации объектов и управляемой кучи для обслуживания крупных объектов.

В главе также было показано, каким образом программно взаимодействовать со сборщиком мусора с применением класса `System.GC`. Как отмечалось, единственным случаем, когда может возникнуть необходимость в подобном взаимодействии, является построение финализируемых или освобождаемых классов, которые имеют дело с неуправляемыми ресурсами.

Вспомните, что финализируемые типы — это классы, которые предоставляют деструктор (переопределяя метод `Finalize()`) для очистки неуправляемых ресурсов во время сборки мусора. С другой стороны, освобождаемые объекты являются классами (или структурами), реализующими интерфейс `IDisposable`, к которому пользователь объекта должен обращаться по завершении работы с ними. Наконец, вы изучили официальный шаблон освобождения, в котором смешаны оба подхода.

В заключение был рассмотрен обобщенный класс по имени `Lazy<>`. Вы узнали, что данный класс позволяет отложить создание затратных (в смысле потребления памяти) объектов до тех пор, пока вызывающая сторона действительно не затребует их. Класс `Lazy<>` помогает сократить количество объектов, хранящихся в управляемой куче, и также обеспечивает создание затратных объектов только тогда, когда они действительно нужны в вызывающем коде.

часть V

Программирование с использованием сборок .NET

В этой части

Глава 14. Построение и конфигурирование библиотек классов

Глава 15. Рефлексия типов, позднее связывание
и программирование на основе атрибутов

Глава 16. Динамические типы и среда DLR

Глава 17. Процессы, домены приложений и объектные контексты

Глава 18. Язык CIL и роль динамических сборок

ГЛАВА 14

Построение и конфигурирование библиотек классов

На протяжении первых четырех частей книги было создано несколько “автономных” исполняемых приложений, в которых вся программная логика упаковывалась в единственный исполняемый файл (*.exe). Такие исполняемые сборки использовали в основном главную библиотеку классов .NET, т.е. mscorlib.dll. В то время как многие простые программы .NET могут быть сконструированы с применением только библиотек базовых классов .NET, многократно используемая программная логика нередко изолируется в специальных библиотеках классов (файлах *.dll), которые могут разделяться между приложениями.

В настоящей главе вы ознакомитесь с различными способами упаковки типов в специальные библиотеки кода. Для начала вы узнаете о разнесении типов по пространствам имен .NET. После этого вы исследуете шаблоны проектов библиотеки классов Visual Studio и разберетесь с разницей между закрытыми и разделяемыми сборками.

Затем вы изучите, как исполняющая среда определяет местонахождение сборки, и освоите глобальный кеш сборок (Global Assembly Cache — GAC), XML-файлы конфигурации приложений (файлы *.config), сборки политик издателя и пространство имен System.Configuration.

Определение специальных пространств имен

Прежде чем погружаться в детали развертывания и конфигурирования библиотек, сначала необходимо узнать, каким образом упаковывать свои специальные типы в пространства имен .NET. Вплоть до этого места в книге создавались небольшие тестовые программы, которые задействовали существующие пространства имен из мира .NET (в частности System). Однако когда строится крупное приложение со многими типами, возможно, будет удобно группировать связанные типы в специальные пространства имен. В C# такая цель достигается с применением ключевого слова namespace. Явное определение специальных пространств имен становится еще более важным при построении .NET-сборок *.dll, т.к. другие разработчики будут нуждаться в ссылке на вашу библиотеку и импортировании специальных пространств имен для использования типов, содержащихся внутри них.

Чтобы исследовать все аспекты непосредственно, начнем с создания нового проекта консольного приложения под названием CustomNamespaces. Предположим, что требует-

ся разработать коллекцию геометрических классов с именами Square (квадрат), Circle (круг) и Hexagon (шестиугольник). Учитывая сходные между ними черты, было бы желательно сгруппировать их вместе в уникальном пространстве имен MyShapes внутри сборки CustomNamespaces.exe. На выбор доступны два базовых подхода. Первый из них предусматривает определение всех классов в единственном файле C# (ShapesLib.cs):

```
// ShapesLib.cs
using System;

namespace MyShapes
{
    // Класс Circle.
    public class Circle { /* Интересные члены... */ }
    // Класс Hexagon.
    public class Hexagon { /* Более интересные члены... */ }
    // Класс Square.
    public class Square { /* Даже еще более интересные члены... */ }
}
```

Хотя компилятор C# без проблем воспримет единственный файл кода C#, содержащий множество типов, могут возникнуть сложности, когда появится желание повторно использовать определения классов в новых проектах. Например, пусть разрабатывается новый проект, в котором необходимо взаимодействовать только с классом Circle. Если все типы определены в единственном файле кода, то вы так или иначе привязаны к целому набору типов. Следовательно, в качестве альтернативы вы можете разнести одно пространство имен по нескольким файлам кода C#. Чтобы обеспечить упаковку типов в ту же самую логическую группу, нужно просто помещать определения заданных классов в область действия одного и того же пространства имен:

```
// Circle.cs
using System;

namespace MyShapes
{
    // Класс Circle.
    public class Circle { /* Интересные методы... */ }
}

// Hexagon.cs
using System;

namespace MyShapes
{
    // Класс Hexagon.
    public class Hexagon { /* Более интересные методы... */ }
}

// Square.cs
using System;

namespace MyShapes
{
    // Класс Square.
    public class Square { /* Даже еще более интересные методы... */ }
}
```

В обоих случаях обратите внимание на то, что пространство MyShapes действует как концептуальный “контейнер” для определяемых в нем классов.

Когда в другом пространстве имен (скажем, CustomNamespaces) необходимо работать с типами из отдельного пространства имен, вы применяете ключевое слово using, как поступали бы в случае использования пространств имен из библиотек базовых классов .NET:

```
// Обратиться к пространству имен из библиотек базовых классов.
using System;

// Использовать типы, определенные в пространстве имен MyShapes.
using MyShapes;

namespace CustomNamespaces
{
    public class Program
    {
        static void Main(string[] args)
        {
            Hexagon h = new Hexagon();
            Circle c = new Circle();
            Square s = new Square();
        }
    }
}
```

В рассматриваемом примере предполагается, что файл или файлы C#, где определено пространство имен MyShapes, являются частью того же самого проекта консольного приложения, что и файл с определением пространства имен CustomNamespaces; другими словами, все эти файлы задействованы для компиляции единственной исполняемой сборки .NET. Если пространство имен MyShapes определено во внешней сборке, то для успешной компиляции потребуется также добавить ссылку на данную библиотеку. На протяжении настоящей главы вы изучите все детали построения приложений, взаимодействующих с внешними библиотеками.

Разрешение конфликтов имен с помощью полностью заданных имен

Говоря формально, вы не обязаны применять ключевое слово `using` языка C# при ссылках на типы, определенные во внешних пространствах имен. Вы можете использовать *полностью заданные имена* типов, которые, как упоминалось в главе 1, представляют собой имена типов, предваренные названиями пространств имен, где типы определены. Например:

```
// Обратите внимание, что пространство имен MyShapes больше не импортируется!
using System;

namespace CustomNamespaces
{
    public class Program
    {
        static void Main(string[] args)
        {
            MyShapes.Hexagon h = new MyShapes.Hexagon();
            MyShapes.Circle c = new MyShapes.Circle();
            MyShapes.Square s = new MyShapes.Square();
        }
    }
}
```

Обычно необходимость в применении полностью заданных имен отсутствует. Они требуют большего объема клавиатурного ввода, но никак не влияют на размер кода и скорость выполнения. На самом деле в коде CIL типы *всегда* определяются с полностью заданными именами. С этой точки зрения ключевое слово `using` языка C# является просто средством экономии времени на наборе.

Тем не менее, полностью заданные имена могут быть полезными (а иногда и необходимыми) для избегания потенциальных конфликтов имен при использовании множеств пространств имен, которые содержат идентично названные типы. Предположим, что есть новое пространство имен `My3DShapes`, где определены три класса, которые способны визуализировать фигуры в трехмерном формате:

```
// Еще одно пространство имен для работы с фигурами.
using System;
namespace My3DShapes
{
    // Класс для представления трехмерного круга.
    public class Circle { }

    // Класс для представления трехмерного шестиугольника.
    public class Hexagon { }

    // Класс для представления трехмерного квадрата.
    public class Square { }
}
```

Если теперь модифицировать класс `Program`, как показано ниже, то компилятор выдаст набор сообщений об ошибках, потому что в обоих пространствах имен определены одинаково именованные классы:

```
// Масса неоднозначностей!
using System;
using MyShapes;
using My3DShapes;

namespace CustomNamespaces
{
    public class Program
    {
        static void Main(string[] args)
        {
            // На какое пространство имен производится ссылка?
            Hexagon h = new Hexagon(); // Ошибка на этапе компиляции!
            Circle c = new Circle();   // Ошибка на этапе компиляции!
            Square s = new Square();    // Ошибка на этапе компиляции!
        }
    }
}
```

Устранить неоднозначности можно за счет применения полностью заданных имен:

```
// Теперь неоднозначности устранены.
static void Main(string[] args)
{
    My3DShapes.Hexagon h = new My3DShapes.Hexagon();
    My3DShapes.Circle c = new My3DShapes.Circle();
    MyShapes.Square s = new MyShapes.Square();
}
```

Разрешение конфликтов имен с помощью псевдонимов

Ключевое слово `using` языка C# также позволяет создавать псевдоним для полностью заданного имени типа. В этом случае определяется метка, которая на этапе компиляции заменяется полностью заданным именем типа. Определение псевдонимов предоставляет второй способ разрешения конфликтов имен. Вот пример:

```

using System;
using MyShapes;
using My3DShapes;
// Устранить неоднозначность, используя специальный псевдоним.
using The3DHexagon = My3DShapes.Hexagon;
namespace CustomNamespaces
{
    class Program
    {
        static void Main(string[] args)
        {
            // На самом деле здесь создается экземпляр класса My3DShapes.Hexagon.
            The3DHexagon h2 = new The3DHexagon();
            ...
        }
    }
}

```

Посредством продемонстрированного альтернативного синтаксиса `using` можно также создавать псевдонимы для пространств имен с очень длинными названиями. Одним из самых длинных пространств имен в библиотеках базовых классов является `System.Runtime.Serialization.Formatters.Binary`, которое содержит член по имени `BinaryFormatter`. При желании экземпляр класса `BinaryFormatter` можно создать следующим образом:

```

using bfHome = System.Runtime.Serialization.Formatters.Binary;
namespace MyApp
{
    class ShapeTester
    {
        static void Main(string[] args)
        {
            bfHome.BinaryFormatter b = new bfHome.BinaryFormatter();
            ...
        }
    }
}

```

либо с использованием традиционной директивы `using`:

```

using System.Runtime.Serialization.Formatters.Binary;
namespace MyApp
{
    class ShapeTester
    {
        static void Main(string[] args)
        {
            BinaryFormatter b = new BinaryFormatter();
            ...
        }
    }
}

```

На данном этапе не нужно беспокоиться о предназначении класса `BinaryFormatter` (он исследуется в главе 20). Сейчас просто запомните, что ключевое слово `using` в C# позволяет создавать псевдонимы для очень длинных полностью заданных имен или, как случается более часто, для разрешения конфликтов имен, которые могут возникать при импорте пространств имен, определяющих типы с идентичными названиями.

На заметку! Имейте в виду, что чрезмерное применение псевдонимов C# в результате может привести к получению запутанной кодовой базы. Если другие программисты в команде не знают о ваших специальных псевдонимах, то они могут полагать, что псевдонимы ссылаются на типы из библиотек базовых классов .NET, и прийти в замешательство, не обнаружив их описания в документации .NET Framework SDK.

Создание вложенных пространств имен

При организации типов допускается определять пространства имен внутри других пространств имен. В библиотеках базовых классов .NET подобное встречается во многих местах и обеспечивает размещение типов на более глубоких уровнях. Например, пространство имен `IO` вложено внутрь пространства имен `System`, давая `System.IO`. Чтобы создать корневое пространство имен, содержащее внутри себя существующее пространство имен `My3DShapes`, необходимо модифицировать код, как показано ниже:

```
// Вложение пространства имен.
namespace Chapter14
{
    namespace My3DShapes
    {
        // Класс для представления трехмерного круга.
        public class Circle{ }

        // Класс для представления трехмерного шестиугольника.
        public class Hexagon{ }

        // Класс для представления трехмерного квадрата.
        public class Square{ }
    }
}
```

Зачастую роль корневого пространства имен заключается просто в предоставлении дополнительного уровня области действия; следовательно, непосредственно в этой области действия определения каких-либо типов могут отсутствовать (как в случае пространства имен `Chapter14`). В такой ситуации вложенное пространство имен может определяться с использованием следующей компактной формы:

```
// Вложение пространства имен (другой способ).
namespace Chapter14.My3DShapes
{
    // Класс для представления трехмерного круга.
    public class Circle{ }

    // Класс для представления трехмерного шестиугольника.
    public class Hexagon{ }

    // Класс для представления трехмерного квадрата.
    public class Square{ }
}
```

Учитывая, что теперь пространство `My3DShapes` вложено внутрь корневого пространства имен `Chapter14`, понадобится обновить все существующие директивы `using` и псевдонимы типов:

```
using Chapter14.My3DShapes;
using The3DHexagon = Chapter14.My3DShapes.Hexagon;
```

Стандартное пространство имен Visual Studio

Относительно пространств имен осталось еще отметить, что по умолчанию при создании нового проекта C# в среде Visual Studio название стандартного пространства имен приложения будет идентичным имени проекта. После этого, когда в проект вставляются новые файлы кода с применением пункта меню Project⇒Add New Item (Проект⇒Добавить новый элемент), типы будут автоматически помещаться внутрь стандартного пространства имен. Если вы хотите изменить название стандартного пространства имен, тогда откройте окно свойств проекта, перейдите в нем на вкладку Application (Приложение) и введите желаемое имя в поле Default namespace (Стандартное пространство имен), как показано на рис. 14.1.

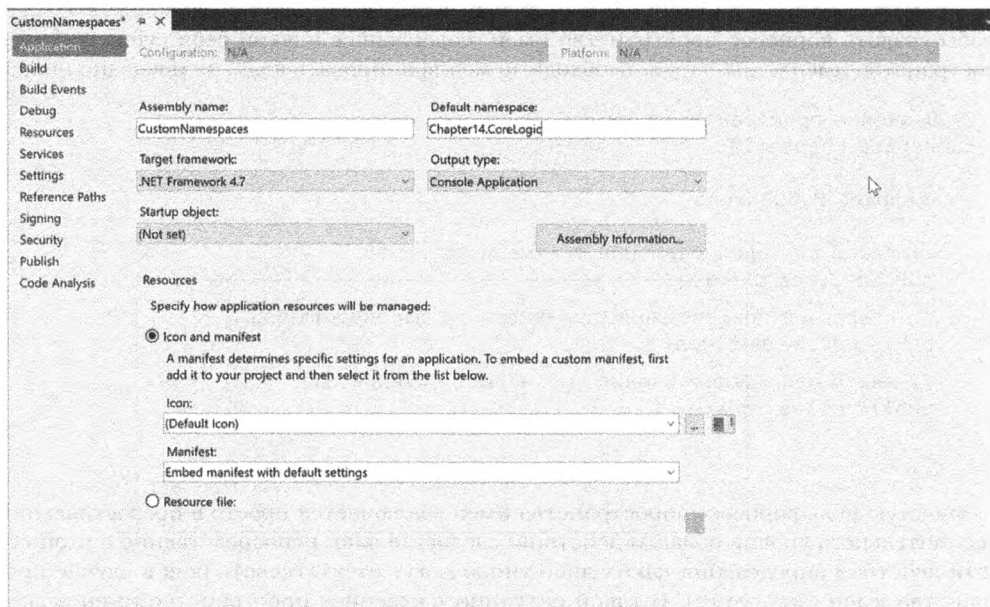


Рис. 14.1. Конфигурирование стандартного пространства имен

После такого изменения любой новый элемент, добавляемый в проект, будет помещаться внутрь пространства имен Chapter14.CoreLogic (и вполне очевидно, что для использования типов из CoreLogic в другом пространстве имен должна применяться корректная директива using).

Теперь, когда вы ознакомились с некоторыми деталями упаковки специальных типов в четко организованные пространства имен, давайте кратко рассмотрим преимущества и формат сборки .NET. Затем мы углубимся в подробности создания, развертывания и конфигурирования специальных библиотек классов.

Исходный код. Проект CustomNamespaces доступен в подкаталоге Chapter_14.

Роль сборок .NET

Приложения .NET конструируются путем соединения в одно целое любого количества *сборок*. Попросту говоря, сборка представляет собой самоописательный двоичный файл, который поддерживает версии и обслуживается средой CLR. Несмотря на то что

сборки .NET имеют такие же файловые расширения (*.exe или *.dll), как и старые двоичные файлы Windows, внутри у них мало общего. Таким образом, для начала давайте выясним, какие преимущества предлагает формат сборки.

Сборки содействуют многократному использованию кода

При построении проектов консольных приложений в предшествующих главах могло показаться, что *вся* функциональность приложений содержалась внутри конструируемых исполняемых сборок. В действительности примеры приложений задействовали многочисленные типы из всегда доступной библиотеки кода .NET по имени `mscorlib.dll` (вспомните, что компилятор C# ссылается на `mscorlib.dll` автоматически), а в ряде случаев — `System.Core.dll`.

Возможно, вы уже знаете, что *библиотека кода* (также называемая *библиотекой классов*) — это файл *.dll, который содержит типы, предназначенные для применения внешними приложениями. При построении исполняемых сборок вы без сомнения будете использовать много системных и специальных библиотек кода по мере создания приложений. Однако имейте в виду, что библиотека кода необязательно должна получать файловое расширение *.dll. Вполне допускается (хотя нечасто), чтобы исполняемая сборка работала с типами, определенными внутри внешнего исполняемого файла. В таком случае ссылаемый файл *.exe также может считаться библиотекой кода.

Независимо от того, как упакована библиотека кода, платформа .NET позволяет многократно применять типы в независимой от языка манере. Например, вы могли бы создать библиотеку кода на C# и повторно использовать ее при написании кода на другом языке программирования .NET. Между языками есть возможность не только выделять память под экземпляры типов, но также и наследовать от них. Базовый класс, определенный в C#, может быть расширен классом, написанным на Visual Basic. Интерфейсы, определенные в F#, могут быть реализованы структурами, определенными в C#, и т.д. Важно понимать, что за счет разбиения единственного монолитного исполняемого файла на несколько сборок .NET достигается возможность многократного использования кода в форме, *нейтральной к языку*.

Сборки устанавливают границы типов

Вспомните, что *полностью заданное имя* типа получается путем предварения имени этого типа (Console) названием пространства имен, где он определен (System). Тем не менее, выражаясь строго, удостоверение типа дополнительно устанавливается сборкой, в которой он находится. Например, если есть две уникально именованные сборки (MyCars.dll и YourCars.dll), которые определяют пространство имен (CarLibrary), содержащее класс по имени SportsCar, то в мире .NET такие типы SportsCar будут рассматриваться как уникальные.

Сборки являются единицами, поддерживающими версии

Сборкам .NET назначается состоящий из четырех частей числовой номер версии в форме <старший номер>.<младший номер>.<номер сборки>.<номер редакции>. (Если номер версии явно не указан, то сборке автоматически назначается версия 1.0.0.0 из-за стандартных настроек проекта в Visual Studio.) Этот номер в сочетании с необязательным значением *открытого ключа* позволяет множеству версий той же самой сборки свободно сосуществовать на одной машине. Формально сборки, которые предоставляют информацию открытого ключа, называются *строго именованными*. Как вы увидите далее в главе, при наличии строгого имени среда CLR способна обеспечивать загрузку корректной версии в интересах вызывающего клиента.

Сборки являются самоописательными

Сборки считаются *самоописательными* частично потому, что содержат информацию обо всех внешних сборках, к которым они должны иметь доступ для корректного функционирования. Таким образом, если сборке требуются библиотеки `System.Windows.Forms.dll` и `System.Core.dll`, то это будет документировано в *манифесте* сборки. Вспомните из главы 1, что манифест представляет собой блок метаданных, которые описывают саму сборку (имя, версия, обязательные внешние сборки и т.д.).

В дополнение к данным манифеста сборка содержит метаданные, которые описывают структуру каждого содержащегося в ней типа (имена членов, реализуемые интерфейсы, базовые классы, конструкторы и т.п.). Благодаря тому, что сборка настолько детально документирована, среда CLR не нуждается в обращении к реестру Windows для выяснения ее местонахождения (что радикально отличается от унаследованной модели программирования COM от Microsoft). Как вы узнаете в текущей главе, для получения информации о местонахождении внешних библиотек кода среда CLR применяет совершенно новую схему.

Сборки являются конфигурируемыми

Сборки могут развертываться как “закрытые” или как “разделяемые”. Закрытые сборки размещаются в том же каталоге (или возможно в подкаталоге), что и клиентское приложение, которое их использует. С другой стороны, разделяемые сборки — это библиотеки, предназначенные для потребления многочисленными приложениями на одиночной машине, которые развертываются в специальном каталоге, называемом *глобальным кешем сборок* (Global Assembly Cache — GAC).

Независимо от того, каким образом развертываются сборки, для них могут быть созданы конфигурационные файлы, основанные на XML. Применяя такие конфигурационные файлы, среду CLR можно заставить “зондировать” сборки в специфичном местоположении, загружать конкретную версию ссылаемой сборки для определенного клиента либо обращаться к произвольному каталогу на локальной машине, сетевому ресурсу или URL-адресу. Вы узнаете больше о конфигурационных XML-файлах далее в главе.

Формат сборки .NET

Теперь, когда вы узнали о многих преимуществах сборок .NET, давайте более детально рассмотрим, как такие сборки устроены внутри. С точки зрения структуры сборки .NET (* .dll или * .exe) состоит из следующих элементов:

- заголовок файла Windows;
- заголовок файла CLR;
- код CIL;
- метаданные типов;
- манифест сборки;
- дополнительные встроенные ресурсы.

Несмотря на то что первые два элемента (заголовки Windows и CLR) представляют собой блоки данных, которые обычно можно игнорировать, краткого рассмотрения они все-таки заслуживают.

Ниже приведен обзор всех перечисленных элементов.

Заголовок файла Windows

Заголовок файла Windows устанавливает факт того, что сборку можно загружать и манипулировать ею в средах операционных систем семейства Windows. Этот заголовок данных также идентифицирует тип приложения (консольное, с графическим пользовательским интерфейсом или библиотека кода *.dll), которое должно обслуживаться Windows. Чтобы просмотреть информацию заголовка Windows сборки, необходимо открыть сборку .NET с помощью утилиты `dumpbin.exe` (в окне командной строки Windows), указав ей флаг `/headers`:

```
dumpbin /headers CarLibrary.dll
```

Ниже показана (неполная) информация заголовка Windows для сборки `CarLibrary.dll`, которая будет построена чуть позже в главе (можете запустить `dumpbin.exe` с именем любого ранее созданного файла *.dll или *.exe вместо `CarLibrary.dll`):

```
Dump of file CarLibrary.dll
PE signature found
File Type: DLL

FILE HEADER VALUES
    14C machine (x86)
      3 number of sections
4B37DCD8 time date stamp Sun Dec 27 16:16:56 2011
      0 file pointer to symbol table
      0 number of symbols
      E0 size of optional header
2102 characteristics
      Executable
      32 bit word machine
      DLL

OPTIONAL HEADER VALUES
    10B magic # (PE32)
    8.00 linker version
    E00 size of code
    600 size of initialized data
      0 size of uninitialized data
    2CDE entry point (00402CDE)
    2000 base of code
    4000 base of data
400000 image base (00400000 to 00407FFF)
    2000 section alignment
    200 file alignment
    4.00 operating system version
    0.00 image version
    4.00 subsystem version
      0 Win32 version
    8000 size of image
    200 size of headers
      0 checksum
      3 subsystem (Windows CUI)

...
```

Дамп файла `CarLibrary.dll`

Обнаружена подпись PE

Тип файла: DLL

Значения заголовка файла

```

    14C машина (x86)
    3 количество разделов
4B37DCD8 дата и время Sun Dec 27 16:16:56 2011
    0 файловый указатель на таблицу символов
    0 количество символов
    E0 размер необязательного заголовка
2102 характеристики
    Исполняемый файл
    Машина с 32-битным словом
    DLL

```

Необязательные значения заголовка

```

    10B магический номер (PE32)
    8.00 версия редактора связей
    E00 размер кода
    600 размер инициализированных данных
    0 размер неинициализированных данных
2CDE точка входа (00402CDE)
    2000 база кода
    4000 база данных
400000 база образа (с 00400000 до 00407FFF)
    2000 выравнивание раздела
    200 выравнивание файла
    4.00 версия операционной системы
    0.00 версия образа
    4.00 версия подсистемы
    0 версия Win32
    8000 размер образа
    200 размер заголовков
    0 контрольная сумма
    3 подсистема (консольный пользовательский интерфейс Windows)
...

```

Запомните, что подавляющему большинству программистов .NET никогда не придется беспокоиться о формате данных заголовков, встроенных в сборку .NET. Если только вы не занимаетесь разработкой нового компилятора языка .NET (в таком случае вы должны позаботиться о подобной информации), то можете не вникать в тонкие детали заголовков. Однако помните, что такая информация используется “за кулисами”, когда операционная система Windows загружает двоичный образ в память.

Заголовок файла CLR

Заголовок CLR — это блок данных, который должны поддерживать все сборки .NET (и благодаря компилятору C# они его поддерживают), чтобы обслуживаться средой CLR. В двух словах: в заголовке CLR определены многочисленные флаги, которые позволяют исполняющей среде воспринимать компоновку управляемого файла. Например, существуют флаги, идентифицирующие местоположение метаданных и ресурсов внутри файла, версию исполняющей среды, для которой была построена сборка, значение (необязательного) открытого ключа и т.д. Чтобы просмотреть данные заголовка CLR для сборки .NET, необходимо открыть сборку в утилите `dumpbin.exe`, указав флаг `/clrheader`:

```
dumpbin /clrheader CarLibrary.dll
```

Вот как выглядит заголовок CLR:

```

Dump of file CarLibrary.dll
File Type: DLL

```

```

clr Header:
    48 cb
    2.05 runtime version
    2164 [    A74] RVA [size] of MetaData Directory
    1 flags
        IL Only
    0 entry point token
    0 [    0] RVA [size] of Resources Directory
    0 [    0] RVA [size] of StrongNameSignature Directory
    0 [    0] RVA [size] of CodeManagerTable Directory
    0 [    0] RVA [size] of VTableFixups Directory
    0 [    0] RVA [size] of ExportAddressTableJumps Directory
    0 [    0] RVA [size] of ManagedNativeHeader Directory

```

```

Summary
    2000 .reloc
    2000 .rsrc
    2000 .text

```

Дамп файла CarLibrary.dll

Тип файла: DLL

```

Заголовок clr:
    48 cb
    2.05 версия исполняющей среды
    2164 [    A74] RVA [размер] каталога MetaData
    1 флаги
        Только IL
    0 маркер записи
    0 [    0] RVA [размер] каталога Resources
    0 [    0] RVA [размер] каталога StrongNameSignature
    0 [    0] RVA [размер] каталога CodeManagerTable
    0 [    0] RVA [размер] каталога VTableFixups
    0 [    0] RVA [размер] каталога ExportAddressTableJumps
    0 [    0] RVA [размер] каталога ManagedNativeHeader

```

```

Сводка
    2000 .reloc
    2000 .rsrc
    2000 .text

```

И снова отметим, что разработчикам приложений .NET не придется беспокоиться о тонких деталях информации заголовка CLR. Просто найдите, что каждая сборка .NET содержит данные такого рода, которые исполняющая среда .NET использует "за кулисами" при загрузке образа в память. Теперь переключим внимание на информацию, которая является намного более полезной при решении повседневных задач программирования.

Код CIL, метаданные типов и манифест сборки

В своей основе сборка содержит код CIL, который, как вы помните, представляет собой промежуточный язык, не зависящий от платформы и процессора. Во время выполнения внутренний код CIL на лету посредством JIT-компилятора компилируется в инструкции, специфичные для конкретной платформы и процессора. Благодаря такому проектному решению сборки .NET действительно могут выполняться под управлением разнообразных архитектур, устройств и операционных систем. (Хотя вы можете благополучно жить и продуктивно работать, не разбираясь в деталях языка программирования CIL, в главе 18 предлагается введение в синтаксис и семантику CIL.)

Сборка также содержит метаданные, полностью описывающие формат внутренних типов и формат внешних типов, на которые сборка ссылается. Исполняющая среда .NET применяет эти метаданные для выяснения местоположения типов (и их членов) внутри двоичного файла, для размещения типов в памяти и для упрощения удаленного вызова методов. Более подробно детали формата метаданных .NET будут раскрыты в главе 15 во время исследования служб рефлексии.

Сборка должна также содержать связанный с ней *манифест* (по-другому называемый *метаданными сборки*). Манифест документирует каждый *модуль* внутри сборки, устанавливает версию сборки и указывает любые *внешние* сборки, на которые ссылается текущая сборка. Как вы увидите далее в главе, среда CLR интенсивно использует манифест сборки в процессе нахождения ссылок на внешние сборки.

Дополнительные ресурсы сборки

Наконец, сборка .NET может содержать любое количество встроенных ресурсов, таких как значки приложения, файлы изображений, звуковые клипы или таблицы строк. На самом деле платформа .NET поддерживает *подчиненные сборки*, которые содержат только локализованные ресурсы и ничего другого. Они могут быть удобны, когда необходимо выделять ресурсы на основе культуры (русской, немецкой, английской и т.д.) при построении интернационального программного обеспечения. Тема создания подчиненных сборок выходит за рамки настоящей книги; если вам интересно, обращайтесь за дополнительными сведениями о подчиненных сборках в документацию .NET 4.7 Framework.

Построение и потребление специальной библиотеки классов

Перед началом исследований мира библиотек классов .NET давайте создадим сборку *.dll (по имени CarLibrary), которая содержит небольшой набор открытых типов. Для построения библиотеки классов в Visual Studio создадим новый проект Class Library (Библиотека классов), выбрав пункт меню File⇒New Project (Файл⇒Новый проект), как показано на рис. 14.2. После создания проекта нужно удалить автоматически сгенерированный файл Class1.cs.

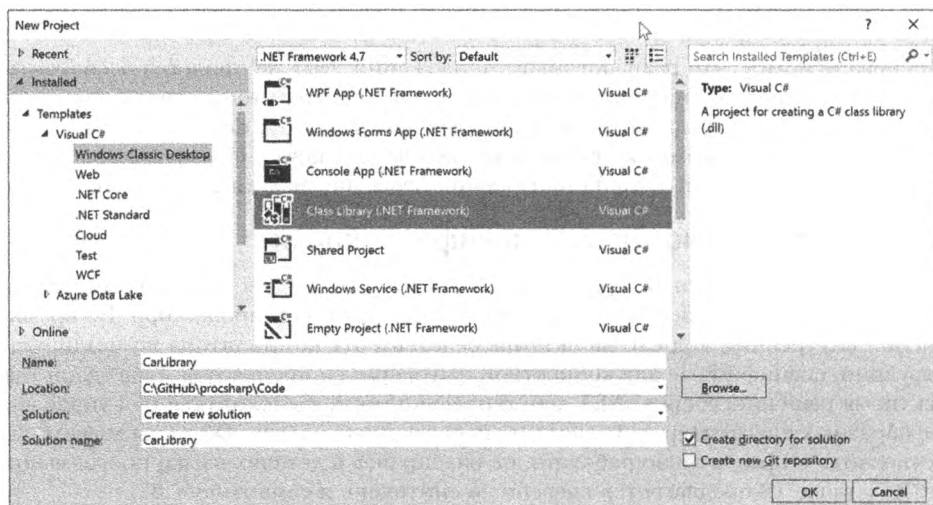


Рис. 14.2. Создание библиотеки классов C#

Проектное решение библиотеки для работы с автомобилями начинается с создания абстрактного базового класса по имени `Car`, который определяет разнообразные данные состояния через синтаксис автоматических свойств. Класс `Car` также имеет единственный абстрактный метод `TurboBoost()`, в котором применяется специальное перечисление (`EngineState`), представляющее текущее состояние двигателя автомобиля. Вставим в проект новый файл класса `C#` по имени `Car.cs` со следующим кодом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CarLibrary
{
    // Представляет состояние двигателя.
    public enum EngineState
    { engineAlive, engineDead }

    // Абстрактный базовый класс в иерархии.
    public abstract class Car
    {
        public string PetName {get; set;}
        public int CurrentSpeed {get; set;}
        public int MaxSpeed {get; set;}
        protected EngineState egnState = EngineState.engineAlive;
        public EngineState EngineState => egnState;
        public abstract void TurboBoost();
        public Car(){}
        public Car(string name, int maxSp, int currSp)
        {
            PetName = name; MaxSpeed = maxSp; CurrentSpeed = currSp;
        }
    }
}
```

Теперь предположим, что есть два непосредственных потомка класса `Car` с именами `MiniVan` (минивэн) и `SportsCar` (спортивный автомобиль). В каждом из них абстрактный метод `TurboBoost()` переопределяется для отображения подходящего сообщения в окне сообщений `Windows Forms`. Вставим в проект новый файл класса `C#` под названием `DerivedCars.cs` с таким кодом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

// Продолжайте чтение главы! Код не скомпилируется до тех пор,
// пока не будет добавлена ссылка на библиотеку .NET.
using System.Windows.Forms;

namespace CarLibrary
{
    public class SportsCar : Car
    {
        public SportsCar(){}
        public SportsCar(string name, int maxSp, int currSp)
            : base (name, maxSp, currSp){ }
    }
}
```

```

public override void TurboBoost()
{
    MessageBox.Show("Ramming speed!", "Faster is better...");
}
}

public class MiniVan : Car
{
    public MiniVan() { }
    public MiniVan(string name, int maxSp, int currSp)
        : base(name, maxSp, currSp) { }
    public override void TurboBoost()
    {
        // Минивэны имеют плохие возможности ускорения!
        engnState = EngineState.engineDead;
        MessageBox.Show("Eek!", "Your engine block exploded!");
    }
}
}

```

Обратите внимание, что каждый подкласс реализует метод `TurboBoost()` с использованием класса `MessageBox` из `Windows Forms`, который определен в сборке `System.Windows.Forms.dll`. Для применения в своей сборке типов, определенных внутри указанной внешней сборки, в проект `CarLibrary` потребуется добавить ссылку на эту сборку посредством диалогового окна `Add Reference` (Добавление ссылки), которое представлено на рис. 14.3 (оно доступно через пункт меню `Project⇒Add Reference` (Проект⇒Добавить ссылку)).

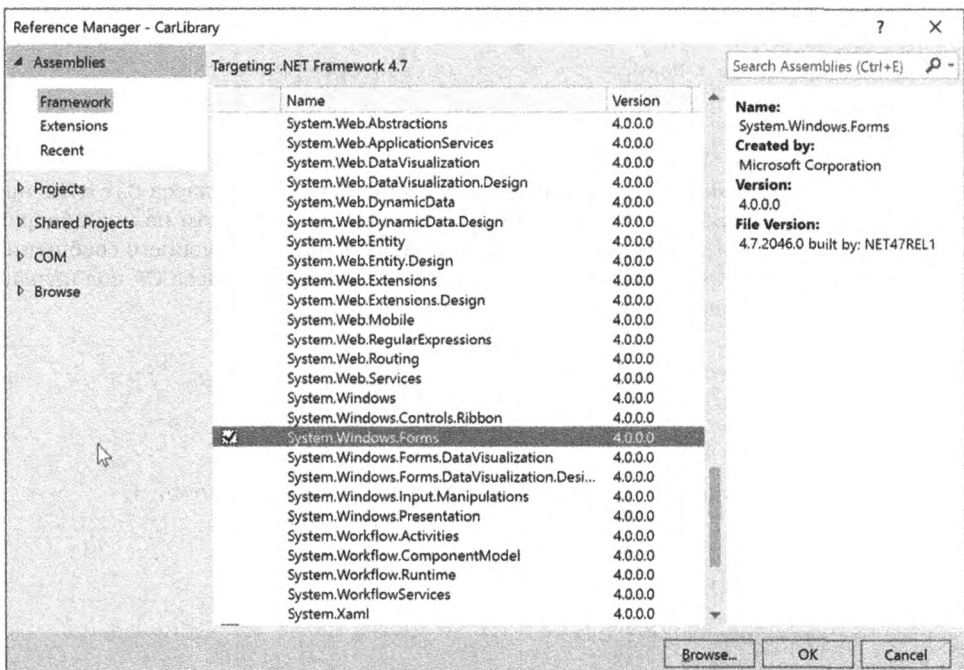


Рис. 14.3. Добавление ссылок на внешние сборки .NET с помощью диалогового окна `Add Reference`

По-настоящему важно понимать, что в области Framework диалогового окна Add Reference присутствуют далеко не все сборки, имеющиеся на машине. Диалоговое окно Add Reference не будет отображать специальные библиотеки, равно как и не будет отображать все библиотеки, расположенные в GAC (причины вы узнаете далее в главе). Взамен просто предлагается список общих сборок, которые среда Visual Studio была изначально запрограммирована отображать. При построении приложений, которые требуют использования сборки, отсутствующей в диалоговом окне Add Reference, понадобится щелкнуть на узле Browse (Обзор) и вручную перейти к интересующему файлу *.dll или *.exe.

На заметку! Имейте в виду, что в области Recent (Недавние) диалогового окна Add Reference поддерживается актуальный список сборок, на которые производилась ссылка ранее. Он может быть удобным, т.к. во многих проектах .NET приходится применять тот же самый основной набор внешних библиотек.

Исследование манифеста

Перед использованием CarLibrary.dll в клиентском приложении давайте посмотрим, как библиотека кода устроена внутри. Предполагая, что проект был скомпилирован, загрузим CarLibrary.dll в утилиту ildasm.exe, выбрав пункт меню File⇒Open (Файл⇒Открыть) и в открывшемся диалоговом окне перейдя в подкаталог bin\Debug каталога проекта CarLibrary. В инструменте дизассемблера IL должна отобразиться только что созданная библиотека (рис. 14.4).

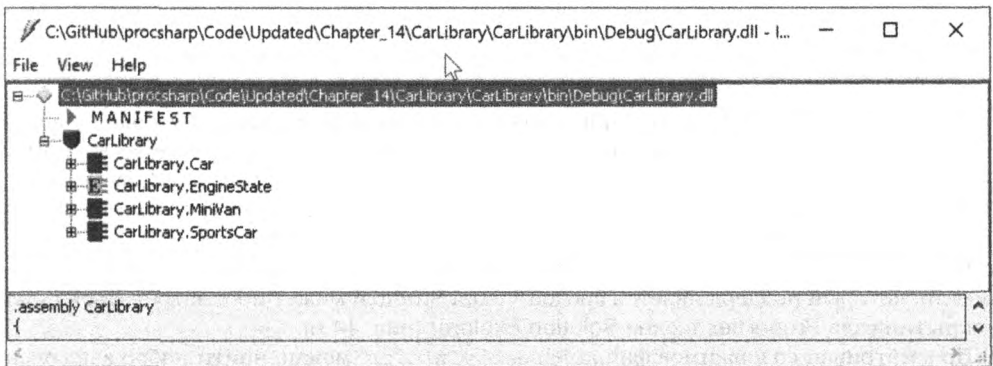


Рис. 14.4. Библиотека CarLibrary.dll, загруженная в ildasm.exe

Теперь откроем манифест сборки CarLibrary.dll, дважды щелкнув на значке MANIFEST (Манифест). В первом блоке кода манифеста указываются все внешние сборки, требуемые текущей сборкой для нормального функционирования. Вспомните, что в CarLibrary.dll применяются типы из внешних сборок mscorlib.dll и System.Windows.Forms.dll, которые перечислены в манифесте с использованием маркера .assembly extern:

```
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
```



```
.assembly extern System.Windows.Forms
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
```

Здесь каждый блок `.assembly extern` уточняется директивами `.publickeytoken` и `.ver`. Инструкция `.publickeytoken` присутствует только в случае, если сборка была сконфигурирована со *строгим именем* (более подробно о строгих именах речь пойдет в разделе “Понятие строгих имен” далее в главе). Маркер `.ver` определяет числовой идентификатор версии ссылаемой сборки.

После внешних ссылок вы обнаружите набор маркеров `.custom`, которые идентифицируют атрибуты уровня сборки (информацию об авторском праве, название компании, версию сборки и т.д.). Ниже приведена (весьма) небольшая часть этой порции данных манифеста:

```
.assembly CarLibrary
{
    .custom instance void ...AssemblyDescriptionAttribute...
    .custom instance void ...AssemblyConfigurationAttribute...
    .custom instance void ...RuntimeCompatibilityAttribute...
    .custom instance void ...TargetFrameworkAttribute...
    .custom instance void ...AssemblyTitleAttribute...
    .custom instance void ...AssemblyTrademarkAttribute...
    .custom instance void ...AssemblyCompanyAttribute...
    .custom instance void ...AssemblyProductAttribute...
    .custom instance void ...AssemblyCopyrightAttribute...
    ...
    .ver 1:0:0:0
}
.module CarLibrary.dll
```

Обычно такие настройки устанавливаются визуально с применением окна свойств текущего проекта. Вернувшись в среду Visual Studio, щелкнем на значке Properties внутри окна Solution Explorer и затем на кнопке Assembly Information (Информация о сборке) внутри (автоматически выбранной) вкладки Application. Откроется диалоговое окно Assembly Information (Информация о сборке), показанное на рис. 14.5.

После внесения и сохранения изменений будет обновлен файл `AssemblyInfo.cs` проекта, который поддерживается средой Visual Studio и может просматриваться путем раскрытия узла Properties в окне Solution Explorer (рис. 14.6).

Просматривая содержимое файла `AssemblyInfo.cs`, можно найти набор заключенных в квадратные скобки *атрибутов* .NET. Вот пример:

```
[assembly: AssemblyTitle("CarLibrary")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("CarLibrary")]
[assembly: AssemblyCopyright("Copyright © 2017")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

Роль атрибутов подробно обсуждается в главе 15, так что на данном этапе не беспокойтесь о деталях. Пока просто знайте, что большинство атрибутов из файла `AssemblyInfo.cs` будет применяться для обновления значений в маркерах `.custom` внутри манифеста сборки.

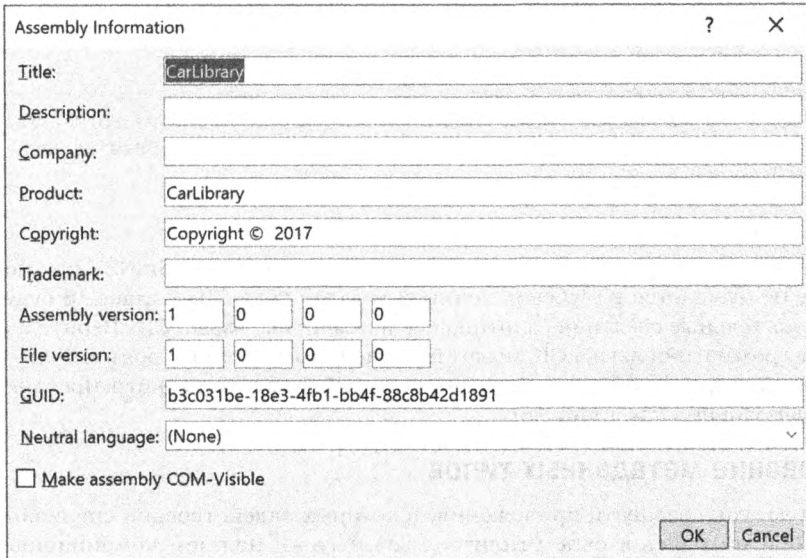


Рис. 14.5. Редактирование информации о сборке в диалоговом окне Assembly Information

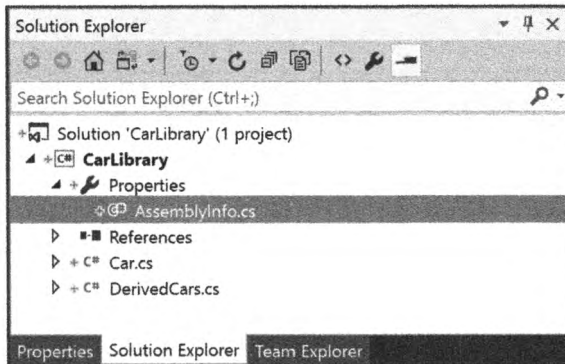


Рис. 14.6. После использования окна свойств проекта обновляется файл AssemblyInfo.cs

Исследование кода CIL

Вспомните, что сборка не содержит инструкций, специфичных для платформы; взамен в ней хранятся инструкции на независимом от платформы общем промежуточном языке (Common Intermediate Language — CIL). Когда исполняющая среда .NET загружает сборку в память, лежащий в ее основе код CIL компилируется (с использованием JIT-компилятора) в инструкции, воспринимаемые целевой платформой. Например, если в утилите ildasm.exe дважды щелкнуть на методе TurboBoost() класса SportsCar, то откроется новое окно, в котором будут отображаться инструкции CIL, реализующие данный метод:

```
.method public hidebysig virtual instance void
  TurboBoost() cil managed
{
  // Code size      18 (0x12)
```

```

.maxstack 8
IL_0000: nop
IL_0001: ldstr  "Ramming speed!"
IL_0006: ldstr  "Faster is better..."
IL_000b: call  valuetype [System.Windows.Forms]System.Windows.Forms.DialogResult
         [System.Windows.Forms]System.Windows.Forms.MessageBox::Show(string, string)
IL_0010: pop
IL_0011: ret
) // end of method SportsCar::TurboBoost

```

Несмотря на то что большая часть разработчиков приложений .NET при повседневной работе не нуждается в глубоких знаниях деталей кода CIL, в главе 18 будут приведены дополнительные сведения о синтаксисе и семантике языка CIL. Верите или нет, но понимание грамматики языка CIL может быть полезным, когда строятся более сложные приложения, которые требуют расширенных служб, таких как конструирование сборок во время выполнения (см. главу 18).

Исследование метаданных типов

Перед тем, как создавать приложения, в которых задействована специальная библиотека .NET, находясь в окне утилиты `ildasm.exe`, нажмем комбинацию клавиш `<Ctrl+M>`: отобразятся метаданные для каждого типа внутри сборки `CarLibrary.dll` (рис. 14.7).

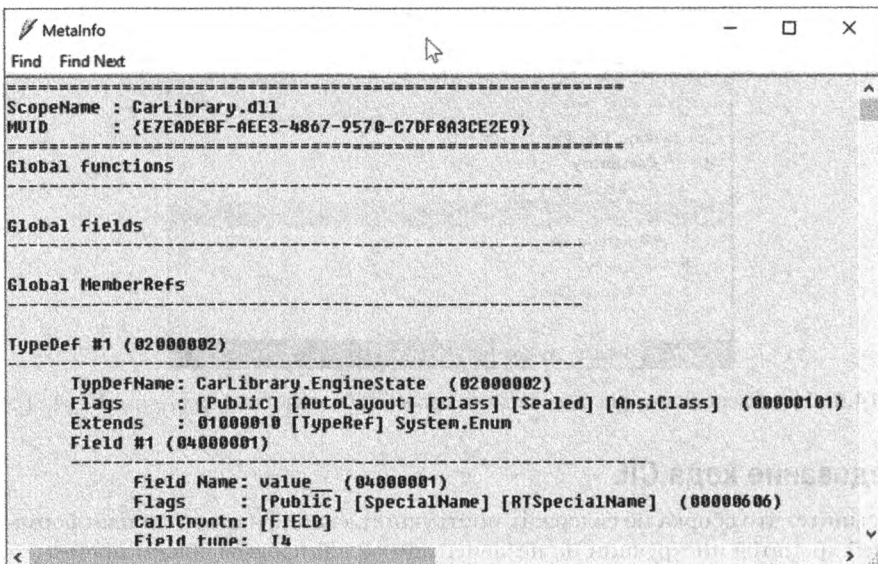


Рис. 14.7. Метаданные для типов в сборке `CarLibrary.dll`

Как объясняется в следующей главе, метаданные сборки являются важным элементом платформы .NET и служат основой для многочисленных технологий (сериализация объектов, позднее связывание, расширяемые приложения и т.д.). В любом случае теперь, когда мы заглянули внутрь сборки `CarLibrary.dll`, можно приступать к построению клиентских приложений, в которых будут применяться типы из сборки.

Построение клиентского приложения C#

Поскольку все типы в CarLibrary были объявлены с ключевым словом `public`, другие приложения .NET имеют возможность пользоваться ими. Вспомните, что типы могут также определяться с применением ключевого слова `internal` языка C# (в действительности это стандартный режим доступа в C#). Внутренние типы могут использоваться только в сборке, где они определены. Внешние клиенты не могут ни видеть, ни создавать экземпляры типов, помеченных ключевым словом `internal`.

Чтобы задействовать функциональность построенной библиотеки, создадим новый проект консольного приложения C# по имени CSharpCarClient. Затем установим ссылку на CarLibrary.dll с применением узла Browse диалогового окна Add Reference (если сборка CarLibrary.dll компилировалась в Visual Studio, то она будет находиться в подкаталоге bin\Debug внутри каталога проекта CarLibrary). Теперь можно строить клиентское приложение для использования внешних типов. Модифицируем начальный файл кода C#, как показано ниже:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

// Не забудьте импортировать пространство имен CarLibrary!
using CarLibrary;

namespace CSharpCarClient
{
    public class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** C# CarLibrary Client App *****");

            // Создать объект SportsCar.
            SportsCar viper = new SportsCar("Viper", 240, 40);
            viper.TurboBoost();

            // Создать объект MiniVan.
            MiniVan mv = new MiniVan();
            mv.TurboBoost();

            Console.WriteLine("Done. Press any key to terminate");
            Console.ReadLine();
        }
    }
}
```

Код выглядит очень похожим на код в других приложениях, которые разрабатывались в книге ранее. Единственный интересный аспект связан с тем, что в клиентском приложении C# теперь применяются типы, определенные внутри отдельной специальной библиотеки. Запустив приложение, можно наблюдать отображение разнообразных окон с сообщениями.

Вас может интересовать, что в точности происходит при добавлении ссылки на CarLibrary.dll в диалоговом окне Add Reference. Щелкнув на кнопке Show All Files (Показать все файлы) в окне Solution Explorer, можно увидеть, что среда Visual Studio добавила копию исходной библиотеки CarLibrary.dll в подкаталог bin\Debug каталога проекта CSharpCarClient (рис. 14.8).

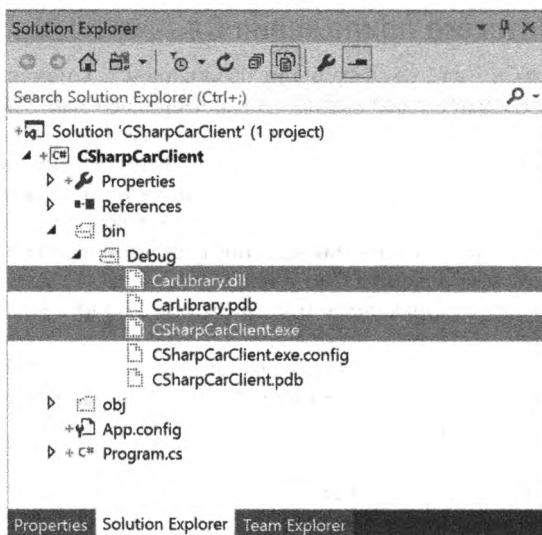


Рис. 14.8. Среда Visual Studio копирует закрытые сборки в каталог клиентского приложения

Как вскоре будет объяснено, библиотека `CarLibrary.dll` сконфигурирована как закрытая сборка (автоматическое поведение для всех проектов библиотек классов Visual Studio). Когда производится ссылка на закрытые сборки в новых приложениях (вроде `CSharpCarClient.exe`), IDE-среда реагирует помещением копии библиотеки в выходной каталог клиентского приложения.

Исходный код. Проект `CSharpCarClient` доступен в подкаталоге `Chapter_14`.

Построение клиентского приложения Visual Basic

Вспомните, что платформа .NET позволяет разработчикам разделять скомпилированный код между языками программирования. Чтобы проиллюстрировать языковую независимость платформы .NET, давайте создадим еще один проект консольного приложения (по имени `VisualBasicCarClient`) с использованием на этот раз языка Visual Basic (рис. 14.9). После создания проекта установим ссылку на `CarLibrary.dll` с применением диалогового окна `Add Reference`, которое открывается выбором пункта меню `Project⇒Add Reference`. Среда Visual Studio запоминает ранее просмотренные файлы, так что файл `CarLibrary.dll` должен находиться в списке `Recent` диалогового окна `Add Reference`.

Подобно C# язык Visual Basic позволяет перечислять все пространства имен, используемые внутри текущего файла. Тем не менее, вместо ключевого слова `using`, применяемого в C#, для такой цели в Visual Basic служит ключевое слово `Imports`, поэтому добавим в файл кода `Module1.vb` следующий оператор `Imports`:

```
Imports CarLibrary

Module Module1
    Sub Main()
    End Sub
End Module
```

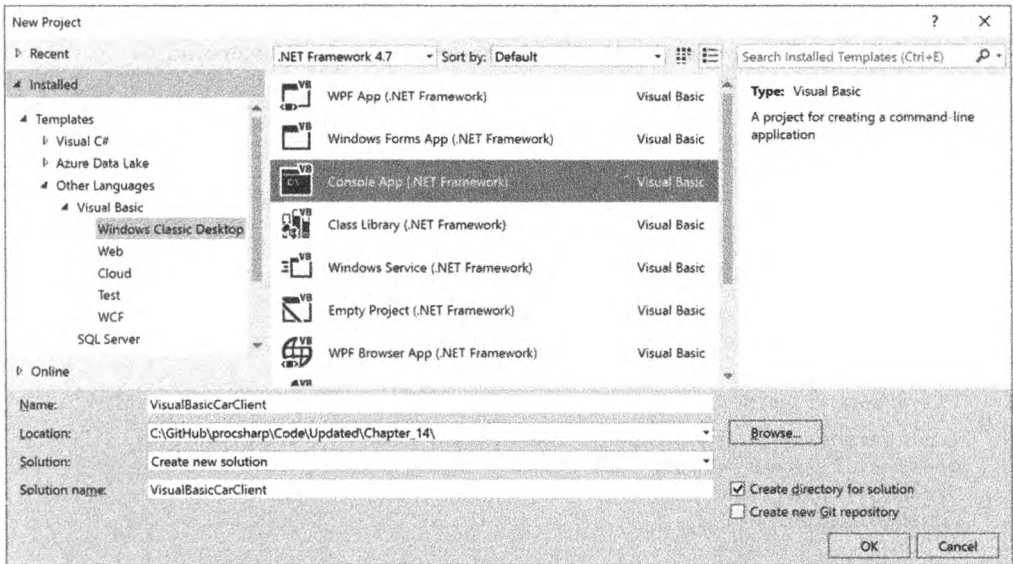


Рис. 14.9. Создание проекта консольного приложения Visual Basic

Обратите внимание, что метод `Main()` определен внутри типа модуля Visual Basic. Выразаясь кратко, модули представляют собой систему обозначений Visual Basic для определения класса, который может содержать только статические методы (очень похоже на статический класс C#). Итак, чтобы испробовать типы `MiniVan` и `SportsCar`, используя синтаксис Visual Basic, модифицируем метод `Main()`, как показано ниже:

```
Sub Main()
    Console.WriteLine("***** VB CarLibrary Client App *****")

    ' Локальные переменные объявляются с применением ключевого слова Dim.
    Dim myMiniVan As New MiniVan()
    myMiniVan.TurboBoost()

    Dim mySportsCar As New SportsCar()
    mySportsCar.TurboBoost()
    Console.ReadLine()
End Sub
```

После компиляции и запуска приложения снова отобразится последовательность окон с сообщениями. Кроме того, новое клиентское приложение имеет собственную локальную копию `CarLibrary.dll` в своем подкаталоге `bin\Debug`.

Межязыковое наследование в действии

Привлекательным аспектом разработки в .NET является понятие *межязыкового наследования*. В целях иллюстрации давайте создадим новый класс Visual Basic, производный от типа `SportsCar` (который был написан на C#). Для начала добавим в текущее приложение Visual Basic новый файл класса по имени `PerformanceCar.vb` (выбрав пункт меню `Project → Add Class` (Проект → Добавить класс)). Модифицируем начальное определение класса, унаследовав его от типа `SportsCar` с применением ключевого слова `Inherits`. Затем переопределим абстрактный метод `TurboBoost()`, используя ключевое слово `Overrides`:

```
Imports CarLibrary

' Этот класс VB унаследован от класса SportsCar, написанного на C#.
Public Class PerformanceCar
    Inherits SportsCar

    Public Overrides Sub TurboBoost()
        Console.WriteLine("Zero to 60 in a cool 4.8 seconds...")
    End Sub
End Class
```

Чтобы протестировать новый тип класса, модифицируем код метода `Main()` в модуле:

```
Sub Main()
    ...
    Dim dreamCar As New PerformanceCar()
    ' Использовать унаследованное свойство.
    dreamCar.PetName = "Hank"
    dreamCar.TurboBoost()
    Console.ReadLine()
End Sub
```

Обратите внимание, что объект `dreamCar` способен обращаться к любому открытому члену (такому как свойство `PetName`), находящемуся выше в цепочке наследования, невзирая на тот факт, что базовый класс был определен на совершенно другом языке и полностью в другой сборке! Возможность расширения классов за пределы границ сборок в независимой от языка манере — естественный аспект цикла разработки в .NET. Он упрощает применение скомпилированного кода, написанного программистами, которые предпочли не создавать свой разделяемый код на языке C#.

Исходный код. Проект `VisualBasicCarClient` доступен в подкаталоге `Chapter_14`.

Понятие закрытых сборок

Говоря формально, библиотеки классов, которые были созданы до сих пор в главе, развертывались как *закрытые сборки*. Закрытые сборки должны размещаться в том же самом каталоге, что и клиентское приложение, которое их использует (в *каталоге приложения*), или в одном из его подкаталогов. Вспомните, что при добавлении ссылки на `CarLibrary.dll` во время построения приложений `CSharpCarClient.exe` и `VbNetCarClient.exe` среда Visual Studio реагировала помещением копии `CarLibrary.dll` в каталоги упомянутых клиентских приложений (по крайней мере, после первой компиляции).

Когда в клиентской программе применяются типы, определенные в этой внешней сборке, среда CLR просто загружает локальную копию `CarLibrary.dll`. Поскольку при поиске внешних сборок исполняющая среда .NET не должна просматривать системный реестр, сборки `CSharpCarClient.exe` (или `VisualBasicCarClient.exe`) и `CarLibrary.dll` можно перемещать в новое местоположение на машине и успешно запускать приложение (что часто называется *развертыванием с помощью Хсору*).

Удаление (или копирование) приложения, в котором используются исключительно закрытые сборки, не требует особого труда: нужно просто удалить (или скопировать) каталог приложения. Важнее всего то, что не приходится переживать по поводу возможного нарушения работы других приложений на машине по причине удаления закрытых сборок.

Удостоверение закрытой сборки

Полное удостоверение закрытой сборки состоит из дружественного имени и числовой версии, которые записываются в манифест сборки. *Дружественное имя* — просто название модуля, содержащего манифест сборки, без файлового расширения. Например, если просмотреть манифест сборки `CarLibrary.dll`, то в нем можно обнаружить следующие данные:

```
.assembly CarLibrary
{
    ...
    .ver 1:0:0:0
}
```

Учитывая изолированную природу закрытой сборки, должно быть понятно, что среда CLR не применяет номер версии при выяснении места ее размещения. Предположение заключается в том, что закрытые сборки не нуждаются в выполнении сложной проверки версий, поскольку клиентское приложение является единственной сущностью, которой известно об их наличии. По указанной причине на одной машине вполне может находиться множество копий той же самой закрытой сборки в разных каталогах приложений.

Понятие процесса зондирования

Исполняющая среда .NET определяет местонахождение закрытой сборки, используя прием под названием *зондирование*, который в действительности не такой навязчивый, как может показаться. Зондирование представляет собой процесс отображения запроса внешней сборки на местоположение требуемого двоичного файла. Строго говоря, запрос на загрузку внешней сборки может быть либо *явным*, либо *неявным*. Неявный запрос загрузки происходит, когда среда CLR исследует манифест для выяснения, где находится сборка, по маркерам `.assembly extern`. Вот пример:

```
// Неявный запрос загрузки.
.assembly extern CarLibrary
{ ... }
```

Явный запрос загрузки производится программно с применением метода `Load()` или `LoadFrom()` класса `System.Reflection.Assembly` обычно в целях позднего связывания или динамического вызова членов типа. Более подробно эти темы рассматриваются в главе 15, а пока ниже приведен пример явного запроса загрузки:

```
// Явный запрос загрузки, основанный на дружественном имени.
Assembly asm = Assembly.Load("CarLibrary");
```

Среда CLR извлекает дружественное имя сборки и начинает зондирование каталога клиентского приложения в поисках файла по имени `CarLibrary.dll`. Если указанный файл обнаружить не удалось, тогда предпринимается попытка найти исполняемую сборку с таким же дружественным именем (например, `CarLibrary.exe`). Если ни один из файлов в каталоге приложения не найден, то исполняющая среда генерирует исключение `FileNotFoundException` во время выполнения.

На заметку! Выражаясь формально, если копию запрашиваемой сборки не удалось найти в каталоге клиентского приложения, тогда среда CLR будет также искать клиентский подкаталог с тем же именем, что и дружественное имя сборки (например, `C:\MyClient\CarLibrary`). Если запрашиваемая сборка размещена в этом подкаталоге, то среда CLR загрузит ее в память.

Конфигурирование закрытых сборок

В то время как допускается развертывать приложение .NET простым копированием всех требуемых сборок в единственный каталог на жестком диске пользователя, скорее всего, вы захотите определить несколько подкаталогов для группирования связанного содержимого. Например, предположим, что имеется каталог приложения по имени `C:\MyApp`, содержащий файл `CSharpCarClient.exe`. Внутри этого каталога может быть создан подкаталог `MyLibraries`, в котором находится сборка `CarLibrary.dll`.

Несмотря на подразумеваемую связь между упомянутыми двумя каталогами, среда CLR *не будет* зондировать подкаталог `MyLibraries`, если только не предоставить ей конфигурационный файл. Конфигурационные файлы содержат разнообразные XML-элементы, которые позволяют влиять на процесс зондирования. Конфигурационные файлы должны иметь такое же имя, как у запускаемого приложения, обладать расширением `*.config` и быть развернутыми в каталоге клиентского приложения. Таким образом, если вы хотите создать конфигурационный файл для приложения `CSharpCarClient.exe`, то он должен получить имя `CSharpCarClient.exe.config` и располагаться (в текущем примере) внутри каталога `C:\MyApp`.

Чтобы проиллюстрировать процесс, создадим на диске `C:` новый каталог по имени `MyApp`, используя проводник Windows. Далее скопируем в него файлы `CSharpCarClient.exe` и `CarLibrary.dll` и запустим программу, дважды щелкнув на исполняемом файле. На данном этапе программа должна запуститься успешно.

Теперь создадим в `C:\MyApp` новый подкаталог по имени `MyLibraries` (рис. 14.10) и переместим в него сборку `CarLibrary.dll`.

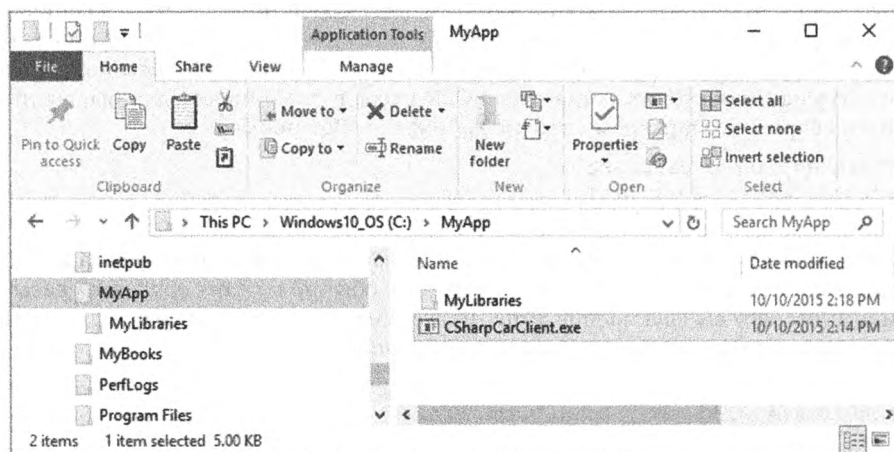


Рис. 14.10. Теперь файл `CarLibrary.dll` находится в подкаталоге `MyLibraries`

Попробуем запустить программу снова, дважды щелкнув на исполняемом файле. Поскольку среде CLR не удастся обнаружить сборку по имени `CarLibrary` непосредственно в каталоге приложения, генерируется необработанное исключение `FileNotFoundException`.

Для того чтобы указать среде CLR на необходимость зондирования подкаталога `MyLibraries`, создадим с помощью любого текстового редактора конфигурационный файл `CSharpCarClient.exe.config` и сохраним его в каталоге, содержащем приложение `CSharpCarClient.exe` (`C:\MyApp` в рассматриваемом примере).

Откроем файл `CSharpCarClient.exe.config` и поместим в него следующее содержимое (не забывайте, что язык XML чувствителен к регистру символов):

```

<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="MyLibraries"/>
    </assemblyBinding>
  </runtime>
</configuration>

```

Файлы *.config в .NET всегда начинаются с корневого элемента <configuration>. Вложенный в него элемент <runtime> может содержать элемент <assemblyBinding>, а тот в свою очередь — вложенный элемент <probing>. В данном примере главный интерес представляет атрибут privatePath, т.к. именно он служит для указания подкаталогов внутри каталога приложения, которые среда CLR должна зондировать.

Завершив создание конфигурационного файла CSharpCarClient.exe.config, снова запустим клиентское приложение. На этот раз выполнение CSharpCarClient.exe должно пройти без проблем (в противном случае необходимо проверить конфигурационный файл на предмет опечаток).

Обязательно обратите внимание на то, что в элементе <probing> не указывается, какая сборка находится в заданном подкаталоге. Другими словами, невозможно указать, что сборка CarLibrary находится в подкаталоге MyLibraries, а сборка MathLibrary — в подкаталоге OtherStuff. Элемент <probing> просто инструктирует среду CLR о том, что она должна просмотреть все перечисленные подкаталоги в поисках запрашиваемой сборки до тех пор, пока не встретится первое совпадение.

На заметку! Очень важно запомнить, что атрибут privatePath не может применяться для указания абсолютного (C:\Каталог\Подкаталог) или относительного (..\Каталог\ДругойПодкаталог) пути! Если нужно указать каталог, находящийся за пределами каталога клиентского приложения, то придется применять совершенно другой XML-элемент под названием <codeBase> (он более подробно рассматривается позже в главе).

Атрибут privatePath допускает указание множества подкаталогов в виде списка значений, разделенных точками с запятой. В настоящий момент в этом нет необходимости, но ниже приведен пример, в котором среда CLR инструктируется на предмет просмотра клиентских подкаталогов MyLibraries и MyLibraries\Tests:

```
<probing privatePath="MyLibraries;MyLibraries\Tests"/>
```

В целях тестирования изменим (произвольным образом) имя конфигурационного файла и попробуем запустить приложение еще раз. Теперь клиентское приложение должно потерпеть неудачу. Вспомните, что файл *.config должен иметь то же имя, что и у связанного клиентского приложения. В качестве последнего теста откроем конфигурационный файл для редактирования и изменим регистр символов любого XML-элемента. После сохранения файла запуск клиентского приложения должен снова оказаться безуспешным (поскольку язык XML чувствителен к регистру символов).

На заметку! Имейте в виду, что среда CLR будет загружать ту сборку, которая во время процесса зондирования обнаруживается первой. Например, если в каталоге C:\MyApp имеется копия CarLibrary.dll, то именно она загрузится в память, а копия, содержащаяся в подкаталоге MyLibraries, будет проигнорирована.

Роль файла App.Config

Хотя всегда есть возможность готовить конфигурационные XML-файлы вручную в любом текстовом редакторе, среда Visual Studio позволяет создавать конфигурацион-

ный файл во время разработки клиентской программы. По умолчанию новый проект Visual Studio содержит конфигурационный файл, предназначенный для редактирования. Если когда-нибудь понадобится добавить его вручную, то можно выбрать пункт меню Project⇒Add New Item. На рис. 14.11 обратите внимание, что для этого файла оставлено предложенное стандартное имя App.config.

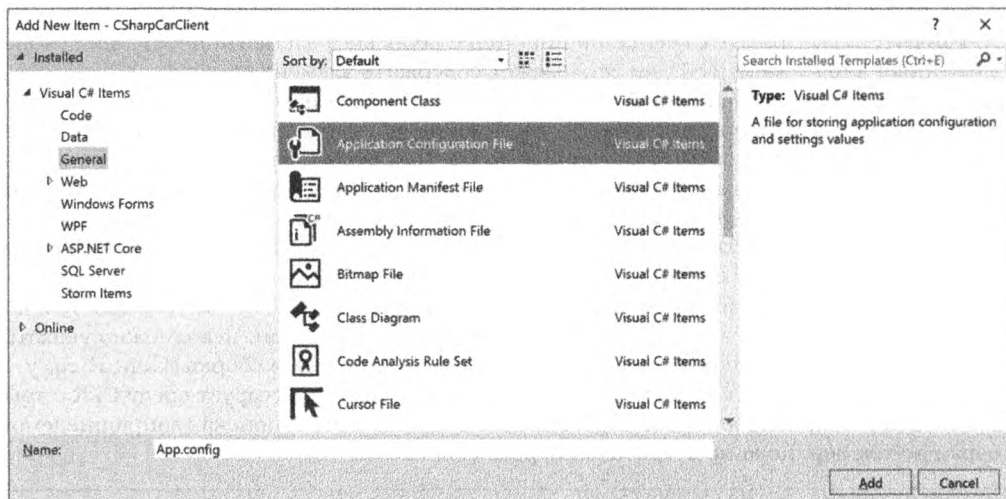


Рис. 14.11. Вставка нового конфигурационного XML-файла

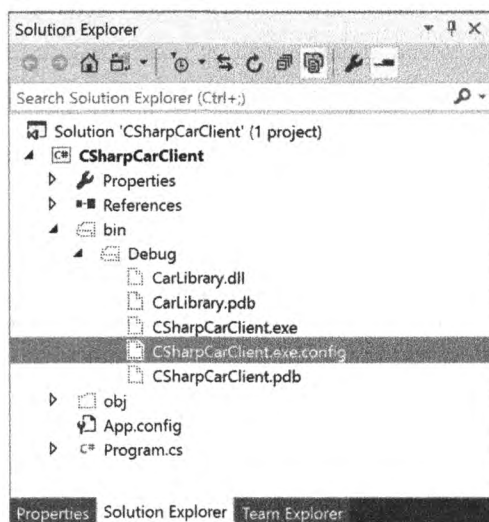


Рис. 14.12. Содержимое App.config будет скопировано в корректно именованный файл *.config внутри выходного каталога проекта

Открыв файл App.config для просмотра, можно увидеть минимальный набор инструкций, к которым вы будете добавлять дополнительные элементы:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0"
      sku=".NETFramework,Version=v4.7" />
  </startup>
</configuration>
```

Необходимо отметить один интересный момент. Каждый раз, когда вы компилируете проект, среда Visual Studio будет автоматически копировать данные из App.config в новый файл, расположенный в каталоге bin\Debug, используя надлежащее соглашение об именовании (вроде CSharpCarClient.exe.config). Однако такое действие будет происходить, только если конфигурационный файл действительно имеет имя App.config (рис. 14.12).

При таком подходе от вас требуется лишь поддерживать файл App.config, а среда Visual Studio позаботится о том, чтобы в каталоге приложения содержались актуальные конфигурационные данные (даже если вы переименуете проект).

Понятие разделяемых сборок

Теперь, когда вы понимаете, каким образом развертывать и конфигурировать закрытые сборки, можно приступить к исследованию роли *разделяемых сборок*. Подобно закрытой сборке разделяемая сборка представляет собой коллекцию типов, предназначенных для многократного использования во множестве проектов. Самое очевидное отличие между разделяемыми и закрытыми сборками заключается в том, что единственная копия разделяемой сборки может быть задействована несколькими приложениями на той же самой машине.

Вспомните, что все приложения, создаваемые в книге, требуют доступа к сборке `mscorlib.dll`. Заглянув в каталог любого из этих клиентских приложений, вы не обнаружите там локальной копии упомянутой сборки .NET. Причина в том, что `mscorlib.dll` развернута как разделяемая сборка. Ясно, что если необходимо создать библиотеку классов для применения в масштабах всей машины, то именно так и следует поступать.

На заметку! Решение о том, каким образом должна развертываться библиотека кода — как закрытая или как разделяемая — является еще одним вопросом, который должен быть обдуман на этапе проектирования, и зависит от многих специфических деталей проекта. Существует эмпирическое правило: при построении библиотек, которые необходимо использовать в разнообразных приложениях, разделяемые сборки могут оказаться более удобными тем, что их легко обновлять до новых версий (как будет показано позже).

Глобальный кеш сборок

Из указанного выше следует, что разделяемая сборка не развертывается внутри того же самого каталога, где находится приложение, в котором она применяется. Взамен разделяемые сборки устанавливаются в глобальный кеш сборок (Global Assembly Cache — GAC). Тем не менее, точное местоположение GAC будет зависеть от версии платформы .NET, установленной на целевом компьютере.

На машинах с версиями, предшествующими .NET 4.0, глобальный кеш сборок размещен в подкаталоге `assembly` внутри каталога Windows (например, `C:\Windows\assembly`). В наши дни упомянутый подкаталог можно считать “историческим GAC”, т.к. он содержит только библиотеки .NET, скомпилированные для версий 1.0, 2.0, 3.0 или 3.5 (рис. 14.13).

На заметку! Устанавливать в GAC исполняемые сборки (*.exe) не разрешено. В качестве разделяемых можно развертывать только сборки с расширением *.dll.

С выходом версии .NET 4.0 в Microsoft решили изолировать библиотеки для .NET 4.0 и последующих версий в отдельном месте, находящемся в `C:\Windows\Microsoft.NET\assembly\GAC_MSIL` (рис. 14.14).

В этом новом каталоге вы обнаружите набор подкаталогов, каждый из которых назван идентично дружественному имени отдельной библиотеки кода (скажем, `\System.Windows.Forms`, `\System.Core` и т.д.). Внутри заданного подкаталога с дружественным именем есть еще один подкаталог, который всегда именуется в соответствие со следующим соглашением:

```
v4.0_старшийНомер.младшийНомер.номерСборки.номерРедакции_значениеМаркераОткрытогоКлюча
```

Префикс `v4.0` обозначает, что библиотека скомпилирована в .NET 4.0 или последующей версии. За префиксом следует одиночный символ подчеркивания (`_`) и версия рассматриваемой библиотеки (например, `1.0.0.0`). После пары символов подчеркивания находится число, называемое *значением маркера открытого ключа*. Как вы увидите в следующем разделе, значение открытого ключа является частью “строного имени” сборки. Наконец, внутри данного подкаталога находится копия интересующей сборки *.dll.

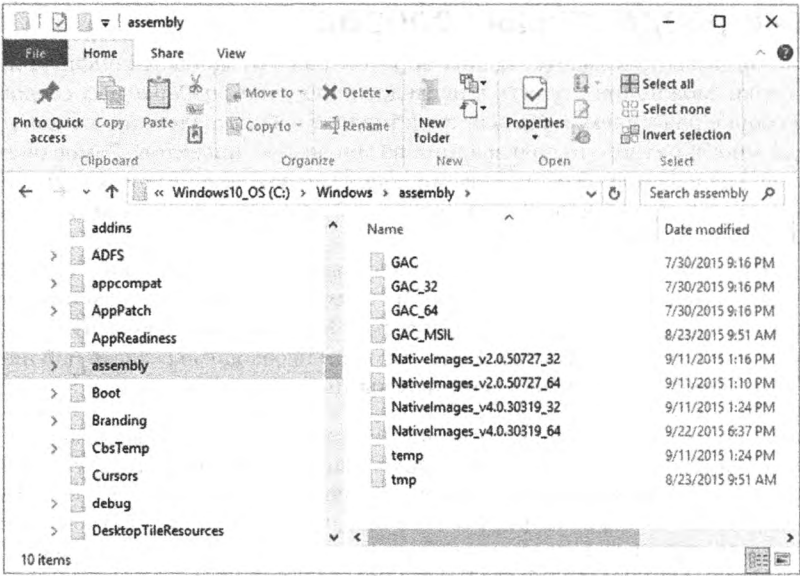


Рис. 14.13. “Исторический” глобальный кеш сборок

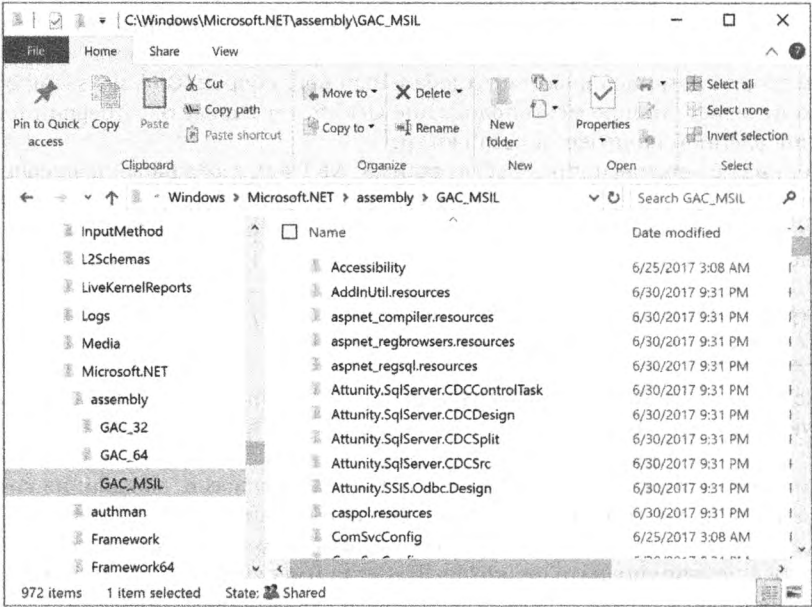


Рис. 14.14. Глобальный кеш сборок в версии .NET 4.0 и новее

В настоящей книге предполагается построение приложений с использованием версии .NET 4.7; таким образом, если вы устанавливаете библиотеку в GAC, то она попадет в каталог C:\Windows\Microsoft.NET\assembly\GAC_MSIL. Однако не забывайте, что если проект библиотеки классов сконфигурирован на компиляцию для платформы .NET 3.5 или более ранней версии, то разделяемые библиотеки следует искать в каталоге C:\Windows\assembly.

Понятие строгих имен

Перед развертыванием сборки в ГАС ей должно быть назначено *строгое имя*, которое применяется для уникальной идентификации издателя заданного двоичного файла .NET. Имейте в виду, что в роли “издателя” может выступать отдельный программист, подразделение внутри компании или компания в целом.

В некоторых отношениях строгие имена являются современным .NET-эквивалентом схемы распознавания с помощью глобально уникальных идентификаторов (globally unique identifier — GUID) из технологии COM. Если вы ранее имели дело с COM, то можете вспомнить, что идентификаторы приложений (AppID) — это идентификаторы GUID, указывающие на конкретные COM-приложения. В отличие от значений GUID в COM, которые представляют собой всего лишь 128-битные числа, строгие имена основаны (отчасти) на двух криптографически связанных ключах (*открытом* и *секретном*), которые характеризуются гораздо более высокой уникальностью и устойчивостью к подделке, нежели простые GUID.

Формально строгое имя образовано из набора связанных данных, большинство которых указывается с использованием перечисленных ниже атрибутов уровня сборки.

- Дружественное имя сборки (представляющее собой, как вы помните, имя сборки без файлового расширения).
- Номер версии сборки (назначенный посредством атрибута [AssemblyVersion]).
- Значение открытого ключа (назначенное с помощью атрибута [AssemblyKeyFile]).
- Необязательное значение, идентифицирующее культуру, для целей локализации (назначенное с применением атрибута [AssemblyCulture]).
- Встроенная цифровая подпись, созданная с использованием хеш-кода содержимого сборки и значения секретного ключа.

Чтобы получить строгое имя для сборки, сначала посредством утилиты `sn.exe` из .NET Framework генерируются данные открытого и секретного ключей. Утилита `sn.exe` генерирует файл (обычно оканчивающийся расширением `*.snk` (Strong Name Key — ключ строгого имени)), который содержит данные для двух разных, но математически связанных ключей — открытого и секретного. После того как компилятору C# указано местоположение результирующего файла `*.snk`, он запишет полное значение открытого ключа в манифест сборки с применением маркера `.publickey`.

Компилятор C# также сгенерирует хеш-код на основе всего содержимого сборки (кода CIL, метаданных и т.д.). Как упоминалось в главе 6, *хеш-код* — это числовое значение, которое является статистически уникальным для фиксированных входных данных. Следовательно, в случае изменения любого аспекта сборки .NET (даже одного символа в каком-нибудь строковом литерале) компилятор выдаст другой хеш-код. Затем хеш-код комбинируется с данными секретного ключа, содержащимися внутри файла `*.snk`, для формирования цифровой подписи, встраиваемой в данные заголовка CLR сборки. Процесс создания строгого имени показан на рис. 14.15.

Важно понимать, что действительные данные секретного ключа нигде в манифесте не встречаются, а используются только для цифрового подписания содержимого сборки (в сочетании со сгенерированным хеш-кодом). Общая идея применения открытого и секретного ключей — гарантия того, что никакие две компании, два подразделения или два программиста не будут иметь то же самое удостоверение в мире .NET. По завершении процесса назначения строгого имени сборка может быть установлена в ГАС.

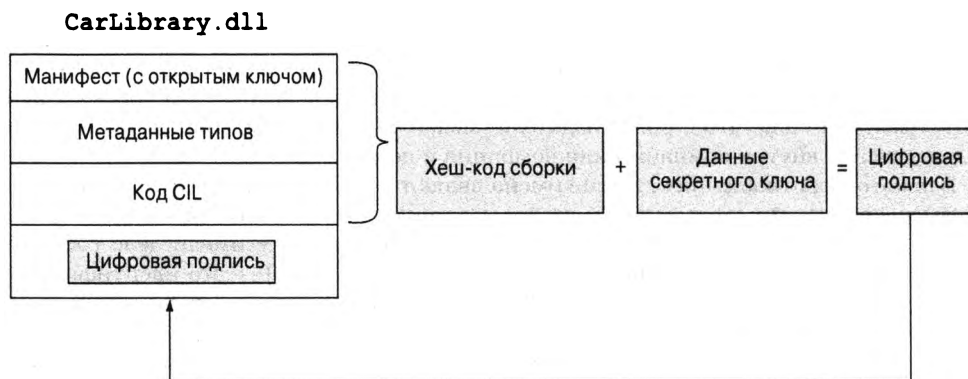


Рис. 14.15. На этапе компиляции генерируется цифровая подпись, основанная на данных открытого и секретного ключей, которая затем вставляется в сборку

На заметку! Строгие имена также обеспечивают уровень защиты от потенциальной подделки злоумышленниками содержимого сборок. С учетом сказанного установившейся практикой .NET считается назначение строгих имен всем сборкам (включая сборки .exe) независимо от того, развертываются они в GAC или нет.

Генерация строгих имен в командной строке

Давайте исследуем процесс назначения строгого имени сборке CarLibrary, созданной ранее в главе. В настоящее время необходимый файл *.snk, скорее всего, будет генерироваться с использованием Visual Studio. Тем не менее, в прошлом (примерно до 2003 года) назначать сборке строгое имя можно было только в командной строке. Посмотрим, как это делается.

В первую очередь необходимо сгенерировать требуемые данные ключей с применением утилиты sn.exe. Хотя инструмент sn.exe поддерживает многочисленные параметры командной строки, нас интересует только флаг -k, который приводит к генерированию нового файла с информацией открытого и секретного ключей.

Создадим на диске C: новый каталог по имени MyTestKeyPair, перейдем в него в окне командной строки Windows и введем следующую команду для генерации файла MyTestKeyPair.snk:

```
sn -k MyTestKeyPair.snk
```

Теперь, имея данные ключей, необходимо проинформировать компилятор C# о том, где находится файл MyTestKeyPair.snk. Как уже упоминалось ранее в главе, при создании любого нового проекта C# в Visual Studio один из первоначальных файлов проекта (отображаемых в узле Properties окна Solution Explorer) имеет имя AssemblyInfo.cs. Он содержит несколько атрибутов, которые описывают саму сборку. Чтобы сообщить компилятору местоположение действительного файла *.snk, в файл AssemblyInfo.cs можно добавить атрибут [AssemblyKeyFile] уровня сборки. Путь просто указывается в виде строкового параметра, например:

```
[assembly: AssemblyKeyFile(@"C:\MyTestKeyPair\MyTestKeyPair.snk")]
```

На заметку! Когда значение атрибута [AssemblyKeyFile] устанавливается вручную, среда Visual Studio выдает предупреждение о том, что нужно либо указать параметр /keyfile для csc.exe, либо установить файл ключей в окне свойств проекта. Мы сделаем это в IDE-среде немного позже, а потому полученное предупреждение можно попросту проигнорировать.

Поскольку версия разделяемой сборки является одним из аспектов строгого имени, выбор номера версии для CarLibrary.dll становится необходимой деталью. В файле AssemblyInfo.cs вы найдете еще один атрибут с именем AssemblyVersion. Первоначально его значение установлено в 1.0.0.0:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Номер версии в .NET состоит из четырех частей (<старший номер>.<младший номер>.<номер сборки>.<номер редакции>). Хотя указание номера версии возлагается полностью на вас, можно заставить Visual Studio автоматически инкрементировать номера сборки и редакции во время каждой компиляции, используя групповой символ. В рассматриваемом примере в этом нет нужды, но взгляните на следующий атрибут:

```
// Формат: <старший номер>.<младший номер>.<номер сборки>.<номер редакции>
// Допустимые значения для каждой части номера версии лежат в диапазоне от 0 до 65535.
[assembly: AssemblyVersion("1.0.*")]
```

Теперь у компилятора C# есть вся информация, необходимая для генерации строгого имени (т.к. значение культуры в атрибуте [AssemblyCulture] не указано, "наследуется" культура, установленная на текущей машине).

Скомпилируем библиотеку кода CarLibrary, откроем ее в ildasm.exe и заглянем в манифест. Теперь можно видеть, что новый маркер .publickey содержит полную информацию об открытом ключе, а маркер .ver хранит номер версии, указанный в атрибуте [AssemblyVersion] (рис. 14.16).

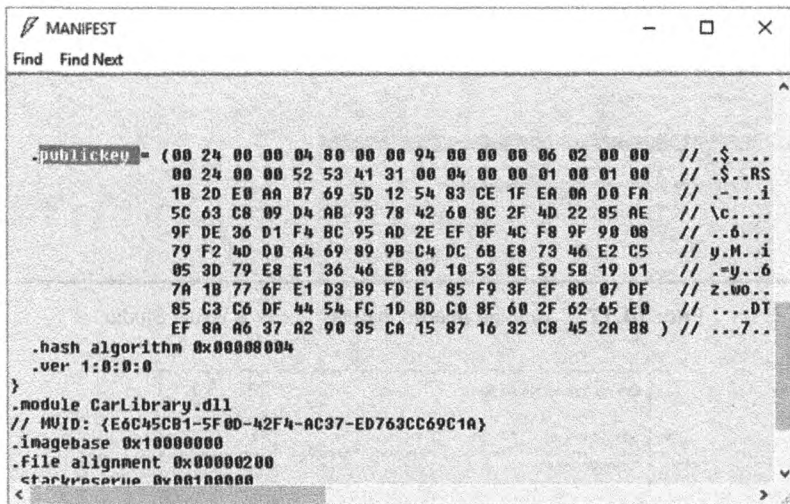


Рис. 14.16. В манифесте строго именованной сборки записана информация об открытом ключе

В данный момент можно было бы развернуть разделяемую сборку CarLibrary.dll в GAC. Однако вспомните, что в настоящее время для создания сборок со строгими именами разработчики приложений .NET могут применять Visual Studio вместо утилиты командной строки sn.exe. Прежде чем взглянуть, как это делается, потребуется удалить (или закомментировать) следующую строку кода в файле AssemblyInfo.cs (предполагая, что она была добавлена вручную):

```
// [assembly: AssemblyKeyFile(@"C:\MyTestKeyPair\MyTestKeyPair.snk")]
```


Генерация строгих имен в Visual Studio

Среда Visual Studio позволяет указывать местоположение существующего файла *.snk в окне свойств проекта, а также генерировать новый файл *.snk. Чтобы создать новый файл *.snk для проекта CarLibrary, дважды щелкните на значке Properties в окне Solution Explorer, в открывшемся окне свойств перейдите на вкладку Signing (Подпись), отметьте флажок Sign the assembly (Подписать сборку) и выберите в раскрывающемся списке вариант <New...> (Новый), как показано на рис. 14.17. Откроется окно с приглашением указать имя для нового файла *.snk (например, myKeyFile.snk) и флажком Protect my key file with a password (Защитить файл ключей с помощью пароля), отмечать который в рассматриваемом примере не требуется (рис. 14.18).

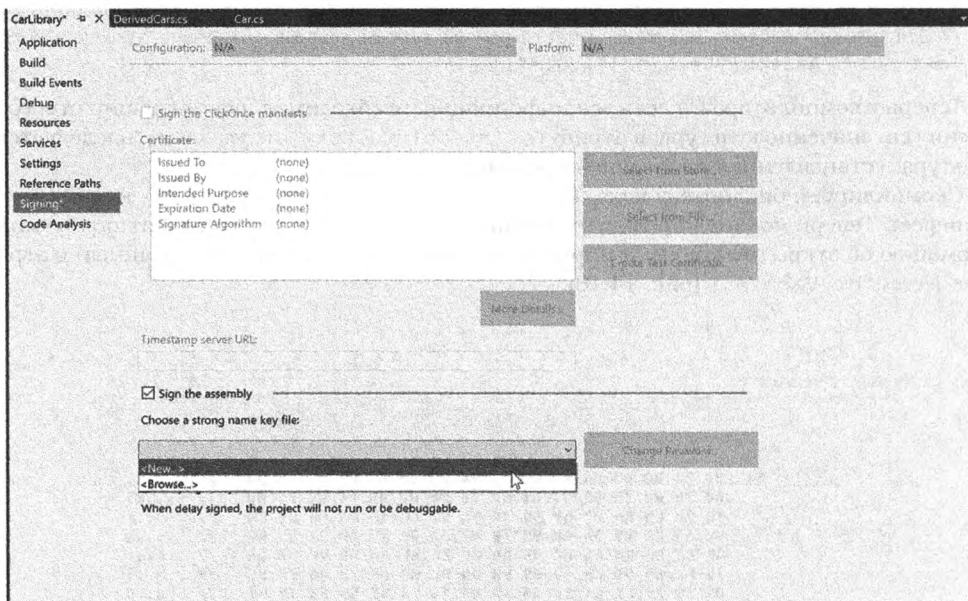


Рис. 14.17. Создание нового файла *.snk в Visual Studio

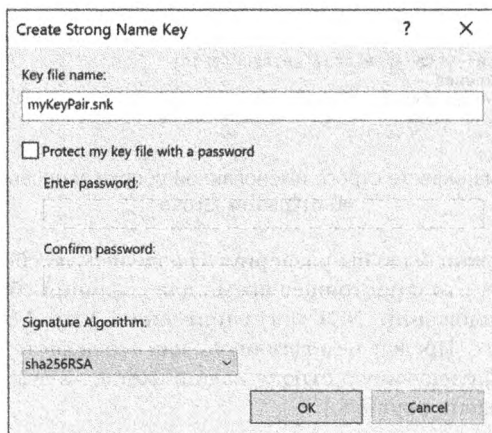


Рис. 14.18. Именованное нового файла *.snk в Visual Studio

Новый файл *.snk появится в окне Solution Explorer (рис. 14.19). Каждый раз, когда приложение компилируется, данные из файла *.snk будут использоваться для назначения сборки надлежащего строгого имени.

На заметку! Вспомните, что на вкладке Application окна свойств проекта имеется кнопка Assembly Information. Щелчок на ней приводит к открытию диалогового окна, которое позволяет устанавливать многочисленные атрибуты уровня сборки, включая номер версии, информацию об авторских правах и т.д.

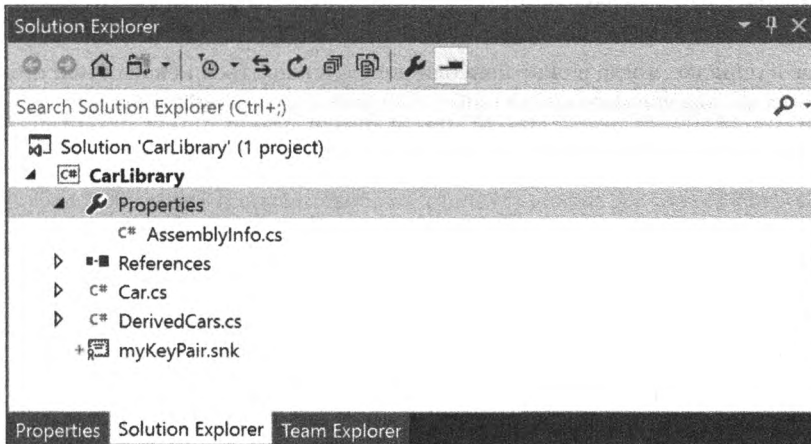


Рис. 14.19. Теперь при каждой компиляции среда Visual Studio будет назначать сборке строгое имя

Установка строго именованных сборок в GAC

Последний шаг заключается в установке (теперь строго именованной) сборки CarLibrary.dll в GAC. Хотя предпочитаемый способ для развертывания сборок в GAC внутри производственной среды предусматривает создание установочного пакета, в составе .NET Framework SDK поставляется инструмент командной строки gacutil.exe, который удобен для проведения быстрых тестов.

На заметку! Для взаимодействия с GAC на своей машине необходимо иметь права администратора. Удостоверьтесь, что открываете окно командной строки от имени учетной записи администратора.

В табл. 14.1 перечислены некоторые наиболее важные параметры gacutil.exe (для вывода полного списка параметров служит флаг /?).

Таблица 14.1. Параметры утилиты gacutil.exe

Параметр	Описание
/i	Устанавливает сборку со строгим именем в GAC
/u	Удаляет сборку из GAC
/l	Отображает список сборок (или конкретную сборку) в GAC

Чтобы установить строго именованную сборку в GAC с помощью `gacutil.exe`, понадобится открыть окно командной строки и перейти в каталог, содержащий файл `CarLibrary.dll`. Вот пример (путь к каталогу у вас может быть другим):

```
cd C:\MyCode\CarLibrary\bin\Debug
```

Затем можно установить библиотеку, используя параметр `/i`:

```
gacutil /i CarLibrary.dll
```

Далее нужно проверить, действительно ли библиотека была развернута, выполнив следующую команду с параметром `/l` (обратите внимание, что расширение файла в случае применения `/l` не указывается):

```
gacutil /l CarLibrary
```

Если все в порядке, тогда в окне консоли должен появиться показанный ниже вывод (как и ожидалось, вы увидите уникальное значение `PublicKeyToken`):

```
The Global Assembly Cache contains the following assemblies:
```

```
CarLibrary, Version=1.0.0.0, Culture=neutral,
```

```
PublicKeyToken=33a2bc294331e8b9, processorArchitecture=MSIL
```

```
Number of items = 1
```

Глобальный кеш сборок содержит следующие сборки:

```
CarLibrary, Version=1.0.0.0, Culture=neutral,
```

```
PublicKeyToken=33a2bc294331e8b9, processorArchitecture=MSIL
```

```
Количество элементов = 1
```

Более того, если вы перейдете в каталог `C:\Windows\Microsoft.NET\assembly\GAC_MSIL`, то обнаружите в нем новый каталог `CarLibrary` с корректной структурой подкаталогов (рис. 14.20).

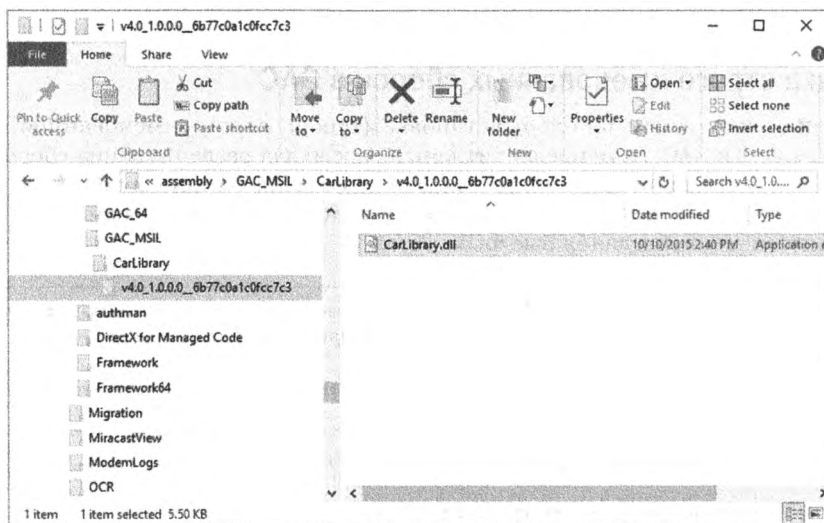


Рис. 14.20. Разделяемая сборка `CarLibrary` в GAC

Потребление разделяемой сборки

При построении приложений, которые используют разделяемую сборку, единственное отличие от случая закрытой сборки связано со способом ссылки на библиотеку в

Visual Studio. На самом деле применяется тот же самый инструмент — диалоговое окно Add Reference.

Когда необходимо сослаться на разделяемую сборку, можно было бы использовать кнопку Browse (Обзор) для перехода в соответствующий подкаталог GAC. Тем не менее, можно также просто перейти к месту хранения строго именованной сборки (такому как каталог bin\Debug проекта библиотеки классов) и сослаться на копию. На рис. 14.21 представлена добавленная ссылка на библиотеку.

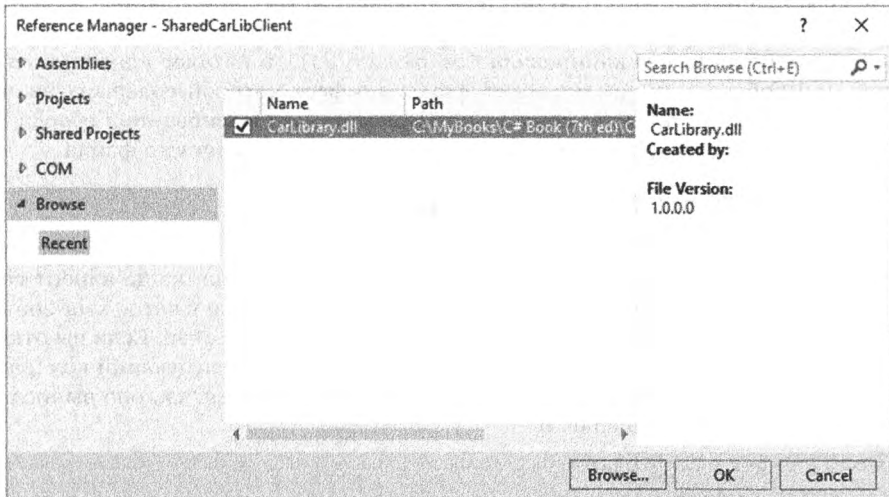


Рис. 14.21. Добавление ссылки на разделяемую сборку CarLibrary (версии 1.0.0.0) в Visual Studio

Хотя среда Visual Studio обнаруживает строго именованную библиотеку, по умолчанию она по-прежнему будет копировать эту библиотеку в выходной каталог клиентского приложения. Чтобы изменить такое поведение, нужно щелкнуть правой кнопкой мыши на ассоциированном файле в окне Reference Manager (Диспетчер ссылок), выбрать в контекстном меню пункт Properties (Свойства) и в открывшемся диалоговом окне изменить свойство Copy Local (Копировать локально) на False.

В целях иллюстрации создадим новый проект консольного приложения по имени SharedCarLibClient и добавим в него ссылку на сборку CarLibrary, как только что было описано. После этого вполне ожидаемо на вкладке References (Ссылки) окна Solution Explorer появится новый значок. Выберем данный значок, откроем окно свойств (через меню View (Вид)) и изменим свойство Copy Local на False. Поместим в клиентское приложение следующий тестовый код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using CarLibrary;

namespace SharedCarLibClient
{
    class Program
    {
        static void Main(string[] args)
```

```

    {
        Console.WriteLine("***** Shared Assembly Client *****");
        SportsCar c = new SportsCar();
        c.TurboBoost();
        Console.ReadLine();
    }
}

```

После компиляции клиентского приложения посредством проводника Windows перейдем в каталог, где хранится файл `SharedCarLibClient.exe`, и удостоверимся в том, что среда Visual Studio не скопировала `CarLibrary.dll` в каталог клиентского приложения. При добавлении ссылки на сборку, манифест которой содержит значение `.publickey`, среда Visual Studio считает, что такая строго именованная сборка будет развернута в GAC, и потому не заботится о копировании ее двоичного файла.

Изучение манифеста SharedCarLibClient

Вспомните, что при генерации строгого имени для сборки в ее манифест записывается полный открытый ключ. В качестве связанного замечания: когда клиент ссылается на строго именованную сборку, в ее манифест помещается сжатое хеш-значение полного открытого ключа, обозначаемое маркером `.publickeytoken`. Если вы откроете манифест `SharedCarLibClient.exe` в `ildasm.exe`, то увидите следующий код (разумеется, значение маркера открытого ключа у вас будет отличаться, т.к. оно вычисляется на основе полного открытого ключа):

```

.assembly extern CarLibrary
{
    .publickeytoken = (33 A2 BC 29 43 31 E8 B9 )
    .ver 1:0:0:0
}

```

Сравнив значение маркера открытого ключа, записанного в манифесте клиента, и значение маркера открытого ключа, отображаемого в GAC, вы обнаружите, что они полностью совпадают. Как упоминалось ранее, открытый ключ представляет один из аспектов удостоверения строго именованной сборки. С учетом этого среда CLR будет загружать только версию 1.0.0.0 сборки по имени `CarLibrary`, имеющую открытый ключ, хеширование которого дает значение `33A2BC294331E8B9`. Если среде CLR не удастся найти сборку, удовлетворяющую такому описанию, в GAC (и закрытую сборку по имени `CarLibrary` в каталоге клиента), тогда она сгенерирует исключение `FileNotFoundException`.

Исходный код. Проект `SharedCarLibClient` доступен в подкаталоге `Chapter_14`.

Конфигурирование разделяемых сборок

Подобно закрытым сборкам разделяемые сборки можно конфигурировать с применением клиентского файла `*.config`. Естественно, поскольку разделяемые сборки развертываются в хорошо известном месте (GAC), элемент `<privatePath>` для них не используется, как это делалось для закрытых сборок (хотя если клиент работает и с разделяемыми, и с закрытыми сборками, то элемент `<privatePath>` может присутствовать в файле `*.config`).

Конфигурационные файлы приложения в сочетании с разделяемыми сборками могут применяться, когда необходимо заставить среду CLR привязаться к *другой* версии

заданной сборки, пропуская значение, которое записано в манифесте клиента. Так поступать удобно по нескольким причинам. Например, представьте, что вы поставили версию 1.0.0.0 сборки, но со временем в ней был выявлен крупный дефект. Одним из корректирующих действий могла бы быть перекомпиляция клиентского приложения для ссылки на версию сборки с устраненным дефектом (скажем, 1.1.0.0) и распространение обновленного приложения и новой сборки на все целевые машины.

Другой вариант предусматривает поставку новой библиотеки кода и файла *.config, который автоматически инструктирует исполняющую среду о необходимости привязки к новой (свободной от дефектов) версии. При условии, что новая версия установлена в ГАС, первоначальное клиентское приложение будет функционировать без перекомпиляции или повторного распространения.

Рассмотрим еще один пример. Предположим, что была поставлена первая версия свободной от дефектов сборки (1.0.0.0), а через пару месяцев вы добавили в сборку новую функциональность, получив версию 2.0.0.0. Очевидно, что существующие клиентские приложения, которые компилировались со сборкой версии 1.0.0.0, не имеют никакого понятия о появившихся новых типах, т.к. их кодовая база не содержит ссылки на них.

Однако новым клиентским приложениям требуется ссылка на новую функциональность в версии 2.0.0.0. В .NET вы можете поставить на целевые машины сборку версии 2.0.0.0 и обеспечить ее работу бок о бок со сборкой версии 1.0.0.0. При необходимости существующие клиенты могут динамически перенаправляться для загрузки версии 2.0.0.0 (чтобы получить доступ к улучшениям реализации) с использованием конфигурационного файла, не требуя повторной компиляции и развертывания клиентского приложения.

Замораживание текущей версии разделяемой сборки

В целях иллюстрации динамической привязки к конкретной версии разделяемой сборки откроем окно проводника Windows и скопируем текущую версию скомпилированной сборки CarLibrary.dll (1.0.0.0) в другой подкаталог (например, CarLibrary Version 1.0.0.0), чтобы символизировать замораживание этой версии (рис. 14.22).

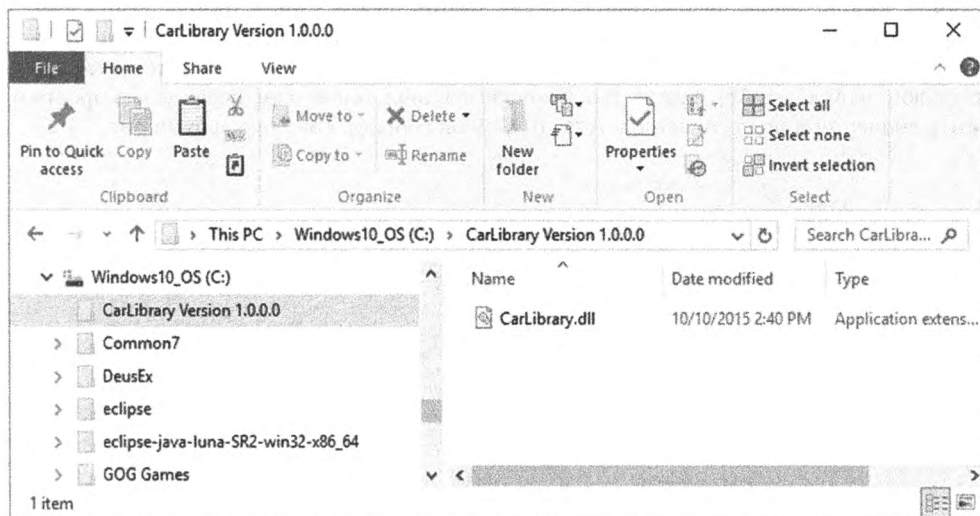


Рис. 14.22. Замораживание текущей версии сборки CarLibrary.dll

Построение разделяемой сборки версии 2.0.0.0

Теперь откроем существующий проект `CarLibrary` и модифицируем кодовую базу, добавив новый тип `enum` по имени `MusicMedia`, который определяет четыре возможных музыкальных устройства:

```
// Тип музыкального устройства, установленного в автомобиле.
public enum MusicMedia
{
    musicCd,
    musicTape,
    musicRadio,
    musicMp3
}
```

Кроме того, добавим в тип `Car` новый открытый метод, который позволяет вызывающему коду включать один из заданных музыкальных проигрывателей (при необходимости импортировав пространство имен `System.Windows.Forms`):

```
public abstract class Car
{
    ...
    public void TurnOnRadio(bool musicOn, MusicMedia mm)
        => MessageBox.Show(musicOn ? $"Jamming {mm}" : "Quiet time...");
}
```

Изменим код конструкторов класса `Car` так, чтобы они отображали окно `MessageBox` с сообщением, подтверждающим применение версии 2.0.0.0 сборки `CarLibrary`:

```
public abstract class Car
{
    ...
    public Car() => MessageBox.Show("CarLibrary Version 2.0!");
    public Car(string name, int maxSp, int currSp)
    {
        MessageBox.Show("CarLibrary Version 2.0!");
        PetName = name; MaxSpeed = maxSp; CurrentSpeed = currSp;
    }
    ...
}
```

Последнее, но не менее важное: перед перекомпиляцией новой библиотеки изменим номер версии с 1.0.0.0 на 2.0.0.0. Вспомните, что это можно делать визуально, дважды щелкнув на значке `Properties` в окне `Solution Explorer` и затем щелкнув на кнопке `Assembly Information` на вкладке `Application`. В открывшемся диалоговом окне необходимо просто изменить значения в полях `Assembly version` (Версия сборки), как показано на рис. 14.23.

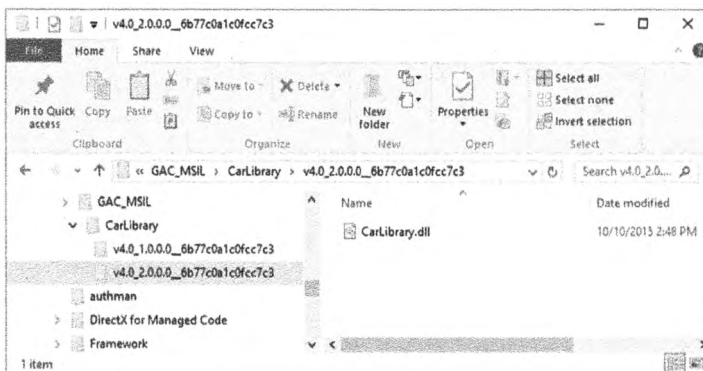


Рис. 14.23. Установка номера версии сборки `CarLibrary.dll` в 2.0.0.0

Заглянув в каталог `bin\Debug` проекта, вы увидите, что в нем появилась сборка новой версии (2.0.0.0), в то время как сборка версии 1.0.0.0 благополучно хранится в подкаталоге `CarLibrary Version 1.0.0.0`. Давайте установим сборку новой версии в GAC для .NET 4.0 с помощью утилиты `gacutil.exe`, как было описано ранее в главе. Обратите внимание, что теперь на машине имеются две версии той же самой сборки (рис. 14.24).

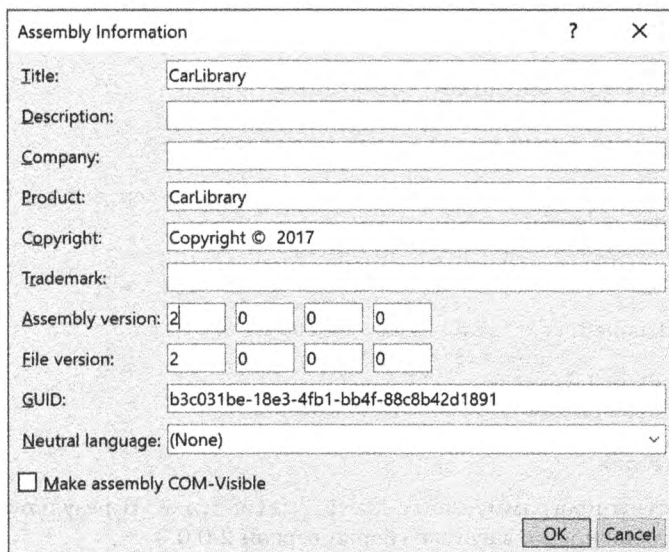


Рис. 14.24. Выполнение бок о бок разделяемой сборки

После запуска приложения `SharedCarLibClient.exe` окно с сообщением "CarLibrary Version 2.0!" не отображается, потому что в манифесте явным образом запрашивается версия 1.0.0.0. Как же тогда указать среде CLR о необходимости привязки к версии 2.0.0.0? Ответ на этот вопрос ищите ниже.

На заметку! При компиляции приложений среда Visual Studio будет автоматически сбрасывать ссылки! Следовательно, если вы запускаете приложение `SharedCarLibClient.exe` внутри Visual Studio, она захватит сборку `CarLibrary.dll` версии 2.0.0.0! Если вы случайно запустили приложение подобным образом, то просто удалите текущую ссылку на `CarLibrary.dll` и выберите сборку версии 1.0.0.0 (которая была помещена в подкаталог `CarLibrary Version 1.0.0.0`).

Динамическое перенаправление на специфичные версии разделяемой сборки

Когда среде CLR нужно сообщить о загрузке разделяемой сборки с версией, отличающейся от указанной в манифесте, можно создать файл `*.config`, который содержит элемент `<dependentAssembly>`. Потребуется создать подэлемент `<assemblyIdentity>` с дружественным именем сборки, которое указано в манифесте клиента (`CarLibrary` в рассматриваемом примере), и необязательным атрибутом культуры (которому можно присвоить пустую строку или вообще опустить, если должна использоваться стандартная культура машины). Кроме того, элемент `<dependentAssembly>` будет содержать подэлемент `<bindingRedirect>` для определения версии, в текущий момент заданной в манифесте (атрибут `oldVersion`), и версии, которая должна загружаться вместо нее из GAC (атрибут `newVersion`).

Модифицируем конфигурационный файл `SharedCarLibClient.exe.config`, находящийся в каталоге приложения `SharedCarLibClient`, как показано ниже.

На заметку! Значение вашего маркера открытого ключа будет отличаться от того, который вы видите в приведенной далее разметке. Чтобы найти маркер открытого ключа, откройте клиентскую сборку в `ildasm.exe`, дважды щелкните на значке **MANIFEST** и скопируйте нужное значение в буфер (не забудьте удалить пробелы).

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <!-- Информация о привязке во время выполнения -->
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="CarLibrary"
                          publicKeyToken="64ee9364749d8328"
                          culture="neutral"/>
        <bindingRedirect oldVersion= "1.0.0.0"
                          newVersion= "2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Теперь запустим программу `SharedCarLibClient.exe`. В результате должно появиться окно с сообщением о загрузке сборки версии 2.0.0.0.

В конфигурационном файле клиента могут присутствовать многочисленные элементы `<dependentAssembly>`. Хотя в текущем примере в этом нет никакой необходимости, предположим, что манифест `SharedCarLibClient.exe` также ссылается на сборку `MathLibrary` версии 2.5.0.0. Если нужно перенаправить на сборку `MathLibrary` версии 3.0.0.0 (в дополнение к перенаправлению на сборку `CarLibrary` версии 2.0.0.0), то содержимое конфигурационного файла `SharedCarLibClient.exe.config` могло бы выглядеть следующим образом:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <!-- Управляет привязкой к CarLibrary -->
      <dependentAssembly>
        <assemblyIdentity name="CarLibrary"
                          publicKeyToken="64ee9364749d8328"
                          culture=""/>
        <bindingRedirect oldVersion= "1.0.0.0" newVersion= "2.0.0.0"/>
      </dependentAssembly>
      <!-- Управляет привязкой к MathLibrary -->
      <dependentAssembly>
        <assemblyIdentity name="MathLibrary"
                          publicKeyToken="64ee9364749d8328"
                          culture=""/>
        <bindingRedirect oldVersion= "2.5.0.0" newVersion= "3.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

На заметку! В атрибуте `oldVersion` допускается указывать диапазон номеров старых версий; например, `<bindingRedirect oldVersion="1.0.0.0-1.2.0.0" newVersion="2.0.0.0"/>` информирует среду CLR о том, что вместо любой старой версии из диапазона от 1.0.0.0 до 1.2.0.0 должна применяться версия 2.0.0.0.

Понятие сборок политик издателя

Рассмотрим еще один аспект, касающийся конфигурации — *сборки политик издателя*. Как объяснялось ранее, с помощью файлов `*.config` можно привязываться к специфической версии разделяемой сборки, обходя тем самым версию, которая записана в манифесте клиента. Но предположим, что вам как администратору требуется переконфигурировать все клиентские приложения на заданной машине с целью привязки к версии 2.0.0.0 сборки `CarLibrary.dll`. Учитывая строгое соглашение об именовании конфигурационных файлов, вам придется дублировать одно и то же XML-содержимое во множестве мест (при условии, что вы действительно знаете местоположение исполняемых файлов, работающих с `CarLibrary`). Понятно, что все может вылиться в настоящий кошмар сопровождения.

Политики издателя позволяют издателю конкретной сборки (в роли которого может выступать программист, подразделение или целая компания) поставлять двоичную версию файла `*.config`, которая устанавливается в GAC вместе с более новой версией связанной с ним сборки. Преимущество такого подхода заключается в том, что каталоги клиентских приложений не нуждаются в наличии специфических файлов `*.config`. Взамен среда CLR будет считывать текущий манифест и пытаться найти сборку запрашиваемой версии в GAC. Тем не менее, если среда CLR обнаружит сборку политик издателя, тогда она прочтает содержащиеся в ней XML-данные и выполнит запрашиваемое перенаправление на уровне GAC.

Сборки политик издателя создаются в командной строке с использованием утилиты .NET под названием `al.exe` (редактор связей сборки). Данный инструмент поддерживает множество параметров, но построение сборки политик издателя требует передачи только нескольких из них:

- местоположение файла `*.config` или `*.xml`, содержащего инструкции перенаправления;
- имя результирующей сборки политик издателя;
- местоположение файла `*.snk`, применяемого для подписания сборки политик издателя;
- номера версии для назначения создаваемой сборке политик издателя.

Чтобы построить сборку политик издателя, которая управляет библиотекой `CarLibrary.dll`, выполните следующую команду (она должна вводиться в одной строке):

```
al /link:CarLibraryPolicy.xml /out:policy.1.0.CarLibrary.dll
/keyf:C:\MyKey\myKey.snk /v:1.0.0.0
```

Здесь указано, что необходимое XML-содержимое находится в файле по имени `CarLibraryPolicy.xml`. Имя выходного файла (которое должно быть представлено в формате `policy.<старший номер>.<младший номер>.имяКонфигурируемойСборки`) задано с помощью флага `/out`. Кроме того, обратите внимание, что посредством флага `/keyf` также должно быть указано имя файла, содержащего пару открытого и секретного ключей. Не забывайте, что файлы политик издателя являются разделяемыми и потому должны иметь строгие имена!

В результате запуска `al.exe` создается новая сборка, которая может быть помещена в GAC, чтобы заставить всех клиентов привязываться к версии 2.0.0.0 сборки `CarLibrary.dll` без использования специального конфигурационного файла для каждого клиентского приложения. При таком подходе можно спроектировать перенаправление в масштабах машины для всех приложений, работающих с конкретной версией (или диапазоном версий) существующей сборки.

Отключение политики издателя

А теперь предположим, что вы (будучи системным администратором) развернули сборку политик издателя (и последнюю версию связанной сборки) в GAC на клиентской машине. Как обычно бывает, девять из десяти задействованных приложений привязались к версии 2.0.0.0 без ошибок, но одно (по ряду причин) при получении доступа к `CarLibrary.dll` версии 2.0.0.0 терпит неудачу. (Как все мы знаем, практически невозможно создать программное обеспечение с обратной совместимостью, которое функционировало бы корректно в любых ситуациях.)

В таком случае для проблемного клиента можно создать конфигурационный файл, который укажет среде CLR о необходимости *игнорировать* любые файлы политик издателя, установленные в GAC. Остальные клиентские приложения, успешно работающие с новой версией сборки .NET, будут просто перенаправляться посредством установленной сборки политик издателя. Чтобы отключить политику издателя на уровне отдельного клиента, потребуется создать файл `*.config` (с подходящим именем), добавить в него элемент `<publisherPolicy>` и установить атрибут `apply` в `no`. После этого среда CLR будет загружать сборку той версии, которая была изначально указана в манифесте клиента:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <publisherPolicy apply="no" />
    </assemblyBinding>
  </runtime>
</configuration>
```

Элемент `<codeBase>`

В конфигурационных файлах приложений можно также указывать *кодовые базы*. Элемент `<codeBase>` позволяет инструктировать среду CLR о необходимости зондирования зависимых сборок, находящихся в произвольных местоположениях (таких как сетевые конечные точки или любые пути на машине за пределами каталога клиентского приложения).

Если в `<codeBase>` указано местоположение на удаленной машине, тогда сборка будет загружаться по требованию в специальный каталог внутри GAC, который называется *кешем загрузки*. С учетом того, что вы уже знаете о развертывании сборок в GAC, должно быть ясно, что сборки, загружаемые из указанного в `<codeBase>` места, нуждаются в строгих именах (иначе среда CLR не смогла бы их устанавливать в GAC). Просмотреть содержимое кеша загрузки на своей машине можно, запустив утилиту `gacutil.exe` с параметром `/ldl`:

```
gacutil /ldl
```

На заметку! Говоря формально, элемент `<codeBase>` может применяться для зондирования сборки, не обладающих строгими именами. Однако в таком случае местоположение сборки должно быть относительным к каталогу клиентского приложения (и элемент `<codeBase>` становится просто альтернативой `<privatePath>`).

Чтобы посмотреть на элемент `<codeBase>` в действии, создадим новый проект консольного приложения по имени `CodeBaseClient`, установим ссылку на сборку `CarLibrary.dll` версии 2.0.0.0 и модифицируем начальный файл кода следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using CarLibrary;
namespace CodeBaseClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Fun with CodeBases *****");
            SportsCar c = new SportsCar();
            Console.WriteLine("Sports car has been allocated.");
            Console.ReadLine();
        }
    }
}
```

Поскольку сборка `CarLibrary.dll` была развернута в GAC, программу можно запускать в том виде, как есть. Тем не менее, для иллюстрации использования элемента `<codeBase>` создадим на диске C: новый каталог (скажем, `C:\MyAsms`) и скопируем в него сборку `CarLibrary.dll` версии 2.0.0.0.

Теперь добавим в проект `CodeBaseClient` файл `App.config` (или отредактируем существующий), как объяснялось ранее в главе, и поместим в него приведенное ниже XML-содержимое (не забывайте, что ваше значение `.publickeytoken` будет отличаться; при необходимости просмотрите его в GAC):

```
<configuration>
...
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <dependentAssembly>
      <assemblyIdentity name="CarLibrary" publicKeyToken="33A2BC294331E8B9"
        culture="neutral"/>
      <codeBase version="2.0.0.0" href="file:///C:/MyAsms/CarLibrary.dll" />
    </dependentAssembly>
  </assemblyBinding>
</runtime>
</configuration>
```

Как видите, элемент `<codeBase>` вложен внутрь элемента `<assemblyIdentity>`, в атрибутах `name` и `publicKeyToken` которого указано дружественное имя сборки и ассоциированное значение маркера открытого ключа. В самом элементе `<codeBase>` задается версия и местоположение (в свойстве `href`) сборки, которая должна загружаться. Если удалить версию 2.0.0.0 сборки `CarLibrary.dll` из GAC, то клиентское приложение по-прежнему будет успешно функционировать, т.к. среда CLR способна найти необходимую внешнюю сборку в каталоге `C:\MyAsms`.

На заметку! Размещая сборки в произвольных местах на машине разработки, вы по существу воссоздаете системный реестр (и связанный с ним “ад DLL”), если учесть, что перемещение или переименование каталога, содержащего двоичные файлы, приведет к нарушению текущей привязки. Следовательно, элемент `<codeBase>` нужно применять осмотрительно.

Элемент `<codeBase>` может также быть удобным при ссылке на сборки, находящиеся на удаленной машине в сети. Предположим, что у вас есть права доступа к каталогу, расположенному на `http://www.MySite.com`. Для помещения удаленного файла `*.dll` в кеш загрузки GAC на локальной машине можно использовать следующий элемент `<codeBase>`:

```
<codeBase version="2.0.0.0"
  href="http://www.MySite.com/Assemblies/CarLibrary.dll" />
```

Исходный код. Проект `CodeBaseClient` доступен в подкаталоге `Chapter_14`.

Пространство имен `System.Configuration`

До сих пор во всех показанных в главе файлах `*.config` применялись хорошо известные XML-элементы, которые среда CLR считывала для выяснения местоположений внешних сборок. В дополнение к этим распознаваемым элементам конфигурационный файл клиента может содержать специфичные для приложения данные, которые не имеют ничего общего с механизмом привязки. Таким образом, не должен вызывать удивления тот факт, что платформа .NET Framework предоставляет пространство имен, которое позволяет программно читать данные из конфигурационного файла клиента.

Пространство имен `System.Configuration` предлагает небольшой набор типов, которые можно использовать для чтения специальных данных из файла `*.config` клиента. Специальные настройки должны помещаться внутрь элемента `<appSettings>`. Элемент `<appSettings>` может содержать любое количество элементов `<add>`, которые определяют пары “ключ-значение”, предназначенные для получения программным образом.

В качестве примера предположим, что имеется файл `*.config`, относящийся к проекту консольного приложения `AppConfigReaderApp`, в котором определены два значения, специфичные для приложения:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7" />
  </startup>
  <'-- Специальные настройки приложения -->
  <appSettings>
    <add key="TextColor" value="Green" />
    <add key="RepeatCount" value="8" />
  </appSettings>
</configuration>
```

Чтение этих значений для работы с ними в клиентском приложении сводится просто к вызову метода `GetValue()` уровня экземпляра типа `System.Configuration.AppSettingsReader`. Следующий код демонстрирует, что первым параметром `GetValue()` является имя ключа в файле `*.config`, а вторым — тип ключа (получаемый посредством операции `typeof`):

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Configuration;

namespace AppConfigReaderApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Reading <appSettings> Data *****\n");

            // Извлечь специальные данные из файла *.config.
            AppSettingsReader ar = new AppSettingsReader();
            int numbOfTimes = (int)ar.GetValue("RepeatCount", typeof(int));
            string textColor = (string)ar.GetValue("TextColor", typeof(string));

            Console.ForegroundColor =
                (ConsoleColor)Enum.Parse(typeof(ConsoleColor), textColor);

            // Вывести сообщение нужное количество раз.
            for (int i = 0; i < numbOfTimes; i++)
                Console.WriteLine("Howdy!");
            Console.ReadLine();
        }
    }
}

```

Исходный код. Проект AppConfigReaderApp доступен в подкаталоге Chapter_14.

Документация по схеме конфигурационного файла

В текущей главе вы узнали о роли конфигурационных XML-файлов. Основное внимание здесь было сосредоточено на нескольких настройках, которые можно добавлять к элементу `<runtime>` для управления тем, как среда CLR будет искать требуемые внешние библиотеки. По мере дальнейшего чтения книги (и после перехода к построению крупномасштабного программного обеспечения) вы быстро заметите, что XML-файлы конфигурации применяются повсеместно.

Действительно, в рамках платформы .NET файлы `*.config` используются в многочисленных API-интерфейсах. Например, в главе 23 вы увидите, что в инфраструктуре Windows Communication Foundation (WCF) конфигурационные файлы применяются для установки сложных настроек сети. Позже в книге, когда будет обсуждаться разработка веб-приложений с помощью ASP.NET, вы узнаете, что файл `web.config` содержит инструкции того же типа, что и файл `App.config` для настольных приложений.

Поскольку конфигурационный файл .NET может содержать большое количество инструкций, вы должны знать, что полная схема этого XML-файла документирована в Интернете. Выполните поиск по ключевым словам *Схема файлов конфигурации для .NET Framework* или перейдите по ссылке <https://docs.microsoft.com/ru-ru/dotnet/framework/configure-apps/file-schema/>.

Резюме

В главе была исследована роль библиотек классов .NET (файлов *.dll для .NET). Вы видели, что библиотеки классов представляют собой двоичные файлы .NET, содержащие логику, которая предназначена для многократного использования в разнообразных проектах. Вспомните, что библиотеки могут быть развернуты двумя основными путями — как закрытые или как разделяемые. Закрытые сборки развертываются в каталоге клиента или в его подкаталоге при условии, что имеется подходящий конфигурационный XML-файл. Разделяемые сборки являются библиотеками, которые могут потребляться любым приложением на машине, и на них также можно оказывать влияние посредством настроек в конфигурационном файле клиентской стороны.

Вы узнали, как назначать разделяемым сборками строгие имена, с помощью которых устанавливается уникальное удостоверение библиотек с точки зрения среды CLR. Кроме того, вы ознакомились с различными инструментами командной строки (`sn.exe` и `gacutil.exe`), которые применяются во время разработки и развертывания разделяемых библиотек.

Глава была завершена рассмотрением роли политик издателя и процесса сохранения и извлечения специальных настроек с использованием пространства имен `System.Configuration`.

ГЛАВА 15

Рефлексия типов, позднее связывание и программирование на основе атрибутов

Как было показано в главе 14, сборки являются базовой единицей развертывания в мире .NET. Используя интегрированный браузер объектов Visual Studio (и многих других IDE-сред), можно просматривать типы внутри набора сборок, на которые ссылается проект. Кроме того, внешние инструменты, такие как утилита `ildasm.exe`, позволяют заглядывать внутрь лежащего в основе кода CIL, метаданных типов и манифеста сборки для заданного двоичного файла .NET. В дополнение к подобному исследованию сборок .NET на этапе проектирования ту же самую информацию можно получить *программно* с применением пространства имен `System.Reflection`. Таким образом, первой задачей настоящей главы является определение роли рефлексии и потребности в метаданных .NET.

Остаток главы посвящен нескольким тесно связанным темам, которые вращаются вокруг служб рефлексии. Например, вы узнаете, как клиент .NET может задействовать динамическую загрузку и позднее связывание для активизации типов, о которых нет никаких сведений на этапе компиляции. Вы также научитесь вставлять специальные метаданные в сборки .NET за счет использования системных и специальных атрибутов. Для практической демонстрации всех этих аспектов в завершение главы приводится пример построения нескольких “объектов-оснасток”, которые можно подключать к расширяемому настольному приложению с графическим пользовательским интерфейсом.

Потребность в метаданных типов

Возможность полного описания типов (классов, интерфейсов, структур, перечислений и делегатов) с помощью метаданных является ключевым элементом платформы .NET. Многочисленным технологиям .NET, таким как Windows Communication Foundation (WCF) и сериализация объектов, требуется способность выяснения формата типов во время выполнения. Кроме того, межъязыковое взаимодействие, многие службы компилятора и средства IntelliSense в IDE-среде опираются на конкретное описание *типа*.

Вспомните из главы 1, что утилита `ildasm.exe` позволяет просматривать метаданные типов сборки по нажатию комбинации клавиш <Ctrl+M>. Таким образом, если вы

откроете в `ildasm.exe` любую из сборок `*.dll` или `*.exe`, которые создавались ранее в книге (например, `CarLibrary.dll` из главы 14), и нажмете <Ctrl+M>, то увидите метаданные типов (рис. 15.1).

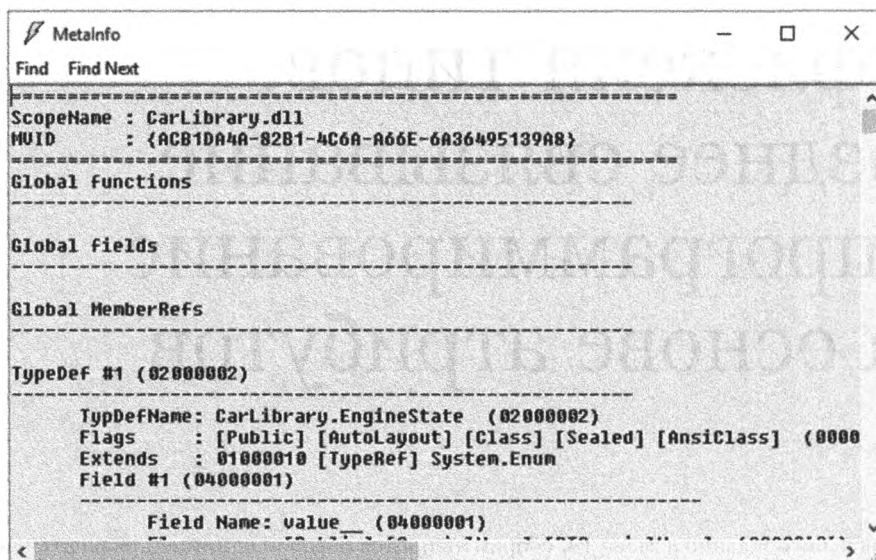


Рис. 15.1. Просмотр метаданных сборки с помощью утилиты `ildasm.exe`

Как видите, утилита `ildasm.exe` отображает метаданные типов .NET очень подробно (фактический двоичный формат гораздо компактнее). В действительности описание всех метаданных сборки `CarLibrary.dll` заняло бы несколько страниц. Однако для понимания вполне достаточно кратко взглянуть на некоторые ключевые описания метаданных сборки `CarLibrary.dll`.

На заметку! Не стоит слишком глубоко вникать в синтаксис каждого фрагмента метаданных .NET, приводимого в нескольких последующих разделах. Важно усвоить, что метаданные .NET являются исключительно дескриптивными и учитывают каждый внутренне определенный (и внешне ссылаемый) тип, который найден в заданной кодовой базе.

Просмотр (частичных) метаданных для перечисления `EngineState`

Каждый тип, определенный внутри текущей сборки, документируется с применением маркера `TypeDef #n` (где `TypeDef` — сокращение от *type definition* (определение типа)). Если описываемый тип использует какой-то тип, определенный в отдельной сборке .NET, тогда ссылаемый тип документируется с помощью маркера `TypeRef #n` (где `TypeRef` — сокращение от *type reference* (ссылка на тип)). Если хотите, то можете считать, что маркер `TypeRef` является указателем на полное определение метаданных ссылаемого типа во внешней сборке. Коротко говоря, метаданные .NET — это набор таблиц, явно помечающих все определения типов (`TypeDef`) и ссылаемые типы (`TypeRef`), которые могут быть просмотрены в окне метаданных утилиты `ildasm.exe`.

В случае сборки `CarLibrary.dll` один из маркеров `TypeDef` представляет описание метаданных перечисления `CarLibrary.EngineState` (номер `TypeDef` у вас может отли-

ваться; нумерация `TypeDef` основана на порядке, в котором компилятор C# обрабатывает файл):

```

TypeDef #2 (02000003)
-----
  TypDefName: CarLibrary.EngineState (02000003)
  Flags      : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass] (00000101)
  Extends    : 01000001 [TypeRef] System.Enum
  Field #1 (04000006)
  -----
    Field Name: value__ (04000006)
    Flags      : [Public] [SpecialName] [RTSpecialName] (00000606)
    CallCnvtn: [FIELD]
    Field type: I4
  Field #2 (04000007)
  -----
    Field Name: engineAlive (04000007)
    Flags      : [Public] [Static] [Literal] [HasDefault] (00008056)
    DefltValue: (I4) 0
    CallCnvtn: [FIELD]
    Field type: ValueClass CarLibrary.EngineState
  ...

```

Маркер `TypDefName` здесь служит для установления имени заданного типа, которым в рассматриваемом случае является специальное перечисление `CarLibrary.EngineState`. Маркер метаданных `Extends` применяется при документировании базового типа для заданного типа .NET (ссылаемого типа `System.Enum` в этом случае). Каждое поле перечисления помечается с использованием маркера `Field #n`. Ради краткости выше были приведены только метаданные для поля `CarLibrary.EngineState.engineAlive`.

Просмотр (частичных) метаданных для типа `Car`

Ниже показана часть метаданных класса `Car`, которая иллюстрирует следующие аспекты:

- как поля определены в терминах метаданных .NET;
- как методы документированы посредством метаданных .NET;
- как автоматическое свойство представлено в метаданных .NET.

```

TypeDef #3 (02000004)
-----
  TypDefName: CarLibrary.Car (02000004)
  Flags      : [Public] [AutoLayout] [Class] [Abstract]
               [AnsiClass] [BeforeFieldInit] (00100081)
  Extends    : 01000002 [TypeRef] System.Object
  ...
  Field #2 (0400000a)
  -----
    Field Name: <PetName>k__BackingField (0400000a)
    Flags      : [Private] (00000001)
    CallCnvtn: [FIELD]
    Field type: String
  ...

```

Method #1 (06000001)

```

MethodName: get_PetName (06000001)
Flags      : [Public] [HideBySig] [ReuseSlot] [SpecialName] (00000886)
RVA        : 0x000020d0
ImplFlags  : [IL] [Managed] (00000000)
CallCnvtn : [DEFAULT]
hasThis
ReturnType: String
No arguments.

```

...

Method #2 (06000002)

```

MethodName: set_PetName (06000002)
Flags      : [Public] [HideBySig] [ReuseSlot] [SpecialName] (00000886)
RVA        : 0x000020e7
ImplFlags  : [IL] [Managed] (00000000)
CallCnvtn : [DEFAULT]
hasThis
ReturnType: Void
1 Arguments
  Argument #1: String
1 Parameters
  (1) ParamToken : (08000001) Name : value flags: [none] (00000000)

```

...

Property #1 (17000001)

```

Prop.Name : PetName (17000001)
Flags     : [none] (00000000)
CallCnvtn : [PROPERTY]
hasThis
ReturnType: String
No arguments.
DefltValue:
Setter : (06000002) set_PetName
Getter : (06000001) get_PetName
0 Others

```

...

Прежде всего, метаданные класса `Car` указывают базовый класс этого типа (`System.Object`) и включают разнообразные флаги, которые описывают то, как тип был сконструирован (например, `[Public]`, `[Abstract]` и т.п.). Описания методов (вроде конструктора `Car`) содержат имя, возвращаемое значение и параметры.

Обратите внимание, что автоматическое свойство дает в результате сгенерированное компилятором закрытое поддерживающее поле (по имени `<PetName>k__BackingField`) и два сгенерированных компилятором метода (в случае свойства для чтения и записи) с именами `get_PetName()` и `set_PetName()`. Наконец, само свойство отображается на внутренние методы получения/установки с применением маркеров `Setter` и `Getter` метаданных .NET.

Исследование блока `TypeRef`

Вспомните, что метаданные сборки будут описывать не только набор внутренних типов (`Car`, `EngineState` и т.д.), но также любые внешние типы, на которые ссылаются

внутренние типы. Например, с учетом того, что в сборке `CarLibrary.dll` определены два перечисления, метаданные типа `System.Enum` будут содержать следующий блок `TypeRef`:

```
TypeRef #1 (01000001)
-----
Token:           0x01000001
ResolutionScope: 0x23000001
TypeRefName:     System.Enum
```

Документирование определяемой сборки

Окно метаданных `ildasm.exe` также позволяет просматривать метаданные .NET, которые описывают саму сборку с использованием маркера `Assembly`. Как показано в приведенном далее (неполном) листинге, информация, документируемая внутри таблицы `Assembly`, совпадает с той, которую можно просматривать по щелчку на значке **MANIFEST** (Манифест). Ниже представлена часть метаданных манифеста сборки `CarLibrary.dll` (версии 2.0.0.0):

```
Assembly
-----
Token: 0x20000001
Name : CarLibrary
Public Key : 00 24 00 00 04 80 00 00 // Etc...

Hash Algorithm : 0x00008004
Major Version: 0x00000002
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
Flags : [PublicKey] ...
```

Документирование ссылаемых сборок

В дополнение к маркеру `Assembly` и набору блоков `TypeDef` и `TypeRef` в метаданных .NET также применяются маркеры `AssemblyRef #n` для документирования каждой внешней сборки. Поскольку в сборке `CarLibrary.dll` используется класс `System.Windows.Forms.MessageBox`, вы обнаружите следующий блок `AssemblyRef` для сборки `System.Windows.Forms`:

```
AssemblyRef #2 (23000002)
-----
Token: 0x23000002
Public Key or Token: b7 7a 5c 56 19 34 e0 89
Name: System.Windows.Forms
Version: 4.0.0.0
Major Version: 0x00000004
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
HashValue Blob:
Flags: [none] (00000000)
```

Документирование строковых литералов

Последний полезный аспект, относящийся к метаданным .NET, связан с тем, что все строковые литералы в кодовой базе документируются внутри маркера User Strings:

```
User Strings
-----
70000001 : (11) L"Jamming {0}"
70000019 : (13) L"Quiet time..."
70000035 : (23) L"CarLibrary Version 2.0!"
70000065 : (14) L"Ramming speed!"
70000083 : (19) L"Faster is better..."
700000ab : ( 4) L"Eek!"
700000cd : (27) L"Your engine block exploded!"
```

На заметку! Как демонстрирует представленный выше листинг метаданных, всегда помните о том, что все строки ясно документируются в метаданных сборки. Это может привести к крупным последствиям в плане безопасности, если вы применяете строковые литералы для хранения паролей, номеров кредитных карт или другой конфиденциальной информации.

У вас может возникнуть вопрос о том, каким образом задействовать такую информацию в разрабатываемых приложениях (в лучшем сценарии) или зачем вообще заботиться о метаданных (в худшем сценарии). Чтобы получить ответ, необходимо ознакомиться со службами рефлексии .NET. Следует отметить, что полезность рассматриваемых далее тем может стать ясной только ближе к концу главы, а потому наберитесь терпения.

На заметку! В окне метаданных ildasm.exe вы также найдете несколько маркеров CustomAttribute, которые документируют атрибуты, применяемые внутри кодовой базы. О роли атрибутов .NET речь пойдет позже в главе.

Понятие рефлексии

В мире .NET *рефлексией* называется процесс обнаружения типов во время выполнения. Службы рефлексии дают возможность получать программно ту же самую информацию о метаданных, которую отображает утилита ildasm.exe, используя дружественную объектную модель. Например, посредством рефлексии можно извлечь список всех типов, содержащихся внутри заданной сборки *.dll или *.exe, в том числе методы, поля, свойства и события, которые определены конкретным типом. Можно также динамически получать набор интерфейсов, поддерживаемых заданным типом, параметры метода и другие относящиеся к ним детали (базовые классы, пространства имен, данные манифеста и т.д.).

Как и любое другое пространство имен, System.Reflection (из сборки mscorlib.dll) содержит набор связанных типов. В табл. 15.1 описаны основные члены System.Reflection, которые необходимо знать.

Чтобы понять, каким образом задействовать пространство имен System.Reflection для программного чтения метаданных .NET, сначала следует ознакомиться с классом System.Type.

Таблица 15.1. Избранные члены пространства имен `System.Reflection`

Тип	Описание
<code>Assembly</code>	Этот абстрактный класс содержит несколько членов, которые позволяют загружать, исследовать и манипулировать сборкой
<code>AssemblyName</code>	Этот класс позволяет выяснить многочисленные детали, связанные с идентичностью сборки (номер версии, информация о культуре и т.д.)
<code>EventInfo</code>	Этот абстрактный класс хранит информацию о заданном событии
<code>FieldInfo</code>	Этот абстрактный класс хранит информацию о заданном поле
<code>MemberInfo</code>	Этот абстрактный базовый класс определяет общее поведение для типов <code>EventInfo</code> , <code>FieldInfo</code> , <code>MethodInfo</code> и <code>PropertyInfo</code>
<code>MethodInfo</code>	Этот абстрактный класс содержит информацию о заданном методе
<code>Module</code>	Этот абстрактный класс позволяет получить доступ к заданному модулю внутри многофайловой сборки
<code>ParameterInfo</code>	Этот класс хранит информацию о заданном параметре
<code>PropertyInfo</code>	Этот абстрактный класс хранит информацию о заданном свойстве

Класс `System.Type`

Класс `System.Type` определяет набор членов, которые могут применяться для исследования метаданных типа, большое количество которых возвращают типы из пространства имен `System.Reflection`. Например, метод `Type.GetMethods()` возвращает массив объектов `MethodInfo`, метод `Type.GetFields()` — массив объектов `FieldInfo` и т.д. Полный перечень членов, доступных в `System.Type`, довольно велик, но в табл. 15.2 приведен список избранных членов, поддерживаемых `System.Type` (за исчерпывающими сведениями обращайтесь в документацию .NET Framework 4.7 SDK).

Таблица 15.2. Избранные члены `System.Type`

Член	Описание
<code>IsAbstract</code>	Эти свойства позволяют выяснять базовые характеристики типа, на который осуществляется ссылка (например, является ли он абстрактной сущностью, массивом, вложенным классом и т.п.)
<code>isArray</code>	
<code>IsClass</code>	
<code>IsCOMObject</code>	
<code>IsEnum</code>	
<code>IsGenericTypeDefinition</code>	
<code>IsGenericParameter</code>	
<code>IsInterface</code>	
<code>IsPrimitive</code>	
<code>IsNestedPrivate</code>	
<code>IsNestedPublic</code>	
<code>IsSealed</code>	
<code>IsValueType</code>	

Член	Описание
GetConstructors()	Эти методы позволяют получать массив интересных элементов (интерфейсов, методов, свойств и т.д.). Каждый метод возвращает связанный массив (например, GetFields() возвращает массив FieldInfo, метод GetMethods() — массив MethodInfo и т.д.). Следует отметить, что для каждого метода предусмотрена версия с именем в единственном числе (например, GetMethod(), GetProperty() и т.п.), которая позволяет извлекать специфический элемент по имени, а не массив всех связанных элементов
GetEvents()	
GetFields()	
GetInterfaces()	
GetMembers()	
GetMethods()	
GetNestedTypes()	
GetProperties()	
FindMembers()	Этот метод возвращает массив объектов MemberInfo на основе указанного критерия поиска
GetType()	Этот статический метод возвращает экземпляр Type с заданным строковым именем
InvokeMember()	Этот метод позволяет выполнять “позднее связывание” для указанного элемента. Вы узнаете о позднем связывании далее в главе

Получение информации о типе с помощью System.Object.GetType()

Экземпляр класса Type можно получать разнообразными способами. Тем не менее, есть одна вещь, которую делать невозможно — создавать объект Type напрямую, используя ключевое слово new, т.к. Type является абстрактным классом. Касательно первого способа вспомните, что в классе System.Object определен метод GetType(), который возвращает экземпляр класса Type, представляющий метаданные текущего объекта:

```
// Получить информацию о типе с применением экземпляра SportsCar.
SportsCar sc = new SportsCar();
Type t = sc.GetType();
```

Очевидно, такой подход будет работать, только если подвергаемый рефлексии тип (SportsCar в данном случае) известен на этапе компиляции и в памяти присутствует его экземпляр. С учетом этого ограничения должно быть понятно, почему инструменты вроде ildasm.exe не получают информацию о типе, непосредственно вызывая метод System.Object.GetType() для каждого типа: ведь утилита ildasm.exe не компилировалась вместе с вашими специальными сборками.

Получение информации о типе с помощью typeof()

Следующий способ получения информации о типе предполагает применение операции typeof:

```
// Получить информацию о типе с использованием операции typeof.
Type t = typeof(SportsCar);
```

В отличие от метода System.Object.GetType() операция typeof удобна тем, что она не требует предварительного создания экземпляра объекта перед получением информации о типе. Однако кодовой базе по-прежнему должно быть известно об исследуемом типе на этапе компиляции, поскольку typeof ожидает получения строго типизированного имени типа.

Получение информации о типе с помощью System.Type.GetType()

Для получения информации о типе в более гибкой манере можно вызывать статический метод GetType() класса System.Type и указывать полностью заданное строковое

имя типа, который планируется изучить. При таком подходе знать тип, из которого будут извлекаться метаданные, на этапе компиляции не нужно, т.к. метод `Type.GetType()` принимает в качестве параметра экземпляр вездесущего класса `System.String`.

На заметку! Когда речь идет о том, что при вызове метода `Type.GetType()` знание типа на этапе компиляции не требуется, имеется в виду тот факт, что данный метод может принимать любое строковое значение (а не строго типизированную переменную). Разумеется, знать имя типа в строковом формате по-прежнему необходимо!

Метод `Type.GetType()` перегружен, позволяя указывать два булевских параметра, из которых один управляет тем, должно ли генерироваться исключение, если тип не удастся найти, а второй отвечает за то, должен ли учитываться регистр символов в строке. В целях иллюстрации рассмотрим следующий код:

```
// Получить информацию о типе с использованием статического метода Type.GetType()
// (не генерировать исключение, если тип SportsCar не удастся найти,
// и игнорировать регистр символов) .
Type t = Type.GetType("CarLibrary.SportsCar", false, true);
```

В приведенном выше примере обратите внимание на то, что в строке, передаваемой методу `GetType()`, никак не упоминается сборка, внутри которой содержится интересующий тип. В этом случае делается предположение о том, что тип определен внутри сборки, выполняющейся в текущий момент. Тем не менее, когда необходимо получить метаданные для типа из внешней закрытой сборки, строковый параметр форматируется с использованием полностью заданного имени типа, за которым следует запятая и дружественное имя сборки, содержащей данный тип:

```
// Получить информацию о типе из внешней сборки.
Type t = Type.GetType("CarLibrary.SportsCar, CarLibrary");
```

Кроме того, в передаваемой методу `GetType()` строке может быть указан символ "плюс" (+) для обозначения вложенного типа. Пусть необходимо получить информацию о типе перечисления (`SpyOptions`), вложенного в класс по имени `JamesBondCar`. В таком случае можно написать следующий код:

```
// Получить информацию о типе для вложенного перечисления внутри текущей сборки.
Type t = Type.GetType("CarLibrary.JamesBondCar+SpyOptions");
```

Построение специального средства для просмотра метаданных

Чтобы продемонстрировать базовый процесс рефлексии (и полезность класса `System.Type`), создадим новый проект консольного приложения по имени `MyTypeViewer`. Приложение будет отображать детали методов, свойств, полей и поддерживаемых интерфейсов (в дополнение к другим интересным данным) для любого типа внутри `mscorlib.dll` (вспомните, что все приложения .NET автоматически получают доступ к этой основной библиотеке классов платформы) или типа внутри самого приложения `MyTypeViewer`. После создания приложения не забудьте импортировать пространство имен `System.Reflection`:

```
// Это пространство имен должно импортироваться для выполнения любой рефлексии!
using System.Reflection;
```


Рефлексия методов

Класс `Program` потребуется модифицировать для определения в нем нескольких статических методов, каждый из которых принимает единственный параметр `System.Type` и возвращает `void`. Сначала определим метод `ListMethods()`, который выводит имена методов, определенных во входном типе. Обратите внимание, что `Type.GetMethods()` возвращает массив объектов `System.Reflection.MethodInfo`, по которому можно осуществлять проход с помощью цикла `foreach`:

```
// Отобразить имена методов в типе.
static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    MethodInfo[] mi = t.GetMethods();
    foreach(MethodInfo m in mi)
        Console.WriteLine("->{0}", m.Name);
    Console.WriteLine();
}
```

Здесь просто выводится имя метода с применением свойства `MethodInfo.Name`. Как не трудно догадаться, класс `MethodInfo` имеет множество дополнительных членов, которые позволяют выяснить, является ли метод статическим, виртуальным, обобщенным или абстрактным. Вдобавок тип `MethodInfo` дает возможность получить информацию о возвращаемом значении и наборе параметров метода. Чуть позже реализация `ListMethods()` будет немного улучшена.

При желании для перечисления имен методов можно было бы также построить подходящий запрос LINQ. Вспомните из главы 12, что технология LINQ to Object позволяет создавать строго типизированные запросы и применять их к коллекциям объектов в памяти. В качестве эмпирического правила запомните, что при обнаружении блоков с программной логикой циклов или принятия решений можно использовать соответствующий запрос LINQ. Скажем, предыдущий метод можно было бы переписать так:

```
static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    var methodNames = from n in t.GetMethods() select n.Name;
    foreach (var name in methodNames)
        Console.WriteLine("->{0}", name);
    Console.WriteLine();
}
```

Рефлексия полей и свойств

Реализация метода `ListFields()` похожа. Единственным заметным отличием является вызов `Type.GetFields()` и результирующий массив элементов `FieldInfo`. И снова для простоты выводятся только имена каждого поля с применением запроса LINQ:

```
// Отобразить имена полей в типе.
static void ListFields(Type t)
{
    Console.WriteLine("***** Fields *****");
    var fieldNames = from f in t.GetFields() select f.Name;
    foreach (var name in fieldNames)
        Console.WriteLine("->{0}", name);
    Console.WriteLine();
}
```

Логика для отображения имен свойств типа аналогична:

```
// Отобразить имена свойств в типе.
static void ListProps(Type t)
{
    Console.WriteLine("***** Properties *****");
    var propNames = from p in t.GetProperties() select p.Name;
    foreach (var name in propNames)
        Console.WriteLine("->{0}", name);
    Console.WriteLine();
}
```

Рефлексия реализованных интерфейсов

Следующим создается метод по имени `ListInterfaces()`, который будет выводить имена любых интерфейсов, поддерживаемых входным типом. Один интересный момент здесь в том, что вызов `GetInterfaces()` возвращает массив объектов `System.Type`! Это вполне логично, поскольку интерфейсы действительно являются типами:

```
// Отобразить имена интерфейсов, которые реализует тип.
static void ListInterfaces(Type t)
{
    Console.WriteLine("***** Interfaces *****");
    var ifaces = from i in t.GetInterfaces() select i;
    foreach (Type i in ifaces)
        Console.WriteLine("->{0}", i.Name);
}
```

На заметку! Имейте в виду, что большинство методов “получения” в `System.Type` (`GetMethods()`, `GetInterfaces()` и т.д.) перегружены, чтобы позволить указывать значения из перечисления `BindingFlags`. В итоге появляется высокий уровень контроля над тем, что в точности необходимо искать (например, только статические члены, только открытые члены, включать закрытые члены и т.д.). За более подробной информацией обращайтесь в документацию .NET Framework 4.7 SDK.

Отображение разнообразных дополнительных деталей

В качестве последнего, но не менее важного действия, осталось реализовать финальный вспомогательный метод, который будет отображать различные статистические данные о входном типе (является ли он обобщенным, какой его базовый класс, запечатан ли он и т.п.):

```
// Просто ради полноты картины.
static void ListVariousStats(Type t)
{
    Console.WriteLine("***** Various Statistics *****");
    Console.WriteLine("Base class is: {0}", t.BaseType);
    // Базовый класс
    Console.WriteLine("Is type abstract? {0}", t.IsAbstract);
    // Абстрактный?
    Console.WriteLine("Is type sealed? {0}", t.IsSealed);
    // Запечатанный?
    Console.WriteLine("Is type generic? {0}", t.IsGenericTypeDefinition);
    // Обобщенный?
    Console.WriteLine("Is type a class type? {0}", t.IsClass); // Класс?
    Console.WriteLine();
}
```

Реализация метода Main ()

Метод Main() класса Program запрашивает у пользователя полностью заданное имя типа. После получения этих строковых данных они передаются методу Type.GetType(), а результирующий объект System.Type отправляется каждому вспомогательному методу. Процесс повторяется до тех пор, пока пользователь не введет Q для прекращения работы приложения.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Welcome to MyTypeViewer *****");
    string typeName = "";
    do
    {
        Console.WriteLine("\nEnter a type name to evaluate");
        // Предложить ввести имя типа
        Console.Write("or enter Q to quit: ");
        // или Q для завершения.

        // Получить имя типа.
        typeName = Console.ReadLine();

        // Пользователь желает завершить программу?
        if (typeName.Equals("Q", StringComparison.OrdinalIgnoreCase))
        {
            break;
        }

        // Попробовать отобразить информацию о типе.
        try
        {
            Type t = Type.GetType(typeName);
            Console.WriteLine("");
            ListVariousStats(t);
            ListFields(t);
            ListProps(t);
            ListMethods(t);
            ListInterfaces(t);
        }
        catch
        {
            Console.WriteLine("Sorry, can't find type"); // Не удастся найти тип.
        }
    } while (true);
}
```

В настоящий момент приложение MyTypeViewer.exe готово к тестовому запуску. Давайте запустим его и введем следующие полностью заданные имена (не забывая, что выбранный способ вызова Type.GetType() требует ввода строковых имен с учетом регистра):

- System.Int32
- System.Collections.ArrayList
- System.Threading.Thread
- System.Void
- System.IO.BinaryWriter

- `System.Math`
- `System.Console`
- `MyTypeViewer.Program`

Ниже показан частичный вывод при указании `System.Math`:

```
***** Welcome to MyTypeViewer *****
Enter a type name to evaluate
or enter Q to quit: System.Math
***** Various Statistics *****
Base class is: System.Object
Is type abstract? True
Is type sealed? True
Is type generic? False
Is type a class type? True
***** Fields *****
->PI
->E
***** Properties *****
***** Methods *****
->Acos
->Asin
->Atan
->Atan2
->Ceiling
->Ceiling
->Cos
...
```

Рефлексия обобщенных типов

При вызове `Type.GetType()` для получения описаний метаданных обобщенных типов должен использоваться специальный синтаксис, включающий символ обратной одинарной кавычки (```), за которым следует числовое значение, представляющее количество поддерживаемых параметров типа. Например, чтобы вывести описание метаданных `System.Collections.Generic.List<T>`, приложению потребуется передать следующую строку:

```
System.Collections.Generic.List`1
```

Здесь указано числовое значение 1, т.к. `List<T>` имеет только один параметр типа. Однако для применения рефлексии к типу `Dictionary<TKey, TValue>` понадобится предоставить значение 2:

```
System.Collections.Generic.Dictionary`2
```

Рефлексия параметров и возвращаемых значений методов

Давайте проведем небольшое усовершенствование приложения. В частности, мы модифицируем вспомогательный метод `ListMethods()` так, чтобы он выводил не только имя заданного метода, но также возвращаемый тип и типы входных параметров. Для решения этих задач тип `MethodInfo` предлагает свойство `ReturnType` и метод `GetParameters()`. В следующем измененном коде обратите внимание на то, что строка с информацией о типе и имени каждого параметра строится с использованием вложенного цикла `foreach` (`LINQ` не применяется):

```

static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    MethodInfo[] mi = t.GetMethods();
    foreach (MethodInfo m in mi)
    {
        // Получить информацию о возвращаемом типе.
        string retVal = m.ReturnType.FullName;
        string paramInfo = " ";

        // Получить информацию о параметрах.
        foreach (ParameterInfo pi in m.GetParameters())
        {
            paramInfo += string.Format("{0} {1} ", pi.ParameterType, pi.Name);
        }
        paramInfo += " ";

        // Отобразить базовую сигнатуру метода.
        Console.WriteLine("->{0} {1} {2}", retVal, m.Name, paramInfo);
    }
    Console.WriteLine();
}

```

Если теперь запустить обновленное приложение, то обнаружится, что методы заданного типа будут описаны более подробно. Например, в случае ввода типа `System.Object` отобразятся следующие описания методов:

```

***** Methods *****
->System.String ToString ( )
->System.Boolean Equals ( System.Object obj )
->System.Boolean Equals ( System.Object objA System.Object objB )
->System.Boolean ReferenceEquals ( System.Object objA System.Object objB )
->System.Int32 GetHashCode ( )
->System.Type GetType ( )

```

Текущая реализация `ListMethods()` удобна тем, что позволяет исследовать непосредственно каждый параметр и возвращаемый тип методов с использованием объектной модели `System.Reflection`. В качестве предельного сокращения имейте в виду, что все типы `XXXInfo` (`MethodInfo`, `PropertyInfo`, `EventInfo` и т.д.) переопределяют метод `ToString()` с целью отображения сигнатуры запрашиваемого элемента. Таким образом, метод `ListMethods()` можно было бы реализовать в следующем виде (здесь снова применяется запрос `LINQ`, выбирающий все объекты `MethodInfo`, а не только значения `Name`):

```

static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    var methodNames = from n in t.GetMethods() select n;
    foreach (var name in methodNames)
        Console.WriteLine("->{0}", name);
    Console.WriteLine();
}

```

Довольно интересно, не так ли? Очевидно, что пространство имен `System.Reflection` и класс `System.Type` позволяют выполнять рефлексии многих других аспектов типа помимо тех, которые в настоящий момент отображает приложение `MyTypeViewer`. Как и можно было ожидать, допускается также получать информацию о событиях, поддерживаемых типом, любых обобщенных параметрах для заданных членов и десятки других деталей.

Итак, мы создали (довольно мощный) браузер объектов. Главное ограничение состоит в том, что он не позволяет подвергать рефлексии ничего кроме текущей сборки (MyTypeViewer) и всегда доступной сборки mscorlib.dll. Возникает вопрос: как построить приложение, способное загружать (и проводить рефлексию) сборки, на которые отсутствуют ссылки на этапе компиляции? Об этом пойдет речь в следующем разделе.

Исходный код. Проект MyTypeViewer доступен в подкаталоге Chapter_15.

Динамическая загрузка сборок

В главе 14 вы узнали все о том, как среда CLR заглядывает в манифест сборки во время зондирования внешних сборок, на которые ссылается текущая сборка. Тем не менее, во многих случаях сборки необходимо загружать на лету программным образом, даже если в манифесте отсутствуют записи о них. Формально процесс загрузки внешних сборок по требованию называется *динамической загрузкой*.

В пространстве имен System.Reflection определен класс Assembly, с использованием которого можно динамически загружать сборку, а также исследовать связанные с ней свойства. С помощью Assembly можно динамически загружать закрытые или разделяемые сборки, равно как и сборки, расположенные в произвольных местах. По существу класс Assembly предлагает методы (в частности, Load() и LoadFrom()), которые позволяют программно предоставлять информацию того же рода, что и в клиентских файлах *.config.

Чтобы проиллюстрировать динамическую загрузку, создадим новый проект консольного приложения по имени ExternalAssemblyReflector. Наша задача связана с написанием метода Main(), который будет запрашивать у пользователя дружественное имя сборки с целью ее динамической загрузки. Ссылка на Assembly будет передаваться вспомогательному методу под названием DisplayTypes(), который просто выведет имена всех содержащихся в сборке классов, интерфейсов, структур, перечислений и делегатов. Вот необходимый код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;
using System.IO; // Для определения FileNotFoundException.

namespace ExternalAssemblyReflector
{
    class Program
    {
        static void DisplayTypesInAsm(Assembly asm)
        {
            Console.WriteLine("\n***** Types in Assembly *****");
            Console.WriteLine("->{0}", asm.FullName);
            Type[] types = asm.GetTypes();
            foreach (Type t in types)
            {
                Console.WriteLine("Type: {0}", t);
                Console.WriteLine("");
            }
        }

        static void Main(string[] args)
        {
            Console.WriteLine("***** External Assembly Viewer *****");
```

```

string asmName = "";
Assembly asm = null;
do
{
    Console.WriteLine("\nEnter an assembly to evaluate");
    // Предложить ввести имя сборки
    Console.Write("or enter Q to quit: ");
    // или Q для завершения.
    // Получить имя сборки.
    asmName = Console.ReadLine();
    // Пользователь желает завершить программу?
    if (asmName.Equals("Q", StringComparison.OrdinalIgnoreCase))
    {
        break;
    }
    // Попробовать загрузить сборку.
    try
    {
        asm = Assembly.Load(asmName);
        DisplayTypesInAsm(asm);
    }
    catch
    {
        Console.WriteLine("Sorry, can't find assembly.");
        // Сборка не найдена.
    }
} while (true);
}
}

```

Обратите внимание, что статическому методу `Assembly.Load()` передается только дружественное имя сборки, которую требуется загрузить в память. Таким образом, чтобы подвергнуть рефлексии сборку `CarLibrary.dll`, понадобится скопировать двойичный файл `CarLibrary.dll` в подкаталог `bin\Debug` внутри каталога приложения `ExternalAssemblyReflector`. Затем можно будет получить примерно такой вывод:

```

***** External Assembly Viewer *****

Enter an assembly to evaluate
or enter Q to quit: CarLibrary

***** Types in Assembly *****
->CarLibrary, Version=2.0.0.0, Culture=neutral, PublicKeyToken=33a2bc294331e8b9
Type: CarLibrary.MusicMedia
Type: CarLibrary.EngineState
Type: CarLibrary.Car
Type: CarLibrary.SportsCar
Type: CarLibrary.Minivan

```

Если вы хотите сделать приложение `ExternalAssemblyReflector` более гибким, тогда модифицируйте код так, чтобы загрузка внешней сборки производилась с применением метода `Assembly.LoadFrom()` вместо `Assembly.Load()`:

```

try
{
    asm = Assembly.LoadFrom(asmName);
    DisplayTypesInAsm(asm);
}

```

В итоге появляется возможность вводить абсолютный путь к интересующей сборке (скажем, C:\MyApp\MyAsm.dll). В сущности метод `Assembly.LoadFrom()` позволяет программно предоставлять значение `<codeBase>`. Теперь консольному приложению можно передавать полный путь. Если сборка `CarLibrary.dll` находится в каталоге C:\MyCode, то будет получен следующий вывод:

```
***** External Assembly Viewer *****
Enter an assembly to evaluate
or enter Q to quit: C:\MyCode\CarLibrary.dll
***** Types in Assembly *****
->CarLibrary, Version=2.0.0.0, Culture=neutral, PublicKeyToken=33a2bc294331e8b9
Type: CarLibrary.EngineState
Type: CarLibrary.Car
Type: CarLibrary.SportsCar
Type: CarLibrary.Minivan
```

Исходный код. Проект `ExternalAssemblyReflector` доступен в подкаталоге `Chapter_15`.

Рефлексия разделяемых сборок

Метод `Assembly.Load()` имеет несколько перегруженных версий. Одна из них разрешает указывать значение культуры (для локализованных сборок), а также номер версии и значение маркера открытого ключа (для разделяемых сборок). Коллективно многочисленные элементы, идентифицирующие сборку, называются *отображаемым именем*. Форматом отображаемого имени является строка пар “имя-значение”, разделенных запятыми, которая начинается с дружественного имени сборки, а за ним следуют необязательные квалификаторы (в любом порядке). Вот как выглядит шаблон (необязательные элементы указаны в круглых скобках):

```
Имя (,Version =
    <старший номер>.<младший номер>.<номер сборки>.<номер редакции>)
    (,Culture = <маркер культуры>) (,PublicKeyToken = <маркер открытого ключа>)
```

При создании отображаемого имени соглашение `PublicKeyToken=null` отражает тот факт, что требуется связывание и сопоставление со сборкой, не имеющей строгого имени. Вдобавок `Culture=""` указывает, что сопоставление должно осуществляться со стандартной культурой целевой машины, например:

```
// Загрузить версию 1.0.0.0 сборки CarLibrary, используя стандартную культуру.
Assembly a =
    Assembly.Load(@"CarLibrary, Version=1.0.0.0, PublicKeyToken=null, Culture=");
```

Кроме того, следует иметь в виду, что пространство имен `System.Reflection` предлагает тип `AssemblyName`, который позволяет представлять показанную выше строковую информацию в удобной объектной переменной. Обычно класс `AssemblyName` применяется вместе с классом `System.Version`, который представляет собой объектно-ориентированную оболочку для номера версии сборки. После создания отображаемого имени его затем можно передавать перегруженной версии метода `Assembly.Load()`:

```
// Применение типа AssemblyName для определения отображаемого имени.
AssemblyName asmName;
asmName = new AssemblyName();
asmName.Name = "CarLibrary";
Version v = new Version("1.0.0.0");
asmName.Version = v;
Assembly a = Assembly.Load(asmName);
```


Для загрузки разделяемой сборки из GAC в параметре метода `Assembly.Load()` должно быть указано значение `PublicKeyToken`. Например, пусть создан новый проект консольного приложения по имени `SharedAsmReflector`, и нужно загрузить версию 4.0.0.0 сборки `System.Windows.Forms.dll`, предоставляемую библиотеками базовых классов .NET. Поскольку количество типов в данной сборке довольно велико, приложение будет выводить только имена открытых перечислений, используя простой запрос LINQ:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
using System.IO;
namespace SharedAsmReflector
{
    public class SharedAsmReflector
    {
        private static void DisplayInfo(Assembly a)
        {
            Console.WriteLine("***** Info about Assembly *****");
            Console.WriteLine("Loaded from GAC? {0}", a.GlobalAssemblyCache);
            // Загружена из GAC?
            Console.WriteLine("Asm Name: {0}", a.GetName().Name);
            // Имя сборки
            Console.WriteLine("Asm Version: {0}", a.GetName().Version);
            // Версия сборки
            Console.WriteLine("Asm Culture: {0}",
                // Культура сборки
                a.GetName().CultureInfo.DisplayName);
            Console.WriteLine("\nHere are the public enums:");
            // Список открытых перечислений
            // Использовать запрос LINQ для нахождения открытых перечислений.
            Type[] types = a.GetTypes();
            var publicEnums = from pe in types where pe.IsEnum &&
                               pe.IsPublic select pe;
            foreach (var pe in publicEnums)
            {
                Console.WriteLine(pe);
            }
        }
        static void Main(string[] args)
        {
            Console.WriteLine("***** The Shared Asm Reflector App *****\n");
            // Загрузить System.Windows.Forms.dll из GAC.
            string displayName = null;
            displayName = "System.Windows.Forms," +
                "Version=4.0.0.0," +
                "PublicKeyToken=b77a5c561934e089," +
                @"Culture=""""";
            Assembly asm = Assembly.Load(displayName);
            DisplayInfo(asm);
            Console.WriteLine("Done!");
            Console.ReadLine();
        }
    }
}
```

Исходный код. Проект SharedAsmReflector доступен в подкаталоге Chapter_15.

К настоящему моменту вы должны уметь работать с некоторыми основными членами пространства имен `System.Reflection` для получения метаданных во время выполнения. Конечно, необходимость в самостоятельном построении специальных браузеров объектов в повседневной практике вряд ли будет возникать часто. Однако не забывайте, что службы рефлексии являются основой для нескольких распространенных действий программирования, включая *позднее связывание*.

Позднее связывание

Позднее связывание представляет собой прием, который позволяет создавать экземпляры заданного типа и обращаться к его членам во время выполнения без необходимости в жестком кодировании факта его существования на этапе компиляции. При построении приложения, в котором производится позднее связывание с типом из внешней сборки, нет причин устанавливать ссылку на эту сборку; следовательно, в манифесте вызывающего кода она прямо не указывается.

На первый взгляд значимость позднего связывания оценить нелегко. Действительно, если есть возможность выполнить "раннее связывание" с объектом (например, добавить ссылку на сборку и выделить память под экземпляр типа с помощью ключевого слова `new`), то именно так следует поступать. Причина в том, что раннее связывание позволяет выявлять ошибки на этапе компиляции, а не во время выполнения. Тем не менее, позднее связывание играет важную роль в любом расширяемом приложении, которое может строиться. Пример построения такого "расширяемого" приложения будет приведен в конце главы, в разделе "Построение расширяемого приложения", а пока займемся исследованием роли класса `Activator`.

Класс `System.Activator`

Класс `System.Activator` (определенный в `mscorlib.dll`) играет ключевую роль в процессе позднего связывания .NET. В текущем примере нам интересен только метод `Activator.CreateInstance()`, который применяется для создания экземпляра типа в стиле позднего связывания. Этот метод имеет несколько перегруженных версий, обеспечивая достаточно высокую гибкость. Самая простая версия метода `CreateInstance()` принимает действительный объект `Type`, описывающий сущность, которую необходимо разместить в памяти на лету.

Создадим новый проект консольного приложения по имени `LateBindingApp` и с помощью ключевого слова `using` импортируем в него пространства имен `System.IO` и `System.Reflection`. Теперь модифицируем класс `Program` следующим образом:

```
// Это приложение будет загружать внешнюю сборку и
// создавать объект, используя позднее связывание.
public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Late Binding *****");
        // Попробовать загрузить локальную копию CarLibrary.
        Assembly a = null;
        try
        {
            a = Assembly.Load("CarLibrary");
        }
    }
}
```

```

        catch (FileNotFoundException ex)
        {
            Console.WriteLine(ex.Message);
            return;
        }
        if (a != null)
            CreateUsingLateBinding(a);
        Console.ReadLine();
    }

    static void CreateUsingLateBinding(Assembly asm)
    {
        try
        {
            // Получить метаданные для типа MiniVan.
            Type miniVan = asm.GetType("CarLibrary.MiniVan");

            // Создать экземпляр MiniVan на лету.
            object obj = Activator.CreateInstance(miniVan);
            Console.WriteLine("Created a {0} using late binding!", obj);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}

```

Перед запуском нового приложения понадобится вручную скопировать файл `CarLibrary.dll` в подкаталог `bin\Debug` внутри каталога данного приложения. Причина в том, что в коде вызывается метод `Assembly.Load()`, т.е. среда CLR будет зондировать только каталог клиента (при желании можно было бы вызвать метод `Assembly.LoadFrom()` и вводить полный путь к сборке, но в текущем случае в этом нет нужды).

На заметку! В рассматриваемом примере не добавляйте ссылку на сборку `CarLibrary.dll` с использованием Visual Studio! Такое действие приведет к помещению в манифест клиента записи о библиотеке `CarLibrary.dll`. Вся суть позднего связывания заключается в попытке создания объекта, который не известен на этапе компиляции.

Обратите внимание, что метод `Activator.CreateInstance()` возвращает экземпляр `System.Object`, а не строго типизированный объект `MiniVan`. Следовательно, если применить к переменной `obj` операцию точки, то члены класса `MiniVan` не будут видны. На первый взгляд может показаться, что проблему удастся решить с помощью явного приведения:

```

// Привести к типу MiniVan, чтобы получить доступ к его членам?
// Нет! Компилятор сообщит об ошибке!
object obj = (MiniVan)Activator.CreateInstance(minivan);

```

Однако из-за того, что в приложение не была добавлена ссылка на сборку `CarLibrary.dll`, использовать ключевое слово `using` для импортирования пространства имен `CarLibrary` нельзя, а значит невозможно и указывать `MiniVan` в операции приведения! Не забывайте, что смысл позднего связывания — создание экземпляров типов, о которых на этапе компиляции ничего не известно. Учитывая сказанное, возникает вопрос: как вызывать методы объекта `MiniVan`, сохраненного в ссылке на `System.Object`? Ответ: конечно же, с помощью рефлексии.

Вызов методов без параметров

Предположим, что требуется вызвать метод `TurboBoost()` объекта `MiniVan`. Как вы наверняка помните, упомянутый метод переводит двигатель в нерабочее состояние и затем отображает окно с соответствующим сообщением. Первый шаг заключается в получении объекта `MethodInfo` для метода `TurboBoost()` посредством `Type.GetMethod()`. Имея результирующий объект `MethodInfo`, можно вызвать `MiniVan.TurboBoost()` с помощью метода `Invoke()`. Метод `MethodInfo.Invoke()` требует указания всех параметров, которые подлежат передаче методу, представленному объектом `MethodInfo`. Параметры задаются в виде массива объектов `System.Object` (т.к. они могут быть самыми разнообразными сущностями).

Поскольку метод `TurboBoost()` не принимает параметров, можно просто передать `null` (т.е. сообщить, что вызываемый метод не имеет параметров). Модифицируем метод `CreateUsingLateBinding()` следующим образом:

```
static void CreateUsingLateBinding(Assembly asm)
{
    try
    {
        // Получить метаданные для типа MiniVan.
        Type miniVan = asm.GetType("CarLibrary.MiniVan");

        // Создать объект MiniVan на лету.
        object obj = Activator.CreateInstance(miniVan);
        Console.WriteLine("Created a {0} using late binding!", obj);

        // Получить информацию о TurboBoost.
        MethodInfo mi = miniVan.GetMethod("TurboBoost");

        // Вызвать метод (null означает отсутствие параметров).
        mi.Invoke(obj, null);
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Теперь после запуска приложения в результате вызова метода `TurboBoost()` отображается окно с сообщением (рис. 15.2).

Вызов методов с параметрами

Когда позднее связывание нужно применять для вызова метода, ожидающего параметры, аргументы потребуется упаковать в слабо типизированный массив `object`. Вспомните, что в версии 2.0.0.0 библиотеки `CarLibrary.dll` был определен следующий метод класса `Car`:

```
public void TurnOnRadio(bool musicOn, MusicMedia mm)
{
    public void TurnOnRadio(bool musicOn, MusicMedia mm)
        => MessageBox.Show(musicOn ? $"Jamming {mm}" : "Quiet time...");
}
```

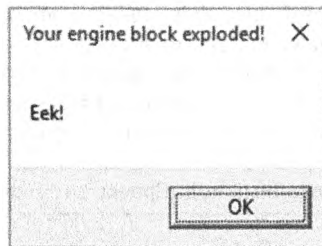


Рис. 15.2. Вызов метода через позднее связывание

Метод `TurnOnRadio()` принимает два параметра: булевское значение, которое указывает, должна ли быть включена музыкальная система в автомобиле, и перечисление, представляющее тип музыкального проигрывателя. Как упоминалось ранее, это перечисление выглядит так:

```
public enum MusicMedia
{
    musicCd,        // 0
    musicTape,      // 1
    musicRadio,     // 2
    musicMp3        // 3
}
```

Ниже приведен код нового метода класса `Program`, в котором вызывается `TurnOnRadio()`. Обратите внимание на использование внутренних числовых значений перечисления `MusicMedia`:

```
static void InvokeMethodWithArgsUsingLateBinding(Assembly asm)
{
    try
    {
        // Получить описание метаданных для типа SportsCar.
        Type sport = asm.GetType("CarLibrary.SportsCar");
        // Создать объект типа SportsCar.
        object obj = Activator.CreateInstance(sport);
        // Вызвать метод TurnOnRadio() с аргументами.
        MethodInfo m1 = sport.GetMethod("TurnOnRadio");
        m1.Invoke(obj, new object[] { true, 2 });
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

В идеале к настоящему времени вы уже видите отношения между рефлексией, динамической загрузкой и поздним связыванием. Естественно, помимо раскрытых здесь возможностей API-интерфейс рефлексии предлагает много дополнительных средств, но вы уже должны быть в хорошей форме, чтобы погрузиться в дальнейшее изучение.

Вас все еще может интересовать вопрос: *когда* описанные приемы должны применяться в разрабатываемых приложениях? Ответ проявится ближе к концу главы, а пока мы займемся исследованием роли атрибутов .NET.

Исходный код. Проект `LateBindingApp` доступен в подкаталоге `Chapter_15`.

Роль атрибутов .NET

Как было показано в начале главы, одной из задач компилятора .NET является генерация описаний метаданных для всех определяемых типов и для типов, на которые имеются ссылки. Помимо стандартных метаданных, содержащихся в любой сборке, платформа .NET предоставляет программистам способ встраивания в сборку дополнительных метаданных с использованием *атрибутов*. Выражаясь кратко, атрибуты — это всего лишь аннотации кода, которые могут применяться к заданному типу (классу, интерфейсу, структуре и т.п.), члену (свойству, методу и т.д.), сборке или модулю.

Атрибуты .NET представляют собой типы классов, расширяющие абстрактный базовый класс `System.Attribute`. По мере изучения пространств имен .NET вы найдете много предопределенных атрибутов, которые можно использовать в своих приложениях. Более того, вы также можете строить собственные атрибуты для дополнительного уточнения поведения своих типов путем создания нового типа, производного от `Attribute`.

Библиотека базовых классов .NET предлагает множество атрибутов в различных пространствах имен. Некоторые (безусловно, далеко не все) предопределенные атрибуты описаны в табл. 15.3.

Таблица 15.3. Небольшое число избранных предопределенных атрибутов

Атрибут	Описание
<code>[CLSCompliant]</code>	Заставляет аннотированный элемент соответствовать правилам CLS (Common Language Specification – общезыко́вая спецификация). Вспомните, что совместимые с CLS типы могут гарантированно использоваться во всех языках программирования .NET
<code>[DllImport]</code>	Позволяет коду .NET обращаться к любой неуправляемой библиотеке кода C или C++, включая API-интерфейс операционной системы. Обратите внимание, что при взаимодействии с программным обеспечением, основанным на COM, этот атрибут не применяется
<code>[Obsolete]</code>	Помечает устаревший тип или член. Если другие программисты попытаются использовать такой элемент, то они получают соответствующее предупреждение от компилятора
<code>[Serializable]</code>	Помечает класс или структуру как сериализируемую, что означает возможность сохранения своего текущего состояния в потоке
<code>[NonSerialized]</code>	Указывает, что заданное поле в классе или структуре не должно сохраняться в процессе сериализации
<code>[ServiceContract]</code>	Помечает метод как контракт, реализованный службой WCF

Важно понимать, что когда вы применяете атрибуты в своем коде, встроенные метаданные по существу бесполезны до тех пор, пока другая часть программного обеспечения явно не запросит такую информацию посредством рефлексии. В противном случае метаданные, встроенные в сборку, игнорируются и не причиняют никакого вреда.

Потребители атрибутов

Как нетрудно догадаться, в составе .NET Framework 4.7 SDK поставляются многочисленные утилиты, которые действительно ищут разнообразные атрибуты. Сам компилятор C# (`csc.exe`) запрограммирован на обнаружение различных атрибутов при проведении компиляции. Например, встретив атрибут `[CLSCompliant]`, компилятор автоматически проверяет помеченный им элемент и удостоверяется в том, что в нем открыт доступ только к конструкциям, совместимым с CLS. Еще один пример: если компилятор обнаруживает элемент с атрибутом `[Obsolete]`, тогда он отображает в окне Error List (Список ошибок) среды Visual Studio сообщение с предупреждением.

В дополнение к инструментам разработки многие методы в библиотеках базовых классов .NET изначально запрограммированы на распознавание определенных атрибутов посредством рефлексии. Например, если нужно сохранить информацию о состоянии объекта в файл, то все, что потребуется сделать — аннотировать класс или структуру атрибутом `[Serializable]`. Если метод `Serialize()` класса `BinaryFormatter` встречает этот атрибут, то объект автоматически сохраняется в файле в компактном двоичном формате.

Наконец, можно строить приложения, способные распознавать специальные атрибуты, а также любые атрибуты из библиотек базовых классов .NET. По сути, тем самым создается набор “ключевых слов”, которые понимает специфичное множество сборок.

Применение атрибутов в C#

Чтобы продемонстрировать процесс применения атрибутов в C#, создадим новый проект консольного приложения по имени `ApplyingAttributes`. Предположим, что необходимо построить класс под названием `Motorcycle` (мотоцикл), который может сохраняться в двоичном формате. Для этого нужно просто применить к определению класса атрибут `[Serializable]`. Если какое-то поле не должно сохраняться, то к нему можно применить атрибут `[NonSerialized]`.

```
// Этот класс может быть сохранен на диске.
[Serializable]
public class Motorcycle
{
    // Однако это поле сохраняться не будет.
    [NonSerialized]
    float weightOfCurrentPassengers;
    // Эти поля остаются сериализуемыми.
    bool hasRadioSystem;
    bool hasHeadSet;
    bool hasSissyBar;
}
```

На заметку! Атрибут применяется только к элементу, находящемуся непосредственно после него. Например, единственным несериализуемым полем в классе `Motorcycle` является `weightOfCurrentPassengers`. Остальные поля будут сериализуемыми, т.к. сам класс аннотирован атрибутом `[Serializable]`.

В данный момент пусть вас не беспокоит фактический процесс сериализации объектов (он подробно рассматривается в главе 20). Просто знайте, что для применения атрибута его имя должно быть помещено в квадратные скобки.

После компиляции класса `Motorcycle` дополнительные метаданные можно просматривать с помощью утилиты `ildasm.exe`. Обратите внимание, что эти атрибуты записаны с использованием маркера `serializable` (взгляните на значок в форме треугольника внутри класса `Motorcycle`) и маркера `notserialized` (в поле `weightOfCurrentPassengers`), как показано на рис. 15.3.

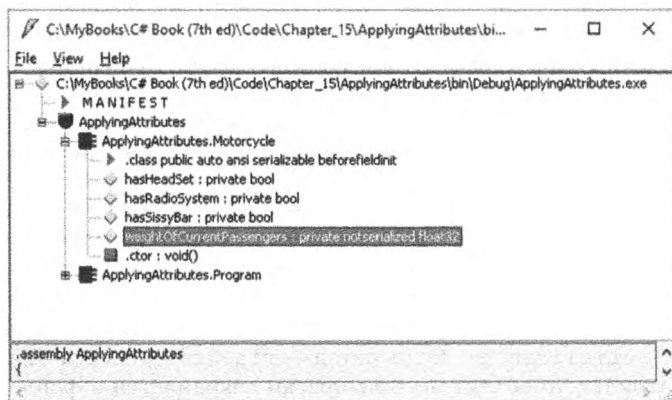


Рис. 15.3. Просмотр атрибутов в `ildasm.exe`

Нетрудно догадаться, что к одиночному элементу можно применять множество атрибутов. Пусть имеется унаследованный тип класса C# (HorseAndBuggy), который был помечен как сериализуемый, но для текущей разработки он считается устаревшим. Вместо того чтобы удалять определение такого класса из кодовой базы (с риском нарушения работы существующего программного обеспечения), его можно пометить атрибутом [Obsolete]. Для применения множества атрибутов к одному элементу просто используйте список с разделителями-запятymi:

```
[Serializable, Obsolete("Use another vehicle!")]
public class HorseAndBuggy
{
    // ...
}
```

В качестве альтернативы применить множество атрибутов к единственному элементу можно также, указывая их друг за другом (конечный результат будет идентичным):

```
[Serializable]
[Obsolete("Use another vehicle!")]
public class HorseAndBuggy
{
    // ...
}
```

Сокращенная система обозначения атрибутов C#

Заглянув в документацию .NET Framework 4.7 SDK, вы можете заметить, что действительным именем класса, представляющего атрибут [Obsolete], является ObsoleteAttribute, а не просто Obsolete. По соглашению имена всех атрибутов .NET (включая специальные атрибуты, которые создаете вы сами) снабжаются суффиксом Attribute. Тем не менее, чтобы упростить процесс применения атрибутов, язык C# не требует обязательного ввода суффикса Attribute. Учитывая это, показанная ниже версия типа HorseAndBuggy идентична предыдущей версии (но влечет за собой более объемный клавиатурный набор):

```
[SerializableAttribute]
[ObsoleteAttribute("Use another vehicle!")]
public class HorseAndBuggy
{
    // ...
}
```

Имейте в виду, что такая сокращенная система обозначения для атрибутов предлагается только в C#. Ее поддерживают не все языки .NET.

Указание параметров конструктора для атрибутов

Обратите внимание, что атрибут [Obsolete] может принимать то, что выглядит как параметр конструктора. Если вы просмотрите формальное определение атрибута [Obsolete], щелкнув на нем правой кнопкой мыши в окне кода и выбрав в контекстном меню пункт Go To Definition (Перейти к определению), то обнаружите, что данный класс на самом деле предоставляет конструктор, принимающий System.String:

```
public sealed class ObsoleteAttribute : Attribute
{
    public ObsoleteAttribute(string message, bool error);
    public ObsoleteAttribute(string message);
    public ObsoleteAttribute();
    public bool IsError { get; }
    public string Message { get; }
}
```


Важно понимать, что когда вы предоставляете атрибуту параметров конструктора, этот атрибут *не* размещается в памяти до тех пор, пока к параметрам не будет применена рефлексия со стороны другого типа или внешнего инструмента. Строковые данные, определенные на уровне атрибутов, просто сохраняются внутри сборки в виде блока метаданных.

Атрибут [Obsolete] в действии

Теперь, когда класс `HorseAndBuggy` помечен как устаревший, следующая попытка выделения памяти под его экземпляр:

```
static void Main(string[] args)
{
    HorseAndBuggy mule = new HorseAndBuggy();
}
```

приводит к тому, что указанные строковые данные извлекаются и отображаются в окне Error List среды Visual Studio, а также в проблемной строке кода при наведении курсора мыши на устаревший тип (рис. 15.4).

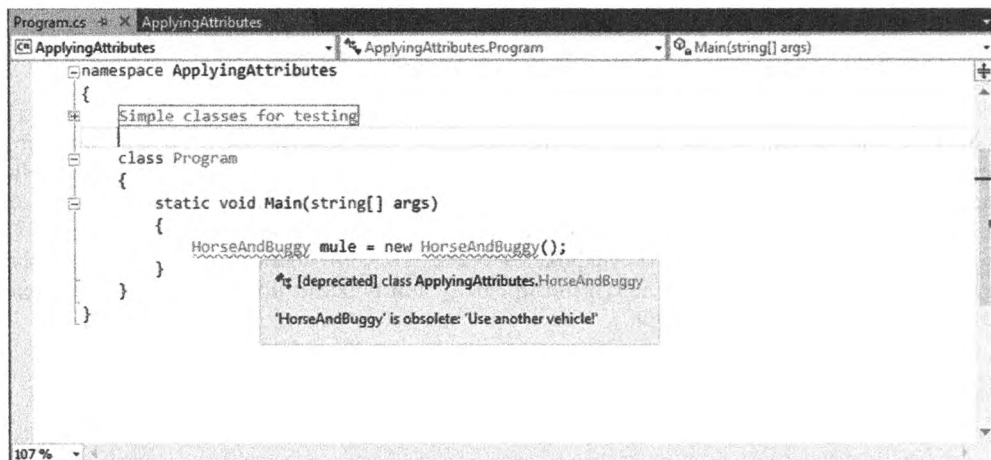


Рис. 15.4. Атрибуты в действии

В этом случае “другой частью программного обеспечения”, которая производит рефлексия атрибута `[Obsolete]`, является компилятор C#. По идее вы уже должны понимать перечисленные ниже ключевые моменты, касающиеся атрибутов .NET:

- атрибуты представляют собой классы, производные от `System.Attribute`;
- атрибуты дают в результате встроенные метаданные;
- атрибуты в основном бесполезны до тех пор, пока другой агент не проведет в их отношении рефлексия;
- атрибуты в языке C# применяются с использованием квадратных скобок.

А теперь давайте посмотрим, как реализовывать собственные специальные атрибуты и создавать специальное программное обеспечение, которое выполняет рефлексия по встроенным метаданным.

Построение специальных атрибутов

Первый шаг при построении специального атрибута предусматривает создание нового класса, производного от `System.Attribute`. Не отклоняясь от автомобильной темы, повсеместно встречающейся в книге, создадим новый проект типа `Class Library` (Библиотека классов) на C# под названием `AttributedCarLibrary`. В этой сборке будет определено несколько классов для представления транспортных средств, каждый из которых описан с использованием специального атрибута по имени `VehicleDescriptionAttribute`:

```
// Специальный атрибут.
public sealed class VehicleDescriptionAttribute : System.Attribute
{
    public string Description { get; set; }
    public VehicleDescriptionAttribute(string vehicalDescription)
        => Description = vehicalDescription;
    public VehicleDescriptionAttribute() { }
}
```

Как видите, класс `VehicleDescriptionAttribute` поддерживает фрагмент строковых данных, которым можно манипулировать с помощью автоматического свойства (`Description`). Помимо того факта, что данный класс является производным от `System.Attribute`, ничего примечательного в его определении нет.

На заметку! По причинам, связанным с безопасностью, установившейся практикой в .NET считается проектирование всех специальных атрибутов как запечатанных. На самом деле среда `Visual Studio` предлагает фрагмент кода под названием `Attribute`, который позволяет сгенерировать в окне редактора кода новый класс, производный от `System.Attribute`. О применении фрагментов кода подробно рассказывалось в главе 2; вероятно вы помните, что для раскрытия любого фрагмента кода необходимо набрать его имя и два раза нажать клавишу `<Tab>`.

Применение специальных атрибутов

С учетом того, что класс `VehicleDescriptionAttribute` является производным от `System.Attribute`, теперь можно аннотировать транспортные средства. В целях тестирования добавим в новую библиотеку классов следующие определения:

```
// Назначить описание с помощью "именованного свойства".
[Serializable]
[VehicleDescription(Description = "My rocking Harley")]
public class Motorcycle
{
}

[Serializable]
[Obsolete ("Use another vehicle!")]
[VehicleDescription("The old gray mare, she ain't what she used to be...")]
public class HorseAndBuggy
{
}

[VehicleDescription("A very long, slow, but feature-rich auto")]
public class Winnebago
{
}
```

Синтаксис именованных свойств

Обратите внимание, что описание назначается классу `Motorcycle` с использованием нового фрагмента синтаксиса, связанного с атрибутами, который называется *именованным свойством*. В конструкторе первого атрибута `[VehicleDescription]` лежащие в основе строковые данные устанавливаются с применением свойства `Description`. Когда внешний агент выполняет рефлексию для этого атрибута, свойству `Description` будет передано указанное значение (синтаксис именованных свойств разрешен, только если атрибут предоставляет поддерживающее запись свойство `.NET`).

По контрасту для типов `HorseAndBuggy` и `Winnebago` синтаксис именованных свойств не используется, а строковые данные просто передаются через специальный конструктор. В любом случае после компиляции сборки `AttributedCarLibrary` с помощью утилиты `ildasm.exe` можно просмотреть добавленные описания метаданных. Например, на рис. 15.5 показано встроенное описание класса `Winnebago`, в частности данные внутри элемента `beforefieldinit`.

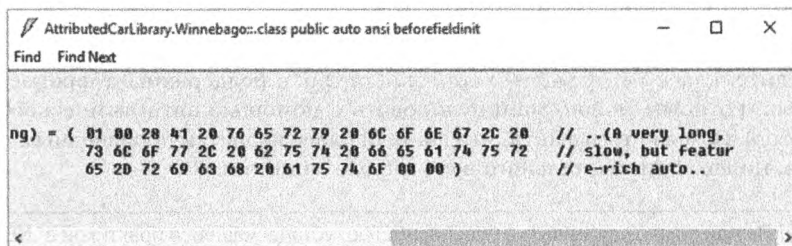


Рис. 15.5. Встроенные данные описания транспортного средства

Ограничение использования атрибутов

По умолчанию специальные атрибуты могут быть применены практически к любому аспекту кода (методам, классам, свойствам и т.д.). Таким образом, если бы это имело смысл, то `VehicleDescription` можно было бы использовать для уточнения методов, свойств или полей (помимо прочего):

```
[VehicleDescription("A very long, slow, but feature-rich auto")]
public class Winnebago
{
    [VehicleDescription("My rocking CD player")]
    public void PlayMusic(bool On)
    {
        ...
    }
}
```

В одних случаях такое поведение является точно таким, какое требуется, но в других может возникнуть желание создать специальный атрибут, применяемый только к избранным элементам кода. Чтобы ограничить область действия специального атрибута, понадобится добавить к его определению атрибут `[AttributeUsage]`, который позволяет предоставлять любую комбинацию значений (посредством операции "ИЛИ") из перечисления `AttributeTargets`:

```
// Это перечисление определяет возможные целевые элементы для атрибута.
public enum AttributeTargets
{

```

```
All, Assembly, Class, Constructor,
Delegate, Enum, Event, Field, GenericParameter,
Interface, Method, Module, Parameter,
Property, ReturnValue, Struct
}
```

Кроме того, атрибут `[AttributeUsage]` допускает необязательную установку именованного свойства (`AllowMultiple`), которое указывает, может ли атрибут применяться к тому же самому элементу более одного раза (стандартным значением является `false`). Вдобавок `[AttributeUsage]` разрешает указывать, должен ли атрибут наследоваться производными классами, с использованием именованного свойства `Inherited` (со стандартным значением `true`).

Модифицируем определение `VehicleDescriptionAttribute` для указания на то, что атрибут `[VehicleDescription]` может применяться только к классу или структуре:

```
// На этот раз для аннотирования специального атрибута
// мы используем атрибут AttributeUsage.
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
                Inherited = false)]
public sealed class VehicleDescriptionAttribute : System.Attribute
{
    ...
}
```

Теперь если разработчик попытается применить атрибут `[VehicleDescription]` не к классу или структуре, то компилятор сообщит об ошибке.

Атрибуты уровня сборки

Атрибуты можно также применять ко всем типам внутри отдельной сборки, используя дескриптор `[assembly:]`. Например, предположим, что необходимо обеспечить совместимость с CLS для всех открытых членов во всех открытых типах, определенных внутри сборки.

На заметку! В главе 1 упоминалось о роли сборок, совместимых с CLS. Вспомните, что совместимая с CLS сборка может быть задействована во всех языках программирования .NET. Если в открытых типах создаются открытые члены, которые предоставляют доступ к несовместимым с CLS конструкциям программирования (таким как данные без знака или параметры типа указателей), то другие языки .NET могут оказаться не в состоянии работать с ними. Следовательно, при построении библиотек кода C#, которые должны применяться во множестве языков .NET, проверка на совместимость с CLS является обязательной.

Для этого нужно просто добавить в начало файла исходного кода C# показанный ниже атрибут уровня сборки. Имейте в виду, что все атрибуты уровня сборки или модуля должны быть указаны за пределами области действия любого пространства имен! При добавлении в проект атрибутов уровня сборки или модуля рекомендуется придерживаться следующей схемы для файла кода:

```
// Первыми перечислить операторы using.
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

// Теперь перечислить атрибуты уровня сборки или модуля.
// Обеспечить совместимость с CLS для всех открытых типов в данной сборке.
```

```
[assembly: CLSCompliant(true)]  
  
// Далее может следовать ваше пространство (пространства) имен и типы.  
namespace AttributedCarLibrary  
{  
    // Типы...  
}
```

Если теперь добавить фрагмент кода, выходящий за рамки спецификации CLS (вроде открытого элемента данных без знака):

```
// Тип ulong не соответствует спецификации CLS.  
public class Winnebago  
{  
    public ulong notCompliant;  
}
```

тогда компилятор выдаст предупреждение.

Файл AssemblyInfo.cs, генерируемый Visual Studio

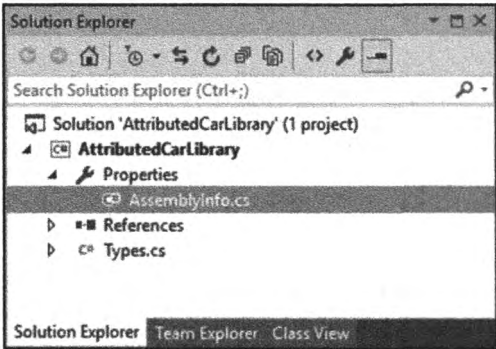


Рис. 15.6. Файл AssemblyInfo.cs

По умолчанию проекты Visual Studio получают файл по имени AssemblyInfo.cs, который можно просмотреть, раскрыв в окне Solution Explorer узел Properties (Свойства), как показано на рис. 15.6.

Файл AssemblyInfo.cs является удобным местом для размещения атрибутов, подлежащих применению на уровне сборки. Во время исследования сборок .NET в главе 14 рассказывалось о том, что в манифесте содержатся метаданные уровня сборки, большая часть которых берется из атрибутов уровня сборки, описанных в табл. 15.4.

Таблица 15.4. Избранные атрибуты уровня сборки

Атрибут	Описание
[AssemblyCompany]	Хранит общую информацию о компании
[AssemblyCopyright]	Хранит любую информацию, касающуюся авторских прав на продукт или сборку
[AssemblyCulture]	Предоставляет информацию о том, какие культуры или языки поддерживает сборка
[AssemblyDescription]	Хранит дружественное описание продукта или модулей, из которых состоит сборка
[AssemblyKeyFile]	Указывает имя файла, в котором содержится пара ключей, используемых для подписания сборки (т.е. создания строгого имени)
[AssemblyProduct]	Предоставляет информацию о продукте
[AssemblyTrademark]	Предоставляет информацию о торговой марке
[AssemblyVersion]	Указывает информацию о версии сборки в формате <старшийНомер.младшийНомер.номерСборки.номерРедакции>

Исходный код. Проект `AttributedCarLibrary` доступен в подкаталоге `Chapter_15`.

Рефлексия атрибутов с использованием раннего связывания

Вспомните, что атрибуты остаются бесполезными до тех пор, пока к их значениям не будет применена рефлексия в другой части программного обеспечения. После обнаружения атрибута другая часть кода может предпринять необходимый образ действий. Подобно любому приложению "другая часть программного обеспечения" может обнаруживать присутствие специального атрибута с использованием либо раннего, либо позднего связывания. Для применения раннего связывания определение интересующего атрибута (в данном случае `VehicleDescriptionAttribute`) должно находиться в клиентском приложении на этапе компиляции. Учитывая то, что специальный атрибут определен в сборке `AttributedCarLibrary` как открытый класс, раннее связывание будет наилучшим выбором.

Чтобы проиллюстрировать процесс рефлексии специальных атрибутов, создадим новый проект консольного приложения по имени `VehicleDescriptionAttributeReader`, добавим в него ссылку на сборку `AttributedCarLibrary` и поместим в первоначальный файл `*.cs` следующий код:

```
// Выполнение рефлексии атрибутов с использованием раннего связывания.
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using AttributedCarLibrary;

namespace VehicleDescriptionAttributeReader
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
            ReflectOnAttributesUsingEarlyBinding();
            Console.ReadLine();
        }

        private static void ReflectOnAttributesUsingEarlyBinding()
        {
            // Получить объект Type, представляющий тип Winnebago.
            Type t = typeof(Winnebago);

            // Получить все атрибуты Winnebago.
            object[] customAtts = t.GetCustomAttributes(false);

            // Вывести описание.
            foreach (VehicleDescriptionAttribute v in customAtts)
                Console.WriteLine("-> {0}\n", v.Description);
        }
    }
}
```

Метод `Type.GetCustomAttributes()` возвращает массив объектов со всеми атрибутами, примененными к члену, который представлен объектом `Type` (булевский па-

раметр управляет тем, должен ли поиск распространяться вверх по цепочке наследования). После получения списка атрибутов осуществляется проход по всем элементам `VehicleDescriptionAttribute` с отображением значения свойства `Description`.

Исходный код. Проект `VehicleDescriptionAttributeReader` доступен в подкаталоге `Chapter_15`.

Рефлексия атрибутов с использованием позднего связывания

В предыдущем примере для вывода описания транспортного средства типа `Winnebago` применялось ранее связывание. Это было возможно благодаря тому, что тип класса `VehicleDescriptionAttribute` определен в сборке `AttributedCarLibrary` как открытый член. Для рефлексии атрибутов также допускается использовать динамическую загрузку и позднее связывание.

Создадим новый проект консольного приложения по имени `VehicleDescriptionAttributeReaderLateBinding` и скопируем сборку `AttributedCarLibrary.dll` в каталог `bin\Debug` проекта. Теперь модифицируем класс `Program`, как показано ниже:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;

namespace VehicleDescriptionAttributeReaderLateBinding
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
            ReflectAttributesUsingLateBinding();
            Console.ReadLine();
        }

        private static void ReflectAttributesUsingLateBinding()
        {
            try
            {
                // Загрузить локальную копию сборки AttributedCarLibrary.
                Assembly asm = Assembly.Load("AttributedCarLibrary");

                // Получить информацию о типе VehicleDescriptionAttribute.
                Type vehicleDesc =
                    asm.GetType("AttributedCarLibrary.VehicleDescriptionAttribute");

                // Получить информацию о типе свойства Description.
                PropertyInfo propDesc = vehicleDesc.GetProperty("Description");

                // Получить все типы в сборке.
                Type[] types = asm.GetTypes();

                // Пройти по всем типам и получить любые атрибуты
                // VehicleDescriptionAttribute.
                foreach (Type t in types)
```

```
{
    object[] objs = t.GetCustomAttributes(vehicleDesc, false);
    // Пройти по каждому VehicleDescriptionAttribute и вывести
    // описание, используя позднее связывание.
    foreach (object o in objs)
    {
        Console.WriteLine("-> {0}: {1}\n",
            t.Name, propDesc.GetValue(o, null));
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
}
```

Если вы прорабатывали примеры, рассмотренные ранее в главе, тогда приведенный код должен быть более или менее понятен. Единственный интересный момент здесь связан с применением метода `PropertyInfo.GetValue()`, который служит для активизации средства доступа к свойству. Вот как выглядит вывод, полученный в результате выполнения текущего примера:

```
***** Value of VehicleDescriptionAttribute *****
-> Motorcycle: My rocking Harley
-> HorseAndBuggy: The old gray mare, she ain't what she used to be...
-> Winnebago: A very long, slow, but feature-rich auto
```

Исходный код. Проект `VehicleDescriptionAttributeReaderLateBinding` доступен в подкаталоге `Chapter_15`.

Практическое использование рефлексии, позднего связывания и специальных атрибутов

Хотя вы видели многочисленные примеры применения этих приемов, вас по-прежнему может интересовать, когда использовать рефлексия, динамическое связывание и специальные атрибуты в своих программах. Действительно, данные темы могут показаться как относящиеся в большей степени к академической стороне программирования (что в зависимости от вашей точки зрения может быть как отрицательным, так и положительным моментом). Для содействия в отображении указанных тем на реальные ситуации необходим более серьезный пример. Предположим, что вы работаете в составе команды программистов, которая занимается построением приложения со следующим требованием:

- продукт должен быть расширяемым за счет использования дополнительных сторонних инструментов.

Что понимается под *расширяемостью*? Возьмем IDE-среду Visual Studio. Когда это приложение разрабатывалось, в его кодовую базу были вставлены многочисленные “зацепки”, чтобы позволить другим производителям программного обеспечения “привязывать” (или подключать) специальные модули к IDE-среде. Очевидно, что у разработчи-

ков Visual Studio отсутствовал какой-либо способ установки ссылок на внешние сборки .NET, которые на тот момент еще не были созданы (таким образом, раннее связывание недоступно), тогда как они обеспечили наличие в приложении необходимых зацепок? Ниже представлен один из возможных способов решения задачи.

1. Во-первых, расширяемое приложение должно предоставлять некоторый механизм ввода, позволяющий пользователю указать модуль для подключения (наподобие диалогового окна или флага командной строки). Это требует динамической загрузки.
2. Во-вторых, расширяемое приложение должно иметь возможность выяснять, поддерживает ли модуль корректную функциональность (такую как набор обязательных интерфейсов), необходимую для его подключения к среде. Это требует рефлексии.
3. В-третьих, расширяемое приложение должно получать ссылку на требуемую инфраструктуру (вроде набора интерфейсных типов) и вызывать члены для запуска лежащей в основе функциональности. Это может требовать позднего связывания.

Попросту говоря, если расширяемое приложение изначально было запрограммировано для запрашивания специфических интерфейсов, то оно в состоянии выяснять во время выполнения, может ли активизироваться интересующий тип. После успешного прохождения такой проверки тип может поддерживать дополнительные интерфейсы, которые формируют полиморфную фабрику его функциональности. Именно этот подход был принят командой разработчиков Visual Studio, и вопреки тому, что вы могли подумать, в нем нет ничего сложного!

Построение расширяемого приложения

В последующих разделах будет рассмотрен пример создания расширяемого приложения, которое может быть дополнено функциональностью внешних сборок. Расширяемое приложение образовано из следующих сборок.

- `CommonSnappableTypes.dll`. Эта сборка содержит определения типов, которые будут использоваться каждым объектом оснастки, и на нее будет напрямую ссылаться расширяемое приложение.
- `CSharpSnapIn.dll`. Оснастка, написанная на C#, в которой задействованы типы из сборки `CommonSnappableTypes.dll`.
- `VbSnapIn.dll`. Оснастка, написанная на Visual Basic, в которой применяются типы из сборки `CommonSnappableTypes.dll`.
- `MyExtendableApp.exe`. Консольное приложение, которое может быть расширено функциональностью каждой оснастки.

В приложении будут использоваться динамическая загрузка, рефлексия и позднее связывание для динамического получения функциональности сборок, о которых заранее ничего не известно.

На заметку! Вы можете подумать о том, что вам вряд ли будет ставиться задача построения консольного приложения, и тут вы совершенно правы! Подавляющее большинство пользовательских приложений, создаваемых с помощью C#, являются либо развитыми клиентами (Windows Forms или WPF), либо веб-приложениями (ASP.NET Web Forms или ASP.NET MVC). Мы применяем консольные приложения, чтобы дать возможность вам как читателю сосредоточить внимание на специфических концепциях примеров, в данном случае — на динамической загрузке, рефлексии и позднем связывании. Позже в книге вы узнаете, как строить “реальные” пользовательские приложения с использованием WPF и ASP.NET MVC.

Построение мультипроектного решения **ExtendableApp**

Ранее в книге каждое приложение было автономным проектом. Мы поступали так для того, чтобы сохранять примеры простыми и четко ориентированными на демонстрируемые в них аспекты. Однако в реальном процессе разработки очень часто приходится иметь дело сразу с несколькими проектами. Среда Visual Studio спроектирована с учетом такого требования, и в рассматриваемом примере мы покажем, как работать одновременно с множеством проектов. Все создаваемые до сих пор проекты также включали решение, охватывающее проект. По умолчанию имя решения совпадает с именем первого проекта, но это вовсе не обязательно (и почти никогда не происходит).

Чтобы создать решение **ExtendableApp**, выберем в меню **File** (Файл) пункт **New Project** (Создать проект). В открывшемся диалоговом окне **New Project** (Новый проект) выберем элемент **Class Library** (Библиотека классов) и в поле **Name** (Имя) введем **CommonSnappableTypes**. Прежде чем щелкнуть на кнопке **OK**, в поле **Solution name** (Имя решения) введем **ExtendableApp** (рис. 15.7).

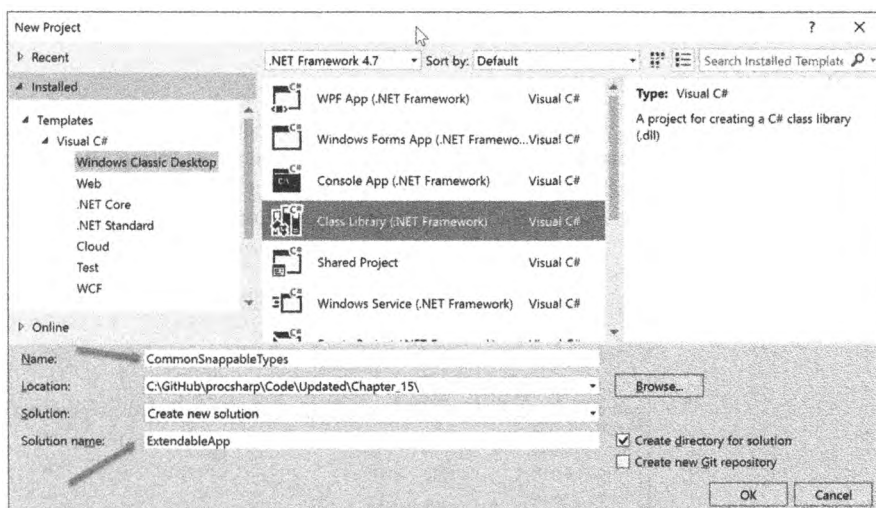


Рис. 15.7. Создание проекта **CommonSnappableTypes** и решения **ExtendableApp**

Далее в примере будет показано, как добавлять дополнительные проекты в решение **ExtendableApp**, указывать ссылки на проекты, устанавливать действия по построению и определять запускаемый проект решения.

Построение сборки **CommonSnappableTypes.dll**

Удалим стандартный файл **Class1.cs** и добавим новый файл по имени **SnappableTypes.cs**. В проекте библиотеки классов **CommonSnappableTypes** определены следующие два типа:

```
namespace CommonSnappableTypes
{
    public interface IAppFunctionality
    {
        void DoIt();
    }

    [AttributeUsage(AttributeTargets.Class)]
```

```
public sealed class CompanyInfoAttribute : System.Attribute
{
    public string CompanyName { get; set; }
    public string CompanyUrl { get; set; }
}
}
```

Интерфейс `IAppFunctionality` предоставляет полиморфный интерфейс для всех оснасток, которые могут потребляться расширяемым приложением. Учитывая, что рассматриваемый пример является полностью иллюстративным, в интерфейсе определен единственный метод под названием `DoIt()`. Более реалистичный интерфейс (или набор интерфейсов) мог бы позволять объекту генерировать код сценария, визуализировать изображение в панели инструментов приложения либо интегрироваться в главное меню размещающего приложения.

Тип `CompanyInfoAttribute` — это специальный атрибут, который может применяться к любому классу, желающему подключиться к контейнеру. Как несложно заметить по определению класса, `[CompanyInfo]` позволяет разработчику оснастки указывать общие сведения о месте происхождения компонента.

Добавление проектов в решение

Далее необходимо создать тип, который реализует интерфейс `IAppFunctionality`. Чтобы добавить в решение еще один проект, щелчком правой кнопкой мыши на элементе `ExtendableApp` в окне `Solution Explorer` (или выберем в меню `File` пункт `Add⇒New Project` (Добавить⇒Новый проект)) и выберем в меню `Add` (Добавить) пункт `New Project` (рис. 15.8).

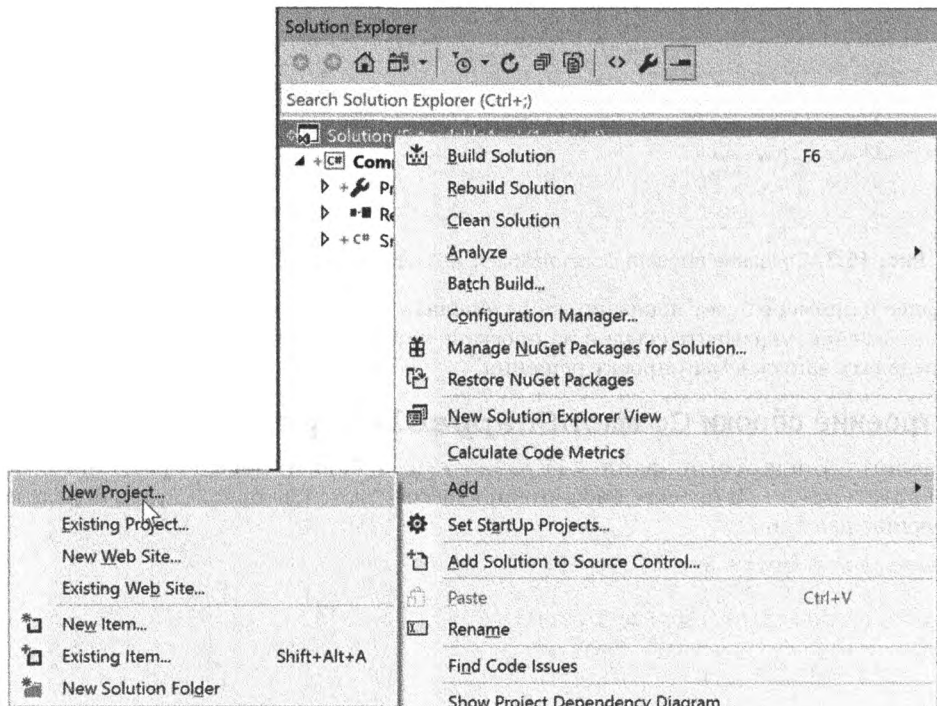


Рис. 15.8. Добавление еще одного проекта в решение

Открывшееся диалоговое окно Add New Project (Добавление нового проекта) не содержит полей ввода, относящихся к решению (рис. 15.9).

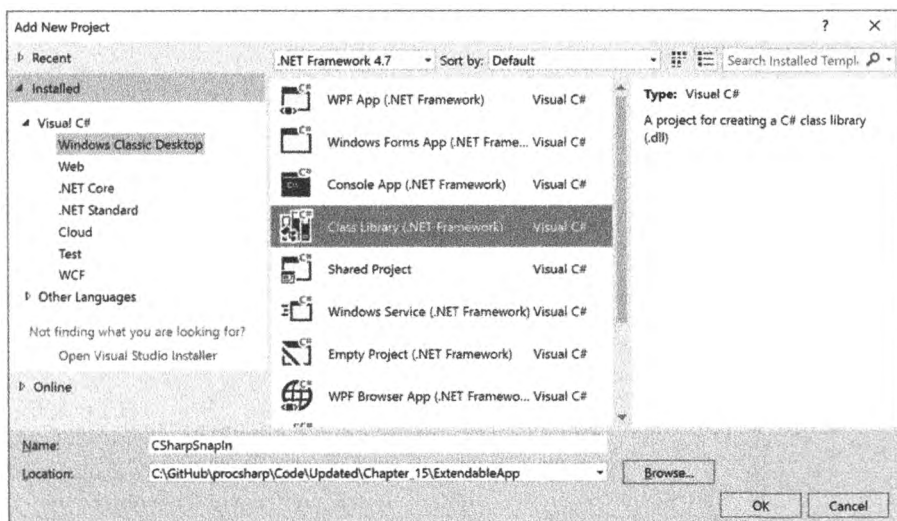


Рис. 15.9. Добавление проекта в существующее решение с помощью диалогового окна Add New Project

Назначим проекту библиотеки классов имя CSharpSnapIn и щелкнем на кнопке ОК.

Добавление ссылок на проекты

В создаваемой библиотеке классов определен класс по имени CSharpModule. Поскольку данный класс должен потреблять типы, определенные в CommonSnappableTypes, необходимо добавить ссылку на проект CommonSnappableTypes (не на сборку). Фактическая ссылка указывает на скомпилированную сборку, но об этом позаботится Visual Studio.

Чтобы добавить ссылку на проект, щелкнем правой кнопкой мыши на узле References (Ссылки) в окне Solution Explorer и выберем из контекстного меню пункт Add Reference (Добавить ссылку). В левой части окна Reference Manager (Диспетчер ссылок) выберем элемент Projects⇒Solution (Проекты⇒Решение), что приведет к выводу списка всех проектов внутри решения (в текущий момент есть только один проект), как показано на рис. 15.10. Выберем проект CommonSnappableTypes в качестве ссылки и щелкнем на кнопке ОК.

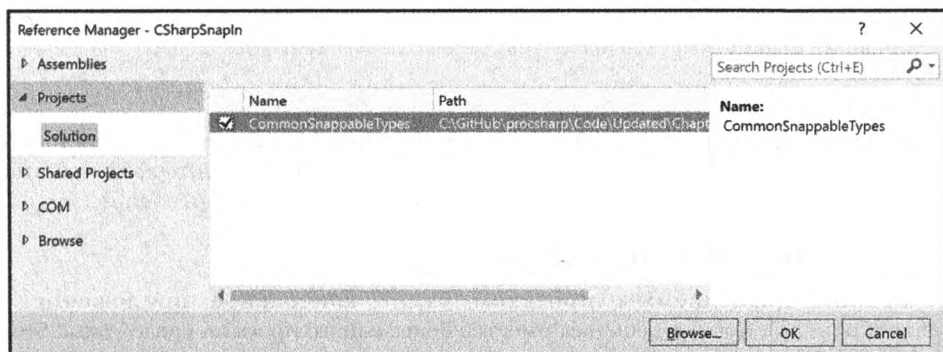


Рис. 15.10. Добавление ссылки на проект

Построение оснастки на C#

Имея ссылку, удалим файл `Class1.cs` из проекта, добавим новый файл по имени `CSharpModule.cs` и поместим в него следующий код (не забыв об операторе `using` для `CommonSnappableTypes`):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using CommonSnappableTypes;

namespace CSharpSnapIn
{
    [CompanyInfo(CompanyName = "FooBar", CompanyUrl = "www.FooBar.com")]
    public class CSharpModule : IAppFunctionality
    {
        void IAppFunctionality.DoIt()
        {
            Console.WriteLine("You have just used the C# snap-in");
        }
    }
}
```

Обратите внимание на явную реализацию интерфейса `IAppFunctionality` (см. главу 9). Это не является требованием; тем не менее, идея заключается в том, что единственной частью системы, которая нуждается в прямом взаимодействии с упомянутым интерфейсным типом, будет размещающее приложение. Благодаря явной реализации интерфейса `IAppFunctionality` метод `DoIt()` не доступен напрямую из типа `CSharpModule`.

Построение оснастки на Visual Basic

Для эмуляции стороннего производителя, предпочитающего работать с языком Visual Basic, а не C#, добавим новую библиотеку классов на Visual Basic (`VbSnapIn`), который ссылается на проект `CommonSnappableTypes` в точности, как предыдущий проект `CSharpSnapIn`. Удалим файл `Class1.vb` и добавим новый файл по имени `VbSnapIn.vb`.

Код Visual Basic столь же прост:

```
Imports CommonSnappableTypes

<CompanyInfo(CompanyName:="Chucky's Software", CompanyUrl:="www.ChuckySoft.com")>
Public Class VbSnapIn
    Implements IAppFunctionality

    Public Sub DoIt() Implements CommonSnappableTypes.IAppFunctionality.DoIt
        Console.WriteLine("You have just used the VB snap in")
    End Sub
End Class
```

Как видите, применение атрибутов в синтаксисе Visual Basic требует указания угловых (<>), а не квадратных ([]) скобок. Кроме того, для реализации интерфейсных типов заданным классом или структурой используется ключевое слово `Implements`.

Установка запускаемого проекта

Последним добавляемым проектом будет консольное приложение `C# (MyExtendableApp)`. Когда в решении имеется более одного проекта, среде Visual Studio нужно сообщить, какой проект (или проекты; мультипроектные решения начнут рас-

смастиваться позже в книге) должен запускаться по щелчку на кнопке Run (Выполнить). По умолчанию это проект, добавленный в решение первым. Проект MyExtendableApp понадобится сделать первым.

Установить запускаемый проект легко. Необходимо щелкнуть правой кнопкой мыши на проекте MyExtendableApp в окне Solution Explorer и выбрать в контекстном меню пункт Set as StartUp Project (Установить как запускаемый проект), как показано на рис. 15.11.

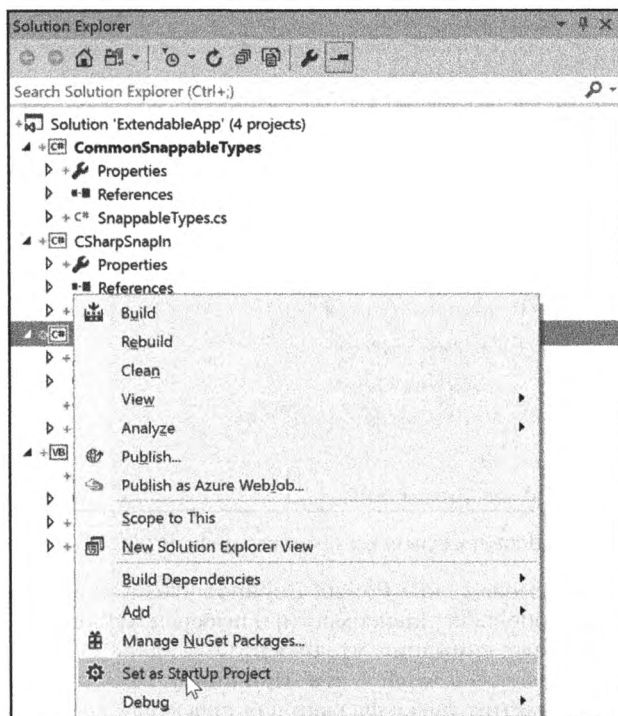


Рис. 15.11. Установка запускаемого проекта

На заметку! Если щелкнуть правой кнопкой мыши не на одном из проектов, а на решении ExtendableApp, тогда в контекстном меню будет отображаться пункт Set StartUp Projects (Установить как запускаемые проекты). В дополнение к запуску только одного проекта при щелчке на Run можно настроить выполнение множества проектов. Такой прием может быть полезен, если решение содержит проект службы REST, созданной посредством Web API, и веб-приложение, которое потребляет эту службу. Для выполнения веб-приложения служба REST также должна быть запущена, что демонстрируется в последующих главах.

Установка порядка построения проектов

После добавления в решение консольного приложения MyExtendableApp и его установки как запускаемого проекта добавим ссылку на проект CommonSnappableTypes, но не на проект CSharpSnapIn.dll или VbSnapIn.dll.

Когда среде Visual Studio выдается команда выполнить решение, запускаемые проекты и все ссылаемые проекты строятся в случае обнаружения любых изменений; однако проекты, ссылки на которые отсутствуют, строиться не будут. Это можно изменить за

счет настройки зависимостей между проектами с помощью пункта контекстного меню Project Build Order (Порядок построения проектов). Оно доступно через щелчок правой кнопкой мыши на имени решения (рис. 15.12).

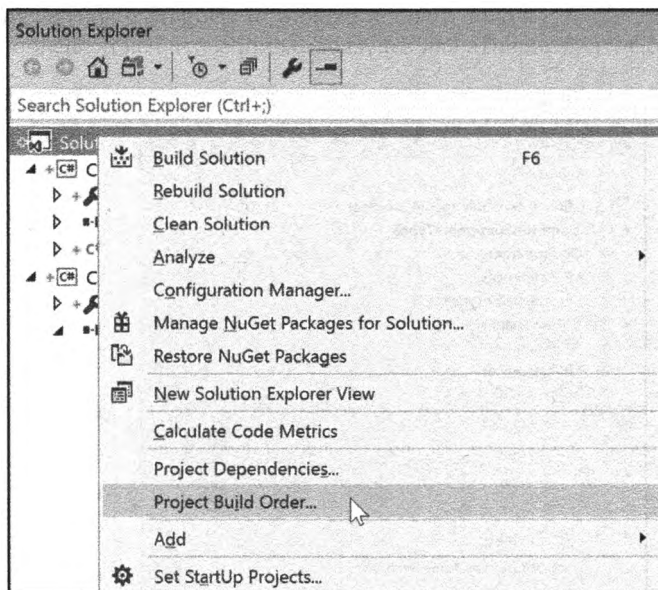


Рис. 15.12. Доступ к пункту контекстного меню Project Build Order

В открывшемся диалоговом окне Project Dependencies (Зависимости проектов) перейдем на вкладку Dependencies (Зависимости) и выберем MyExtendableApp в раскрывающемся списке. Обратите внимание, что проект CommonSnappableTypes уже выбран и соответствующий ему флажок недоступен. Причина в том, что на него производится ссылка напрямую. Отметим также флажки для проектов CSharpSnapIn и VbSnapIn (рис. 15.13).

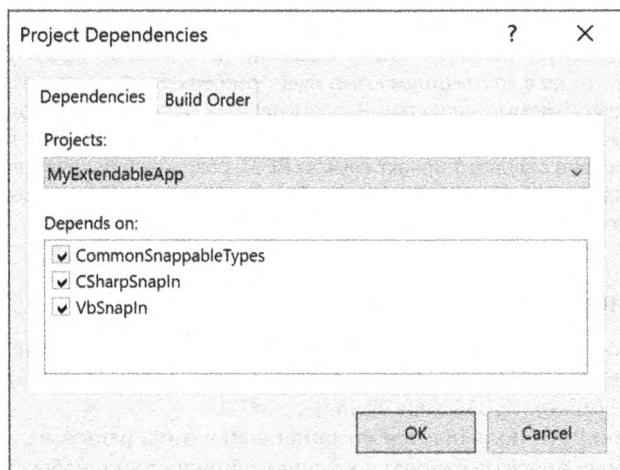


Рис. 15.13. Настройка зависимостей проектов

Теперь каждый раз, когда строится проект `MyExtendableApp`, будут также строиться проекты `CSharpSnapIn` и `VbSnapIn`.

Построение расширяемого консольного приложения

Располагая всей инфраструктурой, самое время заняться созданием расширяемого приложения. Вспомните, что цель этого приложения заключается в применении позднего связывания и рефлексии для выяснения "подключаемости" независимых двоичных файлов, созданных другими производителями.

Добавим в начало файла `Program.cs` операторы `using` для пространств имен `System.Reflection`, `System.Windows.Forms`, `System.IO` и `CommonSnappableTypes`. Далее добавим метод `LoadSnapin()`, который создает диалоговое окно открытия файла, предлагает пользователю выбрать файл и отправляет выбранный файл методу `LoadExternalModule()` на обработку (предполагается, что данный метод находится не в сборке `CommonSnappableTypes.dll`).

```
static void LoadSnapin()
{
    // Предоставить пользователю возможность выбора сборки для загрузки.
    OpenFileDialog dlg = new OpenFileDialog
    {
        // Установить в качестве начального каталога путь к текущему проекту.
        InitialDirectory =
            Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location),
        Filter = "assemblies (*.dll)|*.dll|All files (*.*)|*.*",
        FilterIndex = 1
    };
    if (dlg.ShowDialog() != DialogResult.OK)
    {
        // Пользователь закрыл диалоговое окно, не выбирая сборку.
        Console.WriteLine("User cancelled out of the open file dialog.");
        return;
    }
    if (dlg.FileName.Contains("CommonSnappableTypes"))
        // В CommonSnappableTypes нет оснасток.
        Console.WriteLine("CommonSnappableTypes has no snap-ins!");
    else if (!LoadExternalModule(dlg.FileName))
        // Интерфейс IAppFunctionality не реализован.
        Console.WriteLine("Nothing implements IAppFunctionality!");
}
```

Метод `LoadExternalModule()` выполняет следующие действия:

- динамически загружает в память выбранную сборку;
- определяет, содержит ли сборка типы, реализующие интерфейс `IAppFunctionality`;
- создает экземпляр типа, используя позднее связывание.

Если обнаружен тип, реализующий `IAppFunctionality`, тогда вызывается метод `DoIt()` и найденный тип передается методу `DisplayCompanyData()` для вывода дополнительной информации о нем посредством рефлексии.

```
private static bool LoadExternalModule(string path)
{
    bool foundSnapIn = false;
    Assembly theSnapInAsm = null;
```



```

try
{
    // Динамически загрузить выбранную сборку.
    theSnapInAsm = Assembly.LoadFrom(path);
}
catch (Exception ex)
{
    // Ошибка при загрузке оснастки.
    Console.WriteLine($"An error occurred loading the snapin: {ex.Message}");
    return foundSnapIn;
}

// Получить все совместимые с IAppFunctionality классы в сборке.
var theClassTypes = from t in theSnapInAsm.GetTypes()
    where t.IsClass && (t.GetInterface("IAppFunctionality") != null)
    select t;

// Создать объект и вызвать метод DoIt().
foreach (Type t in theClassTypes)
{
    foundSnapIn = true;
    // Использовать позднее связывание для создания экземпляра типа.
    IAppFunctionality itfApp =
        (IAppFunctionality) theSnapInAsm.CreateInstance(t.FullName, true);
    itfApp?.DoIt();
    // Отобразить информацию о компании.
    DisplayCompanyData(t);
}
return foundSnapIn;
}

```

Финальная задача связана с отображением метаданных, предоставляемых атрибутом [CompanyInfo]. Создадим метод DisplayCompanyData(), который принимает параметр System.Type:

```

private static void DisplayCompanyData(Type t)
{
    // Получить данные [CompanyInfo].
    var compInfo = from ci in t.GetCustomAttributes(false)
        where (ci is CompanyInfoAttribute)
        select ci;

    // Отобразить данные.
    foreach (CompanyInfoAttribute c in compInfo)
    {
        Console.WriteLine($"More info about {c.CompanyName} can be found at {c.CompanyUrl}");
    }
}

```

Наконец, модифицируем метод Main() следующим образом (обратите внимание на добавленный в начало метода атрибут):

```

[STAThread]
static void Main(string[] args)
{
    Console.WriteLine("***** Welcome to MyTypeViewer *****");
    do
    {

```

```
// Запросить о необходимости загрузки оснастки.  
Console.WriteLine("\nWould you like to load a snapin? [Y,N]");  
// Получить имя типа.  
string answer = Console.ReadLine();  
// Желает ли пользователь завершить работу?  
if ('answer.Equals("Y", StringComparison.OrdinalIgnoreCase))  
{  
    break;  
}  
// Попытаться отобразить тип.  
try  
{  
    LoadSnapin();  
}  
catch (Exception ex)  
{  
    // Найти оснастку не удалось.  
    Console.WriteLine("Sorry, can't find snapin");  
}  
}  
while (true);  
}
```

Атрибут `STAThread` необходим для диалогового окна открытия файла; все Windows-приложения с графическим пользовательским интерфейсом являются однопоточными (в плане пользовательского интерфейса), как обсуждается в последующих главах. Пока достаточно знать, что для работы с диалоговым окном открытия файла процесс должен быть помечен атрибутом `STAThread`.

Итак, создание примера расширяемого приложения завершено. Вы смогли увидеть, что представленные в главе приемы могут оказаться весьма полезными, и их применение не ограничивается только разработчиками инструментов.

Исходный код. Решение `ExtendableApp` доступно в подкаталоге `Chapter_15`.

Резюме

Рефлексия является интересным аспектом надежной объектно-ориентированной среды. В мире .NET службы рефлексии вращаются вокруг класса `System.Type` и пространства имен `System.Reflection`. Вы видели, что рефлексия — это процесс помещения типа под “увеличительное стекло” во время выполнения с целью выяснения его характеристик и возможностей.

Позднее связывание представляет собой процесс создания экземпляра типа и обращения к его членам без предварительного знания имен членов типа. Позднее связывание часто является прямым результатом динамической загрузки, которая позволяет программным образом загружать сборку .NET в память. На примере построения расширяемого приложения было продемонстрировано, что это мощный прием, используемый разработчиками инструментов, а также их потребителями.

Кроме того, в главе была исследована роль программирования на основе атрибутов. Снабжение типов атрибутами приводит к дополнению метаданных лежащей в основе сборки.

ГЛАВА 16

Динамические типы и среда DLR

В версии .NET 4.0 язык C# получил новое ключевое слово `dynamic`, которое позволяет внедрять в строго типизированный мир безопасности к типам, точек с запятой и фигурных скобок поведение, характерное для сценариев. Используя такую слабую типизацию, можно значительно упростить решение ряда сложных задач кодирования и получить возможность взаимодействия с несколькими динамическими языками (вроде IronRuby и IronPython), которые поддерживают .NET.

В настоящей главе вы узнаете о ключевом слове `dynamic` и о том, как слабо типизированные вызовы отображаются на корректные объекты в памяти с применением исполняющей среды динамического языка (Dynamic Language Runtime — DLR). После освоения служб, предлагаемых средой DLR, вы увидите примеры использования динамических типов для облегчения вызова методов с поздним связыванием (через службы рефлексии) и простого взаимодействия с унаследованными библиотеками COM.

На заметку! Не путайте ключевое слово `dynamic` языка C# с концепцией *динамической сборки* (объясняемой в главе 18). Хотя ключевое слово `dynamic` может применяться при построении динамической сборки, все же это две совершенно независимые концепции.

Роль ключевого слова `dynamic` языка C#

В главе 3 вы ознакомились с ключевым словом `var`, которое позволяет объявлять локальные переменные таким способом, что их действительные типы данных определяются на основе начального присваивания во время компиляции (вспомните, что результат называется *неявной типизацией*). После того как начальное присваивание выполнено, вы имеете строго типизированную переменную, и любая попытка присвоить ей несовместимое значение приведет к ошибке на этапе компиляции.

Чтобы приступить к исследованию ключевого слова `dynamic` языка C#, создадим новый проект консольного приложения по имени `DynamicKeyword`. Добавим в класс `Program` следующий метод и удостоверимся, что финальный оператор кода действительно генерирует ошибку во время компиляции, если убрать символы комментария:

```
static void ImplicitlyTypedVariable()
{
    // Переменная a имеет тип List<int>.
    var a = new List<int> {90};
    // Этот оператор приведет к ошибке на этапе компиляции!
    // a = "Hello";
}
```

Использование неявной типизации лишь потому, что она возможна, некоторые считают плохим стилем (если известно, что необходима переменная типа `List<int>`, то так и следует ее объявлять). Однако, как было показано в главе 12, неявная типизация удобна в сочетании с LINQ, поскольку многие запросы LINQ возвращают перечисления анонимных классов (через проекции), которые напрямую объявлять в коде C# невозможно. Тем не менее, даже в таких случаях неявно типизированная переменная фактически будет строго типизированной.

В качестве связанного замечания: в главе 6 упоминалось, что `System.Object` является изначальным родительским классом внутри .NET Framework и может представлять все, что угодно. Опять-таки, объявление переменной типа `object` в результате дает строго типизированный фрагмент данных, но то, на что указывает эта переменная в памяти, может отличаться в зависимости от присваиваемой ссылки. Чтобы получить доступ к членам объекта, на который указывает ссылка в памяти, понадобится выполнить явное приведение.

Предположим, что есть простой класс по имени `Person`, в котором определены два автоматических свойства (`FirstName` и `LastName`), инкапсулирующие данные `string`. Взгляните на следующий код:

```
static void UseObjectVarible()
{
    // Пусть имеется класс по имени Person.
    object o = new Person() { FirstName = "Mike", LastName = "Larson" };

    // Для получения доступа к свойствам Person
    // переменную o потребуется привести к Person.
    Console.WriteLine("Person's first name is {0}", ((Person)o).FirstName);
}
```

В результате выхода версии .NET 4.0 в языке C# появилось ключевое слово `dynamic`. С высокоуровневой точки значения ключевое слово `dynamic` можно трактовать как специализированную форму типа `System.Object` — в том смысле, что переменной динамического типа данных может быть присвоено любое значение. На первый взгляд это может привести к серьезной путанице, поскольку теперь получается, что доступны три способа определения данных, внутренний тип которых явно не указан в кодовой базе. Например, следующий метод:

```
static void PrintThreeStrings()
{
    var s1 = "Greetings";
    object s2 = "From";
    dynamic s3 = "Minneapolis";

    Console.WriteLine("s1 is of type: {0}", s1.GetType());
    Console.WriteLine("s2 is of type: {0}", s2.GetType());
    Console.WriteLine("s3 is of type: {0}", s3.GetType());
}
```

в случае вызова в `Main()` приведет к такому выводу:

```
s1 is of type: System.String
s2 is of type: System.String
s3 is of type: System.String
```

Динамическая переменная и переменная, объявленная неявно или через ссылку на `System.Object`, существенно отличаются тем, что динамическая переменная не является строго типизированной. Выражаясь по-другому, динамические данные не типизированы статически. Для компилятора C# ситуация выглядит так, что элементу данных, объявленному с ключевым словом `dynamic`, можно присваивать вообще любое

начальное значение, и на протяжении периода его существования взамен начального значения может быть присвоено любое новое (возможно, не связанное) значение. Рассмотрим показанный ниже метод и результирующий вывод:

```
static void ChangeDynamicDataType()
{
    // Объявить одиночный динамический элемент данных по имени t.
    dynamic t = "Hello!";
    Console.WriteLine("t is of type: {0}", t.GetType());

    t = false;
    Console.WriteLine("t is of type: {0}", t.GetType());

    t = new List<int>();
    Console.WriteLine("t is of type: {0}", t.GetType());
}

t is of type: System.String
t is of type: System.Boolean
t is of type: System.Collections.Generic.List`1[System.Int32]
```

Имейте в виду, что приведенный выше код успешно скомпилировался и дал бы идентичный результат, если бы переменная `t` была объявлена с типом `System.Object`. Однако, как вскоре будет показано, ключевое слово `dynamic` предлагает много дополнительных возможностей.

Вызов членов на динамически объявленных данных

Учитывая то, что динамическая переменная способна принимать идентичность любого типа на лету (подобно переменной типа `System.Object`), у вас может возникнуть вопрос о способе обращения к членам такой переменной (свойствам, методам, индексаторам, событиям и т.п.). С точки зрения синтаксиса отличий нет. Нужно просто применить операцию точки к динамической переменной, указать открытый член и предоставить любые аргументы (если они требуются).

Но (и это очень важное “но”) допустимость указываемых членов компилятор проверять не будет! Вспомните, что в отличие от переменной, определенной с типом `System.Object`, динамические данные не являются статически типизированными. Вплоть до времени выполнения не будет известно, поддерживают ли вызываемые динамические данные указанный член, переданы ли корректные параметры, правильно ли записано имя члена, и т.д. Таким образом, хотя это может показаться странным, следующий метод благополучно скомпилируется:

```
static void InvokeMembersOnDynamicData()
{
    dynamic textData1 = "Hello";
    Console.WriteLine(textData1.ToUpper());

    // Здесь можно было бы ожидать ошибки на этапе компиляции!
    // Однако все компилируется нормально.
    Console.WriteLine(textData1.toupper());
    Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
}
```

Обратите внимание, что во втором вызове `WriteLine()` предпринимается попытка обращения к методу по имени `toupper()` на динамическом элементе данных (при записи имени метода использовался неправильный регистр символов; оно должно выглядеть как `ToUpper()`). Как видите, переменная `textData1` имеет тип `string`, а потому известно, что у этого типа отсутствует метод с именем, записанным полностью в нижнем

регистре. Более того, тип `string` определенно не имеет метода по имени `Foo()`, который принимает параметры `int`, `string` и объект `DateTime`!

Тем не менее, компилятор C# ни о каких ошибках не сообщает. Однако если вызвать метод `InvokeMembersOnDynamicData()` в `Main()`, то возникнет ошибка времени выполнения с примерно таким сообщением:

```
Unhandled Exception: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
'string' does not contain a definition for 'toupper'
```

Необработанное исключение: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: string не содержит определения для toupper

Другое очевидное отличие между обращением к членам динамических и строго типизированных данных связано с тем, что когда к элементу динамических данных применяется операция точки, ожидаемое средство IntelliSense среды Visual Studio не активизируется. Взамен IDE-среда позволит вводить любое имя члена, какое только может прийти вам на ум.

Отсутствие возможности доступа к средству IntelliSense для динамических данных должно быть понятным. Тем не менее, как вы наверняка помните, это означает необходимость соблюдения предельной аккуратности при наборе кода C# для таких элементов данных. Любая опечатка или символ неправильного регистра в имени члена приведет к ошибке времени выполнения, в частности к генерации исключения типа `RuntimeBinderException`.

Роль сборки `Microsoft.CSharp.dll`

Для каждого создаваемого проекта C# среда Visual Studio автоматически устанавливает ссылку на сборку по имени `Microsoft.CSharp.dll` (ее можно увидеть, заглянув в папку References (Ссылки) внутри окна Solution Explorer). В этой небольшой библиотеке определено единственное пространство имен (`Microsoft.CSharp.RuntimeBinder`) с двумя классами (рис. 16.1).

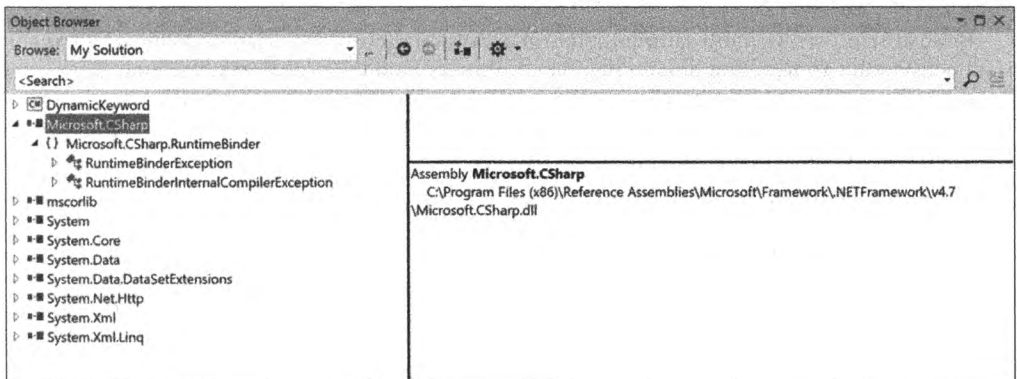


Рис. 16.1. Сборка `Microsoft.CSharp.dll`

Как можно догадаться по именам, классы являются строго типизированными исключениями. Более общий класс, `RuntimeBinderException`, представляет ошибку, которая будет сгенерирована при попытке обращения к несуществующему члену динамического типа данных (как в рассмотренной ситуации с методами `toupper()` и `Foo()`). Та же самая ошибка будет инициирована в случае указания некорректных данных параметров для члена, который существует.

Поскольку динамические данные настолько изменчивы, любые обращения к членам переменной, объявленной с ключевым словом `dynamic`, могут быть помещены внутрь подходящего блока `try/catch` для элегантной обработки ошибок:

```
static void InvokeMembersOnDynamicData()
{
    dynamic textData1 = "Hello";
    try
    {
        Console.WriteLine(textData1.ToUpper());
        Console.WriteLine(textData1.toupper());
        Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
    }
    catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Если вызвать метод `InvokeMembersOnDynamicData()` снова, то можно заметить, что вызов `ToUpper()` (обратите внимание на заглавные буквы "Т" и "U") работает корректно, но затем на консоль выводится сообщение об ошибке:

```
HELLO
'string' does not contain a definition for 'toupper'
string не содержит определение для toupper
```

Конечно, процесс помещения всех динамических обращений к методам в блоки `try/catch` довольно утомителен. Если вы тщательно следите за написанием кода и передачей параметров, тогда поступать так необязательно. Однако перехват исключений удобен, когда вы заранее не знаете, присутствует ли интересующий член в целевом типе.

Область применения ключевого слова `dynamic`

Вспомните, что неявно типизированные данные (объявленные с ключевым словом `var`) возможны только для локальных переменных в области действия члена. Ключевое слово `var` никогда не может использоваться с возвращаемым значением, параметром или членом класса/структуры. Тем не менее, это не касается ключевого слова `dynamic`. Взгляните на следующее определение класса:

```
class VeryDynamicClass
{
    // Динамическое поле.
    private static dynamic myDynamicField;

    // Динамическое свойство.
    public dynamic DynamicProperty { get; set; }

    // Динамический тип возврата и динамический тип параметра.
    public dynamic DynamicMethod(dynamic dynamicParam)
    {
        // Динамическая локальная переменная.
        dynamic dynamicLocalVar = "Local variable";
        int myInt = 10;
        if (dynamicParam is int)
        {
            return dynamicLocalVar;
        }
    }
}
```

```

    else
    {
        return myInt;
    }
}

```

Теперь можно было бы обращаться к открытым членам ожидаемым образом; однако при работе с динамическими методами и свойствами нет полной уверенности в том, каким будет тип данных! По правде говоря, определение `VeryDynamicClass` может оказаться не особенно полезным в реальном приложении, но оно иллюстрирует область, где допускается применять ключевое слово `dynamic`.

Ограничения ключевого слова `dynamic`

Невзирая на то, что с использованием ключевого слова `dynamic` можно определять разнообразные сущности, с ним связаны некоторые ограничения. Хотя они не настолько впечатляющие, следует помнить, что элементы динамических данных не могут применять лямбда-выражения или анонимные методы C# при вызове метода. Например, показанный ниже код всегда будет давать в результате ошибки, даже если целевой метод на самом деле принимает параметр типа делегата, который в свою очередь принимает значение `string` и возвращает `void`:

```

dynamic a = GetDynamicObject();

// Ошибка! Методы на динамических данных не могут использовать лямбда-выражения!
a.Method(arg => Console.WriteLine(arg));

```

Чтобы обойти упомянутое ограничение, понадобится работать с лежащим в основе делегатом напрямую, используя приемы из главы 10. Еще одно ограничение заключается в том, что динамический элемент данных не может воспринимать расширяющие методы (см. главу 11). К сожалению, сказанное касается также всех расширяющих методов из API-интерфейсов LINQ. Следовательно, переменная, объявленная с ключевым словом `dynamic`, имеет ограниченное применение в рамках LINQ to Objects и других технологий LINQ:

```

dynamic a = GetDynamicObject();

// Ошибка! Динамические данные не могут найти расширяющий метод Select()!
var data = from d in a select d;

```

Практическое использование ключевого слова `dynamic`

С учетом того, что динамические данные не являются строго типизированными, не проверяются на этапе компиляции, не имеют возможности запускать средство `IntelliSense` и не могут быть целью запроса LINQ, совершенно корректно предположить, что применение ключевого слова `dynamic` лишь по причине его существования представляет собой плохую практику программирования.

Тем не менее, в редких обстоятельствах ключевое слово `dynamic` может радикально сократить объем вводимого вручную кода. В частности, при построении приложения .NET, в котором интенсивно используется позднее связывание (через рефлекссию), ключевое слово `dynamic` может сэкономить время на наборе кода. Аналогично при разработке приложения .NET, которое должно взаимодействовать с унаследованными библиотеками COM (вроде тех, что входят в состав продуктов Microsoft Office), за счет использования ключевого слова `dynamic` можно значительно упростить кодовую базу.

В качестве финального примера можно привести веб-сайты, построенные с применением шаблона проектирования MVC: они часто используют тип `ViewBag`, к которому также допускается производить доступ в упрощенной манере с помощью ключевого слова `dynamic`.

Как с любым “сокращением”, прежде чем его использовать, необходимо взвесить все “за” и “против”. Применение ключевого слова `dynamic` — компромисс между краткостью кода и безопасностью к типам. В то время как C# в своей основе является строго типизированным языком, динамическое поведение можно задействовать (или нет) от вызова к вызову. Всегда помните, что использовать ключевое слово `dynamic` необязательно. Тот же самый конечный результат можно получить, написав альтернативный код вручную (правда, обычно намного большего объема).

Исходный код. Проект `DynamicKeyword` доступен в подкаталоге `Chapter_16`.

Роль исполняющей среды динамического языка

Теперь, когда вы лучше понимаете сущность “динамических данных”, давайте посмотрим, как их обрабатывать. Начиная с версии .NET 4.0 общезыковая исполняющая среда (`Common Language Runtime` — CLR) получила дополняющую среду времени выполнения, которая называется *исполняющей средой динамического языка* (`Dynamic Language Runtime` — DLR). Концепция “динамической исполняющей среды” определенно не нова. На самом деле ее много лет используют такие языки программирования, как JavaScript, LISP, Ruby и Python. Выражаясь кратко, динамическая исполняющая среда предоставляет динамическим языкам возможность обнаруживать типы полностью во время выполнения без каких-либо проверок на этапе компиляции.

Если у вас есть опыт работы со строго типизированными языками (включая C# без динамических типов), тогда идея такой исполняющей среды может показаться неподходящей. В конце концов, вы обычно хотите выявлять ошибки на этапе компиляции, а не во время выполнения, когда только возможно. Тем не менее, динамические языки и исполняющие среды предлагают ряд интересных средств, включая перечисленные ниже.

- Чрезвычайно гибкая кодовая база. Можно проводить рефакторинг кода, не внося многочисленных изменений в типы данных.
- Простой способ взаимодействия с разнообразными типами объектов, которые построены на разных платформах и языках программирования.
- Способ добавления или удаления членов типа в памяти во время выполнения.

Одна из задач среды DLR заключается в том, чтобы позволить различным динамическим языкам работать с исполняющей средой .NET и предоставлять им возможность взаимодействия с другим кодом .NET. Двумя популярными динамическими языками, которые используют DLR, являются IronPython и IronRuby. Указанные языки находятся в “динамической вселенной”, где типы определяются целиком во время выполнения. К тому же данные языки имеют доступ ко всему богатству библиотек базовых классов .NET. А еще лучше то, что благодаря наличию ключевого слова `dynamic` их кодовые базы могут взаимодействовать с языком C# (и наоборот).

На заметку! В настоящей главе вопросы применения среды DLR для интеграции с динамическими языками не обсуждаются.

Роль деревьев выражений

Для описания динамического вызова в нейтральных терминах среда DLR использует *деревья выражений*. Например, когда среда DLR встречает код C# вроде следующего:

```
dynamic d = GetSomeData();
d.SuperMethod(12);
```

она автоматически строит дерево выражения, которое по существу гласит: “Вызвать метод по имени *SuperMethod* на объекте *d* с передачей числа 12 в качестве аргумента”. Затем эта информация (формально называемая *полезной нагрузкой*) передается корректному связывателю времени выполнения, который может быть динамическим связывателем C#, динамическим связывателем IronPython или даже (как вскоре будет объяснено) унаследованным объектом COM.

Далее запрос отображается на необходимую структуру вызовов для целевого объекта. Деревья выражений обладают одной замечательной характеристикой (помимо того, что их не приходится создавать вручную): они позволяют писать фиксированный оператор кода C# и не беспокоиться о том, какой будет действительная цель (объект COM, кодовая база IronPython или IronRuby и т.д.). На рис. 16.2 проиллюстрирована высокоуровневая концепция деревьев выражений.

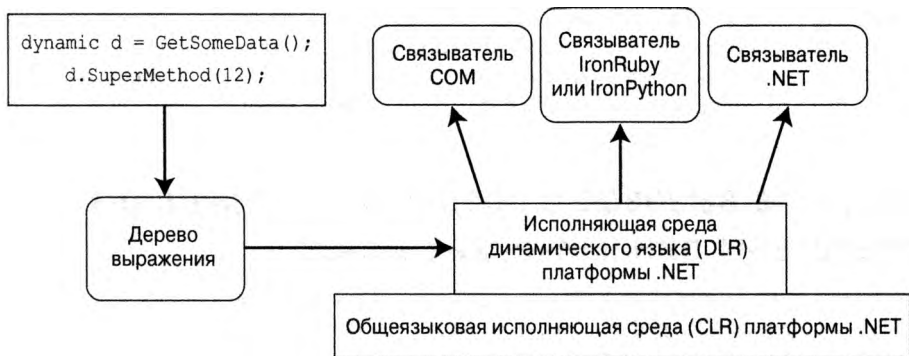


Рис. 16.2. Деревья выражений фиксируют динамические вызовы в нейтральных терминах и обрабатываются связывателями

Роль пространства имен *System.Dynamic*

Сборка *System.Core.dll* содержит пространство имен под названием *System.Dynamic*. По правде говоря, шансы, что вам когда-либо придется применять типы из этого пространства имен, весьма невелики. Однако если вы являетесь производителем языка и желаете обеспечить своему динамическому языку возможность взаимодействия со средой DLR, то сможете задействовать пространство имен *System.Dynamic* для построения специального связывателя времени выполнения.

Типы из пространства имен *System.Dynamic* здесь подробно не рассматриваются, но в случае заинтересованности обращайтесь в документацию *.NET Framework 4.7 SDK*. Для практических нужд просто знайте, что пространство имен *System.Dynamic* предоставляет необходимую инфраструктуру, которая делает динамический язык “осведомленным о платформе .NET”.

Динамический поиск в деревьях выражений во время выполнения

Как уже объяснялось, среда DLR будет передавать деревья выражений целевому объекту; тем не менее, на этот процесс отправки влияет несколько факторов. Если динамический тип данных указывает в памяти на объект COM, то дерево выражения отправляется реализации низкоуровневого интерфейса COM по имени `IDispatch`. Как вам может быть известно, упомянутый интерфейс представляет собой способ, которым COM внедряет собственный набор динамических служб. Однако объекты COM можно использовать в приложении .NET без применения DLR или ключевого слова `dynamic` языка C#. Тем не менее, такой подход (как вы увидите) сопряжен с написанием более сложного кода на C#.

Если динамические данные не указывают на объект COM, тогда дерево выражения может быть передано объекту, реализующему интерфейс `IDynamicObject`. Указанный интерфейс используется “за кулисами”, чтобы позволить языку вроде IronRuby принимать дерево выражения DLR и отображать его на специфические средства языка Ruby.

Наконец, если динамические данные указывают на объект, который не является объектом COM и не реализует интерфейс `IDynamicObject`, то это нормальный повседневный объект .NET. В таком случае дерево выражения передается на обработку связывателю исполняющей среды C#. Процесс отображения дерева выражений на специфические средства платформы .NET вовлекает в дело службы рефлексии.

После того как дерево выражения обработано определенным связывателем, динамические данные преобразуются в реальный тип данных в памяти, после чего вызывается корректный метод со всеми необходимыми параметрами. Теперь давайте рассмотрим несколько практических применений DLR, начав с упрощения вызовов .NET с поздним связыванием.

Упрощение вызовов с поздним связыванием посредством динамических типов

Одним из случаев, когда имеет смысл использовать ключевое слово `dynamic`, может быть работа со службами рефлексии, а именно — вызов методов с поздним связыванием. В главе 15 приводилось несколько примеров, когда вызовы методов такого рода могут быть полезными — чаще всего при построении расширяемого приложения. Там вы узнали, как применять метод `Activator.CreateInstance()` для создания объекта типа, о котором ничего не известно на этапе компиляции (помимо его отображаемого имени). Затем с помощью типов из пространства имен `System.Reflection` можно обращаться к членам объекта через механизм позднего связывания. Вспомните показанный ниже пример из главы 15:

```
static void CreateUsingLateBinding(Assembly asm)
{
    try
    {
        // Получить метаданные для типа MiniVan.
        Type miniVan = asm.GetType("CarLibrary.MiniVan");
        // Создать экземпляр MiniVan на лету.
        object obj = Activator.CreateInstance(miniVan);
        // Получить информацию о TurboBoost.
        MethodInfo mi = miniVan.GetMethod("TurboBoost");
        // Вызвать метод (null означает отсутствие параметров).
        mi.Invoke(obj, null);
    }
}
```

```

catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
}

```

В то время как приведенный код функционирует ожидаемым образом, нельзя не отметить его некоторую громоздкость. Здесь приходится вручную работать с классом `MethodInfo`, вручную запрашивать метаданные и т.д. В следующей версии того же метода используется ключевое слово `dynamic` и среда DLR:

```

static void InvokeMethodWithDynamicKeyword(Assembly asm)
{
    try
    {
        // Получить метаданные для типа Minivan.
        Type miniVan = asm.GetType("CarLibrary.Minivan");

        // Создать экземпляр Minivan на лету и вызвать метод.
        dynamic obj = Activator.CreateInstance(miniVan);
        obj.TurboBoost();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

```

За счет объявления переменной `obj` с ключевым словом `dynamic` вся рутинная работа, связанная с рефлексией, перекладывается на DLR.

Использование ключевого слова `dynamic` для передачи аргументов

Полезность среды DLR становится еще более очевидной, когда нужно выполнять вызовы методов с поздним связыванием, которые принимают параметры. В случае применения “многословных” обращений к рефлексии аргументы нуждаются в упаковке внутрь массива экземпляров `object`, который передается методу `Invoke()` класса `MethodInfo`.

Чтобы проиллюстрировать использование на примере, создадим новый проект консольного приложения C# по имени `LateBindingWithDynamic`. Добавим к текущему решению проект библиотеки классов (с помощью пункта меню `File⇒Add⇒New Project (Файл⇒Добавить⇒Новый проект)`) под названием `MathLibrary` (удостоверившись в том, что `LateBindingWithDynamic` — по-прежнему запускаемый проект решения). Переименуем первоначальный файл `Class1.cs` в проекте `MathLibrary` на `SimpleMath.cs` и реализуем класс, как показано ниже:

```

public class SimpleMath
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}

```

После компиляции сборки `MathLibrary.dll` поместим ее копию в подкаталог `bin\Debug` проекта `LateBindingWithDynamic`. (Щелкнув на кнопке `Show All Files` (Показать все файлы) для нужных проектов в окне `Solution Explorer`, можно просто перетащить файл из одного проекта в другой). В данный момент окно `Solution Explorer` должно выглядеть так, как показано на рис. 16.3.

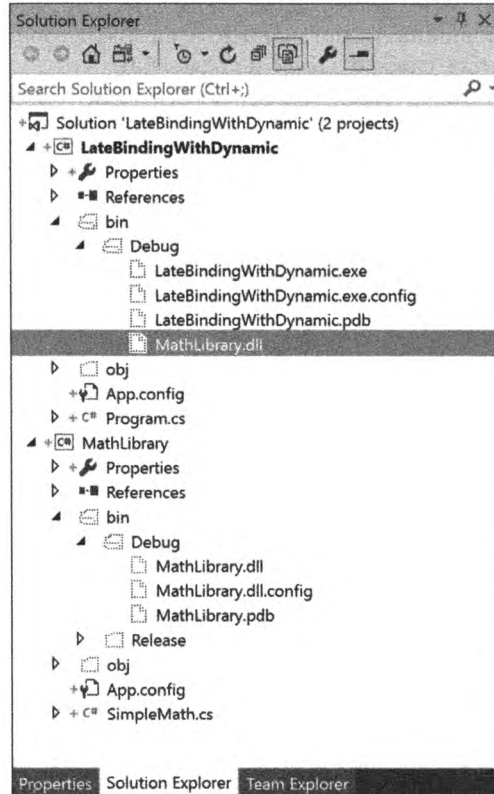


Рис. 16.3. Проект `LateBindingWithDynamic` содержит закрытую копию сборки `MathLibrary.dll`

На заметку! Помните, что главная цель позднего связывания — позволить приложению создать объект типа, для которого не предусмотрено записи в манифесте. Именно потому требуется вручную копировать сборку `MathLibrary.dll` в выходную папку консольного приложения C#, а не устанавливать ссылку на проект в Visual Studio.

Теперь импортируем пространство имен `System.Reflection` в файл `Program.cs` проекта консольного приложения. Добавим в класс `Program` следующий метод, который вызывает метод `Add()` с применением типичных обращений к API-интерфейсу рефлексии:

```
private static void AddWithReflection()
{
    Assembly asm = Assembly.Load("MathLibrary");
```

```

try
{
    // Получить метаданные для типа SimpleMath.
    Type math = asm.GetType("MathLibrary.SimpleMath");

    // Создать объект SimpleMath на лету.
    object obj = Activator.CreateInstance(math);

    // Получить информацию о методе Add().
    MethodInfo mi = math.GetMethod("Add");

    // Вызвать метод (с параметрами).
    object[] args = { 10, 70 };
    Console.WriteLine("Result is: {0}", mi.Invoke(obj, args));
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
}

```

Ниже показано, как можно упростить предыдущую логику, используя ключевое слово `dynamic`:

```

private static void AddWithDynamic()
{
    Assembly asm = Assembly.Load("MathLibrary");

    try
    {
        // Получить метаданные для типа SimpleMath.
        Type math = asm.GetType("MathLibrary.SimpleMath");

        // Создать объект SimpleMath на лету.
        dynamic obj = Activator.CreateInstance(math);

        // Обратите внимание, насколько легко теперь вызывать метод Add().
        Console.WriteLine("Result is: {0}", obj.Add(10, 70));
    }
    catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
    {
        Console.WriteLine(ex.Message);
    }
}

```

В результате вызова обоих методов в `Main()` получается идентичный вывод. Однако в случае применения ключевого слова `dynamic` сокращается объем кода. Благодаря динамически определяемым данным вам больше не придется вручную упаковывать аргументы внутрь массива экземпляров `object`, запрашивать метаданные сборки либо иметь дело с другими деталями подобного рода. При построении приложения, в котором интенсивно используется динамическая загрузка и позднее связывание, экономия на кодировании со временем становится еще более ощутимой.

Упрощение взаимодействия с COM посредством динамических данных

Давайте рассмотрим еще один полезный сценарий для ключевого слова `dynamic` в рамках проекта взаимодействия с COM. Если у вас нет опыта разработки для COM, то имейте в виду, что скомпилированная библиотека COM содержит метаданные подобно библиотеке .NET, но ее формат совершенно другой. По указанной причине, когда программа .NET нуждается во взаимодействии с объектом COM, первым делом потребуются сгенерировать так называемую *сборку взаимодействия* (описанную ниже). Задача довольно проста. Сначала нужно создать новое консольное приложение по имени `ExportDataToOfficeApp`, открыть диалоговое окно `Add Reference` (Добавление ссылки), перейти на вкладку `COM` и отыскать библиотеку COM, которую желательно применять (рис. 16.4).

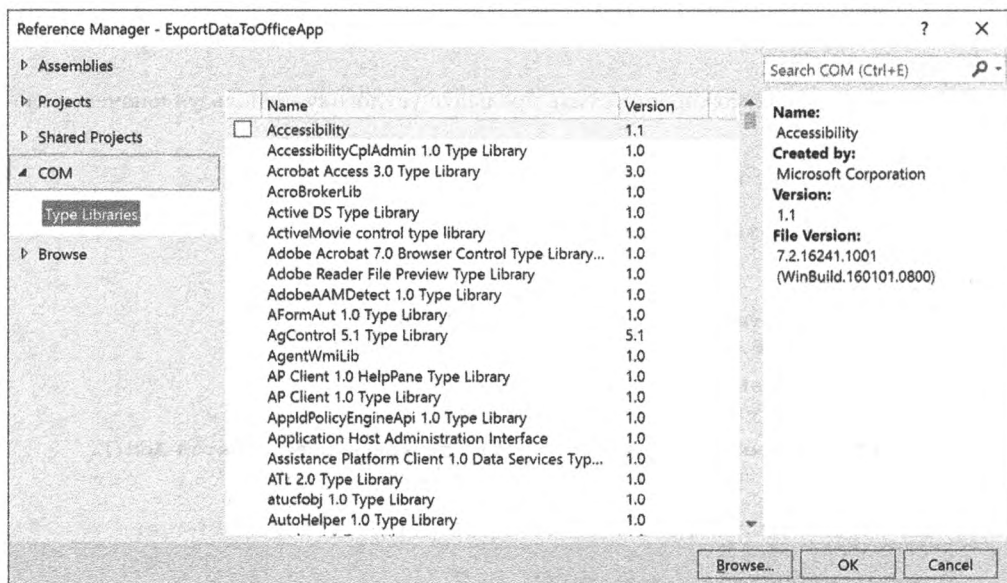


Рис. 16.4. На вкладке `COM` диалогового окна `Add Reference` отображаются все библиотеки COM, зарегистрированные на машине

На заметку! Имейте в виду, что некоторые важные объектные модели Microsoft (включая продукты Office) в настоящее время доступны только через взаимодействие с COM. Таким образом, даже если вы не имеете опыта непосредственного построения приложений COM, то может возникнуть необходимость потреблять их внутри программы .NET.

После выбора COM-библиотеки IDE-среда отреагирует генерацией новой сборки, которая включает описания .NET метаданных COM. Формально она называется *сборкой взаимодействия* и не содержит какого-либо кода реализации кроме небольшой порции кода, который помогает транслировать события COM в события .NET. Тем не менее, сборки взаимодействия полезны тем, что защищают кодовую базу .NET от сложностей внутреннего механизма COM.

В коде C# можно напрямую работать со сборкой взаимодействия, позволяя среде CLR (а если используется ключевое слово `dynamic`, то среде DLR) автоматически отображать типы данных .NET на типы COM и наоборот. “За кулисами” данные маршализируются между приложениями .NET и COM с применением вызываемой оболочки времени выполнения (Runtime Callable Wrapper — RCW), которая по существу является динамически сгенерированным прокси. Прокси RCW будет маршализировать и трансформировать типы данных .NET в типы COM и отображать любые возвращаемые значения COM на их эквиваленты .NET.

На рис. 16.5 изображена общая картина взаимодействия .NET с COM.

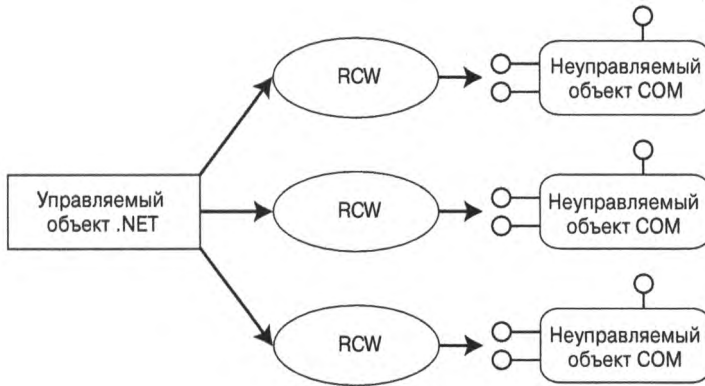


Рис. 16.5. Программы .NET взаимодействуют с объектами COM, используя прокси под названием RCW

Роль основных сборок взаимодействия

Многие библиотеки COM, созданные поставщиками библиотек COM (вроде библиотек Microsoft COM, обеспечивающих доступ к объектной модели продуктов Microsoft Office), предоставляют “официальную” сборку взаимодействия, которая называется *основной сборкой взаимодействия* (primary interop assembly — PIA). Сборки PIA — это оптимизированные сборки взаимодействия, которые приводят в порядок (и возможно расширяют) код, обычно генерируемый при добавлении ссылки на библиотеку COM с помощью диалогового окна Add Reference.

Сборки PIA обычно перечислены в разделе Assemblies (Сборки) диалогового окна Add Reference (в области Extensions (Расширения)). В действительности, если вы ссылаетесь на библиотеку COM из вкладки COM диалогового окна Add Reference, то Visual Studio не генерирует новую библиотеку взаимодействия, как делалось бы обычно, а взамен применяет предоставленную сборку PIA. На рис. 16.6 показана сборка PIA объектной модели Microsoft Office Excel, которая будет использоваться в следующем примере.

На заметку! Если вы не устанавливали отдельный компонент VSTO (Visual Studio Tools For Office — инструменты Visual Studio для Office) или рабочую нагрузку Office/SharePoint development (Разработка для Office/SharePoint), тогда это придется сделать, чтобы проработать материалы данного раздела. Можете запустить программу установки и выбрать недостающий компонент или воспользоваться полем Quick Launch (Быстрый запуск) среды Visual Studio, доступным по нажатию <Ctrl+Q>. В поле Quick Launch введите Visual Studio Tools for Office и выберите параметр Install (Установить).

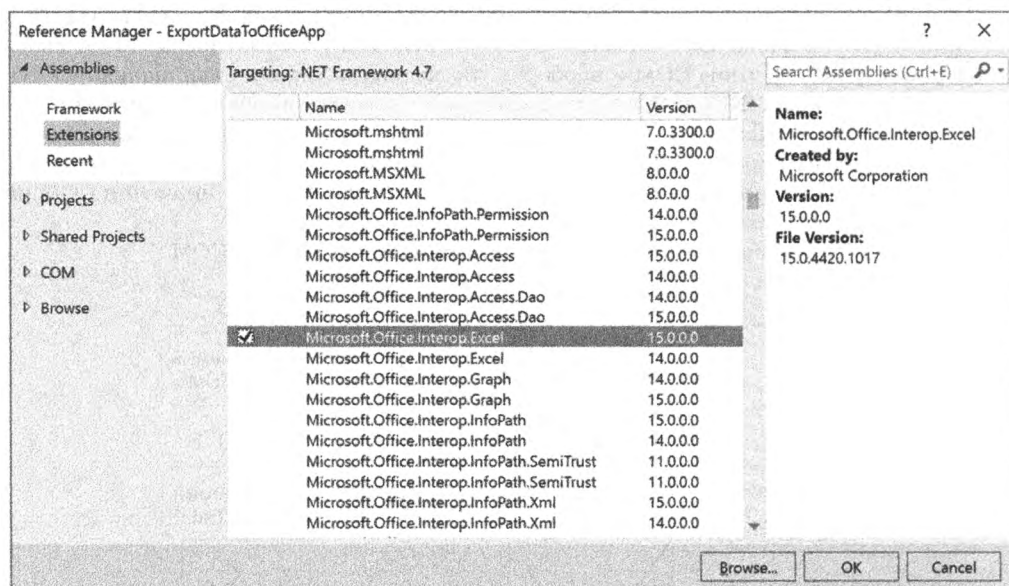


Рис. 16.6. Сборки PIA перечислены в области Extensions раздела Assemblies внутри диалогового окна Add Reference

Встраивание метаданных взаимодействия

До выхода версии .NET 4.0, когда приложение C# задействовало библиотеку COM (через PIA или нет), на клиентской машине необходимо было обеспечить наличие копии сборки взаимодействия. Помимо увеличения размера установочного пакета приложения сценарий установки должен был также проверять, присутствуют ли сборки PIA, и если нет, тогда устанавливать их копии в GAC.

Однако в .NET 4.0 и последующих версиях данные взаимодействия теперь можно встраивать прямо в скомпилированное приложение .NET. В таком случае поставлять копию сборки взаимодействия вместе с приложением .NET больше не понадобится, потому что все необходимые метаданные взаимодействия жестко закодированы в приложении .NET.

По умолчанию после выбора в диалоговом окне Add Reference библиотеки COM (со сборкой PIA или без нее) IDE-среда автоматически устанавливает свойство Embed Interop Types (Встраивать типы взаимодействия) для библиотеки в True. Чтобы увидеть эту настройку, нужно выбрать ссылаемую сборку взаимодействия в папке References (Ссылки) внутри окна Solution Explorer и открыть ее окно свойств (рис. 16.7).

Компилятор C# будет включать только те части библиотеки взаимодействия, которые действительно используются. Таким образом, даже если реальная библиотека взаимодействия содержит описания .NET сотен объектов COM, в приложение попадет только подмножество определений, которые фактически применяются в написанном коде C#. Помимо сокращения размера приложения, предоставляемого клиенту, упрощается и процесс установки, т.к. не придется устанавливать сборки PIA, которые отсутствуют на целевой машине.

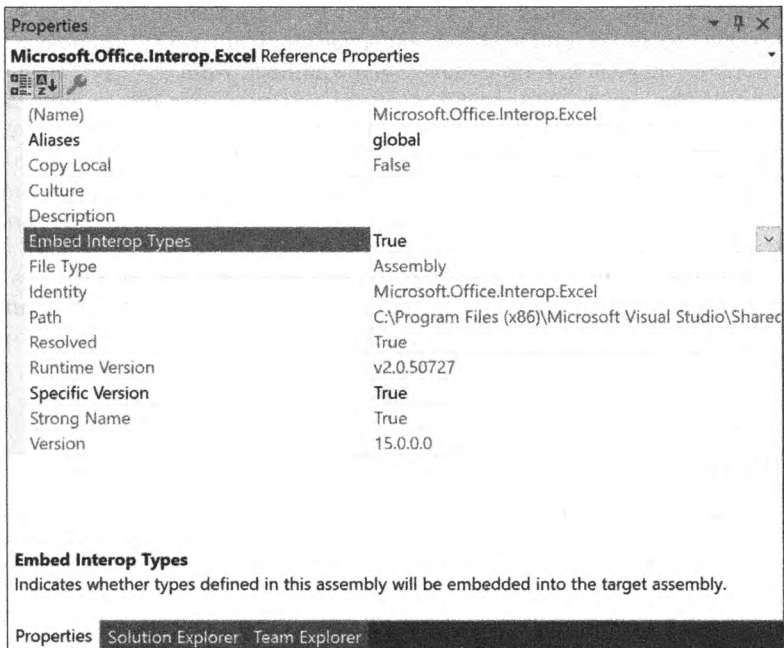


Рис. 16.7. Логику сборки взаимодействия можно встроить непосредственно в приложение .NET

Общие сложности взаимодействия с COM

Давайте перед переходом к следующему примеру рассмотрим еще одну подготовительную тему. До выпуска DLR при написании кода C#, обращающегося к библиотеке COM (через сборку взаимодействия), неизбежно возникало несколько сложностей. Например, многие библиотеки COM определяли методы, принимающие необязательные аргументы, которые вплоть до версии .NET 3.5 в языке C# не поддерживались. Это требовало указания значения `Type.Missing` для каждого вхождения необязательного аргумента. Скажем, если метод COM имел пять аргументов, и все они были необязательными, приходилось писать следующий код C#, чтобы принимать стандартные значения:

```
myComObj.SomeMethod(Type.Missing, Type.Missing, Type.Missing,
                    Type.Missing, Type.Missing);
```

К счастью, теперь появилась возможность записывать показанный ниже упрощенный код, поскольку если не указано специфичное значение, то на этапе компиляции вместо него вставляется `Type.Missing`:

```
myComObj.SomeMethod();
```

В качестве связанного замечания: многие методы COM поддерживают именованные аргументы, которые, как объяснялось в главе 4, позволяют передавать значения членам в любом порядке. Учитывая наличие поддержки той же самой возможности в языке C#, допускается просто "пропускать" необязательные аргументы, которые неважны, и устанавливать только те из них, которые нужны в текущий момент.

Еще одна распространенная сложность взаимодействия с COM была связана с тем фактом, что многие методы COM проектировались так, чтобы принимать и возвращать специфический тип данных по имени `Variant`. Во многом похоже на ключевое слово

dynamic языка C#, типу данных Variant может быть присвоен на лету любой тип данных COM (строка, ссылка на интерфейс, числовое значение и т.д.). До появления ключевого слова dynamic передача и прием элементов данных типа Variant требовали немалых ухищрений, обычно связанных с многочисленными операциями приведения.

Когда свойство Embed Interop Types установлено в True, все COM-типы Variant автоматически отображаются на динамические данные. В итоге не только сокращается потребность в паразитных операциях приведения при работе с типами данных Variant, но также еще больше скрываются некоторые сложности, присущие COM, вроде работы с индексами COM.

Чтобы продемонстрировать, каким образом необязательные аргументы, именованные аргументы и ключевое слово dynamic совместно способствуют упрощению взаимодействия с COM, мы построим приложение, в котором используется объектная модель Microsoft Office. В рассматриваемом примере будет шанс применить новые средства, а также обойтись без них, и затем сравнить объем работы в обоих случаях.

Взаимодействие с COM с использованием динамических данных C#

Вернемся к созданию приложения. Добавим новый класс по имени Car.cs со следующим кодом:

```
namespace ExportDataToOfficeApp
{
    public class Car
    {
        public string Make { get; set; }
        public string Color { get; set; }
        public string PetName { get; set; }
    }
}
```

Далее создадим список записей Car в методе Main() класса Program:

```
List<Car> carsInStock = new List<Car>
{
    new Car {Color="Green", Make="VW", PetName="Mary"},
    new Car {Color="Red", Make="Saab", PetName="Mel"},
    new Car {Color="Black", Make="Ford", PetName="Hank"},
    new Car {Color="Yellow", Make="BMW", PetName="Davie"}
};
```

Добавим в файл Program.cs приведенный ниже псевдоним пространства имен. Имейте в виду, что при взаимодействии с библиотеками COM псевдоним определять не обязательно. Тем не менее, поступая так, вы получаете квалификатор для всех импортированных объектов COM, который удобен, когда объекты COM имеют имена, конфликтующие с именами ваших типов .NET.

```
// Создать псевдоним для объектной модели Excel.
using Excel = Microsoft.Office.Interop.Excel;
```

Поскольку библиотека COM была импортирована с применением Visual Studio, сборка PIA автоматически сконфигурирована так, что используемые метаданные будут встраиваться в приложение .NET (вспомните о роли свойства Embed Interop Types). Таким образом, все типы данных Variant из COM реализуются как типы данных dynamic. Более того, можно применять необязательные и именованные аргументы C#. С учетом всего сказанного вот как будет выглядеть реализация метода ExportToExcel():

```

static void ExportToExcel(List<Car> carsInStock)
{
    // Загрузить Excel и затем создать новую пустую рабочую книгу.
    Excel.Application excelApp = new Excel.Application();
    excelApp.Workbooks.Add();
    // В этом примере используется единственный рабочий лист.
    Excel.Worksheet workSheet = excelApp.ActiveSheet;
    // Установить заголовки столбцов в ячейках.
    workSheet.Cells[1, "A"] = "Make";
    workSheet.Cells[1, "B"] = "Color";
    workSheet.Cells[1, "C"] = "Pet Name";
    // Отобразить все данные из List<Car> на ячейки электронной таблицы.
    int row = 1;
    foreach (Car c in carsInStock)
    {
        row++;
        workSheet.Cells[row, "A"] = c.Make;
        workSheet.Cells[row, "B"] = c.Color;
        workSheet.Cells[row, "C"] = c.PetName;
    }
    // Придать симпатичный вид табличным данным.
    workSheet.Range["A1"].AutoFormat(
        Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
    // Сохранить файл, завершить работу Excel и отобразить сообщение пользователю.
    workSheet.SaveAs($"{Environment.CurrentDirectory}\\Inventory.xlsx");
    excelApp.Quit();
    Console.WriteLine("The Inventory.xlsx file has been saved to your app folder");
    // Файл Inventory.xlsx сохранен в папке приложения.
}

```

Метод `ExportToExcel()` начинается с загрузки приложения Excel в память; однако на рабочем столе оно не отобразится. В данном приложении нас интересует только работа с внутренней объектной моделью Excel. Тем не менее, если необходимо отобразить пользовательский интерфейс Excel, тогда метод понадобится дополнить следующим кодом:

```

static void ExportToExcel(List<Car> carsInStock)
{
    // Загрузить Excel и затем создать новую пустую рабочую книгу.
    Excel.Application excelApp = new Excel.Application();
    // Сделать пользовательский интерфейс Excel видимым на рабочем столе.
    excelApp.Visible = true;
    ...
}

```

После создания пустого рабочего листа добавляются три столбца, именованные в соответствии со свойствами класса `Car`. Затем ячейки наполняются данными `List<Car>`, и файл сохраняется с жестко закодированным именем `Inventory.xlsx`.

Если теперь запустить приложение, добавить несколько записей и экспортировать данные в Excel, то в подкаталоге `bin\Debug` консольного приложения появится файл `Inventory.xlsx`, который можно открыть в приложении Excel (рис. 16.8).

Взаимодействие с COM без использования динамических данных C#

В случае выбора сборки `Microsoft.Office.Interop.Excel.dll` (в окне `Solution Explorer`) и установки ее свойства `Embed Interop Type` в `False` появятся новые сообщения об ошибках компиляции, т.к. COM-данные `Variant` больше не воспринимаются как динамические данные, а считаются переменными типа `System.Object`. Это потребует добавления в метод `ExportToExcel()` нескольких явных операций приведения.

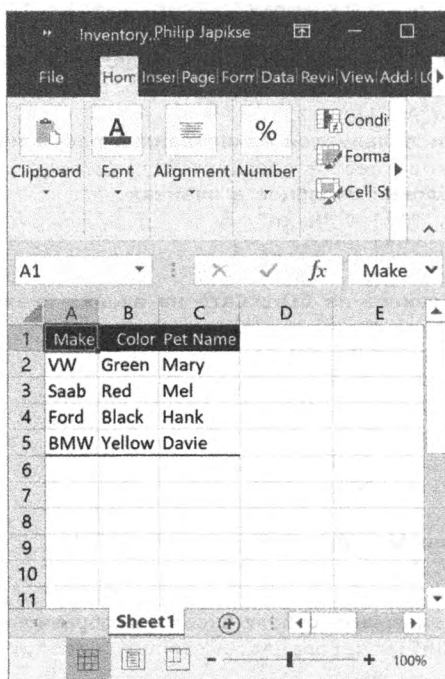


Рис. 16.8. Экспортирование данных в файл Excel

Кроме того, если скомпилировать проект для .NET 3.5 или предшествующих версий, то утратятся преимущества необязательных/именованных параметров и потребуются явно помечать все пропущенные аргументы. Вот версия метода экспортирования данных в файл Excel, предназначенная для более ранних версий C# (обратите внимание на возросшую сложность кода):

```
static void ExportToExcelManual(List<Car> carsInStock)
{
    Excel.Application excelApp = new Excel.Application();
    // Потребуется пометить пропущенные параметры!
    excelApp.Workbooks.Add(Type.Missing);
    // Потребуется привести объект Object к _Worksheet!
    Excel._Worksheet workSheet = (Excel._Worksheet)excelApp.ActiveSheet;
    // Потребуется привести каждый объект Object к Range
    // и затем обратиться к низкоуровневому свойству Value2!
    ((Excel.Range)excelApp.Cells[1, "A"]).Value2 = "Make";
    ((Excel.Range)excelApp.Cells[1, "B"]).Value2 = "Color";
    ((Excel.Range)excelApp.Cells[1, "C"]).Value2 = "Pet Name";
    int row = 1;
    foreach (Car c in carsInStock)
    {
        row++;
        // Потребуется привести каждый объект Object к Range
        // и затем обратиться к низкоуровневому свойству Value2!
        ((Excel.Range)workSheet.Cells[row, "A"]).Value2 = c.Make;
        ((Excel.Range)workSheet.Cells[row, "B"]).Value2 = c.Color;
        ((Excel.Range)workSheet.Cells[row, "C"]).Value2 = c.PetName;
    }
}
```

```

// Потребуется вызвать метод get_Range()
// с указанием всех пропущенных аргументов!
excelApp.get_Range("A1", Type.Missing).AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2,
    Type.Missing, Type.Missing, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing);

// Потребуется указать все пропущенные необязательные аргументы!
workSheet.SaveAs($"{Environment.CurrentDirectory}\InventoryManual.xlsx",
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing);
excelApp.Quit();
Console.WriteLine("The InventoryManual.xlsx file has been saved to your
app folder"); // Файл InventoryManual.xlsx сохранен в папке приложения.
}

```

Несмотря на то что конечный результат выполнения программы идентичен, такая версия метода очевидно намного более многословна. На этом рассмотрение ключевого слова `dynamic` языка C# и среды DLR завершено. Вы увидели, насколько данные средства способны упростить сложные задачи программирования, и (что возможно даже важнее) уяснили сопутствующие компромиссы. Делая выбор в пользу динамических данных, вы теряете изрядную часть безопасности типов, и ваша кодовая база предрасположена к намного большему числу ошибок времени выполнения.

В то время как о среде DLR можно еще рассказать многое, основное внимание в главе было сосредоточено на темах, практических и полезных при повседневном программировании. Если вы хотите изучить расширенные средства DLR, такие как интеграция с языками написания сценариев, тогда обратитесь в документацию .NET Framework 4.7 SDK (начните с поиска темы “Dynamic Language Runtime Overview” (“Обзор исполняющей среды динамического языка”)).

Исходный код. Проект `ExportDataToOfficeApp` доступен в подкаталоге `Chapter_16`.

Резюме

Появившееся в версии C# 4.0 ключевое слово `dynamic` позволяет определять данные, настоящая идентичность которых не известна вплоть до времени выполнения. При обработке новой исполняющей средой динамического языка (DLR) автоматически создаваемое “дерево выражения” будет передаваться подходящему связывателю динамического языка, причем полезная нагрузка будет распакована и отправлена правильному члену объекта.

С применением динамических данных и среды DLR многие сложные задачи программирования C# могут быть радикально упрощены, особенно действие по включению библиотек COM в состав приложений .NET. Кроме того, в главе было показано, что платформа .NET 4.0 и последующих версий предлагает несколько дальнейших упрощений взаимодействия с COM (которые не имеют отношения к динамическим данным), в том числе встраивание данных взаимодействия COM в разрабатываемые приложения, необязательные, а также именованные аргументы.

Хотя все рассмотренные средства определенно могут упростить код, всегда помните о том, что динамические данные существенно снижают безопасность к типам в коде C# и открывают возможности для возникновения ошибок времени выполнения. Тщательно взвешивайте все “за” и “против” использования динамических данных в своих проектах C# и должным образом тестируйте их!

глава 17

Процессы, домены приложений и объектные контексты

В главах 14 и 15 вы изучили шаги, которые среда CLR предпринимает при выяснении местоположения ссылаемой внешней сборки, а также роль метаданных .NET. В настоящей главе будут представлены детали обслуживания сборки средой CLR, а также отношения между процессами, доменами приложений и объектными контекстами.

Выражаясь кратко, *домены приложений* (Application Domain или просто AppDomain) представляют собой логические подразделы внутри заданного процесса, обслуживающего набор связанныхборок .NET. Как вы увидите, каждый домен приложения в дальнейшем подразделяется на *контекстные границы*, которые используются для группирования вместе похожих по смыслу объектов .NET. Благодаря понятию контекста среда CLR способна обеспечивать надлежащую обработку объектов со специальными требованиями времени выполнения.

Хотя вполне справедливо утверждать, что многие повседневные задачи программирования не предусматривают работу с процессами, доменами приложений или объектными контекстами напрямую, их понимание важно при взаимодействии с многочисленными API-интерфейсами .NET, включая Windows Communication Foundation (WCF), многопоточную и параллельную обработку, а также сериализацию объектов.

Роль процесса Windows

Концепция “процесса” существовала в операционных системах Windows задолго до выпуска платформы .NET. Пользуясь простыми терминами, *процесс* — это выполняющаяся программа. Тем не менее, формально процесс является концепцией уровня операционной системы, которая применяется для описания набора ресурсов (таких как внешние библиотеки кода и главный поток) и необходимых распределений памяти, используемой функционирующим приложением. Для каждого загруженного в память файла *.exe операционная система создает отдельный изолированный процесс для применения на протяжении всего его времени существования.

При использовании такого подхода к изоляции приложений в результате получается намного более надежная и устойчивая исполняющая среда, поскольку отказ одного процесса не влияет на работу других процессов. Более того, данные в одном процессе не доступны напрямую другим процессам, если только не применяется программный API-интерфейс для распределенных вычислений вроде WCF. С учетом указанных мо-

ментов процесс можно рассматривать как фиксированную и безопасную границу для выполняющегося приложения.

Каждый процесс Windows получает уникальный идентификатор процесса (process identifier — PID) и может по мере необходимости независимо загружаться и выгружаться операционной системой (а также программно). Как вам возможно известно, в окне диспетчера задач Windows (открываемом по нажатию комбинации клавиш <Ctrl+Shift+Esc>) имеется вкладка Processes (Процессы), на которой можно просматривать разнообразные статические данные о процессах, функционирующих на машине. На вкладке Details (Подробности) можно видеть назначенный идентификатор PID и имя образа (рис. 17.1).

Name	PID	Status	User name	CPU	Memory (pri...	Description
Adobe CEF Helper.exe	13744	Running	Skimedic	00	2,492 K	Adobe CEF Helper
Adobe Desktop Servi...	8184	Running	Skimedic	00	13,540 K	Creative Cloud
AdobeIPCBroker.exe	480	Running	Skimedic	00	2,248 K	Adobe IPC Broker
AdobeUpdateService....	4644	Running	SYSTEM	00	520 K	Adobe Update Service
AGSService.exe	4664	Running	SYSTEM	00	780 K	Adobe Genuine Software Integrity Service
ApplicationFrameHos...	10984	Running	Skimedic	00	4,908 K	Application Frame Host
armsvc.exe	4636	Running	SYSTEM	00	456 K	Adobe Acrobat Update Service
audiodg.exe	38340	Running	LOCAL SERV...	00	4,988 K	Windows Audio Device Graph Isolation
BingSvc.exe	13148	Running	Skimedic	00	2,856 K	Microsoft Bing Service
BlueJeans.exe	12800	Running	Skimedic	00	632 K	Blue Jeans Application
BoxSync.exe	12524	Running	Skimedic	00	24,380 K	Box Sync
BoxSyncMonitor.exe	13284	Running	Skimedic	00	244 K	BoxSyncMonitor.exe
CCLibrary.exe	12668	Running	Skimedic	00	176 K	CCLibraries
CCXProcess.exe	16100	Running	Skimedic	00	176 K	CCXProcess
com.docker.db.exe	16008	Running	Skimedic	00	3,648 K	com.docker.db.exe
com.docker.proxy.exe	15112	Running	Skimedic	00	2,684 K	com.docker.proxy.exe
com.docker.service	4696	Running	SYSTEM	00	3,284 K	Docker.Service
conhost.exe	15928	Running	Skimedic	00	428 K	Console Window Host
conhost.exe	15336	Running	Skimedic	00	428 K	Console Window Host
conhost.exe	13292	Running	Skimedic	00	556 K	Console Window Host
conhost.exe	15436	Running	Skimedic	00	440 K	Console Window Host
conhost.exe	15452	Running	SYSTEM	00	388 K	Console Window Host

Рис. 17.1. Диспетчер задач Windows

Роль потоков

Каждый процесс Windows содержит начальный “поток”, который действует как точка входа для приложения. Особенности построения многопоточных приложений на платформе .NET рассматриваются в главе 19; однако для понимания материала настоящей главы необходимо ознакомиться с несколькими рабочими определениями. Поток представляет собой путь выполнения внутри процесса. Выразаясь формально, первый поток, созданный точкой входа процесса, называется *главным потоком*. В любой исполняемой программе .NET (консольном приложении, Windows-службе, приложении WPF и т.д.) точка входа помечается с помощью метода `Main()`, при вызове которого главный поток создается автоматически.

Процессы, которые содержат единственный главный поток выполнения, по своей сути *безопасны в отношении потоков*, т.к. в каждый момент времени доступ к данным приложения может получать только один поток. Тем не менее, однопоточный процесс (особенно с графическим пользовательским интерфейсом) часто замедленно реагирует на действия пользователя, когда его единственный поток выполняет сложную операцию (наподобие печати длинного текстового файла, сложных математических вычислений или попытки подключения к удаленному серверу, находящемуся на расстоянии тысяч километров).

Учитывая такой потенциальный недостаток однопоточных приложений, API-интерфейс Windows (а также платформа .NET) предоставляет главному потоку возможность порождения дополнительных вторичных потоков (называемых *рабочими потоками*) с использованием нескольких функций из API-интерфейса Windows, таких как `CreateThread()`. Каждый поток (первичный или вторичный) становится уникальным путем выполнения в процессе и имеет параллельный доступ ко всем разделяемым элементам данных внутри этого процесса.

Нетрудно догадаться, что разработчики обычно создают дополнительные потоки для улучшения общей степени отзывчивости программы. Многопоточные процессы обеспечивают иллюзию того, что выполнение многочисленных действий происходит более или менее одновременно. Например, приложение может порождать дополнительный рабочий поток для выполнения трудоемкой единицы работы (вроде вывода на печать крупного текстового файла). После запуска вторичного потока главный поток продолжает реагировать на пользовательский ввод, что дает всему процессу возможность достигать более высокой производительности. Однако на самом деле так происходит не всегда: применение слишком большого количества потоков в одном процессе может приводить к *ухудшению* производительности из-за того, что центральный процессор должен переключаться между активными потоками внутри процесса (а это отнимает время).

На некоторых машинах многопоточность по большей части является иллюзией, обеспечиваемой операционной системой. Машины с единственным (не поддерживающим гиперпотоки) центральным процессором не обладают возможностью обработки множества потоков в одно и то же время. Взамен один центральный процессор выполняет по одному потоку за единицу времени (называемую *квантом времени*), частично основываясь на приоритете потока. По истечении выделенного кванта времени выполнение существующего потока приостанавливается, позволяя выполнять работу другому потоку. Чтобы поток не “забывал”, что происходило до того, как его выполнение было приостановлено, ему предоставляется возможность записывать данные в локальное хранилище потоков (Thread Local Storage — TLS) и выделяется отдельный стек вызовов (рис. 17.2).

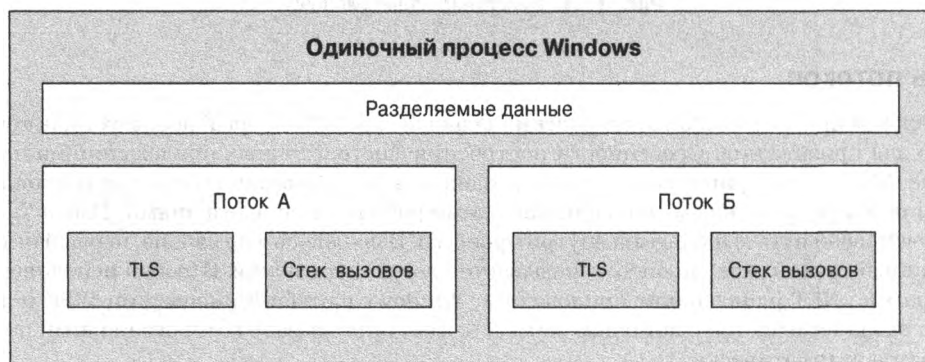


Рис. 17.2. Отношения между потоками и процессами в Windows

Если тема потоков для вас нова, то не стоит беспокоиться о деталях. На данном этапе просто запомните, что любой поток представляет собой уникальный путь выполнения внутри процесса Windows. Каждый процесс имеет главный поток (созданный посредством точки входа исполняемого файла) и может содержать дополнительные потоки, которые создаются программно.

Взаимодействие с процессами на платформе .NET

Несмотря на то что с процессами и потоками не связано ничего нового, способ взаимодействия с ними в рамках платформы .NET значительно изменился (в лучшую сторону). Чтобы подготовить почву для понимания области построения многопоточных сборок (см. главу 19), давайте начнем с выяснения способов взаимодействия с процессами, используя библиотеки базовых классов .NET.

В пространстве имен `System.Diagnostics` определено несколько типов, которые позволяют программно взаимодействовать с процессами и разнообразными типами, связанными с диагностикой, такими как журнал событий системы и счетчики производительности. В текущей главе нас интересуют только типы, связанные с процессами, которые описаны в табл. 17.1.

Таблица 17.1. Избранные типы пространства имен `System.Diagnostics`

Тип	Описание
<code>Process</code>	Предоставляет доступ к локальным и удаленным процессам, а также позволяет программно запускать и останавливать процессы
<code>ProcessModule</code>	Представляет модуль (*.dll или *.exe), загруженный в определенный процесс. Важно понимать, что тип <code>ProcessModule</code> может представлять <i>любой</i> модуль, т.е. двоичные сборки, основанные на COM, .NET или традиционном языке C
<code>ProcessModuleCollection</code>	Предоставляет строго типизированную коллекцию объектов <code>ProcessModule</code>
<code>ProcessStartInfo</code>	Указывает набор значений, применяемых при запуске процесса с помощью метода <code>Process.Start()</code>
<code>ProcessThread</code>	Представляет поток внутри заданного процесса. Имейте в виду, что тип <code>ProcessThread</code> используется для диагностирования набора потоков процесса, но не для порождения новых потоков выполнения в рамках процесса
<code>ProcessThreadCollection</code>	Предоставляет строго типизированную коллекцию объектов <code>ProcessThread</code>

Класс `System.Diagnostics.Process` позволяет анализировать процессы, выполняющиеся на заданной машине (локальные или удаленные). В классе `Process` также определены члены, предназначенные для программного запуска и завершения процессов, просмотра (или модификации) уровня приоритета процесса и получения списка активных потоков и/или загруженных модулей внутри указанного процесса. В табл. 17.2 перечислены некоторые основные свойства класса `System.Diagnostics.Process`.

Таблица 17.2. Избранные свойства класса `Process`

Свойство	Описание
<code>ExitTime</code>	Позволяет извлекать метку времени, ассоциированную с процессом, который был завершен (представленную с помощью типа <code>DateTime</code>)
<code>Handle</code>	Возвращает дескриптор (представленный типом <code>IntPtr</code>), который был назначен процессу операционной системой. Это может быть полезно при построении приложений .NET, нуждающихся во взаимодействии с неуправляемым кодом
<code>Id</code>	Позволяет получать идентификатор PID связанного процесса
<code>MachineName</code>	Позволяет получать имя компьютера, на котором выполняется связанный процесс
<code>MainWindowTitle</code>	Позволяет получать заголовок главного окна процесса (если у процесса нет главного окна, то возвращается пустая строка)
<code>Modules</code>	Предоставляет доступ к строго типизированной коллекции <code>ProcessModuleCollection</code> , представляющей набор модулей (*.dll или *.exe), которые были загружены внутри текущего процесса
<code>ProcessName</code>	Позволяет получать имя процесса (которое, как и можно было предполагать, представляет собой имя самого приложения)
<code>Responding</code>	Позволяет получать значение, которое указывает, реагирует ли пользовательский интерфейс процесса на пользовательский ввод (или в текущий момент находится в "зависшем" состоянии)
<code>StartTime</code>	Позволяет получать значение времени, когда был запущен процесс (представленное с помощью типа <code>DateTime</code>)
<code>Threads</code>	Позволяет получать набор потоков, выполняемых в связанном процессе (представленный посредством коллекции объектов <code>ProcessThread</code>)

Кроме перечисленных выше свойств в классе `System.Diagnostics.Process` определено несколько полезных методов (табл. 17.3).

Таблица 17.3. Избранные методы класса `Process`

Метод	Описание
<code>CloseMainWindow()</code>	Этот метод закрывает процесс, который содержит пользовательский интерфейс, отправляя его главному окну сообщение о закрытии
<code>GetCurrentProcess()</code>	Этот статический метод возвращает новый объект <code>Process</code> , который представляет процесс, активный в текущий момент
<code>GetProcesses()</code>	Этот статический метод возвращает массив объектов <code>Process</code> , представляющих процессы, которые выполняются на заданной машине
<code>Kill()</code>	Этот метод немедленно останавливает связанный процесс
<code>Start()</code>	Этот метод запускает процесс

Перечисление выполняющихся процессов

Для иллюстрации способа манипулирования объектами `Process` создадим новый проект консольного приложения C# по имени `ProcessManipulator` и определим в классе `Program` следующий вспомогательный статический метод (не забудьте импортировать в файл кода пространство имен `System.Diagnostics`):

```
static void ListAllRunningProcesses()
{
    // Получить все процессы на локальной машине, упорядоченные по PID.
    var runningProcs = from proc in Process.GetProcesses(".")
                       orderby proc.Id select proc;

    // Вывести для каждого процесса идентификатор PID и имя.
    foreach(var p in runningProcs)
    {
        string info = $"-> PID: {p.Id}\tName: {p.ProcessName}";
        Console.WriteLine(info);
    }
    Console.WriteLine("*****\n");
}
```

Статический метод `Process.GetProcesses()` возвращает массив объектов `Process`, которые представляют выполняющиеся процессы на целевой машине (передаваемая методу строка `"."` обозначает локальный компьютер). После получения массива объектов `Process` можно обращаться к любым членам, описанным в табл. 17.2 и 17.3. Здесь просто для каждого процесса выводятся идентификатор PID и имя с упорядочением по PID. Добавив в `Main()` вызов метода `ListAllRunningProcesses()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Processes *****\n");
    ListAllRunningProcesses();
    Console.ReadLine();
}
```

можно будет увидеть список имен и идентификаторов PID всех процессов, запущенных на локальной машине. Ниже показана часть вывода (ваш вывод наверняка будет отличаться):

```
***** Fun with Processes *****
-> PID: 0      Name: Idle
-> PID: 4      Name: System
-> PID: 108     Name: iexplore
-> PID: 268     Name: smss
-> PID: 432     Name: csrss
-> PID: 448     Name: svchost
-> PID: 472     Name: wininit
-> PID: 504     Name: csrss
-> PID: 536     Name: winlogon
-> PID: 560     Name: services
-> PID: 584     Name: lsass
-> PID: 592     Name: lsm
-> PID: 660     Name: devenv
-> PID: 684     Name: svchost
-> PID: 760     Name: svchost
-> PID: 832     Name: svchost
-> PID: 844     Name: svchost
```

```

-> PID: 856      Name: svchost
-> PID: 900      Name: svchost
-> PID: 924      Name: svchost
-> PID: 956      Name: VMwareService
-> PID: 1116     Name: spoolsv
-> PID: 1136     Name: ProcessManipulator.vshost
*****

```

Исследование конкретного процесса

В дополнение к полному списку всех выполняющихся процессов на заданной машине статический метод `Process.GetProcessById()` позволяет получать одиночный объект `Process` по ассоциированному с ним идентификатору PID. В случае запроса несуществующего PID генерируется исключение `ArgumentException`. Например, чтобы получить объект `Process`, который представляет процесс с PID, равным 987, можно написать следующий код:

```

// Если процесс с PID, равным 987, не существует,
// то сгенерируется исключение во время выполнения.
static void GetSpecificProcess()
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(987);
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
    }
}

```

К настоящему моменту вы уже знаете, как получить список всех процессов, а также специфический процесс на машине посредством поиска по PID. Наряду с выяснением идентификаторов PID и имен процессов класс `Process` позволяет просматривать набор текущих потоков и библиотек, применяемых внутри заданного процесса. Давайте посмотрим, как это делается.

Исследование набора потоков процесса

Набор потоков представлен в виде строго типизированной коллекции `ProcessThreadCollection`, которая содержит определенное количество отдельных объектов `ProcessThread`. Для примера предположим, что в текущее приложение добавлен приведенный ниже вспомогательный статический метод:

```

static void EnumThreadsForPid(int pID)
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(pID);
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
        return;
    }
}

```

```
// Вывести статистические сведения по каждому потоку в указанном процессе.
Console.WriteLine("Here are the threads used by: {0}",
    theProc.ProcessName);
ProcessThreadCollection theThreads = theProc.Threads;
foreach(ProcessThread pt in theThreads)
{
    string info =
        $"-> Thread ID: {pt.Id}\tStart Time:
(pt.StartTime.ToShortTimeString())\tPriority:{pt.PriorityLevel}";
    Console.WriteLine(info);
}
Console.WriteLine("*****\n");
}
```

Как видите, свойство `Threads` в типе `System.Diagnostics.Process` обеспечивает доступ к классу `ProcessThreadCollection`. Здесь для каждого потока внутри указанного клиентом процесса выводится назначенный идентификатор потока, время запуска и уровень приоритета. Обновим метод `Main()` в классе `Program` так, чтобы он запрашивал у пользователя идентификатор PID подлежащего исследованию процесса:

```
static void Main(string[] args)
{
    ...
    // Запросить у пользователя PID и вывести набор активных потоков.
    Console.WriteLine("***** Enter PID of process to investigate *****");
    Console.Write("PID: ");
    string pID = Console.ReadLine();
    int theProcID = int.Parse(pID);

    EnumThreadsForPid(theProcID);
    Console.ReadLine();
}
```

После запуска приложения можно вводить PID любого процесса на машине и просматривать имеющиеся внутри него потоки. В следующем выводе показаны потоки, используемые процессом с PID, равным 22952, который (так случилось) обслуживает браузер Firefox:

```
***** Enter PID of process to investigate *****
PID: 22952
Here are the threads used by: firefox
-> Thread ID: 680      Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 2040     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 880      Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 3380     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 3376     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 3448     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 3476     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 2264     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 2380     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 2384     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 2308     Start Time: 9:05 AM      Priority: Normal
-> Thread ID: 3096     Start Time: 9:07 AM      Priority: Highest
-> Thread ID: 3600     Start Time: 9:45 AM      Priority: Normal
-> Thread ID: 1412     Start Time: 10:02 AM     Priority: Normal
```

Помимо `Id`, `StartTime` и `PriorityLevel` тип `ProcessThread` содержит дополнительные члены, наиболее интересные из которых перечислены в табл. 17.4.

Таблица 17.4. Избранные члены типа `ProcessThread`

Член	Описание
<code>CurrentPriority</code>	Получает текущий приоритет потока
<code>Id</code>	Получает уникальный идентификатор потока
<code>IdealProcessor</code>	Устанавливает предпочитаемый процессор для выполнения заданного потока
<code>PriorityLevel</code>	Получает или устанавливает уровень приоритета потока
<code>ProcessorAffinity</code>	Устанавливает процессоры, на которых может выполняться связанный поток
<code>StartAddress</code>	Получает адрес в памяти функции, вызванной операционной системой, которая запустила данный поток
<code>StartTime</code>	Получает время, когда операционная система запустила поток
<code>ThreadState</code>	Получает текущее состояние потока
<code>TotalProcessorTime</code>	Получает общее время, потраченное данным потоком на использование процессора
<code>WaitReason</code>	Получает причину, по которой поток находится в состоянии ожидания

Прежде чем двигаться дальше, необходимо уяснить, что тип `ProcessThread` не является сущностью, применяемой для создания, приостановки или уничтожения потоков на платформе .NET. Тип `ProcessThread` скорее представляет собой средство, позволяющее получать диагностическую информацию по активным потокам Windows внутри выполняющегося процесса. Более подробные сведения о том, как создавать многопоточные приложения с использованием пространства имен `System.Threading`, приводятся в главе 19.

Исследование набора модулей процесса

Теперь давайте посмотрим, как реализовать проход по загруженным модулям, которые размещены внутри конкретного процесса. Когда речь идет о процессах, модуль представляет собой общий термин, применяемый для описания заданной сборки *.dll (или самого файла *.exe), которая обслуживается специфичным процессом. Когда производится доступ к коллекции `ProcessModuleCollection` через свойство `Process.Modules`, появляется возможность перечисления *всех модулей*, размещенных внутри процесса: библиотек на основе .NET, COM и традиционного языка C. Взгляните на показанный ниже дополнительный вспомогательный метод, который будет перечислять модули в процессе с указанным идентификатором PID:

```
static void EnumModsForPid(int pID)
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(pID);
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
        return;
    }
}
```

```

Console.WriteLine("Here are the loaded modules for: {0}",
    theProc.ProcessName);
ProcessModuleCollection theMods = theProc.Modules;
foreach(ProcessModule pm in theMods)
{
    string info = $"-> Mod Name: {pm.ModuleName}";
    Console.WriteLine(info);
}
Console.WriteLine("*****\n");
}

```

Чтобы получить какой-то вывод, давайте посмотрим загружаемые модули для процесса, обслуживающего программу текущего примера (ProcessManipulator). Для этого нужно запустить приложение, выяснить идентификатор PID, назначенный ProcessManipulator.exe (посредством диспетчера задач), и передать значение PID методу EnumModsForPid() (соответствующим образом обновив метод Main()). Вас может удивить, что с простым консольным приложением связан настолько внушительный список библиотек *.dll (GDI32.dll, USER32.dll, ole32.dll и т.д.):

```

Here are the loaded modules for: ProcessManipulator
-> Mod Name: ProcessManipulator.exe
-> Mod Name: ntdll.dll
-> Mod Name: MSCOREE.DLL
-> Mod Name: KERNEL32.dll
-> Mod Name: KERNELBASE.dll
-> Mod Name: ADVAPI32.dll
-> Mod Name: msvcrt.dll
-> Mod Name: sechost.dll
-> Mod Name: RPCRT4.dll
-> Mod Name: SspiCli.dll
-> Mod Name: CRYPTBASE.dll
-> Mod Name: mscoree.dll
-> Mod Name: SHLWAPI.dll
-> Mod Name: GDI32.dll
-> Mod Name: USER32.dll
-> Mod Name: LPK.dll
-> Mod Name: USP10.dll
-> Mod Name: IMM32.DLL
-> Mod Name: MSCTF.dll
-> Mod Name: clr.dll
-> Mod Name: MSVCR100_CLR0400.dll
-> Mod Name: mscorlib.ni.dll
-> Mod Name: nlssorting.dll
-> Mod Name: ole32.dll
-> Mod Name: clrjit.dll
-> Mod Name: System.ni.dll
-> Mod Name: System.Core.ni.dll
-> Mod Name: psapi.dll
-> Mod Name: shfolder.dll
-> Mod Name: SHELL32.dll
*****

```

Запуск и останов процессов программным образом

Финальными аспектами класса System.Diagnostics.Process, которые мы здесь исследуем, являются методы Start() и Kill(). Они позволяют программно запускать и завершать процесс. В качестве примера создадим вспомогательный статический метод StartAndKillProcess() с приведенным ниже кодом.

На заметку! Для того чтобы запускать новые процессы, среда Visual Studio должна быть запущена от имени учетной записи администратора. В противном случае возникнет ошибка во время выполнения.

```
static void StartAndKillProcess()
{
    Process ffProc = null;
    // Запустить Firefox и перейти на сайт facebook.com.
    try
    {
        ffProc = Process.Start("Firefox.exe", "www.facebook.com");
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine(ex.Message);
    }

    Console.WriteLine("--> Hit enter to kill {0}...", ffProc.ProcessName);
    Console.ReadLine();

    // Уничтожить процесс firefox.
    try
    {
        ffProc.Kill();
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Статический метод `Process.Start()` имеет несколько перегруженных версий. Как минимум, понадобится указать дружественное имя запускаемого процесса (наподобие `Firefox.exe` для браузера Firefox). В рассматриваемом примере используется версия метода `Start()`, которая позволяет задавать любые дополнительные аргументы, подлежащие передаче в точку входа программы (т.е. методу `Main()`).

В результате вызова метода `Start()` возвращается ссылка на новый запущенный процесс. Чтобы завершить данный процесс, потребуется просто вызвать метод `Kill()` уровня экземпляра. Здесь вызовы `Start()` и `Kill()` помещены внутрь блока `try/catch` с обработкой любых исключений `InvalidOperationException`. Это особенно важно при вызове метода `Kill()`, потому что такое исключение генерируется, если процесс был завершен до вызова `Kill()`.

Управление запуском процесса с использованием класса `ProcessStartInfo`

Метод `Start()` позволяет также передавать объект типа `System.Diagnostics.ProcessStartInfo` для указания дополнительной информации относительно запуска определенного процесса. Ниже приведено частичное определение `ProcessStartInfo` (полное определение можно найти в документации .NET Framework 4.7 SDK):

```
public sealed class ProcessStartInfo : object
{
    public ProcessStartInfo();
    public ProcessStartInfo(string fileName);
    public ProcessStartInfo(string fileName, string arguments);
```

```
public string Arguments { get; set; }
public bool CreateNoWindow { get; set; }
public StringDictionary EnvironmentVariables { get; }
public bool ErrorDialog { get; set; }
public IntPtr ErrorDialogParentHandle { get; set; }
public string FileName { get; set; }
public bool LoadUserProfile { get; set; }
public SecureString Password { get; set; }
public bool RedirectStandardError { get; set; }
public bool RedirectStandardInput { get; set; }
public bool RedirectStandardOutput { get; set; }
public Encoding StandardErrorEncoding { get; set; }
public Encoding StandardOutputEncoding { get; set; }
public bool UseShellExecute { get; set; }
public string Verb { get; set; }
public string[] Verbs { get; }
public ProcessWindowStyle WindowStyle { get; set; }
public string WorkingDirectory { get; set; }
}
```

Чтобы проиллюстрировать настройку запуска процесса, модифицируем метод `StartAndKillProcess()` для загрузки браузера Firefox, перехода на сайт `www.facebook.com` и отображения окна браузера в развернутом на весь экран виде:

```
static void StartAndKillProcess()
{
    Process ffProc = null;
    // Запустить браузер Firefox и перейти на сайт
    // facebook.com с развернутым на весь экран окном.
    try
    {
        ProcessStartInfo startInfo = new
            ProcessStartInfo("Firefox.exe", "www.facebook.com");
        startInfo.WindowStyle = ProcessWindowStyle.Maximized;
        ffProc = Process.Start(startInfo);
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine(ex.Message);
    }
    ...
}
```

Теперь, когда вы понимаете роль процессов Windows и знаете способы взаимодействия с ними из кода C#, можно переходить к исследованию концепции доменов приложений .NET.

Исходный код. Проект `ProcessManipulator` доступен в подкаталоге `Chapter_17`.

Домены приложений .NET

На платформе .NET исполняемые файлы не размещаются прямо внутри процесса Windows, как в случае традиционных неуправляемых приложений. Взамен исполняемый файл .NET попадает в отдельный логический раздел внутри процесса, который

называется *доменом приложения*. Вы увидите, что один процесс может содержать несколько доменов приложений, каждый из которых обслуживает свой исполняемый файл .NET. Такое дополнительное разделение традиционного процесса Windows обеспечивает несколько преимуществ.

- Домены приложений являются ключевым аспектом нейтральной к операционным системам природы платформы .NET, поскольку такое логическое разделение абстрагирует отличия в том, как лежащая в основе операционная система представляет загруженный исполняемый файл.
- Домены приложений оказываются гораздо менее затратными в смысле вычислительных ресурсов и памяти по сравнению с полноценными процессами. Таким образом, среда CLR способна загружать и выгружать домены приложений намного быстрее, чем формальный процесс, и может значительно улучшить масштабируемость серверных приложений.
- Домены приложений обеспечивают более глубокий уровень изоляции при размещении загруженных приложений. Если один домен приложения внутри процесса терпит отказ, то остальные домены приложений остаются работоспособными.

Как уже упоминалось, одиночный процесс может размещать любое количество доменов приложений, каждый из которых полностью изолирован от остальных доменов внутри текущего (или любого другого) процесса. Учитывая такой факт, имейте в виду, что приложение, выполняющееся в одном домене приложения, не может получать данные любого рода (глобальные переменные или статические поля) из другого домена приложения, если только не применяется какой-нибудь протокол распределенного программирования (вроде WCF).

Хотя в одном процессе *может* находиться множество доменов приложений, обычно подобное не происходит. Как минимум процесс операционной системы будет обслуживать так называемый *стандартный домен приложения*. Этот специфический домен приложения автоматически создается средой CLR во время запуска процесса. Затем среда CLR по мере необходимости создает дополнительные домены приложений.

Класс `System.AppDomain`

Платформа .NET позволяет программно отслеживать домены приложений, создавать новые домены приложений (или выгружать их) во время выполнения, загружать сборки в домены приложений и решать целый ряд других задач с использованием класса `AppDomain` из пространства имен `System`, которое находится в сборке `mscorlib.dll`. В табл. 17.5 описаны некоторые полезные методы этого класса (полная информация доступна в документации .NET Framework 4.7 SDK).

На заметку! Платформа .NET не позволяет выгружать конкретную сборку из памяти. Единственный способ программной выгрузки библиотек предусматривает уничтожение размещающего домена приложения посредством метода `Unload()`.

Вдобавок класс `AppDomain` определяет набор свойств, которые могут быть удобными при мониторинге действий заданного домена приложения. Наиболее интересные свойства описаны в табл. 17.6.

Таблица 17.5. Избранные методы класса AppDomain

Метод	Описание
CreateDomain()	Этот статический метод позволяет создавать новый домен приложения в текущем процессе
CreateInstance()	Этот метод позволяет создавать экземпляр типа из внешней сборки после загрузки данной сборки в вызывающий домен приложения
ExecuteAssembly()	Этот метод выполняет сборку *.exe внутри домена приложения, имея ее имя файла
GetAssemblies()	Этот метод получает набор сборок .NET, которые были загружены в данный домен приложения (двоичные сборки на основе COM и C игнорируются)
GetCurrentThreadId()	Этот статический метод возвращает идентификатор активного потока в текущем домене приложения
Load()	Этот метод применяется для динамической загрузки сборки в текущий домен приложения
Unload()	Этот статический метод позволяет выгрузить указанный домен приложения из заданного процесса

Таблица 17.6. Избранные свойства класса AppDomain

Свойство	Описание
BaseDirectory	Это свойство позволяет получить путь к каталогу, который распознаватель сборок использует для зондирования сборок
CurrentDomain	Это статическое свойство позволяет получить домен приложения, применяемый для текущего выполняющегося потока
FriendlyName	Это свойство позволяет получить дружественное имя текущего домена приложения
MonitoringIsEnabled	Это свойство позволяет получить или установить значение, которое указывает, включен ли мониторинг ресурсов центрального процессора и памяти, потребляемых доменами приложений, для текущего процесса. После того, как мониторинг для процесса включен, отключить его невозможно
SetupInformation	Это свойство позволяет получить детали конфигурации для указанного домена приложения, представленные в виде объекта AppDomainSetup

Наконец, класс AppDomain поддерживает набор событий, которые соответствуют различным аспектам жизненного цикла домена приложения. Некоторые наиболее полезные события, доступные для привязки, представлены в табл. 17.7.

Таблица 17.7. Избранные события класса AppDomain

Событие	Описание
AssemblyLoad	Происходит, когда сборка загружается в память
AssemblyResolve	Возникает, когда распознаватель сборок не может найти местоположение обязательной сборки
DomainUnload	Происходит перед началом выгрузки домена приложения из обслуживающего процесса
FirstChanceException	Позволяет получать уведомление о том, что в домене приложения было сгенерировано исключение, перед тем как среда CLR начнет поиск подходящего оператора catch
ProcessExit	Возникает в стандартном домене приложения, когда его родительский процесс завершается
UnhandledException	Происходит, когда исключение не было перехвачено обработчиком исключений

Взаимодействие со стандартным доменом приложения

Вспомните, что после запуска исполняемого файла .NET среда CLR автоматически помещает его в стандартный домен приложения размещающего процесса. Все делается автоматически и прозрачно, и писать какой-то специальный код не придется. Тем не менее, с помощью статического свойства `AppDomain.CurrentDomain` можно получать доступ к стандартному домену приложения. При наличии такой точки доступа появляется возможность привязываться к любым интересующим событиям либо использовать методы и свойства `AppDomain` для проведения диагностики во время выполнения.

Чтобы научиться взаимодействовать со стандартным доменом приложения, начнем с создания нового проекта консольного приложения по имени `DefaultAppDomainApp`. Модифицируем класс `Program`, поместив в него следующий код, который просто выводит детальные сведения о стандартном домене приложения с применением нескольких членов класса `AppDomain`:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with the default AppDomain *****\n");
        DisplayDADStats();
        Console.ReadLine();
    }

    private static void DisplayDADStats()
    {
        // Получить доступ к домену приложения для текущего потока.
        AppDomain defaultAD = AppDomain.CurrentDomain;

        // Вывести разнообразные статистические данные об этом домене.
        Console.WriteLine("Name of this domain: {0}", defaultAD.FriendlyName);
        // Дружественное имя
        Console.WriteLine("ID of domain in this process: {0}", defaultAD.Id);
        // Идентификатор
```

```

Console.WriteLine("Is this the default domain?: {0}",
    defaultAD.IsDefaultAppDomain()); // Является ли стандартным
Console.WriteLine("Base directory of this domain: {0}",
    defaultAD.BaseDirectory);        // Базовый каталог
}
}

```

Ниже приведен вывод:

```
***** Fun with the default AppDomain *****
```

```
Name of this domain: DefaultAppDomainApp.exe
```

```
ID of domain in this process: 1
```

```
Is this the default domain?: True
```

```
Base directory of this domain: E:\MyCode\DefaultAppDomainApp\bin\Debug\
```

Обратите внимание, что имя стандартного домена приложения будет идентичным имени содержащегося внутри него исполняемого файла (DefaultAppDomainApp.exe в этом примере). Кроме того, значение базового каталога, которое будет использоваться для зондирования обязательных внешних закрытых сборок, отображается на текущее местоположение развернутого исполняемого файла.

Перечисление загруженных сборок

С применением метода `GetAssemblies()` уровня экземпляра можно просмотреть все сборки .NET, загруженные в указанный домен приложения. Метод возвращает массив объектов типа `Assembly`, который, как демонстрировалось в главе 15, является членом пространства имен `System.Reflection` (так что не забудьте импортировать его в файл кода C#).

В целях иллюстрации определим в классе `Program` новый вспомогательный метод по имени `ListAllAssembliesInAppDomain()`. Он будет получать список всех загруженных сборок и выводить для каждой из них дружественное имя и номер версии:

```

static void ListAllAssembliesInAppDomain()
{
    // Получить доступ к домену приложения для текущего потока.
    AppDomain defaultAD = AppDomain.CurrentDomain;

    // Извлечь все сборки, загруженные в стандартный домен приложения.
    Assembly[] loadedAssemblies = defaultAD.GetAssemblies();
    Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
        defaultAD.FriendlyName);
    foreach(Assembly a in loadedAssemblies)
    {
        Console.WriteLine("-> Name: {0}", a.GetName().Name);           // Имя
        Console.WriteLine("-> Version: {0}\n", a.GetName().Version);   // Версия
    }
}

```

Добавив в метод `Main()` вызов метода `ListAllAssembliesInAppDomain()`, можно увидеть все библиотеки .NET, которые используются в домене приложения, обслуживающем исполняемую сборку:

```

***** Here are the assemblies loaded in DefaultAppDomainApp.exe *****

-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: DefaultAppDomainApp
-> Version: 1.0.0.0

```

Важно понимать, что список загруженных сборок может изменяться в любой момент по мере написания нового кода C#. Например, предположим, что метод `ListAllAssembliesInAppDomain()` модифицирован так, чтобы задействовать запрос LINQ, который упорядочивает загруженные сборки по имени:

```
static void ListAllAssembliesInAppDomain()
{
    // Получить доступ к домену приложения для текущего потока.
    AppDomain defaultAD = AppDomain.CurrentDomain;
    // Извлечь все сборки, загруженные в стандартный домен приложения.
    var loadedAssemblies = from a in defaultAD.GetAssemblies()
        orderby a.GetName().Name select a;
    Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
        defaultAD.FriendlyName);
    foreach (var a in loadedAssemblies)
    {
        Console.WriteLine("-> Name: {0}", a.GetName().Name);           // Имя
        Console.WriteLine("-> Version: {0}\n", a.GetName().Version);    // Версия
    }
}
```

Запустив приложение еще раз, можно заметить, что в память также были загружены сборки `System.Core.dll` и `System.dll`, т.к. они требуются для API-интерфейса LINQ to Objects:

```
***** Here are the assemblies loaded in DefaultAppDomainApp.exe *****
-> Name: DefaultAppDomainApp
-> Version: 1.0.0.0
-> Name: mscorlib
-> Version: 4.0.0.0
-> Name: System
-> Version: 4.0.0.0
-> Name: System.Core
-> Version: 4.0.0.0
```

Получение уведомлений о загрузке сборки

Если вы хотите получать от среды CLR уведомление о загрузке новой сборки в заданный домен приложения, тогда должны обработать событие `AssemblyLoad`. Это событие относится к типу делегата `AssemblyLoadEventHandler`, который может указывать на любой метод, принимающий `System.Object` в первом параметре и `AssemblyLoadEventArgs` во втором.

Давайте добавим в текущий класс `Program` последний метод по имени `InitDAD()`, который будет инициализировать стандартный домен приложения, обрабатывая событие `AssemblyLoad` посредством подходящего лямбда-выражения:

```
private static void InitDAD()
{
    // Эта логика будет выводить имя любой сборки, загруженной
    // в домен приложения после его создания.
    AppDomain defaultAD = AppDomain.CurrentDomain;
    defaultAD.AssemblyLoad += (o, s) =>
    {
        Console.WriteLine("{0} has been loaded!", s.LoadedAssembly.GetName().Name);
    };
}
```



```

Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
    ad.FriendlyName);
foreach (var a in loadedAssemblies)
{
    Console.WriteLine("-> Name: {0}", a.GetName().Name);           // Имя
    Console.WriteLine("-> Version: {0}\n", a.GetName().Version); // Версия
}
}
}

```

Запустив приложение, вы увидите, что в стандартный домен приложения (CustomAppDomains.exe) были загружены сборки mscorlib.dll, System.dll, System.Core.dll и CustomAppDomains.exe, учитывая кодовую базу C# текущего проекта. Однако новый домен приложения содержит только сборку mscorlib.dll, которая, как вы помните, является одной из сборок .NET, всегда загружаемых средой CLR в любой домен приложения.

```

***** Fun with Custom AppDomains *****

***** Here are the assemblies loaded in CustomAppDomains.exe *****

-> Name: CustomAppDomains
-> Version: 1.0.0.0

-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: System
-> Version: 4.0.0.0

-> Name: System.Core
-> Version: 4.0.0.0

***** Here are the assemblies loaded in SecondAppDomain *****

-> Name: mscorlib
-> Version: 4.0.0.0

```

На заметку! Если вы начнете отладку этого проекта (нажатием <F5>), то обнаружите, что во все домены приложений загружаются многие дополнительные сборки, которые задействованы процессом отладки Visual Studio. Запуск проекта на выполнение (нажатием <Ctrl+F5>) приводит к отображению только сборок, напрямую загруженных в каждый домен приложения.

При наличии опыта построения традиционных приложений Windows такое поведение может показаться нелогичным (предполагается, что оба домена приложений имеют доступ к одному и тому же набору сборок). Тем не менее, вспомните, что сборка загружается в *домен приложения*, а не прямо в сам процесс.

Загрузка сборок в специальные домены приложений

Среда CLR всегда, когда требуется, будет загружать сборки в стандартный домен приложения. Однако в случае создания новых доменов приложений с помощью метода `AppDomain.Load()` можно загружать сборки в указанный домен вручную. Кроме того, не следует забывать о возможности вызова метода `AppDomain.ExecuteAssembly()`, который позволяет загрузить сборку *.exe и выполнить метод `Main()`.

Предположим, что необходимо загрузить сборку `CarLibrary.dll` в новый вторичный домен приложения. При условии, что библиотека `CarLibrary.dll` скопирована в папку `bin\Debug` текущего приложения, модифицируем метод `MakeNewAppDomain()` следующим образом (не забыв импортировать пространство имен `System.IO` для получения доступа к классу `FileNotFoundException`):

```
private static void MakeNewAppDomain()
{
    // Создать новый домен приложения в текущем процессе.
    AppDomain newAD = AppDomain.CreateDomain("SecondAppDomain");
    try
    {
        // Загрузить CarLibrary.dll в этот новый домен.
        newAD.Load("CarLibrary");
    }
    catch (FileNotFoundException ex)
    {
        Console.WriteLine(ex.Message);
    }
    // Вывести список всех сборок.
    ListAllAssembliesInAppDomain(newAD);
}
```

Вот как на этот раз выглядит вывод приложения (обратите внимание на присутствие сборки CarLibrary.dll):

```
***** Fun with Custom AppDomains *****
***** Here are the assemblies loaded in CustomAppDomains.exe *****
-> Name: CustomAppDomains
-> Version: 1.0.0.0
-> Name: mscorlib
-> Version: 4.0.0.0
-> Name: System
-> Version: 4.0.0.0
-> Name: System.Core
-> Version: 4.0.0.0
***** Here are the assemblies loaded in SecondAppDomain *****
-> Name: CarLibrary
-> Version: 2.0.0.0
-> Name: mscorlib
-> Version: 4.0.0.0
```

На заметку! Не забывайте, что во время отладки приложения в каждый домен приложения будут загружаться многие дополнительные библиотеки.

Выгрузка доменов приложений программным образом

Важно отметить, что среда CLR не разрешает выгружать индивидуальные сборки .NET. Тем не менее, с помощью метода `AppDomain.Unload()` можно избирательно выгрузить заданный домен приложения из размещающего процесса. В таком случае вместе с доменом приложения по очереди будут выгружаться содержащиеся в нем сборки.

Вспомните, что в типе `AppDomain` определено событие `DomainUnload`, которое инициируется при выгрузке специального домена приложения из содержащего его процесса. Еще одним интересным событием является `ProcessExit`, которое возникает, когда стандартный домен приложения выгружается из процесса (что вполне очевидно влечет за собой завершение самого процесса).

Чтобы реализовать программную выгрузку домена `newAD` из размещающего процесса с получением уведомления об его уничтожении, добавим в метод `MakeNewAppDomain()` приведенную ниже логику:

```
private static void MakeNewAppDomain()
{
    // Создать новый домен приложения в текущем процессе.
    AppDomain newAD = AppDomain.CreateDomain("SecondAppDomain");
    newAD.DomainUnload += (o, s) =>
    {
        Console.WriteLine("The second AppDomain has been unloaded!");
        // Второй домен приложения выгружен!
    };
    try
    {
        // Загрузить CarLibrary.dll в этот новый домен.
        newAD.Load("CarLibrary");
    }
    catch (FileNotFoundException ex)
    {
        Console.WriteLine(ex.Message);
    }
    // Вывести список всех сборок.
    ListAllAssembliesInAppDomain(newAD);
    // Избавиться от этого домена приложения.
    AppDomain.Unload(newAD);
}
```

Если нужно обеспечить уведомление при выгрузке стандартного домена приложения, тогда метод `Main()` понадобится модифицировать для обработки события `ProcessEvent` упомянутого домена:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Custom AppDomains *****\n");
    // Вывести все сборки, загруженные в стандартный домен приложения.
    AppDomain defaultAD = AppDomain.CurrentDomain;
    defaultAD.ProcessExit += (o, s) =>
    {
        Console.WriteLine("Default AD unloaded!");
        // Стандартный домен приложения выгружен!
    };
    ListAllAssembliesInAppDomain(defaultAD);
    MakeNewAppDomain();
    Console.ReadLine();
}
```

Итак, рассмотрение доменов приложений .NET завершено. Напоследок мы рассмотрим еще один уровень разделения, который применяется для группирования объектов в контекстные границы.

Исходный код. Проект `CustomAppDomains` доступен в подкаталоге `Chapter_17`.

Контекстные границы объектов

Как было показано выше, домены приложений — это логические разделы внутри процесса, используемого для размещения сборок .NET. Однако каждый домен приложения может быть дополнительно разделен на многочисленные контекстные границы. По существу контекст .NET предоставляет одиночному домену приложения возможность установки “специфического местоположения” для заданного объекта.

На заметку! В то время как понимание концепции процессов и доменов приложений довольно важно, в большинстве приложений .NET работа с объектными контекстами не требуется. Обзорный материал по объектным контекстам здесь включен лишь для предоставления более полной картины.

С применением контекста среда CLR способна обеспечить надлежащую и согласованную обработку объектов, которые выдвигают специальные требования времени выполнения, за счет перехвата обращений к методам в и из конкретного контекста. Этот уровень перехвата позволяет среде CLR подстраивать текущий вызов метода так, чтобы он соответствовал контекстным настройкам заданного объекта. Например, если определен класс C#, который требует автоматической безопасности к потокам (с использованием атрибута [Synchronization]), то во время выделения памяти для его экземпляра среда CLR будет создавать "синхронизированный контекст".

Подобно тому, как процесс определяет стандартный домен приложения, каждый домен приложения имеет стандартный контекст. Такой стандартный контекст (иногда называемый *контекстом 0*, поскольку в домене приложения он всегда создается первым) применяется для группирования вместе объектов .NET, которые не имеют никаких специфических или уникальных контекстных потребностей. Как и можно было ожидать, подавляющее большинство объектов .NET загружается именно в контекст 0. Если среда CLR определяет, что вновь созданный объект предъявляет специальные требования, тогда она создает внутри размещающего домена приложения новую контекстную границу. На рис. 17.3 показаны отношения между процессом, доменом приложения и контекстом.

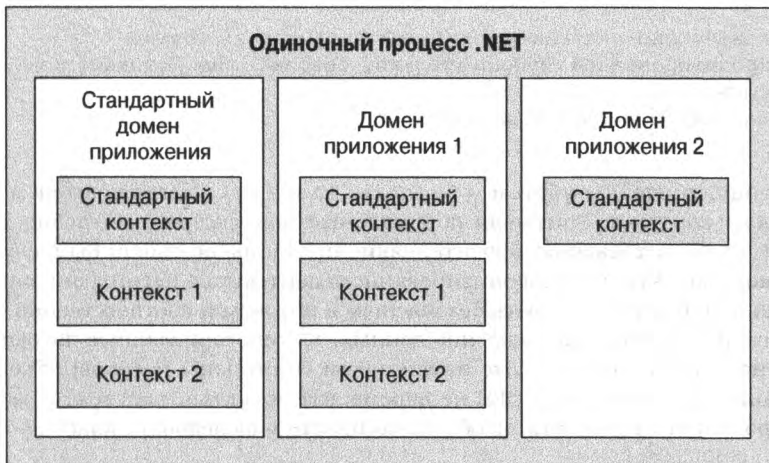


Рис. 17.3. Процессы, домены приложений и контекстные границы

Контекстно-свободные и контекстно-связанные типы

Объекты .NET, которые не нуждаются в какой-то особой контекстной трактовке, называются *контекстно-свободными*. Такие объекты могут быть доступны из любого места внутри размещающего домена приложения без влияния со стороны требований времени выполнения, связанных с объектами. Контекстно-свободные объекты строятся легко, потому что для этого просто ничего не нужно делать (в частности, не декорировать тип контекстными атрибутами и не порождать его от базового класса `System.ContextBoundObject`).

Вот пример:

```
// Контекстно-свободный объект загружается в контекст 0.
class SportsCar{}
```

С другой стороны, объекты, действительно требующие выделения контекста, называются *контекстно-связанными* и должны быть производными от базового класса `System.ContextBoundObject`, который закрепляет тот факт, что интересующий объект может функционировать надлежащим образом только внутри контекста, где он был создан. Учитывая роль контекста .NET, должно быть ясно, что в случае, если по ряду причин контекстно-связанный объект попадает в несовместимый контекст, тогда обязательно произойдет что-то неприемлемое, причем в самый неподходящий момент.

В дополнение к порождению от `System.ContextBoundObject` контекстно-связанный тип будет также декорирован атрибутами .NET специальной категории, которая (вполне ожидаемо) называется *контекстными атрибутами*. Все контекстные атрибуты являются производными от базового класса `ContextAttribute`. Давайте рассмотрим пример.

Определение контекстно-связанного объекта

Предположим, что необходимо определить класс (`SportsCarTS`), который по своей природе автоматически является безопасным к потокам, даже если внутри реализации его членов не была жестко закодирована логика синхронизации потоков. Для этого класс `SportsCarTS` нужно унаследовать от `ContextBoundObject` и применить к нему атрибут `[Synchronization]`:

```
using System.Runtime.Remoting.Contexts;

// Этот контекстно-связанный тип будет загружаться только
// в синхронизированный (следовательно, безопасный к потокам) контекст.
[Synchronization]
class SportsCarTS : ContextBoundObject
{ }
```

Типы, оснащенные атрибутом `[Synchronization]`, загружаются в безопасный к потокам контекст. Учитывая специальные контекстные потребности класса `MyThreadSafeObject`, только вообразите, какие проблемы возникли бы в случае переноса созданного объекта из синхронизированного контекста в несинхронизированный. Объект неожиданно перестает быть безопасным к потокам и соответственно становится кандидатом на массовое повреждение данных, т.к. многочисленные потоки пытаются взаимодействовать с (теперь уже изменчивым со стороны потоков) объектом. Для гарантирования того, что среда CLR не переместит объекты `SportsCarTS` за пределы синхронизированного контекста, необходимо просто унаследовать класс `SportsCarTS` от `ContextBoundObject`.

Исследование контекста объекта

Хотя необходимость в программном взаимодействии с контекстом будет возникать лишь в немногих разрабатываемых вами приложениях, мы рассмотрим иллюстративный пример. Создадим новый проект консольного приложения по имени `ObjectContextApp` и определим в нем контекстно-свободный класс `SportsCar`, а также контекстно-связанный класс `SportsCarTS`:

```
using System;
using System.Runtime.Remoting.Contexts;    // Для типа Context.
using System.Threading;                     // Для типа Thread.
```

```
// Класс SportsCar не имеет никаких специальных
// контекстных потребностей и будет загружаться
// в стандартный контекст домена приложения.
class SportsCar
{
    public SportsCar()
    {
        // Получить информацию о контексте и вывести идентификатор контекста.
        Context ctx = Thread.CurrentContext;
        Console.WriteLine("{0} object in context {1}",
            this.ToString(), ctx.ContextID);
        foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
            Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
    }
}

// SportsCarTS требует загрузки в синхронизированный контекст.
[Synchronization]
class SportsCarTS : ContextBoundObject
{
    public SportsCarTS()
    {
        // Получить информацию о контексте и вывести идентификатор контекста.
        Context ctx = Thread.CurrentContext;
        Console.WriteLine("{0} object in context {1}",
            this.ToString(), ctx.ContextID);
        foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
            Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
    }
}
```

Обратите внимание, что каждый конструктор получает объект Context из текущего потока выполнения с помощью статического свойства Thread.CurrentContext. Имея объект Context, можно вывести статистические данные о контекстной границе, такие как назначенный идентификатор и набор дескрипторов, полученных через свойство Context.ContextProperties. Это свойство возвращает массив объектов, реализующих интерфейс IContextProperty, который открывает доступ к каждому дескриптору через свойство Name. Добавим в метод Main() код для размещения экземпляра каждого класса в памяти:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Context *****\n");
    // Объекты при создании будут отображать контекстную информацию.
    SportsCar sport = new SportsCar();
    Console.WriteLine();

    SportsCar sport2 = new SportsCar();
    Console.WriteLine();

    SportsCarTS synchroSport = new SportsCarTS();
    Console.ReadLine();
}
```

По мере создания объектов конструкторы классов будут выводить различные фрагменты контекстной информации (выводимое свойство LeaseLifeTimeServiceProperty представляет собой низкоуровневый аспект уровня удаленной обработки .NET и может быть проигнорировано):

```

***** Fun with Object Context *****
ObjectContextApp.SportsCar object in context 0
-> Ctx Prop: LeaseLifeTimeServiceProperty
ObjectContextApp.SportsCar object in context 0
-> Ctx Prop: LeaseLifeTimeServiceProperty
ObjectContextApp.SportsCarTS object in context 1
-> Ctx Prop: LeaseLifeTimeServiceProperty
-> Ctx Prop: Synchronization

```

Поскольку класс `SportsCar` не был снабжен атрибутом контекста, среда CLR разместила объекты `sport` и `sport2` в контексте 0 (т.е. в стандартном контексте). Тем не менее, объект `SportsCarTS` загрузился в уникальную контекстную границу (которой был назначен идентификатор контекста, равный 1) с учетом того, что данный контекстно-связанный тип был декорирован атрибутом `[Synchronization]`.

Исходный код. Проект `ObjectContextApp` доступен в подкаталоге `Chapter_17`.

Итоговые сведения о процессах, доменах приложений и контекстах

К настоящему времени вы должны иметь намного лучшее представление о том, как сборка .NET размещается средой CLR. Ниже перечислены основные моменты.

- Процесс .NET размещает один и более доменов приложений. Каждый домен приложения способен содержать любое количество связанных сборок .NET. Домены приложений могут независимо загружаться и выгружаться средой CLR (или программно с помощью типа `System.AppDomain`).
- Домен приложения может состоять из одного и более контекстов. Используя контекст, среда CLR имеет возможность помещать объект с "особыми потребностями" в логический контейнер, чтобы обеспечить удовлетворение его требований времени выполнения.

Если изложенный материал показался слишком низкоуровневым, то не переживайте. По большей части среда CLR самостоятельно занимается всеми деталями процессов, доменов приложений и контекстов. Однако эта информация формирует хороший фундамент для понимания многопоточного программирования на платформе .NET.

Резюме

Задачей главы было исследование особенностей размещения образа исполняемой сборки .NET платформой .NET. Как вы видели, давно существующее понятие процесса Windows было внутренне изменено и адаптировано под потребности CLR. Одиночный процесс (которым можно программно манипулировать посредством типа `System.Diagnostics.Process`) теперь состоит из одного или большего числа доменов приложений, которые представляют изолированные и независимые границы внутри процесса.

Вы узнали, что одиночный процесс может обслуживать несколько доменов приложений, каждый из которых способен размещать и выполнять любое количество связанных сборок. Кроме того, один домен приложения может содержать любое число контекстных границ. Благодаря такому дополнительному уровню изоляции типов среда CLR обеспечивает надлежащую обработку объектов с особыми потребностями во время выполнения.

глава 18

Язык CIL и роль динамических сборок

При построении полномасштабного приложения .NET вы почти наверняка будете использовать C# (или другой управляемый язык, такой как Visual Basic) из-за присущей ему продуктивности и простоты применения. Однако в первой главе было показано, что роль управляемого компилятора заключается в трансляции файлов кода *.cs в код CIL, метаданные типов и манифест сборки. Как выяснилось, CIL представляет собой полноценный язык программирования .NET, который имеет собственный синтаксис, семантику и компилятор (ilasm.exe).

В настоящей главе будет предложен краткий экскурс по этому родному языку платформы .NET. Здесь вы узнаете о различиях между *директивой*, *атрибутом* и *кодом операции* CIL. Затем вы ознакомитесь с ролью возвратного проектирования сборки .NET и разнообразных инструментов программирования на CIL. Остаток главы посвящен основам определения пространств имен, типов и членов с использованием грамматики CIL. В завершение главы исследуется роль пространства имен System.Reflection.Emit и объясняется, как можно динамически конструировать сборки (с помощью инструкций CIL) во время выполнения.

Конечно, необходимость работать с низкоуровневым кодом CIL на повседневной основе будет возникать только у очень немногих программистов. Глава начинается с описания причин, по которым изучение синтаксиса и семантики такого языка .NET может оказаться полезным.

Причины для изучения грамматики языка CIL

Язык CIL является истинным родным языком платформы .NET. При построении сборки .NET с помощью выбранного управляемого языка (C#, VB, F# и т.д.) соответствующий компилятор транслирует исходный код в инструкции CIL. Подобно любому языку программирования CIL предлагает многочисленные лексемы, связанные со структурированием и реализацией. Поскольку CIL представляет собой просто еще один язык программирования .NET, не должен вызывать удивление тот факт, что сборки .NET можно создавать прямо на CIL и компилировать их посредством компилятора CIL (ilasm.exe), который поставляется в составе .NET Framework SDK.

Хотя и верно утверждение о том, что построением полного приложения .NET прямо на CIL занимаются лишь немногие программисты (если вообще такие находятся), изучение этого языка все равно является чрезвычайно интересным занятием. Попросту говоря, чем лучше вы понимаете грамматику CIL, тем больше способны погрузиться в мир расширенной разработки приложений .NET.

Обратившись к конкретным примерам, можно утверждать, что разработчики, разбирающиеся в CIL, обладают следующими навыками.

- Умеют дизассемблировать существующую сборку .NET, редактировать код CIL в ней и заново компилировать обновленную кодовую базу в модифицированный двоичный файл .NET. Скажем, некоторые сценарии могут требовать модификации кода CIL для взаимодействия с расширенными средствами COM.
- Умеют строить динамические сборки с применением пространства имен System.Reflection.Emit. Данный API-интерфейс позволяет генерировать в памяти сборку .NET, которая дополнительно может быть сохранена на диск. Это полезный прием для разработчиков инструментов, которым необходимо генерировать сборки на лету.
- Понимают аспекты CTS, которые не поддерживаются высокоуровневыми управляемыми языками, но существуют на уровне CIL. На самом деле CIL является единственным языком .NET, который позволяет получать доступ ко всем аспектам CTS. Например, за счет использования низкоуровневого кода CIL появляется возможность определения членов и полей глобального уровня (которые не разрешены в C#).

Ради полной ясности нужно еще раз подчеркнуть, что овладеть мастерством работы с языком C# и библиотеками базовых классов .NET можно и без изучения деталей кода CIL. Во многих отношениях знание CIL аналогично знанию языка ассемблера программистом на C (и C++). Те, кто разбирается в низкоуровневых деталях, способны создавать более совершенные решения поставленных задач и глубже понимают лежащую в основе среду программирования (и выполнения). Таким образом, если вы готовы принять вызов, тогда давайте приступим к исследованию внутренних деталей CIL.

На заметку! Имейте в виду, что настоящая глава не планировалась как всеобъемлющее руководство по синтаксису и семантике CIL. Если вам требуется полное изложение темы, то рекомендуется загрузить официальную спецификацию ECMA (ecma-335.pdf) из веб-сайта ECMA International (www.ecma-international.org).

Директивы, атрибуты и коды операций CIL

Когда вы начинаете изучение низкоуровневых языков, таких как CIL, то гарантированно встретите новые (и часто пугающие) названия для знакомых концепций. Например, к этому моменту приведенный ниже набор элементов вы почти наверняка посчитаете ключевыми словами языка C# (и это правильно):

```
{new, public, this, base, get, set, explicit, unsafe, enum, operator, partial}
```

Тем не менее, внимательнее присмотревшись к элементам набора, вы сможете заметить, что в то время как каждый из них действительно является ключевым словом C#, он имеет радикально отличающуюся семантику. Скажем, ключевое слово `enum` определяет производный от `System.Enum` тип, а ключевые слова `this` и `base` позволяют ссылаться на текущий объект и его родительский класс. Ключевое слово `unsafe` применяется для установления блока кода, который не может напрямую отслеживаться средой CLR, а ключевое слово `operator` дает возможность создать скрытый (специально именованный) метод, который будет вызываться, когда используется специфическая операция C# (такая как знак “плюс”).

По разительному контрасту с высокоуровневым языком вроде C# в CIL не просто определен общий набор ключевых слов сам по себе. Напротив, набор лексем, распоз-

наваемых компилятором CIL, подразделяется на следующие три обширные категории, основываясь на их семантике:

- директивы CIL;
- атрибуты CIL;
- коды операций CIL.

Лексемы CIL каждой категории выражаются с применением отдельного синтаксиса и комбинируются для построения допустимой сборки .NET.

Роль директив CIL

Прежде всего, существует набор хорошо известных лексем CIL, которые используются для описания общей структуры сборки .NET. Такие лексемы называются *директивами*. Директивы CIL позволяют информировать компилятор CIL о том, каким образом определять пространства имен, типы и члены, которые будут заполнять сборку.

Синтаксически директивы представляются с применением префикса в виде точки (.), например, `.namespace`, `.class`, `.publickeytoken`, `.override`, `.method`, `.assembly` и т.д. Таким образом, если в файле с расширением `*.il` (общепринятое расширение для файлов кода CIL) указана одна директива `.namespace` и три директивы `.class`, то компилятор CIL сгенерирует сборку, в которой определено единственное пространство имен, содержащее три класса .NET.

Роль атрибутов CIL

Во многих случаях директивы CIL сами по себе недостаточно описательны для того, чтобы полностью выразить определение заданного типа .NET или члена типа. С учетом этого факта многие директивы CIL могут сопровождаться разнообразными *атрибутами* CIL, которые уточняют способ обработки директивы. Например, директива `.class` может быть снабжена атрибутом `public` (для установления видимости типа), атрибутом `extends` (для явного указания базового класса типа) и атрибутом `implements` (для перечисления набора интерфейсов, поддерживаемых данным типом).

На заметку! Не путайте атрибут .NET (глава 15) и атрибут CIL, которые являются двумя совершенно разными понятиями.

Роль кодов операций CIL

После того как сборка .NET, пространство имен и набор типов определены в терминах языка CIL с использованием различных директив и связанных атрибутов, остается только предоставить логику реализации для типов. Это работа *кодов операций*. В традициях других низкоуровневых языков программирования многие коды операций CIL обычно имеют непонятный и совершенно нечитательный вид. Например, для загрузки в память переменной `string` применяется код операции, который имеет не дружественное имя наподобие `LoadString`, а выглядит как `ldstr`.

Справедливости ради следует отметить, что некоторые коды операций CIL довольно естественно отображаются на свои аналоги в C# (например, `box`, `unbox`, `throw` и `sizeof`). Вы увидите, что коды операций CIL всегда используются внутри области реализации члена и в отличие от директив никогда не записываются с префиксом-точкой.

Разница между кодами операций и их мнемоническими эквивалентами в CIL

Как только что объяснялось, для реализации членов отдельно взятого типа применяются коды операций вроде `ldstr`. Однако на самом деле такие лексемы, как `ldstr`, являются *мнемоническими эквивалентами CIL* действительных двоичных кодов операций CIL. Чтобы прояснить разницу, напишем следующий метод C#:

```
static int Add(int x, int y)
{
    return x + y;
}
```

Действие сложения двух чисел в терминах CIL выражается посредством кода операции `0x58`. В том же духе вычитание двух чисел выражается с помощью кода операции `0x59`, а действие по размещению нового объекта в управляемой куче записывается с использованием кода операции `0x73`. С учетом описанной реальности “код CIL”, обрабатываемый JIT-компилятором, в действительности представляет собой не более чем порцию двоичных данных.

К счастью, для каждого двоичного кода операции CIL имеется соответствующий мнемонический эквивалент. Например, вместо кода `0x58` может применяться мнемонический эквивалент `add`, вместо `0x59` — `sub`, а вместо `0x73` — `newobj`. С учетом такой разницы между кодами операций и их мнемоническими эквивалентами декомпиляторы CIL, подобные `ildasm.exe`, транслируют двоичные коды операций сборки в соответствующие им мнемонические эквиваленты CIL. Вот как `ildasm.exe` представит в CIL предыдущий метод `Add()`, написанный на языке C# (в зависимости от версии .NET вывод может отличаться):

```
.method private hidebysig static int32 Add(int32 x,
    int32 y) cil managed
{
    // Code size 9 (0x9)
    .maxstack 2
    .locals init ([0] int32 CS$1$0000)
    IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: ldarg.1
    IL_0003: add
    IL_0004: stloc.0
    IL_0005: br.s IL_0007
    IL_0007: ldloc.0
    IL_0008: ret
}
```

Если вы не занимаетесь разработкой исключительно низкоуровневого программного обеспечения .NET (вроде специального управляемого компилятора), то иметь дело с числовыми двоичными кодами операций CIL никогда не придется. В практическом смысле программисты .NET, говоря о “кодах операций CIL”, имеют в виду набор дружественных строковых мнемонических эквивалентов (что и делается в настоящей книге), а не лежащие в основе числовые значения.

Заталкивание и выталкивание: основанная на стеке природа CIL

В языках .NET высокого уровня (таких как C#) предпринимается попытка насколько возможно скрыть из виду низкоуровневые детали CIL. Один из особенно хорошо скрываемых аспектов — тот факт, что CIL является языком программирования, основанным на использовании стека. Вспомните из исследования пространств имен коллекций (см. главу 9), что класс `Stack<T>` может применяться для помещения значения в стек, а также для извлечения самого верхнего значения из стека с целью последующего использования. Разумеется, программисты на языке CIL не работают с объектом типа `Stack<T>` для загрузки и выгрузки вычисляемых значений, но применяют образ действий, похожий на заталкивание и выталкивание.

Формально сущность, используемая для хранения набора вычисляемых значений, называется *виртуальным стеком выполнения*. Вы увидите, что CIL предоставляет несколько кодов операций, которые служат для помещения значения в стек; такой процесс именуется *загрузкой*. Кроме того, в CIL определен набор дополнительных кодов операций, которые перемещают самое верхнее значение из стека в память (скажем, в локальную переменную), применяя процесс под названием *сохранение*.

В мире CIL невозможно напрямую получать доступ к элементам данных, включая локально определенные переменные, входные аргументы методов и данные полей типа. Вместо этого элемент данных должен быть явно загружен в стек и затем извлекаться оттуда для использования в более позднее время (запомните упомянутое требование, поскольку оно содействует в понимании, почему блок кода CIL может выглядеть несколько избыточным).

На заметку! Вспомните, что код CIL не выполняется напрямую, а компилируется по требованию. Во время компиляции кода CIL многие избыточные аспекты реализации оптимизируются. Более того, если для текущего проекта включена оптимизация кода (на вкладке Build (Сборка) окна свойств проекта в Visual Studio), то компилятор будет также удалять разнообразные избыточные детали CIL.

Чтобы понять, каким образом CIL задействует модель обработки на основе стека, создадим простой метод C# по имени `PrintMessage()`, который не принимает аргументов и возвращает `void`. Внутри его реализации будет просто выводиться в стандартный выходной поток значение локальной переменной:

```
public void PrintMessage()
{
    string myMessage = "Hello.";
    Console.WriteLine(myMessage);
}
```

Если просмотреть код CIL, который получился в результате трансляции метода `PrintMessage()` компилятором C#, то первым делом обнаружится, что в нем определяется ячейка памяти для локальной переменной с помощью директивы `.locals`. Затем локальная строка загружается и сохраняется в этой локальной переменной с применением кодов операций `ldstr` (загрузить строку) и `stloc.0` (сохранить текущее значение в локальной переменной, находящейся в ячейке 0).

Далее с помощью кода операции `ldloc.0` (загрузить локальный аргумент по индексу 0) значение (по индексу 0) загружается в память для использования в вызове метода `System.Console.WriteLine()`, представленном кодом операции `call`. Наконец, посредством кода операции `ret` производится возвращение из функции. Ниже показан

(прокомментированный) код CIL для метода `PrintMessage()` (ради краткости из листинга были удалены коды операций `nop`):

```
.method public hidebysig instance void PrintMessage() cil managed
{
    .maxstack 1
    // Определить локальную переменную типа string (по индексу 0).
    .locals init ([0] string myMessage)

    // Загрузить в стек строку со значением "Hello.".
    ldstr "Hello."

    // Сохранить строковое значение из стека в локальной переменной.
    stloc.0

    // Загрузить значение по индексу 0.
    ldloc.0

    // Вызвать метод с текущим значением.
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

На заметку! Как видите, язык CIL поддерживает синтаксис комментариев в виде двойной ко-
сой черты (и вдобавок синтаксис `/* ... */`). Как и в C#, комментарии в коде компилятор CIL
игнорирует.

Теперь, когда вы знаете основы директив, атрибутов и кодов операций CIL, давайте приступим к практическому программированию на CIL, начав с рассмотрения темы возвратного проектирования.

Возвратное проектирование

В главе 1 было показано, как применять утилиту `ildasm.exe` для просмотра кода CIL, сгенерированного компилятором C#. Тем не менее, вы можете даже не подозревать, что эта утилита позволяет сбрасывать код CIL, содержащийся внутри загруженной в нее сборки, во внешний файл. Полученный подобным образом код CIL можно редактировать и компилировать заново с помощью компилятора CIL (`ilasm.exe`).

Выражаясь формально, такой прием называется *возвратным проектированием* и может быть полезен в избранных обстоятельствах, которые перечислены ниже.

- Вам необходимо модифицировать сборку, исходный код которой больше не доступен.
- Вы работаете с далеким от идеала компилятором языка .NET, который генерирует неэффективный (или явно некорректный) код CIL, поэтому нужно изменить кодовую базу.
- Вы конструируете библиотеку взаимодействия с COM и хотите учесть ряд атрибутов COM IDL, которые были утеряны во время процесса преобразования (такие как COM-атрибут `[helpstring]`).

Чтобы продемонстрировать процесс возвратного проектирования, создадим в простом текстовом редакторе новый файл кода C# (`HelloProgram.cs`) и определим в нем следующий тип класса. (При желании можете создать новый проект консольного приложения в Visual Studio, но только удалите файл `AssemblyInfo.cs`, чтобы уменьшить объем генерируемого кода CIL.)

```
// Простое консольное приложение C#.
using System;

// Обратите внимание, что с целью упрощения генерируемого
// кода CIL класс не помещается в пространство имен.
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello CIL code!");
        Console.ReadLine();
    }
}
```

Сохраним файл в удобном месте (например, C:\RoundTrip) и скомпилируем программу, используя csc.exe:

```
csc HelloProgram.cs
```

Теперь откроем файл HelloProgram.exe в ildasm.exe и путем выбора пункта меню File⇒Dump (Файл⇒Сбросить) сохраним низкоуровневый код CIL в новый файл *.il (HelloProgram.il) внутри той же папки, в которой находится скомпилированная сборка (оставив нетронутыми стандартные настройки в диалоговом окне сохранения).

На заметку! Во время сбрасывания содержимого сборки в файл утилита ildasm.exe также генерирует файл *.res. Такие ресурсные файлы можно игнорировать (и удалять), поскольку в текущей главе они не применяются. В них содержится низкоуровневая информация, касающаяся безопасности CLR (помимо прочих данных).

Теперь можно просмотреть файл HelloProgram.il в любом текстовом редакторе. Вот его содержимое (для удобства оно слегка переформатировано и снабжено комментариями):

```
// Ссылаемые сборки.
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}

// Наша сборка.
.assembly HelloProgram
{
    /**** Ради ясности данные TargetFrameworkAttribute удалены! ****/
    .hash algorithm 0x00008004
    .ver 0:0:0:0
}

.module HelloProgram.exe
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003
.corflags 0x00000003

// Определение класса Program.
.class private auto ansi beforefieldinit Program
    extends [mscorlib]System.Object
{

```

```
.method private hidebysig static void Main(string[] args) cil managed
{
    // Помечает этот метод как точку входа исполняемой сборки.
    .entrypoint
    .maxstack 8
    IL_0000: nop
    IL_0001: ldstr "Hello CIL code!"
    IL_0006: call void [mscorlib]System.Console::WriteLine(string)
    IL_000b: nop
    IL_000c: call string [mscorlib]System.Console::ReadLine()
    IL_0011: pop
    IL_0012: ret
}

// Стандартный конструктор.
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call instance void [mscorlib]System.Object::.ctor()
    IL_0006: ret
}
}
```

Обратите внимание, что файл *.il начинается с объявления всех внешних сборок, на которые ссылается текущая скомпилированная сборка. Здесь имеется единственная лексема `.assembly extern`, установленная для постоянно присутствующей сборки `mscorlib.dll`. Конечно, если бы в библиотеке классов использовались типы из других сборок, тогда были бы определены дополнительные директивы `.assembly extern`.

Далее обнаруживается формальное определение сборки `HelloProgram.exe`, которой был назначен стандартный номер версии `0.0.0.0` (потому что не было указано какое-либо значение с помощью атрибута `[AssemblyVersion]`). Вдобавок сборка описана с применением разнообразных директив CIL (`.module`, `.imagebase` и т.д.).

После документирования внешних ссылаемых сборок и определения текущей сборки находится определение типа `Program`. Обратите внимание, что директива `.class` имеет различные атрибуты (многие из которых на самом деле необязательны) вроде приведенного ниже атрибута `extends`, который указывает базовый класс для типа:

```
.class private auto ansi beforefieldinit Program
    extends [mscorlib]System.Object
{ ... }
```

Основной объем кода CIL представляет реализацию стандартного конструктора класса и метода `Main()`, которые оба определены (частично) посредством директивы `.method`. После того, как эти члены были определены с использованием корректных директив и атрибутов, они реализуются с применением разнообразных кодов операций.

Важно понимать, что при взаимодействии с типами .NET (такими как `System.Console`) в CIL всегда необходимо использовать полностью заданное имя типа. Более того, полностью заданное имя типа всегда должно предваряться префиксом в форме дружественного имени сборки, где определен тип (в квадратных скобках). Взгляните на следующую реализацию метода `Main()` в CIL:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 8
```

```

IL_0000: nop
IL_0001: ldstr "Hello CIL code!"
IL_0006: call void [mscorlib]System.Console::WriteLine(string)
IL_000b: nop
IL_000c: call string [mscorlib]System.Console::ReadLine()
IL_0011: pop
IL_0012: ret
}

```

В реализации стандартного конструктора на языке CIL применяется еще одна инструкция, связанная с загрузкой — `ldarg.0`. В данном случае значение, загружаемое в стек, представляет собой не специальную переменную, указанную вами, а ссылку на текущий объект (более подробно об этом речь пойдет позже). Также обратите внимание, что в стандартном конструкторе явно производится вызов конструктора базового класса, которым в данном случае является хорошо знакомый класс `System.Object`:

```

.method public hidebysig specialname rtspecialname
  instance void .ctor() cil managed
{
  .maxstack 8
  IL_0000: ldarg.0
  IL_0001: call instance void [mscorlib]System.Object::.ctor()
  IL_0006: ret
}

```

Роль меток в коде CIL

Вы определенно заметили, что каждая строка в коде реализации предваряется лексемой в форме `IL_XXX:` (например, `IL_0000:`, `IL_0001:` и т.д.). Такие лексемы называются *метками кода* и могут именоваться в любой выбранной вами манере (при условии, что они не дублируются внутри области действия члена). При сбросе содержимого сборки в файл утилита `ildasm.exe` автоматически генерирует метки кода, которые следуют соглашению об именовании вида `IL_XXX:`. Однако их можно заменить более описательными маркерами, например:

```

.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack 8
  Nothing_1: nop
  Load_String: ldstr "Hello CIL code!"
  PrintToConsole: call void [mscorlib]System.Console::WriteLine(string)
  Nothing_2: nop
  WaitFor_KeyPress: call string [mscorlib]System.Console::ReadLine()
  RemoveValueFromStack: pop
  Leave_Function: ret
}

```

Говоря по существу, большая часть меток кода совершенно не обязательна. Единственный случай, когда метки кода по-настоящему необходимы, связан с написанием кода CIL, в котором используются разнообразные конструкции ветвления или закикливания, т.к. с помощью меток можно указывать, куда должен быть направлен поток логики. В текущем примере все автоматически сгенерированные метки кода можно удалить безо всяких последствий:


```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 8
    nop
    ldstr "Hello CIL code!"
    call void [mscorlib]System.Console::WriteLine(string)
    nop
    call string [mscorlib]System.Console::ReadLine()
    pop
    ret
}
```

Взаимодействие с CIL: модификация файла *.il

Теперь, когда вы имеете представление о том, из чего состоит базовый файл CIL, давайте завершим эксперимент с возвратным проектированием. Нашей целью является модификация кода CIL в существующем файле *.il описанным далее образом.

1. Добавление ссылки на сборку System.Windows.Forms.dll.
2. Загрузка локальной строки внутри метода Main().
3. Вызов метода System.Windows.Forms.MessageBox.Show() с передачей локальной строковой переменной в качестве аргумента.

Первым делом понадобится добавить новую директиву .assembly (уточненную атрибутом extern), которая указывает, что нашей сборке требуется сборка System.Windows.Forms.dll. Для этого непосредственно после ссылки на внешнюю сборку mscorlib в файле *.il необходимо поместить такой код:

```
.assembly extern System.Windows.Forms
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
    .ver 4:0:0:0
}
```

Имейте в виду, что значение, назначаемое директиве .ver, может отличаться в зависимости от версии платформы .NET, установленной на машине разработки. Здесь вы видите, что применяется сборка System.Windows.Forms.dll версии 4.0.0.0, которая имеет маркер открытого ключа B77A5C561934E089. Открыв GAC (см. главу 14) и отыскав версию сборки System.Windows.Forms.dll, можно просто скопировать оттуда корректный номер версии и маркер открытого ключа.

Теперь нужно изменить текущую реализацию метода Main(), для чего следует найти данный метод внутри файла *.il и удалить текущий код реализации (оставив незатронутыми только директивы .maxstack и .entrypoint):

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 8
    // Написать новый код CIL!
}
```

Цель в том, чтобы поместить новую строку в стек и вызвать метод MessageBox.Show() вместо Console.WriteLine(). Вспомните, что при ссылке на внешний тип должно указываться его полностью заданное имя (в сочетании с дружественным именем сборки). Также обратите внимание, что в терминах CIL каждый вызов метода снабжается полностью заданным возвращаемым типом.

Учитывая сказанное, приведем метод `Main()` к следующему виду:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 8
    ldstr "CIL is way cool"
    call valuetype [System.Windows.Forms]
        System.Windows.Forms.DialogResult
        [System.Windows.Forms]
        System.Windows.Forms.MessageBox::Show(string)
    pop
    ret
}
```

Фактически код CIL был модифицирован для соответствия такому определению класса C#:

```
class Program
{
    static void Main(string[] args)
    {
        System.Windows.Forms.MessageBox.Show("CIL is way cool");
    }
}
```

Компиляция кода CIL с помощью `ilasm.exe`

После сохранения обновленного файла `*.il` можно скомпилировать новую сборку .NET, используя утилиту `ilasm.exe` (компилятор CIL). Компилятор CIL поддерживает многочисленные параметры командной строки (их можно вывести, указав опцию `-?`), наиболее интересные из которых описаны в табл. 18.1.

Таблица 18.1. Распространенные параметры командной строки утилиты `ilasm.exe`

Параметр	Описание
<code>/debug</code>	Позволяет включить отладочную информацию (такую как имена локальных переменных и аргументов, а также номера строк)
<code>/dll</code>	Позволяет генерировать в качестве вывода файл <code>*.dll</code>
<code>/exe</code>	Позволяет генерировать в качестве вывода файл <code>*.exe</code> . Это стандартная опция, которую можно не указывать
<code>/key</code>	Позволяет компилировать сборку со строгим именем, применяя заданный файл <code>*.snk</code>
<code>/output</code>	Позволяет указать имя и расширение выходного файла. Если флаг <code>/output</code> не используется, то результирующее имя файла (без расширения) будет таким же, как имя первого исходного файла

Чтобы скомпилировать модифицированный файл `HelloProgram.il` в новую .NET-сборку `*.exe`, необходимо ввести в окне командной строки следующую команду:

```
ilasm /exe HelloProgram.il /output=NewAssembly.exe
```

В случае успешной компиляции на экране появится такой отчет:

```
Microsoft (R) .NET Framework IL Assembler. Version 4.7.2528.0
Copyright (c) Microsoft Corporation. All rights reserved.
Assembling 'HelloProgram.il' to EXE --> 'NewAssembly.exe'
Source file is UTF-8

Assembled method Program::Main
```

```

Assembled method Program::.ctor
Creating PE file
Emitting classes:
Class 1: Program

Emitting fields and methods:
Global
Class 1 Methods: 2;

Emitting events and properties:
Global
Class 1
Writing PE file
Operation completed successfully

```

На этом этапе новое приложение можно запустить. Естественно, сообщение теперь отображается в отдельном окне сообщения, а не в окне консоли. Хотя приведенный простой пример не является особенно впечатляющим, он иллюстрирует один из сценариев применения возвратного проектирования на CIL.

Роль инструмента `peverify.exe`

При построении либо изменении сборок с использованием кода CIL рекомендуется всегда проверять, является ли скомпилированный двоичный образ правильно оформленным образом .NET, с помощью утилиты командной строки `peverify.exe`:

```
peverify NewAssembly.exe
```

Утилита `peverify.exe` проверяет достоверность всех кодов операций CIL внутри указанной сборки. Например, CIL требует, чтобы перед выходом из функции стек вычислений был пустым. Если вы забудете извлечь любые оставшиеся значения, то компилятор `ilasm.exe` все равно сгенерирует сборку (поскольку компилятор заботит только синтаксис). С другой стороны, утилита `peverify.exe` контролирует правильность семантики, и если стек не был опустошен перед выходом из функции, тогда она уведомит об этом до запуска кодовой базы.

Исходный код. Пример `RoundTrip` доступен в подкаталоге `Chapter_18`.

Директивы и атрибуты CIL

После демонстрации применения утилит `ildasm.exe` и `ilasm.exe` при возвратном проектировании можно переходить к более детальному исследованию синтаксиса и семантики CIL. В последующих разделах будет поэтапно рассматриваться процесс создания специального пространства имен, содержащего набор типов. Тем не менее, для простоты типы пока не будут иметь логики реализации своих членов. Разобравшись с созданием простых типов, внимание можно будет переключить на процесс определения “реальных” членов с использованием кодов операций CIL.

Указание ссылок на внешние сборки в CIL

Создадим в текстовом редакторе новый файл по имени `CILTypes.il`. Первой задачей в проекте CIL является перечисление внешних сборок, которые будут задействованы текущей сборкой. В рассматриваемом примере применяются только типы, находящиеся внутри сборки `mscorlib.dll`. В новом файле понадобится указать директиву `.assembly` с уточняющим атрибутом `external`. При добавлении ссылки на сборку со строгим именем, подобную `mscorlib.dll`, должны быть также указаны директивы `.publickeytoken` и `.ver:`

```
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
```

На заметку! Строго говоря, явно добавлять ссылку на внешнюю сборку `mscorlib.dll` не обязательно, потому что компилятор `ilasm.exe` сделает это автоматически. Однако для всех остальных внешних библиотек .NET, требующихся в проекте CIL, должны быть предусмотрены соответствующие директивы `.assembly extern`.

Определение текущей сборки в CIL

Следующее действие заключается в определении создаваемой сборки с использованием директивы `.assembly`. В простейшем случае сборка может быть определена за счет указания дружественного имени двоичного файла:

```
// Наша сборка.
.assembly CILTypes { }
```

В то время как такой код действительно определяет новую сборку .NET, обычно внутри объявления будут помещаться дополнительные директивы. В рассматриваемом примере определение сборки необходимо снабдить номером версии 1.0.0.0 посредством директивы `.ver` (обратите внимание, что числа в номере версии отделяются друг от друга двоеточиями, а не точками, как принято в C#):

```
// Наша сборка.
.assembly CILTypes
{
    .ver 1:0:0:0
}
```

Из-за того, что сборка `CILTypes` является однофайловой (см. главу 14), ее определение завершается с применением следующей директивы `.module`, которая обозначает официальное имя двоичного файла .NET, `CILTypes.dll`:

```
// Наша сборка.
.assembly CILTypes
{
    .ver 1:0:0:0
}
// Модуль нашей однофайловой сборки.
.module CILTypes.dll
```

Кроме `.assembly` и `.module` существуют директивы CIL, которые позволяют дополнительно уточнять общую структуру создаваемого двоичного файла .NET. В табл. 18.2 перечислены некоторые распространенные директивы подобного рода.

Таблица 18.2. Дополнительные директивы, связанные со сборками

Директива	Описание
<code>.resources</code>	Если сборка использует внутренние ресурсы (такие как растровые изображения или таблицы строк), то данная директива применяется для указания имени файла, в котором содержатся ресурсы, подлежащие встраиванию в сборку
<code>.subsystem</code>	Эта директива CIL служит для указания предпочитаемого пользовательского интерфейса, в рамках которого желательно запускать сборку. Например, значение 2 указывает, что сборка должна выполняться в приложении с графическим пользовательским интерфейсом, а значение 3 — в консольном приложении

Определение пространств имен в CIL

После определения внешнего вида и поведения сборки (а также обязательных внешних ссылок) можно создать пространство имен .NET (MyNamespace), используя директиву `.namespace`:

```
// Наша сборка имеет единственное пространство имен.
.namespace MyNamespace {}
```

Подобно C# определения пространств имен CIL могут быть вложены в другие пространства имен. Хотя здесь нет нужды определять корневое пространство имен, ради интереса посмотрим, как создать корневое пространство имен MyCompany:

```
.namespace MyCompany
{
    .namespace MyNamespace {}
}
```

Как и C#, язык CIL позволяет определить вложенное пространство имен следующим образом:

```
// Определение вложенного пространства имен.
.namespace MyCompany.MyNamespace {}
```

Определение типов классов в CIL

Пустые пространства имен не особенно интересны, поэтому давайте рассмотрим процесс определения типов классов в CIL. Для определения нового типа класса предназначена директива `.class`. Тем не менее, эта простая директива может быть декорирована многочисленными дополнительными атрибутами, уточняющими природу типа. В целях иллюстрации добавим в наше пространство имен открытый класс под названием MyBaseClass. Как и в C#, если базовый класс явно не указан, то тип автоматически становится производным от `System.Object`:

```
.namespace MyNamespace
{
    // Предполагается базовый класс System.Object.
    .class public MyBaseClass {}
}
```

При построении типа, производного не от класса `System.Object`, применяется атрибут `extends`. Для ссылки на тип, определенный внутри той же самой сборки, язык CIL требует использования полностью заданного имени (однако если базовый тип находится внутри той же самой сборки, то префикс в виде дружественного имени сборки можно не указывать). Следовательно, демонстрируемая ниже попытка расширения MyBaseClass в результате дает ошибку на этапе компиляции:

```
// Этот код не скомпилируется!
.namespace MyNamespace
{
    .class public MyBaseClass {}
    .class public MyDerivedClass
        extends MyBaseClass {}
}
```

Чтобы корректно определить родительский класс для MyDerivedClass, потребуется указать полностью заданное имя MyBaseClass:

```
// Уже лучше!
namespace MyNamespace
{
    .class public MyBaseClass {}
    .class public MyDerivedClass
        extends MyNamespace.MyBaseClass {}
}
```

В дополнение к атрибутам `public` и `extends` определение класса CIL может иметь множество добавочных квалификаторов, которые управляют видимостью типа, компоновкой полей и т.д. В табл. 18.3 описаны избранные атрибуты, которые могут применяться в сочетании с директивой `.class`.

Таблица 18.3. Избранные атрибуты, используемые вместе с директивой `.class`

Атрибут	Описание
<code>public, private, nested assembly, nested famandassem, nested family, nested famorassem, nested public, nested private</code>	Эти атрибуты применяются для указания видимости заданного типа. Нетрудно заметить, что язык CIL предлагает много других возможностей помимо доступных в C#. За дополнительными сведениями обращайтесь в документ ECMA 335
<code>abstract, sealed</code>	Эти два атрибута могут быть присоединены к директиве <code>.class</code> для определения соответственно абстрактного или запечатанного класса
<code>auto, sequential, explicit</code>	Эти атрибуты инструктируют среду CLR о том, как размещать данные полей в памяти. Для типов классов подходит стандартный флаг <code>auto</code> . Изменение стандартной установки может быть удобно в случае использования <code>P/Invoke</code> для обращения к неуправляемому коду C
<code>extends, implements</code>	Эти атрибуты позволяют определять базовый класс для типа (посредством <code>extends</code>) и реализовывать интерфейс в типе (с помощью <code>implements</code>)

Определение и реализация интерфейсов в CIL

Несколько странно, но типы интерфейсов в CIL определяются с применением директивы `.class`. Тем не менее, когда директива `.class` декорирована атрибутом `interface`, тип трактуется как интерфейсный тип CTS. После определения интерфейс можно приывать к типу класса или структуры с использованием атрибута `implements`:

```
.namespace MyNamespace
{
    // Определение интерфейса.
    .class public interface IMyInterface {}

    // Простой базовый класс.
    .class public MyBaseClass {}

    // Теперь MyDerivedClass реализует IMyInterface
    // и расширяет MyBaseClass.
    .class public MyDerivedClass
        extends MyNamespace.MyBaseClass
        implements MyNamespace.IMyInterface {}
}
```

На заметку! Конструкция `extends` должна предшествовать конструкции `implements`. Кроме того, в конструкции `implements` может содержаться список интерфейсов с разделителями запятыми.

Вспомните из главы 8, что интерфейсы могут выступать в роли базовых для других типов интерфейсов, позволяя строить иерархии интерфейсов. Однако вопреки возможному ожиданию применять атрибут `extends` для порождения интерфейса A от интерфейса B в CIL нельзя. Атрибут `extends` используется только для указания базового класса типа. Когда интерфейс необходимо расширить, снова будет применяться атрибут `implements`, например:

```
// Расширение интерфейсов в CIL.
.class public interface IMyInterface {}

.class public interface IMyOtherInterface
    implements MyNamespace.IMyInterface {}
```

Определение структур в CIL

Директива `.class` может использоваться для определения любой структуры CTS, если тип расширяет `System.ValueType`. Кроме того, такая директива `.class` должна уточняться атрибутом `sealed` (учитывая, что структуры никогда не могут выступать в роли базовых для других типов значений). Если попытаться поступить иначе, тогда компилятор `ilasm.exe` выдаст сообщение об ошибке.

```
// Определение структуры всегда является запечатанным.
.class public sealed MyStruct
    extends [mscorlib]System.ValueType{}
```

Имейте в виду, что в CIL предусмотрен сокращенный синтаксис для определения типа структуры. В случае применения атрибута `value` новый тип автоматически становится производным от `[mscorlib]System.ValueType`. Следовательно, тип `MyStruct` можно было бы определить и так:

```
// Сокращенный синтаксис объявления структуры.
.class public sealed value MyStruct{}
```

Определение перечислений в CIL

Перечисления .NET порождены от класса `System.Enum`, который является `System.ValueType` (и потому также должен быть запечатанным). Чтобы определить перечисление в CIL, необходимо просто расширить `[mscorlib]System.Enum`:

```
// Перечисление.
.class public sealed MyEnum
    extends [mscorlib]System.Enum{}
```

Подобно структурам перечисления могут быть определены с помощью сокращенного синтаксиса, используя атрибут `enum`:

```
// Сокращенный синтаксис определения перечисления.
.class public sealed enum MyEnum{}
```

Вскоре будет показано, как указывать пары “имя-значение” перечисления.

На заметку! Еще один фундаментальный тип .NET — делегат — также имеет специфическое представление в CIL. Подробности приведены в главе 10.

Определение обобщений в CIL

Обобщенные типы также имеют собственное представление в синтаксисе CIL. Вспомните из главы 9, что обобщенный тип или член может иметь один и более параметров типа. Например, в типе `List<T>` определен один параметр типа, а в `Dictionary<TKey, TValue>` — два. В CIL количество параметров типа указывается с применением символа обратной одиночной кавычки (```), за которым следует число, представляющее количество параметров типа. Как и в C#, действительные значения параметров типа заключаются в угловые скобки.

На заметку! На большинстве клавиатур символ ``` находится на клавише, расположенной над клавишей `<Tab>` (и слева от клавиши `<1>`).

Например, предположим, что требуется создать переменную `List<T>`, где `T` — тип `System.Int32`. В C# пришлось бы написать такой код:

```
void SomeMethod()
{
    List<int> myInts = new List<int>();
}
```

В CIL необходимо поступить следующим образом (этот код может находиться внутри любого метода CIL):

```
// В C#: List<int> myInts = new List<int>();
newobj instance void class [mscorlib]
    System.Collections.Generic.List`1<int32>::.ctor()
```

Обратите внимание, что обобщенный класс определен как `List`1<int32>`, поскольку `List<T>` имеет единственный параметр типа. А вот как определить тип `Dictionary<string, int>`:

```
// В C#: Dictionary<string, int> d = new Dictionary<string, int>();
newobj instance void class [mscorlib]
    System.Collections.Generic.Dictionary`2<string,int32>::.ctor()
```

Рассмотрим еще один пример: пусть имеется обобщенный тип, использующий в качестве параметра типа другой обобщенный тип. Код CIL выглядит следующим образом:

```
// В C#: List<List<int>> myInts = new List<List<int>>();
newobj instance void class [mscorlib]
    System.Collections.Generic.List`1<class
        [mscorlib]System.Collections.Generic.List`1<int32>>::.ctor()
```

Компиляция файла `CilTypes.il`

Несмотря на то что к определенным ранее типам пока не были добавлены члены или код реализации, файл `*.il` можно скомпилировать в .NET-сборку `*.dll` (так и нужно поступать ввиду отсутствия метода `Main()`). Для этого необходимо открыть окно командной строки и ввести показанную ниже команду для запуска `ilasm.exe`:

```
ilasm /dll CilTypes.il
```

Затем можно открыть скомпилированную сборку в `ildasm.exe`, чтобы удостовериться в создании каждого типа. После проверки содержимого сборки понадобится запустить в ее отношении утилиту `peverify.exe`:

```
peverify CilTypes.dll
```


Обратите внимание на сообщения об ошибках, т.к. все типы пусты. Вот частичный вывод:

```

Microsoft (R) .NET Framework PE Verifier. Version 4.0.30319.33440
Copyright (c) Microsoft Corporation. All rights reserved.

[MD]: Error: Value class has neither fields nor size parameter. [token:0x02000005]
    Ошибка: Класс значения не имеет ни полей, ни параметра размера.
[MD]: Error: Enum has no instance field. [token:0x02000006]
    Ошибка: Перечисление не имеет полей экземпляра.

...

```

Чтобы понять, каким образом заполнить тип содержимым, сначала необходимо ознакомиться с фундаментальными типами данных CIL.

Соответствия между типами данных в библиотеке базовых классов .NET, C# и CIL

В табл. 18.4 показано, как базовые классы .NET отображаются на соответствующие ключевые слова C#, а ключевые слова C# — на их представления в CIL. Кроме того, для каждого типа CIL приведено сокращенное константное обозначение. Как вы вскоре увидите, на такие константы часто ссылаются многие коды операций CIL.

Таблица 18.4. Отображение базовых классов .NET на ключевые слова C# и ключевых слов C# на CIL

Базовый класс .NET	Ключевое слово C#	Представление CIL	Константное обозначение CIL
System.SByte	sbyte	int8	I1
System.Byte	byte	unsigned int8	U1
System.Int16	short	int16	I2
System.UInt16	ushort	unsigned int16	U2
System.Int32	int	int32	I4
System.UInt32	uint	unsigned int32	U4
System.Int64	long	int64	I8
System.UInt64	ulong	unsigned int64	U8
System.Char	char	char	CHAR
System.Single	float	float32	R4
System.Double	double	float64	R8
System.Boolean	bool	bool	BOOLEAN
System.String	string	string	–
System.Object	object	object	–
System.Void	void	void	VOID

На заметку! Типы System.IntPtr и System.UIntPtr отображаются на собственные типы int и unsigned int в CIL (это полезно знать, т.к. они интенсивно применяются во многих сценариях взаимодействия с COM и P/Invoke).

Определение членов типов в CIL

Как вам уже известно, типы .NET могут поддерживать разнообразные члены. Перечисления содержат набор пар “имя-значение”. Структуры и классы могут иметь конструкторы, поля, методы, свойства, статические члены и т.д. В предшествующих семнадцати главах книги вы уже видели частичные определения в CIL упомянутых элементов, но давайте еще раз кратко повторим, каким образом различные члены отображаются на примитивы CIL.

Определение полей данных в CIL

Перечисления, структуры и классы могут поддерживать поля данных. Во всех случаях для их определения будет использоваться директива `.field`. Например, добавим к перечислению `MyEnum` следующие три пары “имя-значение” (обратите внимание, что значения указаны в круглых скобках):

```
.class public sealed enum MyEnum
{
    .field public static literal valuetype
    MyNamespace.MyEnum A = int32(0)
    .field public static literal valuetype
    MyNamespace.MyEnum B = int32(1)
    .field public static literal valuetype
    MyNamespace.MyEnum C = int32(2)
}
```

Поля, находящиеся внутри области действия производного от `System.Enum` типа .NET, уточняются с применением атрибутов `static` и `literal`. Как не трудно догадаться, эти атрибуты указывают, что данные полей должны быть фиксированными значениями, доступными только из самого типа (например, `MyEnum.A`).

На заметку! Значения, присваиваемые полям в перечислении, могут быть также представлены в шестнадцатеричном формате с префиксом `0x`.

Конечно, когда нужно определить элемент поля данных внутри класса или структуры, вы не ограничены только открытыми статическими литеральными данными. Например, класс `MyBaseClass` можно было бы модифицировать для поддержки двух закрытых полей данных уровня экземпляра со стандартными значениями:

```
.class public MyBaseClass
{
    .field private string stringField = "hello!"
    .field private int32 intField = int32(42)
}
```

Как и в C#, поля данных класса будут автоматически инициализироваться подходящими стандартными значениями. Чтобы предоставить пользователю объекта возможность указывать собственные значения во время создания закрытых полей данных, потребуется создать специальные конструкторы.

Определение конструкторов типа в CIL

Спецификация CTS поддерживает создание конструкторов как уровня экземпляра, так и уровня класса (статических). В CIL конструкторы уровня экземпляра представляются с использованием лексемы `.ctor`, тогда как конструкторы уровня класса — посредством лексемы `.cctor` (class constructor — конструктор класса). Обе лексемы CIL

должны сопровождаться атрибутами `rtspecialname` (return type special name — специальное имя возвращаемого типа) и `specialname`. Упомянутые атрибуты применяются для обозначения специфической лексемы CIL, которая может трактоваться уникальным образом в любом отдельно взятом языке .NET. Например, в языке C# конструкторы не определяют возвращаемый тип, но в CIL возвращаемым значением конструктора на самом деле является `void`:

```
.class public MyBaseClass
{
    .field private string stringField
    .field private int32 intField

    .method public hidebysig specialname rtspecialname
        instance void .ctor(string s, int32 i) cil managed
    {
        // Добавить код реализации...
    }
}
```

Обратите внимание, что директива `.ctor` снабжена атрибутом `instance` (поскольку конструктор не статический). Атрибуты `cil managed` указывают на то, что внутри данного метода содержится код CIL, а не управляемый код, который может использоваться при выполнении запросов `P/Invoke`.

Определение свойств в CIL

Свойства и методы также имеют специфические представления в CIL. В качестве примера модифицируем класс `MyBaseClass` с целью поддержки открытого свойства по имени `TheString`, написав следующий код CIL (обратите внимание на применение атрибута `specialname`):

```
.class public MyBaseClass
{
    ...
    .method public hidebysig specialname
        instance string get_TheString() cil managed
    {
        // Добавить код реализации...
    }

    .method public hidebysig specialname
        instance void set_TheString(string 'value') cil managed
    {
        // Добавить код реализации...
    }

    .property instance string TheString()
    {
        .get instance string
            MyNamespace.MyBaseClass::get_TheString()
        .set instance void
            MyNamespace.MyBaseClass::set_TheString(string)
    }
}
```

В терминах CIL свойство отображается на пару методов, имеющих префиксы `get_` и `set_`. В директиве `.property` используются связанные директивы `.get` и `.set` для отображения синтаксиса свойств на подходящие “специально именованные” методы.

На заметку! Обратите внимание, что входной параметр метода `set_` в свойстве помещен в одинарные кавычки и представляет имя лексемы, которая должна применяться в правой части операции присваивания внутри области определения метода.

Определение параметров членов

Коротко говоря, параметры в CIL указываются (более или менее) идентично тому, как это делается в C#. Например, каждый параметр определяется путем указания его типа данных, за которым следует имя параметра. Более того, подобно C# язык CIL позволяет определять входные, выходные и передаваемые по ссылке параметры. Вдобавок в CIL допускается определять массив параметров (соответствует ключевому слову `params` в C#), а также необязательные параметры.

Чтобы проиллюстрировать процесс определения параметров в низкоуровневом коде CIL, предположим, что необходимо построить метод, который принимает параметр `int32` (по значению), параметр `int32` (по ссылке), параметр `[mscorlib]System.Collection.ArrayList` и один выходной параметр (типа `int32`). В C# метод выглядел бы примерно так:

```
public static void MyMethod(int inputInt,
    ref int refInt, ArrayList ar, out int outputInt)
{
    outputInt = 0; // Просто чтобы удовлетворить компилятор C#...
}
```

После отображения метода `MyMethod()` на код CIL вы обнаружите, что ссылочные параметры C# помечаются символом амперсанда (&), который дополняет лежащий в основе тип данных (`int32&`).

Выходные параметры также снабжаются суффиксом &, но дополнительно уточняются лексемой `[out]` языка CIL. Вдобавок, если параметр относится к ссылочному типу (`[mscorlib]System.Collections.ArrayList`), то перед типом данных указывается лексема `class` (не путайте ее с директивой `.class`):

```
.method public hidebysig static void MyMethod(int32 inputInt,
    int32& refInt,
    class [mscorlib]System.Collections.ArrayList ar,
    [out] int32& outputInt) cil managed
{
    ...
}
```

Исследование кодов операций CIL

Последний аспект кода CIL, который будет здесь рассматриваться, связан с ролью разнообразных кодов операций. Вспомните, что код операции — это просто лексема CIL, используемая при построении логики реализации для заданного члена. Все коды операций CIL (которых довольно много) могут быть разделены на три обширные категории:

- коды операций, которые управляют потоком выполнения программы;
- коды операций, которые вычисляют выражения;
- коды операций, которые получают доступ к значениям в памяти (через параметры, локальные переменные и т.д.).

В табл. 18.5 описаны наиболее полезные коды операций, имеющие прямое отношение к логике реализации членов; они сгруппированы по функциональности.

Таблица 18.5. Различные коды операций CIL, связанные с реализацией членов

Коды операций	Описание
add, sub, mul, div, rem	Позволяют выполнять сложение, вычитание, умножение и деление двух значений (rem возвращает остаток от деления)
and, or, not, xor	Позволяют выполнять побитовые операции над двумя значениями
ceq, cgt, clt	Позволяют сравнивать два значения в стеке разными способами, например: <ul style="list-style-type: none"> ceq — сравнение на равенство cgt — сравнение “больше чем” clt — сравнение “меньше чем”
box, unbox	Применяются для преобразования между ссылочными типами и типами значений
ret	Используется для выхода из метода и возврата значения вызывающему коду (при необходимости)
beq, bgt, ble, blt, switch	Применяются для управления логикой ветвления внутри метода (вдобавок ко многим другим связанным кодам операций), например: <ul style="list-style-type: none"> beq — переход к метке в коде, если равно bgt — переход к метке в коде, если больше чем ble — переход к метке в коде, если меньше или равно blt — переход к метке в коде, если меньше чем Все коды операций, связанные с ветвлением, требуют указания в коде CIL метки для перехода в случае, если результат проверки оказывается истинным
call	Используется для вызова члена заданного типа
newarr, newobj	Позволяют размещать в памяти новый массив или новый объект (соответственно)

Коды операций из следующей обширной категории (подмножество которых описано в табл. 18.6) применяются для загрузки (заталкивания) аргументов в виртуальный стек выполнения. Обратите внимание, что все эти ориентированные на загрузку коды операций имеют префикс ld (load — загрузить).

Таблица 18.6. Основные коды операций CIL, связанные с загрузкой в стек

Код операции	Описание
ldarg (и многочисленные вариации)	Загружает в стек аргумент метода. Помимо общей формы ldarg (которая работает с индексом, идентифицирующим аргумент), существует множество других вариаций. Например, коды операций ldarg, которые имеют числовой суффикс (ldarg_0), жестко кодируют загружаемый аргумент. Кроме того, вариации ldarg позволяют жестко кодировать тип данных, используя константное обозначение CIL из табл. 18.4 (скажем, ldarg_I4 для int32), а также тип данных и значение (к примеру, ldarg_I4_5 для загрузки int32 со значением 5)
ldc (и многочисленные вариации)	Загружает в стек константное значение
ldfld (и многочисленные вариации)	Загружает в стек значение поля уровня экземпляра
ldloc (и многочисленные вариации)	Загружает в стек значение локальной переменной
ldobj	Получает все значения, собранные размещенным в куче объектом, и помещает их в стек
ldstr	Загружает в стек строковое значение

В дополнение к набору кодов операций, связанных с загрузкой, CIL предоставляет многочисленные коды операций, которые явно извлекают из стека самое верхнее значение. Как было показано в нескольких начальных примерах, извлечение значения из стека обычно предусматривает его сохранение во временном локальном хранилище с целью дальнейшего использования (наподобие параметра для предстоящего вызова метода). Многие коды операций, извлекающие текущее значение из виртуального стека выполнения, снабжены префиксом *st* (store — сохранить). В табл. 18.7 описаны некоторые распространенные коды операций.

Таблица 18.7. Распространенные коды операций CIL, связанные с извлечением из стека

Код операции	Описание
<code>pop</code>	Удаляет значение, которое в текущий момент находится на вершухе стека вычислений, но не заботится о его сохранении
<code>starg</code>	Сохраняет значение из вершухи стека в аргументе метода с определенным индексом
<code>stloc</code> (и многочисленные вариации)	Извлекает текущее значение из вершухи стека вычислений и сохраняет его в списке локальных переменных по указанному индексу
<code>stobj</code>	Копирует значение указанного типа из стека вычислений по заданному адресу в памяти
<code>stsfld</code>	Заменяет значение статического поля значением из стека вычислений

Имейте в виду, что различные коды операций CIL будут неявно извлекать значения из стека во время выполнения своих задач. Например, при вычитании одного числа из другого с применением кода операции `sub` должно быть очевидным, что перед самим вычислением операция `sub` должна извлечь из стека два следующих доступных значения. Результат вычисления снова помещается в стек.

Директива `.maxstack`

При написании кода реализации методов на низкоуровневом языке CIL необходимо помнить о специальной директиве под названием `.maxstack`. С ее помощью устанавливается максимальное количество переменных, которые могут находиться внутри стека в любой заданный момент времени на протяжении периода выполнения метода. Хорошая новость в том, что директива `.maxstack` имеет стандартное значение (8), которое должно подходить для подавляющего большинства создаваемых методов. Тем не менее, если вы хотите указывать все явно, то можете вручную подсчитать количество локальных переменных в стеке и определить это значение явно:

```
.method public hidebysig instance void
    Speak() cil managed
{
    // Внутри области действия этого метода в стеке находится
    // в точности одно значение (строковый литерал).
    .maxstack 1
    ldstr "Hello there..."
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

Объявление локальных переменных в CIL

Теперь давайте посмотрим, как объявлять локальные переменные. Предположим, что необходимо построить в CIL метод по имени `MyLocalVariables()`, который не принимает аргументов и возвращает `void`, и определить в нем три локальные переменные с типами `System.String`, `System.Int32` и `System.Object`. В C# такой метод выглядел бы следующим образом (вспомните, что локальные переменные не получают стандартные значения и потому перед использованием должны быть инициализированы):

```
public static void MyLocalVariables()
{
    string myStr = "CIL code is fun!";
    int myInt = 33;
    object myObj = new object();
}
```

А вот как реализовать метод `MyLocalVariables()` на языке CIL:

```
.method public hidebysig static void
MyLocalVariables() cil managed
{
    .maxstack 8
    // Определить три локальные переменные.
    .locals init ([0] string myStr, [1] int32 myInt, [2] object myObj)
    // Загрузить строку в виртуальный стек выполнения.
    ldstr "CIL code is fun!"
    // Извлечь текущее значение и сохранить его в локальной переменной [0].
    stloc.0
    // Загрузить константу типа 14 (сокращение для int32) со значением 33.
    ldc.i4 33
    // Извлечь текущее значение и сохранить его в локальной переменной [1].
    stloc.1
    // Создать новый объект и поместить его в стек.
    newobj instance void [mscorlib]System.Object::.ctor()
    // Извлечь текущее значение и сохранить его в локальной переменной [2].
    stloc.2
    ret
}
```

Первым шагом при размещении локальных переменных с помощью CIL является применение директивы `.locals` в паре с атрибутом `init`. Внутри квадратных скобок, связанных с каждой переменной, понадобится указать определенный числовой индекс ([0], [1] и [2]). Каждый индекс идентифицируется типом данных и необязательным именем переменной. После определения локальных переменных значения загружаются в стек (с использованием различных кодов операций загрузки) и сохраняются в этих локальных переменных (с помощью кодов операций сохранения).

Отображение параметров на локальные переменные в CIL

Вы уже видели, каким образом объявляются локальные переменные в CIL с применением директивы `.local init`; однако осталось еще взглянуть на то, как входные параметры отображаются на локальные переменные. Рассмотрим показанный ниже статический метод C#:

```
public static int Add(int a, int b)
{
    return a + b;
}
```

Такой с виду невинный метод требует немалого объема кодирования на языке CIL. Во-первых, входные аргументы (a и b) должны быть помещены в виртуальный стек выполнения с использованием кода операции `ldarg` (load argument — загрузить аргумент). Во-вторых, с помощью кода операции `add` из стека будут извлечены следующие два значения и просуммированы с сохранением результата обратно в стек. В-третьих, сумма будет извлечена из стека и возвращена вызывающему коду посредством кода операции `ret`. Дизассемблировав этот метод C# с применением `ildasm.exe`, вы обнаружите множество дополнительных лексем, которые вставил компилятор `csc.exe`, но основная часть кода CIL довольно проста:

```
.method public hidebysig static int32 Add(int32 a,
    int32 b) cil managed
{
    .maxstack 2
    ldarg.0 // Загрузить a в стек.
    ldarg.1 // Загрузить b в стек.
    add     // Сложить оба значения.
    ret
}
```

Скрытая ссылка `this`

Обратите внимание, что ссылка на два входных аргумента (a и b) в коде CIL производится с использованием их индексных позиций (0 и 1), т.к. индексация в виртуальном стеке выполнения начинается с нуля.

Во время исследования или написания кода CIL нужно помнить о том, что каждый нестатический метод, принимающий входные аргументы, автоматически получает неявный дополнительный параметр, который представляет собой ссылку на текущий объект (аналогичный ключевому слову `this` в C#). Скажем, если бы метод `Add()` был определен как нестатический:

```
// Больше не является статическим!
public int Add(int a, int b)
{
    return a + b;
}
```

то входные аргументы a и b загружались бы с применением кодов операций `ldarg.1` и `ldarg.2` (a не ожидаемых `ldarg.0` и `ldarg.1`). Причина в том, что ячейка 0 в действительности содержит неявную ссылку `this`. Взгляните на следующий псевдокод:

```
// Это ТОЛЬКО псевдокод!
.method public hidebysig static int32 AddTwoIntParams(
    MyClass_HiddenThisPointer this, int32 a, int32 b) cil managed
{
    ldarg.0 // Загрузить MyClass_HiddenThisPointer в стек.
    ldarg.1 // Загрузить a в стек.
    ldarg.2 // Загрузить b в стек.
    ...
}
```

Представление итерационных конструкций в CIL

Итерационные конструкции в языке программирования C# реализуются посредством ключевых слов `for`, `foreach`, `while` и `do`, каждое из которых имеет специальное представление в CIL. Для примера рассмотрим следующий классический цикл `for`:


```
public static void CountToTen()
{
    for(int i = 0; i < 10; i++)
    ;
}
```

Вспомните, что для управления прекращением потока выполнения, когда удовлетворено некоторое условие, используются коды операций `br` (`br`, `blt` и т.д.). В приведенном примере указано условие, согласно которому выполнение цикла `for` должно прекращаться, когда значение локальной переменной `i` становится больше или равно 10. С каждым проходом к значению `i` добавляется 1, после чего проверяемое условие оценивается заново.

Также вспомните, что в случае применения любого кода операции CIL, предназначенного для ветвления, должна быть определена специфичная метка кода (или две), обозначающая место, куда будет произведен переход при истинном значении условия. С учетом всего сказанного рассмотрим показанный ниже (и дополненный комментариями) код CIL, который сгенерирован утилитой `ildasm.exe` (вместе с автоматически созданными метками):

```
.method public hidebysig static void CountToTen() cil managed
{
    .maxstack 2
    .locals init ([0] int32 i) // Инициализировать локальную целочисленную
                               // переменную i.
    IL_0000: ldc.i4.0           // Загрузить это значение в стек.
    IL_0001: stloc.0           // Сохранить это значение по индексу 0.
    IL_0002: br.s IL_0008      // Перейти на метку IL_0008.
    IL_0004: ldloc.0           // Загрузить значение переменной по индексу 0.
    IL_0005: ldc.i4.1         // Загрузить значение 1 в стек.
    IL_0006: add              // Добавить текущее значение в стеке по индексу 0.
    IL_0007: stloc.0          // Сохранить это значение по индексу 0.
    IL_0008: ldloc.0          // Загрузить значение по индексу 0.
    IL_0009: ldc.i4.s 10      // Загрузить значение 10 в стек.
    IL_000b: blt.s IL_0004     // Меньше чем? Если да, то перейти на метку IL_0004.
    IL_000d: ret
}
```

Код CIL начинается с определения локальной переменной типа `int32` и ее загрузки в стек. Затем производятся переходы туда и обратно между метками `IL_0008` и `IL_0004`, во время каждого из которых значение `i` увеличивается на 1 и проверяется на предмет того, что оно все еще меньше 10. Как только условие будет нарушено, осуществляется выход из метода.

Исходный код. Пример `CilTypes` доступен в подкаталоге `Chapter_18`

Построение сборки .NET на CIL

После ознакомления с синтаксисом и семантикой языка CIL самое время закрепить изученный материал, построив приложение .NET с использованием утилиты `ilasm.exe` и текстового редактора по своему выбору. Приложение будет состоять из закрытой однофайловой сборки `*.dll`, содержащей определения двух типов классов, и консольной программы `*.exe`, которая взаимодействует с этими типами.

Построение сборки CILCars.dll

В первую очередь понадобится построить сборку *.dll, которую будет потреблять клиент. Создадим в текстовом редакторе файл *.il по имени CILCars.il. Внутри однофайловой сборки задействуются две внешние сборки .NET. Модифицируем код следующим образом:

```
// Ссылки на mscorlib.dll и System.Windows.Forms.dll.
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
.assembly extern System.Windows.Forms
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
// Определить однофайловую сборку.
.assembly CILCars
{
    .hash algorithm 0x00008004
    .ver 1:0:0:0
}
.module CILCars.dll
```

Сборка будет содержать два типа класса. Первый тип, CILCar, определяет два поля данных (для простоты открытых) и специальный конструктор. Второй тип, CarInfoHelper, определяет единственный статический метод по имени DisplayCarInfo(), который принимает параметр типа CILCar и возвращает void. Оба типа находятся в пространстве имен CILCars. Вот как может быть реализован тип CILCar на языке CIL:

```
// Реализация типа CILCars.CILCar.
.namespace CILCars
{
    .class public auto ansi beforefieldinit CILCar
        extends [mscorlib]System.Object
    {
        // Поля данных CILCar.
        .field public string petName
        .field public int32 currSpeed

        // Специальный конструктор просто позволяет
        // вызывающему коду присваивать поля данных.
        .method public hidebysig specialname rtspecialname
            instance void .ctor(int32 c, string p) cil managed
        {
            .maxstack 8

            // Загрузить первый аргумент в стек и вызвать конструктор базового класса.
            ldarg.0 // объект this, не значение int32!
            call instance void [mscorlib]System.Object::.ctor()

            // Загрузить первый и второй аргументы в стек.
            ldarg.0 // объект this
            ldarg.1 // аргумент int32

            // Сохранить самый верхний элемент стека (int32) в поле currSpeed.
            stfld int32 CILCars.CILCar::currSpeed
```

```

    // Загрузить строковый аргумент и сохранить в поле petName.
    ldarg.0 // объект this
    ldarg.2 // аргумент string
    stfld string CILCars.CILCar::petName
    ret
}
}
}

```

Учитывая то, что реальным первым аргументом для любого нестатического члена является ссылка на текущий объект, в первом блоке CIL просто загружается ссылка на текущий объект и вызывается конструктор базового класса. Затем входные аргументы конструктора помещаются в стек и сохраняются в соответствующих полях данных с помощью кода операции stfld (store in field — сохранить в поле).

Теперь реализуем второй тип из пространства имен CILCars — CILCarInfo. Главным в нем будет статический метод Display(). Роль метода заключается в том, чтобы принять входной параметр CILCar, извлечь значения его полей данных и отобразить их в окне сообщений Windows Forms. Ниже приведена полная реализация типа CILCarInfo, а после нее — необходимые пояснения:

```

.class public auto ansi beforefieldinit CILCarInfo
    extends [mscorlib]System.Object
{
    .method public hidebysig static void
        Display(class CILCars.CILCar c) cil managed
    {
        .maxstack 8

        // Нам нужна локальная строковая переменная.
        .locals init ([0] string caption)

        // Загрузить строку и входной параметр CILCar в стек.
        ldstr "{0}'s speed is:"
        ldarg.0

        // Поместить в стек значение поля petName из CILCar
        // и вызвать статический метод String.Format().
        ldfld string CILCars.CILCar::petName
        call string [mscorlib]System.String::Format(string, object)
        stloc.0

        // Загрузить значение поля currSpeed и получить его строковое
        // представление (обратите внимание на вызов ToString()).
        ldarg.0
        ldflda int32 CILCars.CILCar::currSpeed
        call instance string [mscorlib]System.Int32::ToString()
        ldloc.0

        // Вызвать метод MessageBox.Show() с загруженными значениями.
        call valuetype [System.Windows.Forms]
            System.Windows.Forms.DialogResult
            [System.Windows.Forms]
            System.Windows.Forms.MessageBox::Show(string, string)
        pop
        ret
    }
}

```

Хотя объем кода CIL здесь немного больше, чем в реализации CILCar, он по-прежнему прямолинеен. Поскольку определяется статический метод, беспокоиться о скрытой ссылке на текущий объект больше не требуется (таким образом, код операции `ldarg.0` действительно загружает входной аргумент CILCar).

Метод начинается с загрузки в стек строки `"{0}'s speed is:"` и следом за ней аргумента CILCar. Затем загружается значение `petName` и вызывается статический метод `System.String.Format()` для замены заполнителя в фигурных скобках дружественным именем CILCar.

С помощью той же самой общей процедуры обрабатывается поле `currSpeed`, но на этот раз применяется код операции `ldflda`, который приводит к загрузке в стек адреса аргумента. Далее вызывается метод `System.Int32.ToString()` с целью преобразования находящегося по указанному адресу значения в строковый тип. И, наконец, после того, как обе строки сформатированы должным образом, вызывается метод `MessageBox.Show()`.

Теперь можно скомпилировать новую сборку `*.dll` с помощью `ilasm.exe`:

```
ilasm /dll CILCars.il
```

и проверить содержащийся внутри нее код CIL посредством утилиты `peverify.exe`:

```
peverify CILCars.dll
```

Построение сборки CILCarClient.exe

Далее можно построить простую сборку `*.exe` с методом `Main()`, который будет:

- создавать объект CILCar;
- передавать объект статическому методу `CILCarInfo.Display()`.

Создадим новый файл `CarClient.il`, добавим в него ссылки на внешние сборки `mscorlib.dll` и `CILCars.dll` (не забыв поместить копию `CILCars.dll` в каталог клиентского приложения) и определим в нем единственный тип (`Program`), который будет манипулировать сборкой `CILCars.dll`. Вот полный код:

```
// Ссылки на внешние сборки.
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
    .ver 4:0:0:0
}
.assembly extern CILCars
{
    .ver 1:0:0:0
}

// Наша исполняемая сборка.
.assembly CarClient
{
    .hash algorithm 0x00008004
    .ver 1:0:0:0
}
.module CarClient.exe

// Реализация типа Program.
.namespace CarClient
{
    .class private auto ansi beforefieldinit Program
    extends [mscorlib]System.Object
```

```

{
    .method private hidebysig static void
    Main(string[] args) cil managed
    {
        // Пометить точку входа в сборке *.exe.
        .entrypoint
        .maxstack 8

        // Объявить локальную переменную типа CILCar и поместить
        // значения в стек для вызова конструктора.
        .locals init ([0] class
        [CILCars]CILCars.CILCar myCilCar)
        ldc.i4 55
        ldstr "Junior"

        // Создать новый объект CILCar; сохранить и загрузить ссылку на него.
        newobj instance void
        [CILCars]CILCars.CILCar::.ctor(int32, string)
        stloc.0
        ldloc.0

        // Вызвать метод Display(), передав ему самое верхнее значение из стека.
        call void [CILCars]
        CILCars.CILCarInfo::Display(
            class [CILCars]CILCars.CILCar)
        ret
    }
}

```

Здесь важно отметить код операции `.entrypoint`. Как упоминалось ранее в главе, он используется для пометки метода в сборке `*.exe`, который должен служить точкой входа. Учитывая, что посредством `.entrypoint` среда CLR идентифицирует начальный метод для выполнения, данный метод на самом деле может иметь любое имя, хотя для него было выбрано стандартное имя `Main()`. В оставшемся коде CIL метода `Main()` производятся типичные операции по помещению и извлечению значений из стека.

Тем не менее, для создания объекта `CILCar` применяется код операции `.newobj`. В связи с этим вспомните, что для вызова члена типа в коде CIL используется синтаксис в виде двойного двоеточия и, как обычно, указывается полностью заданное имя типа. Далее можно скомпилировать новый файл с помощью `ilasm.exe`, проверить сборку с применением `peverify.exe` и запустить программу. Введите в окне командной строки следующие команды:

```

ilasm CarClient.il
peverify CarClient.exe
CarClient.exe

```

Исходный код. Пример `CilCars` доступен в подкаталоге `Chapter_18`.

Динамические сборки

Естественно, процесс построения сложных приложений .NET на языке CIL окажется довольно-таки неблагодарным трудом. С одной стороны, CIL является чрезвычайно выразительным языком программирования, который позволяет взаимодействовать со всеми программными конструкциями, разрешенными CTS.

С другой стороны, написание низкоуровневого кода CIL утомительно, сопряжено с большими затратами времени и подвержено ошибкам. Хотя и правда, что знание — сила, вас может интересовать, насколько важно держать в уме все правила синтаксиса CIL. Ответ: зависит от ситуации. Разумеется, в большинстве случаев при программировании приложений .NET просматривать, редактировать или писать код CIL не требуется. Однако знание основ языка CIL означает готовность перейти к исследованию мира динамическихборок (как противоположности статическим сборкам) и роли пространства имен `System.Reflection.Emit`.

Первым может возникнуть вопрос: чем отличаются статические сборки от динамических? По определению *статической сборки* называется двоичная сборка .NET, которая загружается прямо из дискового хранилища, т.е. на момент запроса средой CLR она находится где-то на жестком диске в физическом файле (или в наборе файлов, если сборка многофайловая). Как и можно было предположить, при каждой компиляции исходного кода C# в результате получается статическая сборка.

В свою очередь, *динамическая сборка* создается в памяти на лету с использованием типов из пространства имен `System.Reflection.Emit`, которое делает возможным построение сборки и ее модулей, определений типов и логики реализации на языке CIL *во время выполнения*. Затем сборку, расположенную в памяти, можно сохранить на диск, получив в результате новую статическую сборку. Ясно, что процесс создания динамическихборок с помощью пространства имен `System.Reflection.Emit` требует понимания природы кодов операций CIL.

Несмотря на то что создание динамическихборок является довольно сложной (и редкой) задачей программирования, оно может быть удобным в разнообразных обстоятельствах. Ниже перечислены примеры.

- Вы строите инструмент программирования .NET, который должен быть способен генерировать сборки по требованию на основе пользовательского ввода.
- Вы создаете приложение, которое нуждается в генерации прокси для удаленных типов на лету, основываясь на полученных метаданных.
- Вам необходима возможность загрузки статической сборки и динамической вставки в двоичный образ новых типов.

Итак, давайте посмотрим, какие типы доступны в пространстве имен `System.Reflection.Emit`.

Исследование пространства имен `System.Reflection.Emit`

Создание динамической сборки требует некоторых знаний кодов операций CIL, но типы из пространства имен `System.Reflection.Emit` максимально возможно скрывают сложность языка CIL. Скажем, вместо непосредственного указания необходимых директив и атрибутов CIL для определения типа класса можно просто применять класс `TypeBuilder`.

Аналогично, если нужно определить новый конструктор уровня экземпляра, то не придется задавать лексему `specialname, rtspecialname` или `.ctor`; взамен можно использовать класс `ConstructorBuilder`. Основные члены пространства имен `System.Reflection.Emit` описаны в табл. 18.8.

В целом типы из пространства имен `System.Reflection.Emit` позволяют представлять низкоуровневые лексемы CIL программным образом во время построения динамической сборки. Вы увидите многие из них в рассматриваемом далее примере; тем не менее, тип `ILGenerator` заслуживает специального внимания.

Таблица 18.8. Избранные члены пространства имен `System.Reflection.Emit`

Член	Описание
<code>AssemblyBuilder</code>	Применяется для создания сборки (*.dll или *.exe) во время выполнения. В сборках *.exe должен вызываться метод <code>ModuleBuilder.SetEntryPoint()</code> для установки метода, который является точкой входа в модуль. Если ни одной точки входа не указано, тогда будет генерироваться файл *.dll
<code>ModuleBuilder</code>	Используется для определения набора модулей внутри текущей сборки
<code>EnumBuilder</code>	Применяется для создания типа перечисления .NET
<code>TypeBuilder</code>	Используется для создания классов, интерфейсов, структур и делегатов внутри модуля во время выполнения
<code>MethodBuilder</code> <code>LocalBuilder</code> <code>PropertyBuilder</code> <code>FieldBuilder</code> <code>ConstructorBuilder</code> <code>CustomAttributeBuilder</code> <code>ParameterBuilder</code> <code>EventBuilder</code>	Применяются для создания членов типов (таких как методы, локальные переменные, свойства, конструкторы и атрибуты) во время выполнения
<code>ILGenerator</code>	Выпускает коды операций CIL для указанного члена типа
<code>OpCodes</code>	Предоставляет многочисленные поля, которые отображаются на коды операций CIL. Используется вместе с разнообразными членами типа <code>System.Reflection.Emit.ILGenerator</code>

Роль типа `System.Reflection.Emit.ILGenerator`

Роль типа `ILGenerator` заключается во вставке кодов операций CIL внутрь заданного члена типа. Однако создавать объекты `ILGenerator` напрямую невозможно, т.к. этот тип не имеет открытых конструкторов. Взамен объекты `ILGenerator` должны получаться путем вызова специфических методов типов, относящихся к строителям (вроде `MethodBuilder` и `ConstructorBuilder`). Вот пример:

```
// Получить объект ILGenerator из объекта ConstructorBuilder
// по имени myCtorBuilder.
ConstructorBuilder myCtorBuilder =
    new ConstructorBuilder(/* ...разнообразные аргументы... */);
ILGenerator myCILGen = myCtorBuilder.GetILGenerator();
```

Имея объект `ILGenerator`, можно выпускать низкоуровневые коды операций CIL с помощью любых его методов. Некоторые (но не все) методы `ILGenerator` кратко описаны в табл. 18.9.

Основным методом класса `ILGenerator` является `Emit()`, который работает в сочетании с типом `System.Reflection.Emit.OpCodes`. Как упоминалось ранее в главе, данный тип открывает доступ к множеству полей только для чтения, которые отображаются на низкоуровневые коды операций CIL. Полный набор этих членов документирован в онлайн-овой справочной системе, и далее в главе вы неоднократно встретите примеры их использования.

Таблица 18.9. Избранные методы класса ILGenerator

Метод	Описание
BeginCatchBlock()	Начинает блок catch
BeginExceptionBlock()	Начинает блок исключения
BeginFinallyBlock()	Начинает блок finally
BeginScope()	Начинает лексическую область
DeclareLocal()	Объявляет локальную переменную
DefineLabel()	Объявляет новую метку
Emit()	Многokrатно перегружен, чтобы позволить выпускать коды операций CIL
EmitCall()	Помещает в поток CIL код операции call или callvirt
EmitWriteLine()	Выпускает вызов Console.WriteLine() со значениями разных типов
EndExceptionBlock()	Завершает блок исключения
EndScope()	Завершает лексическую область
ThrowException()	Выпускает инструкцию для генерации исключения
UsingNamespace()	Указывает пространство имен, которое должно применяться при оценке локальных и наблюдаемых значений в текущей активной лексической области

Выпуск динамической сборки

Чтобы проиллюстрировать процесс определения сборки .NET во время выполнения, давайте рассмотрим процесс создания однофайловой динамической сборки по имени MyAssembly.dll. Внутри модуля находится класс HelloWorld. Класс HelloWorld поддерживает стандартный конструктор и специальный конструктор, который применяется для присваивания значения закрытой переменной-члена (theMessage) типа string. Вдобавок в классе HelloWorld имеется открытый метод экземпляра под названием SayHello(), который выводит приветственное сообщение в стандартный поток ввода-вывода, и еще один метод экземпляра по имени GetMsg(), возвращающий внутреннюю закрытую строку. По существу мы собираемся программно сгенерировать следующий тип класса:

```
// Этот класс будет создаваться во время выполнения с использованием
// пространства имен System.Reflection.Emit.
public class HelloWorld
{
    private string theMessage;
    HelloWorld() {}
    HelloWorld(string s) {theMessage = s;}

    public string GetMsg() {return theMessage;}
    public void SayHello()
    {
        System.Console.WriteLine("Hello from the HelloWorld class!");
    }
}
```


Создадим в Visual Studio новый проект консольного приложения по имени MyAsmBuilder, импортируем в него пространства имен System.Reflection, System.Reflection.Emit и System.Threading, после чего определим в классе Program статический метод по имени CreateMyAsm(). Этот единственный метод будет отвечать за решение следующих задач:

- определение характеристик динамической сборки (имя, версия и т.п.);
- реализация типа HelloWorld;
- сохранение находящейся в памяти сборки в физическом файле.

Кроме того, обратите внимание, что метод CreateMyAsm() принимает единственный параметр типа System.AppDomain, который будет использоваться для получения доступа к объекту типа AssemblyBuilder, ассоциированному с текущим доменом приложения (домены приложений обсуждались в главе 17). Ниже показан полный код.

```
// Объект AppDomain отправляется вызывающим кодом.
public static void CreateMyAsm(AppDomain curAppDomain)
{
    // Установить общие характеристики сборки.
    AssemblyName assemblyName = new AssemblyName();
    assemblyName.Name = "MyAssembly";
    assemblyName.Version = new Version("1.0.0.0");

    // Создать новую сборку внутри текущего домена приложения.
    AssemblyBuilder assembly =
        curAppDomain.DefineDynamicAssembly(assemblyName,
            AssemblyBuilderAccess.Save);

    // Поскольку строится однофайловая сборка, имя модуля
    // будет таким же, как у сборки.
    ModuleBuilder module =
        assembly.DefineDynamicModule("MyAssembly", "MyAssembly.dll");

    // Определить открытый класс по имени HelloWorld.
    TypeBuilder helloWorldClass = module.DefineType("MyAssembly.HelloWorld",
        TypeAttributes.Public);

    // Определить закрытую переменную-член типа String по имени theMessage.
    FieldBuilder msgField =
        helloWorldClass.DefineField("theMessage", Type.GetType("System.String"),
            FieldAttributes.Private);

    // Создать специальный конструктор.
    Type[] constructorArgs = new Type[1];
    constructorArgs[0] = typeof(string);
    ConstructorBuilder constructor =
        helloWorldClass.DefineConstructor(MethodAttributes.Public,
            CallingConventions.Standard,
            constructorArgs);
    ILGenerator constructorIL = constructor.GetILGenerator();
    constructorIL.Emit(OpCodes.Ldarg_0);
    Type objectClass = typeof(object);
    ConstructorInfo superConstructor = objectClass.GetConstructor(new Type[0]);
    constructorIL.Emit(OpCodes.Call, superConstructor);
    constructorIL.Emit(OpCodes.Ldarg_0);
    constructorIL.Emit(OpCodes.Ldarg_1);
    constructorIL.Emit(OpCodes.Stfld, msgField);
    constructorIL.Emit(OpCodes.Ret);
}
```

```

// Создать стандартный конструктор.
helloWorldClass.DefineDefaultConstructor(MethodAttributes.Public);

// Создать метод GetMsg().
MethodBuilder getMsgMethod =
    helloWorldClass.DefineMethod("GetMsg", MethodAttributes.Public,
        typeof(string), null);
ILGenerator methodIL = getMsgMethod.GetILGenerator();
methodIL.Emit(OpCodes.Ldarg_0);
methodIL.Emit(OpCodes.Ldfld, msgField);
methodIL.Emit(OpCodes.Ret);

// Создать метод SayHello().
MethodBuilder sayHiMethod =
    helloWorldClass.DefineMethod("SayHello",
        MethodAttributes.Public, null, null);
methodIL = sayHiMethod.GetILGenerator();
methodIL.EmitWriteLine("Hello from the HelloWorld class!");
methodIL.Emit(OpCodes.Ret);

// Выпустить класс HelloWorld.
helloWorldClass.CreateType();

// (Дополнительно) сохранить сборку в файле.
assembly.Save("MyAssembly.dll");
}

```

Выпуск сборки и набора модулей

Тело метода начинается с установления минимального набора характеристик сборки с применением типов `AssemblyName` и `Version` (определенных в пространстве имен `System.Reflection`). Затем производится получение объекта типа `AssemblyBuilder` посредством метода `AppDomain.DefineDynamicAssembly()` уровня экземпляра (вспомните, что вызывающий код будет передавать методу `CreateMyAsm()` ссылку на `AppDomain`):

```

// Установить общие характеристики сборки.
// и получить доступ к объекту AssemblyBuilder.
public static void CreateMyAsm(AppDomain curAppDomain)
{
    AssemblyName assemblyName = new AssemblyName();
    assemblyName.Name = "MyAssembly";
    assemblyName.Version = new Version("1.0.0.0");

    // Создать новую сборку внутри текущего домена приложения.
    AssemblyBuilder assembly =
        curAppDomain.DefineDynamicAssembly(assemblyName,
            AssemblyBuilderAccess.Save);

    ...
}

```

Как видите, при вызове метода `AppDomain.DefineDynamicAssembly()` должен быть указан режим доступа к определяемой сборке; его наиболее распространенные значения представлены в табл. 18.10.

Следующей задачей будет определение набора модулей для новой сборки. С учетом того, что сборка является однофайловой единицей, необходимо определить только один модуль. В случае построения многофайловой сборки с применением метода `DefineDynamicModule()` пришлось бы указывать обязательный второй параметр, который представляет имя заданного модуля (например, `myMod.dotnetmodule`).

Таблица 18.10. Часто используемые значения перечисления `AssemblyBuilderAccess`

Значение	Описание
<code>ReflectionOnly</code>	Указывает, что динамическая сборка предназначена только для рефлексии
<code>Run</code>	Указывает, что динамическая сборка может выполняться в памяти, но не сохраняться на диске
<code>RunAndSave</code>	Указывает, что динамическая сборка может выполняться в памяти и сохраняться на диске
<code>Save</code>	Указывает, что динамическая сборка может сохраняться на диске, но не выполняться в памяти

Тем не менее, когда создается однофайловая сборка, имя модуля совпадает с именем самой сборки. Метод `DefineDynamicModule()` возвращает ссылку на действительный объект типа `ModuleBuilder`:

```

// Однофайловая сборка.
ModuleBuilder module =
    assembly.DefineDynamicModule("MyAssembly", "MyAssembly.dll");
    
```

Роль типа `ModuleBuilder`

Тип `ModuleBuilder` играет ключевую роль во время разработки динамических сборок. Как и можно было ожидать, `ModuleBuilder` поддерживает несколько членов, которые позволяют определять набор типов, содержащихся внутри модуля (классы, интерфейсы, структуры и т.д.), а также набор встроенных ресурсов (таблицы строк, изображения и т.п.). В табл. 18.11 кратко описаны некоторые методы, связанные с созданием. (Обратите внимание, что каждый метод возвращает объект связанного типа, который представляет тип, подлежащий созданию.)

Таблица 18.11. Избранные методы типа `ModuleBuilder`

Метод	Описание
<code>DefineEnum()</code>	Выпускает определение перечисления .NET
<code>DefineResource()</code>	Определяет управляемый встроенный ресурс, который должен храниться в данном модуле
<code>DefineType()</code>	Конструирует объект <code>TypeBuilder</code> , который позволяет определять типы значений, интерфейсы и типы классов (включая делегаты)

Основным членом класса `ModuleBuilder` является метод `DefineType()`. Кроме указания имени типа (в виде простой строки) с помощью перечисления `System.Reflection.TypeAttributes` можно описывать формат этого типа. В табл. 18.12 приведены избранные члены перечисления `TypeAttributes`.

Таблица 18.12. Избранные члены перечисления `TypeAttributes`

Член	Описание
<code>Abstract</code>	Указывает, что тип является абстрактным
<code>Class</code>	Указывает, что тип является классом
<code>Interface</code>	Указывает, что тип является интерфейсом

Член	Описание
NestedAssembly	Указывает, что класс находится в области видимости сборки, поэтому доступен только методам внутри его сборки
NestedFamANDAssem	Указывает, что класс находится в области видимости сборки и семейства, поэтому доступен только методам, которые относятся к пересечению семейства и сборки
NestedFamily	Указывает, что класс находится в области видимости семейства, поэтому доступен только методам, которые содержатся внутри собственного типа и любых подтипов
NestedFamORAssem	Указывает, что класс находится в области видимости семейства или сборки, поэтому доступен только методам, которые относятся к объединению семейства и сборки
NestedPrivate	Указывает, что класс является вложенным и закрытым
NestedPublic	Указывает, класс является вложенным и открытым
NotPublic	Указывает, что класс не является открытым
Public	Указывает, что класс является открытым
Sealed	Указывает, что класс является конкретным и не может быть расширен
Serializable	Указывает, что класс может быть сериализован

Выпуск типа **HelloClass** и строковой переменной-члена

Теперь, когда вы лучше понимаете роль метода `ModuleBuilder.CreateType()`, давайте посмотрим, как можно выпустить открытый тип класса `HelloWorld` и закрытую строковую переменную:

```
// Определить открытый класс по имени MyAssembly.HelloWorld.
TypeBuilder helloWorldClass = module.DefineType("MyAssembly.HelloWorld",
    TypeAttributes.Public);

// Определить закрытую переменную-член типа String по имени theMessage.
FieldBuilder msgField =
    helloWorldClass.DefineField("theMessage",
        Type.GetType("System.String"),
        FieldAttributes.Private);
```

Обратите внимание, что метод `TypeBuilder.DefineField()` предоставляет доступ к объекту типа `FieldBuilder`. В классе `TypeBuilder` также определены дополнительные методы, которые обеспечивают доступ к другим типам "построителей". Например, метод `DefineConstructor()` возвращает объект типа `ConstructorBuilder`, метод `DefineProperty()` — объект типа `PropertyBuilder` и т.д.

Выпуск конструкторов

Как упоминалось ранее, для определения конструктора текущего типа можно применять метод `TypeBuilder.DefineConstructor()`. Однако когда дело доходит до реализации конструктора `HelloClass`, в тело конструктора необходимо вставить низкоуровневый код CIL, который будет отвечать за присваивание входного параметра внутренней закрытой строке. Чтобы получить объект типа `ILGenerator`, понадобится вызвать метод `GetILGenerator()` из соответствующего типа "построителя" (в данном случае `ConstructorBuilder`).

Помещение кода CIL в реализацию членов осуществляется с помощью метода `Emit()` класса `ILGenerator`. В самом методе `Emit()` часто используется тип класса `OpCodes`, который открывает доступ к набору кодов операций CIL через свойства только для чтения. Например, свойство `OpCodes.Ret` обозначает возвращение из вызова метода, `OpCodes.Stfld` создает присваивание значения переменной-члену, а `OpCodes.Call` применяется для вызова заданного метода (конструктора базового класса в данном случае). Итак, логика для реализации конструктора будет выглядеть следующим образом:

```
// Создать специальный конструктор, принимающий
// единственный аргумент типа System.String.
Type[] constructorArgs = new Type[1];
constructorArgs[0] = typeof(string);
ConstructorBuilder constructor =
    helloWorldClass.DefineConstructor(MethodAttributes.Public,
        CallingConventions.Standard, constructorArgs);

// Выпустить необходимый код CIL для конструктора.
ILGenerator constructorIL = constructor.GetILGenerator();
constructorIL.Emit(OpCodes.Ldarg_0);
Type objectClass = typeof(object);
ConstructorInfo superConstructor = objectClass.GetConstructor(new Type[0]);
constructorIL.Emit(OpCodes.Call, superConstructor); // Вызвать конструктор
                                                    // базового класса.

// Загрузить в стек указатель this объекта.
constructorIL.Emit(OpCodes.Ldarg_0);

// Загрузить входной аргумент в стек и сохранить его в msgField.
constructorIL.Emit(OpCodes.Ldarg_1);
constructorIL.Emit(OpCodes.Stfld, msgField); //Присвоить значение полю msgField.
constructorIL.Emit(OpCodes.Ret);           // Возвращение.
```

Как теперь уже хорошо известно, в результате определения специального конструктора для типа стандартный конструктор молча удаляется. Чтобы снова определить конструктор без аргументов, нужно просто вызвать метод `DefineDefaultConstructor()` типа `TypeBuilder`:

```
// Вставить заново стандартный конструктор.
helloWorldClass.DefineDefaultConstructor(MethodAttributes.Public);
```

Этот единственный вызов выпускает стандартный код CIL, используемый для определения стандартного конструктора:

```
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    .maxstack 1
    ldarg.0
    call instance void [mscorlib]System.Object::.ctor()
    ret
}
```

Выпуск метода SayHello()

Наконец, исследуем процесс выпуска метода `SayHello()`. Первая задача связана с получением объекта типа `MethodBuilder` из переменной `helloWorldClass`. После этого можно определить сам метод и получить внутренний объект типа `ILGenerator` для вставки необходимых инструкций CIL:

```
// Создать метод SayHello.
MethodBuilder sayHiMethod =
    HelloWorldClass.DefineMethod("SayHello",
        MethodAttributes.Public, null, null);
methodIL = sayHiMethod.GetILGenerator();

// Вывести строку на консоль.
methodIL.EmitWriteLine("Hello from the HelloWorld class!");
methodIL.Emit(OpCodes.Ret);
```

Здесь был определен открытый метод (т.к. указано значение `MethodAttributes.Public`), который не принимает параметров и ничего не возвращает (на что указывают значения `null` в вызове `DefineMethod()`). Также обратите внимание на вызов `EmitWriteLine()`. Посредством данного вспомогательного метода класса `ILGenerator` можно записать строку в стандартный поток вывода, приложив минимальные усилия.

Использование динамически сгенерированной сборки

Имея готовую логику для создания и сохранения сборки, осталось только разработать класс, который запустит эту логику. Определим в текущем проекте класс по имени `AsmReader`. Внутри `Main()` посредством метода `Thread.GetDoMain()` необходимо получить ссылку на текущий домен приложения, который будет применяться для размещения динамически созданной сборки. Благодаря такой ссылке появится возможность вызывать метод `CreateMyAsm()`.

Чтобы сделать пример немного интереснее, после вызова `CreateMyAsm()` мы задействуем позднее связывание (см. главу 15) для загрузки созданной сборки в память и взаимодействия с членами класса `HelloWorld`. Модифицируем метод `Main()`, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** The Amazing Dynamic Assembly Builder App *****");

    // Получить домен приложения для текущего потока.
    AppDomain curAppDomain = Thread.GetDomain();

    // Создать динамическую сборку с помощью нашего вспомогательного метода.
    CreateMyAsm(curAppDomain);
    Console.WriteLine("-> Finished creating MyAssembly.dll.");

    // Загрузить новую сборку из файла.
    Console.WriteLine("-> Loading MyAssembly.dll from file.");
    Assembly a = Assembly.Load("MyAssembly");

    // Получить тип HelloWorld.
    Type hello = a.GetType("MyAssembly.HelloWorld");

    // Создать объект HelloWorld и вызвать корректный конструктор.
    Console.WriteLine("-> Enter message to pass HelloWorld class: ");
    string msg = Console.ReadLine();
    object[] ctorArgs = new object[1];
    ctorArgs[0] = msg;
    object obj = Activator.CreateInstance(hello, ctorArgs);

    // Вызвать SayHello() и отобразить возвращенную строку.
    Console.WriteLine("-> Calling SayHello() via late binding.");
    MethodInfo mi = hello.GetMethod("SayHello");
    mi.Invoke(obj, null);

    // Вызвать метод GetMsg().
    mi = hello.GetMethod("GetMsg");
    Console.WriteLine(mi.Invoke(obj, null));
}
```

Фактически только что была построена сборка .NET, которая способна создавать и запускать другие сборки .NET во время выполнения. На этом исследование языка CIL и роли динамическихборок завершено. Настоящая глава должна была помочь углубить знания системы типов .NET, синтаксиса и семантики языка CIL, а также способа обработки кода компилятором C# в процессе его компиляции.

Исходный код. Проект `DynamicAsmBuilder` доступен в подкаталоге `Chapter_18`.

Резюме

В главе был представлен обзор синтаксиса и семантики языка CIL. В отличие от управляемых языков более высокого уровня, таких как C#, в CIL не просто определяется набор ключевых слов, а предоставляются директивы (используемые для определения конструкции сборки и ее типов), атрибуты (дополнительно уточняющие данные директивы) и коды операций (применяемые для реализации членов типов).

Вы ознакомились с несколькими инструментами, связанными с программированием на CIL, и узнали, как изменять содержимое сборки .NET за счет добавления новых инструкций CIL, используя возвратное проектирование. Кроме того, вы изучили способы установления текущей (и ссылаемой) сборки, пространств имен, типов и членов. Был рассмотрен простой пример построения библиотеки кода .NET и исполняемого файла с применением CIL и соответствующих инструментов командной строки.

Наконец, вы получили начальное представление о процессе создания *динамической сборки*. Используя пространство имен `System.Reflection.Emit`, сборку .NET можно определять в памяти во время выполнения. Вы видели, что работа с этим пространством имен требует знания семантики кода CIL. Хотя построение динамическихборок не является распространенной задачей при разработке большинства приложений .NET, оно может быть полезно в случае создания инструментов поддержки и различных утилит для программирования.

ЧАСТЬ VI

Введение в библиотеки базовых классов .NET

В этой части

Глава 19. Многопоточное, параллельное и асинхронное программирование

Глава 20. Файловый ввод-вывод и сериализация объектов

Глава 21. Доступ к данным с помощью ADO.NET

Глава 22. Введение в Entity Framework 6

Глава 23. Введение в Windows Communication Foundation

ГЛАВА 19

Многопоточное, параллельное и асинхронное программирование

Вряд ли кому-то понравится работать с приложением, которое притормаживает во время выполнения. Аналогично, никто не будет в восторге от того, что запуск какой-то задачи внутри приложения (возможно, по щелчку на элементе в панели инструментов) снижает отзывчивость других частей приложения. До выхода платформы .NET построение приложений, способных выполнять сразу несколько задач, обычно требовало написания сложного кода на языке C++, в котором использовались API-интерфейсы многопоточности Windows. К счастью, платформа .NET предложила несколько способов построения программного обеспечения, которое может совершать сложные операции по уникальным путям выполнения, с намного меньшими сложностями.

Глава начинается с определения общей природы “многопоточного приложения”. Затем мы снова обратимся к типу делегата .NET, чтобы исследовать его внутреннюю поддержку *асинхронных вызовов методов*. Как вы увидите, такой прием позволяет вызывать метод во вторичном потоке выполнения без необходимости в ручном создании или конфигурировании самого потока.

После этого будет представлено первоначальное пространство имен для многопоточности, поставляемое со времен версии .NET 1.0 и называемое `System.Threading`. Вы ознакомитесь с многочисленными типами (`Thread`, `ThreadStart` и т.д.), которые позволяют явно создавать дополнительные потоки выполнения и синхронизировать разделяемые ресурсы, обеспечивая совместное использование данных несколькими потоками в неизменчивой манере.

В оставшихся разделах главы будут рассматриваться три более новых технологии, которые разработчики приложений .NET могут применять для построения многопоточного программного обеспечения: библиотека параллельных задач (`Task Parallel Library` — TPL), технология PLINQ (`Parallel LINQ` — параллельный LINQ) и появившиеся в версии C# 6 ключевые слова, связанные с асинхронной обработкой (`async` и `await`). Вы увидите, что указанные средства помогают существенно упростить процесс создания отзывчивых многопоточных программных приложений.

Отношения между процессом, доменом приложения, контекстом и потоком

В главе 17 *поток* был определен как путь выполнения внутри исполняемого приложения. Хотя многие приложения .NET могут успешно и продуктивно работать, будучи однопоточными, первичный поток сборки (создаваемый средой CLR при выполнении метода `Main()`) в любое время может порождать вторичные потоки для выполнения дополнительных единиц работы. За счет создания дополнительных потоков можно строить более отзывчивые (но не обязательно быстрее выполняющиеся на однопоточных машинах) приложения.

Пространство имен `System.Threading` появилось в версии .NET 1.0 и предлагает один из подходов к построению многопоточных приложений. Главным типом в этом пространстве имен, пожалуй, можно назвать класс `Thread`, поскольку он представляет отдельный поток. Если необходимо программно получить ссылку на поток, который в текущий момент выполняет заданный член, то нужно просто обратиться к статическому свойству `Thread.CurrentThread`:

```
static void ExtractExecutingThread()
{
    // Получить поток, который в настоящий
    // момент выполняет данный метод.
    Thread currThread = Thread.CurrentThread;
}
```

Внутри платформы .NET *не существует* прямого соответствия “один к одному” между доменами приложений и потоками. В действительности заданный домен приложения может иметь многочисленные потоки, выполняющиеся в каждый конкретный момент времени. Более того, отдельный поток на протяжении своего времени жизни не ограничен единственным доменом приложения. Потоки могут пересекать границы доменов приложений, когда планировщик потоков Windows и среда CLR сочтут это подходящим.

Несмотря на то что активные потоки могут перемещаться между границами доменов приложений, каждый поток в любой момент времени может выполняться только внутри одного домена приложения (другими словами, одиночный поток не может выполнять работу в более чем одном домене приложения одновременно). Чтобы программно получить доступ к домену приложения, который обслуживает текущий поток, понадобится вызвать статический метод `Thread.GetDomain()`:

```
static void ExtractAppDomainHostingThread()
{
    // Получить домен приложения, обслуживающий текущий поток.
    AppDomain ad = Thread.GetDomain();
}
```

Одиночный поток в любой момент может быть также перенесен в определенный контекст и перемещаться внутри нового контекста по прихоти среды CLR. Для получения текущего контекста, в котором выполняется поток, используется статическое свойство `Thread.CurrentContext` (которое возвращает объект `System.Runtime.Remoting.Contexts.Context`):

```
static void ExtractCurrentThreadContext()
{
    // Получить контекст, в котором работает текущий поток.
    Context ctx = Thread.CurrentContext;
}
```

Еще раз: за перемещение потоков между доменами приложений и контекстами отвечает среда CLR. Как разработчик приложений .NET, вы всегда остаетесь в блаженном неведении относительно того, где завершается каждый конкретный поток (или когда он в точности помещается внутрь новых границ). Тем не менее, вы должны быть осведомлены о разнообразных способах получения лежащих в основе примитивов.

Проблема параллелизма

Один из многих болезненных аспектов многопоточного программирования связан с ограниченным контролем над тем, как операционная система или среда CLR задействует потоки. Например, написав блок кода, который создает новый поток выполнения, нельзя гарантировать, что этот поток запустится немедленно. Взамен такой код только инструктирует операционную систему или CLR о необходимости запуска потока как можно скорее (что обычно происходит, когда планировщик потоков доберется до него).

Более того, учитывая, что потоки могут перемещаться между границами приложений и контекстов, как требуется среде CLR, вы должны представлять, какие аспекты приложения являются *изменчивыми в потоках* (например, подвергаются многопоточному доступу), а какие операции считаются *атомарными* (изменчивые в потоках операции опасны).

Чтобы проиллюстрировать проблему, давайте предположим, что поток вызывает метод специфичного объекта. Теперь представим, что поток приостановлен планировщиком потока, чтобы позволить другому потоку обратиться к тому же методу того же самого объекта.

Если исходный поток еще не полностью завершил свою операцию, тогда второй входящий поток может увидеть объект в частично модифицированном состоянии. В таком случае второй поток по существу читает фиктивные данные, что определенно может привести к очень странным (и трудно обнаруживаемым) ошибкам, которые еще труднее воспроизвести и отладить.

С другой стороны, атомарные операции в многопоточной среде всегда безопасны. К сожалению, в библиотеках базовых классов .NET есть лишь несколько гарантированно атомарных операций. Даже действие по присваиванию значения переменной-члену не является атомарным! Если только в документации .NET Framework 4.7 SDK специально не сказано об атомарности операции, то вы обязаны считать ее изменчивой в потоках и предпринимать меры предосторожности.

Роль синхронизации потоков

К настоящему моменту должно быть ясно, что многопоточные программы сами по себе довольно изменчивы, т.к. множество потоков могут оперировать разделяемыми ресурсами (более или менее) одновременно. Чтобы защитить ресурсы приложений от возможного повреждения, разработчики приложений .NET должны применять любое количество потоковых примитивов (таких как блокировки, мониторы, атрибут [Synchronization] или поддержка языковых ключевых слов) для управления доступом между выполняющимися потоками.

Несмотря на то что платформа .NET не может полностью скрыть сложности, связанные с построением надежных многопоточных приложений, сам процесс был значительно упрощен. Используя типы из пространства имен `System.Threading`, библиотеку TPL и ключевые слова `async` и `await` языка C#, можно работать с множеством потоков, прикладывая минимальные усилия.

Прежде чем погрузиться в детали пространства имен `System.Threading`, библиотеки TPL и ключевых слов `async` и `await` языка C#, мы начнем с выяснения того, каким образом можно применять тип делегата .NET для вызова метода в асинхронной манере. Хотя,

несомненно, справедливо утверждать, что начиная с версии .NET 4.6 ключевые слова `async` и `await` предлагают более простую альтернативу асинхронным делегатам, по-прежнему важно знать способы взаимодействия с кодом, использующим этот подход (в производственной среде имеется масса кода, в котором применяются асинхронные делегаты).

Краткий обзор делегатов .NET

Вспомните, что делегат .NET по существу представляет собой безопасный в отношении типов объектно-ориентированный указатель на функцию. На объявление типа делегата .NET компилятор C# отвечает построением запечатанного класса, производного от `System.MulticastDelegate` (который в свою очередь унаследован от `System.Delegate`). Упомянутые базовые классы предоставляют каждому делегату возможность поддерживать список адресов методов, которые могут быть вызваны в более позднее время. Рассмотрим следующий делегат `BinaryOp`, впервые определенный в главе 10:

```
// Тип делегата C#.
public delegate int BinaryOp(int x, int y);
```

Исходя из определения, тип `BinaryOp` может указывать на любой метод, который принимает два целых числа (по значению) в качестве аргументов и возвращает целое число. После компиляции сборка с определением делегата содержит полноценное определение класса, сгенерированного динамически на основе объявления делегата при компиляции проекта. В случае `BinaryOp` класс похож на показанный ниже псевдокод:

```
public sealed class BinaryOp : System.MulticastDelegate
{
    public BinaryOp(object target, uint functionAddress);
    public int Invoke(int x, int y);
    public IAsyncResult BeginInvoke(int x, int y,
        AsyncCallback cb, object state);
    public int EndInvoke(IAsyncResult result);
}
```

Вспомните, что сгенерированный метод `Invoke()` используется для вызова методов, поддерживаемых объектом делегата, в *синхронной манере*. Следовательно, вызывающий поток (подобный первичному потоку приложения) вынужден ждать до тех пор, пока не завершится вызов делегата. Также вспомните, что в языке C# метод `Invoke()` не должен вызываться в коде напрямую, а может быть запущен косвенно, “за кулисами”, когда применяется “нормальный” синтаксис вызова методов.

Рассмотрим следующий проект консольного приложения (`SyncDelegateReview`), в котором статический метод `Add()` вызывается в синхронном (т.е. блокирующем) режиме (не забудьте импортировать в файл кода C# пространство имен `System.Threading`, т.к. будет вызываться метод `Thread.Sleep()`):

```
namespace SyncDelegateReview
{
    public delegate int BinaryOp(int x, int y);
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Synch Delegate Review *****");
            // Вывести идентификатор выполняющегося потока.
            Console.WriteLine("Main() invoked on thread {0}.",
                Thread.CurrentThread.ManagedThreadId);
```

```

// Вызвать Add() в синхронном режиме.
BinaryOp b = new BinaryOp(Add);

// Можно было бы также написать b.Invoke(10, 10);
int answer = b(10, 10);

// Эти строки кода не выполняются до тех пор,
// пока не завершится метод Add().
Console.WriteLine("Doing more work in Main()");
Console.WriteLine("10 + 10 is {0}.", answer);
Console.ReadLine();
}

static int Add(int x, int y)
{
    // Вывести идентификатор выполняющегося потока.
    Console.WriteLine("Add() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);

    // Сделать паузу для моделирования длительной операции.
    Thread.Sleep(5000);
    return x + y;
}
}
}

```

Внутри метода `Add()` вызывается статический метод `Thread.Sleep()`, чтобы приостановить вызывающий поток приблизительно на 5 секунд для моделирования длительно выполняющейся задачи. Поскольку метод `Add()` вызывается в *синхронной* манере, метод `Main()` не будет выводить результат операции до тех пор, пока не завершится метод `Add()`.

Обратите внимание, что метод `Main()` получает доступ к текущему потоку (через свойство `Thread.CurrentThread`) и выводит идентификатор потока посредством свойства `ManagedThreadId`. Та же самая логика повторяется в статическом методе `Add()`. Как и можно было ожидать, учитывая, что вся работа в этом приложении выполняется исключительно в первичном потоке, на консоль выводится одно и то же значение идентификатора:

```

***** Synch Delegate Review *****
Main() invoked on thread 1.
Add() invoked on thread 1.
Doing more work in Main()!
10 + 10 is 20.

Press any key to continue . . .

```

Запустив программу, вы должны заметить пятисекундную задержку перед тем, как выполнится последний вызов `Console.WriteLine()` в методе `Main()`. Хотя многие (если не большинство) методов могут вызываться синхронно без болезненных последствий, при необходимости делегаты .NET можно инструктировать на вызов их методов асинхронным образом.

Асинхронная природа делегатов

Если тема многопоточности для вас в новинку, тогда может возникнуть вопрос: что собой представляет *асинхронный* вызов метода? Как вам без сомнения известно, выполнение некоторых программных операций требует времени. Несмотря на то что предыдущий метод `Add()` был исключительно иллюстративным, представьте себе, что вы строите однопоточное приложение, вызывающее метод удаленной веб-службы, который инициирует длительный запрос к базе данных, загружает крупный документ либо выводит 500 строк текста во внешний файл. Пока эти операции выполняются, приложение может выглядеть зависшим в течение некоторого периода времени. До завершения задачи все другие поведенческие аспекты программы (вроде выбора пунктов меню, щелчков в панели инструментов или вывода на консоль) приостанавливаются (что может раздражать пользователей).

По указанной причине возникает вопрос: как сообщить делегату о вызове его метода в отдельном потоке выполнения, чтобы эмулировать “одновременное” выполнение многочисленных задач? К счастью, каждый тип делегата .NET автоматически оснащен такой возможностью. А еще лучше то, что для этого вы не обязаны углубляться в детали типов пространства имен `System.Threading` (хотя такие сущности могут вполне естественно работать рука об руку).

Методы `BeginInvoke()` и `EndInvoke()`

Когда компилятор C# обрабатывает ключевое слово `delegate`, он динамически генерирует класс, в котором определены два метода с именами `BeginInvoke()` и `EndInvoke()`. Учитывая определение делегата `BinaryOp`, методы будут прототипированы следующим образом:

```
public sealed class BinaryOp : System.MulticastDelegate
{
    ...
    // Используется для асинхронного вызова метода.
    public IAsyncResult BeginInvoke(int x, int y,
        AsyncCallback cb, object state);

    // Используется для извлечения возвращаемого
    // значения вызванного метода.
    public int EndInvoke(IAsyncResult result);
}
```

Первый набор параметров, передаваемый методу `BeginInvoke()`, будет основан на формате делегата C# (два целых числа в случае `BinaryOp`). Последними двумя аргументами всегда будут `System.AsyncCallback` и `System.Object`. Вскоре вы узнаете о роли этих параметров, а пока передадим в каждом из них значение `null`. Также обратите внимание, что возвращаемое значение `EndInvoke()` является целочисленным, согласно возвращаемому типу `BinaryOp`, тогда как единственный параметр данного метода всегда имеет тип `IAsyncResult`.

Интерфейс `System.IAsyncResult`

Метод `BeginInvoke()` всегда возвращает объект, реализующий интерфейс `IAsyncResult`, а методу `EndInvoke()` требуется единственный параметр совместимого с `IAsyncResult` типа. Совместимый с `IAsyncResult` объект, возвращаемый из `BeginInvoke()` — это по существу связующий механизм, который позволяет вызывающему потоку получать результат вызова асинхронного метода в более позднее время посредством `EndInvoke()`.

Интерфейс `IAsyncResult` (находящийся в пространстве имен `System`) определен следующим образом:

```
public interface IAsyncResult
{
    object AsyncState { get; }
    WaitHandle AsyncWaitHandle { get; }
    bool CompletedSynchronously { get; }
    bool IsCompleted { get; }
}
```

В простейшем случае прямого обращения к этим членам можно избежать. Все, что потребуется сделать — сохранить совместимый с `IAsyncResult` объект, возвращенный методом `BeginInvoke()`, и передать его методу `EndInvoke()` при готовности к получению результата вызова метода. Как будет показано, члены совместимого с `IAsyncResult` объекта можно вызывать, когда требуется “более высокая вовлеченность” в процесс выборки значения, возвращаемого методом.

На заметку! Метод, возвращающий `void`, можно просто вызвать асинхронно и забыть. В таких случаях нет необходимости сохранять совместимый с `IAsyncResult` объект или обращаться к `EndInvoke()`, т.к. нет возвращаемого значения, которое требуется получить.

Асинхронный вызов метода

Чтобы проинструктировать делегат `BinaryOp` о вызове метода `Add()` асинхронным образом, модифицируем логику предыдущего проекта (можно добавить код к имеющемуся проекту, но в коде примеров для главы доступен новый проект консольного приложения по имени `AsyncDelegate`). Обновите предыдущий метод `Main()`, как показано ниже:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Async Delegate Invocation *****");
    // Вывести идентификатор выполняющегося потока.
    Console.WriteLine("Main() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
    // Вызвать Add() во вторичном потоке.
    BinaryOp b = new BinaryOp(Add);
    IAsyncResult ar = b.BeginInvoke(10, 10, null, null);
    // Выполнить другую работу в первичном потоке...
    Console.WriteLine("Doing more work in Main()!");
    // По готовности получить результат выполнения метода Add().
    int answer = b.EndInvoke(ar);
    Console.WriteLine("10 + 10 is {0}.", answer);
    Console.ReadLine();
}
```

Запуск приложения приводит к выводу на консоль двух уникальных идентификаторов потоков, поскольку в текущем домене приложения функционирует множество потоков:

```
***** Async Delegate Invocation *****
Main() invoked on thread 1.
Doing more work in Main()!
Add() invoked on thread 3.
10 + 10 is 20.
```

В дополнение к уникальным идентификаторам при выполнении приложения вы заметите, что сообщение `Doing more work in Main()!` выводится практически мгновенно, в то время как вторичный поток продолжает свою работу.

Синхронизация вызывающего потока

Внимательно проанализировав текущую реализацию `Main()`, можно обнаружить, что промежуток времени между вызовами `BeginInvoke()` и `EndInvoke()` однозначно меньше пяти секунд. Следовательно, как только сообщение `Doing more work in Main()!` выводится на консоль, вызывающий поток блокируется и ожидает завершения вторичного потока, чтобы получить результат вызова метода `Add()`. Таким образом, в действительности происходит еще один *синхронный вызов*.

```
static void Main(string[] args)
{
    ...
    BinaryOp b = new BinaryOp(Add);
    // После обработки следующего оператора вызывающий поток
    // блокируется, пока не будет завершен BeginInvoke().
    IAsyncResult ar = b.BeginInvoke(10, 10, null, null);
    // Этот вызов занимает намного меньше пяти секунд!
    Console.WriteLine("Doing more work in Main()!");
    // Снова происходит ожидание завершения другого потока!
    int answer = b.EndInvoke(ar);
    ...
}
```

Очевидно, что асинхронные делегаты утратят свою привлекательность, если вызывающий поток в определенных обстоятельствах окажется заблокированным. Чтобы позволить вызывающему потоку выяснять, завершил ли работу асинхронно вызванный метод, в интерфейсе `IAsyncResult` предусмотрено свойство `IsCompleted`. С его помощью вызывающий поток может определять, действительно ли асинхронный вызов был завершен, прежде чем обращаться к методу `EndInvoke()`.

Если метод еще не завершился, тогда свойство `IsCompleted` возвращает `false`, и вызывающий поток может продолжить заниматься своей работой. Когда `IsCompleted` возвращает `true`, вызывающий поток может получить результат в “наименее блокирующей манере”. Взгляните на следующую модификацию метода `Main()`:

```
static void Main(string[] args)
{
    ...
    BinaryOp b = new BinaryOp(Add);
    IAsyncResult ar = b.BeginInvoke(10, 10, null, null);
    // Это сообщение продолжит выводиться до тех пор,
    // пока не будет завершен метод Add().
    while(!ar.IsCompleted)
    {
        Console.WriteLine("Doing more work in Main()!");
        Thread.Sleep(1000);
    }
    // Теперь известно, что метод Add() завершен.
    int answer = b.EndInvoke(ar);
    ...
}
```


Здесь организован цикл, который продолжает выполнять оператор `Console.WriteLine()` до тех пор, пока не завершится вторичный поток. Когда это произойдет, можно получить результат выполнения метода `Add()`, точно зная, что он закончил работу. Вызов `Thread.Sleep(1000)` не обязателен для корректного функционирования данного отдельно взятого приложения; однако, вынуждая первичный поток ожидать приблизительно секунду на каждой итерации, мы предотвращаем вывод на консоль слишком большого количества одного и того же сообщения. Ниже показан вывод (в зависимости от быстродействия машины и времени запуска потоков он может слегка варьироваться):

```
***** Async Delegate Invocation *****
Main() invoked on thread 1.
Doing more work in Main()!
Add() invoked on thread 3.
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
10 + 10 is 20.
```

Помимо свойства `IsCompleted` интерфейс `IAsyncResult` предлагает свойство `AsyncWaitHandle`, предназначенное для реализации более гибкой логики ожидания. Свойство `AsyncWaitHandle` возвращает экземпляр типа `WaitHandle`, который открывает доступ к методу по имени `WaitOne()`. Преимущество использования `WaitHandle.WaitOne()` заключается в том, что можно указывать максимальное время ожидания. По истечении заданного времени метод `WaitOne()` возвратит `false`. Взгляните на следующий измененный цикл `while`, в котором больше не применяется вызов `Thread.Sleep()`:

```
while (!ar.AsyncWaitHandle.WaitOne(1000, true))
{
    Console.WriteLine("Doing more work in Main()!");
}
```

Хотя рассмотренные свойства интерфейса `IAsyncResult` предоставляют способ синхронизации вызывающего потока, все же они не обеспечивают самый эффективный подход. Во многих отношениях свойство `IsCompleted` похоже на надоедливую менеджера (или коллегу), постоянно спрашивающего: "Вы уже сделали это?". К счастью, делегаты предлагают несколько дополнительных (и более элегантных) приемов получения результата из метода, который был вызван асинхронным образом.

Исходный код. Проект `AsyncDelegate` доступен в подкаталоге `Chapter_19`.

Роль делегата `AsyncCallback`

Вместо опроса делегата с целью определения, завершился ли асинхронно вызванный метод, было бы более эффективно заставить вторичный поток информировать вызывающий поток о завершении выполнения задачи. Чтобы сделать такое поведение возможным, понадобится передать методу `BeginInvoke()` в качестве параметра экземпляр делегата `System.AsyncCallback`; до сих пор для этого параметра указывалось значение `null`. Тем не менее, когда предоставляется объект `AsyncCallback`, делегат будет автоматически вызывать указанный метод после завершения асинхронного вызова.

На заметку! Метод обратного вызова будет вызываться во вторичном потоке, а не в первичном. Данный факт имеет важные последствия для потоков внутри графического пользовательского интерфейса (WPF или Windows Forms), т.к. элементы управления привязаны к потоку, в котором они были созданы, и могут обрабатываться только в нем. При рассмотрении библиотеки TPL и новых ключевых слов `async` и `await` языка C# далее в главе будут представлены некоторые примеры работы потоков из графического пользовательского интерфейса.

Как и любой делегат, `AsyncCallback` может вызывать только методы, соответствующие определенному шаблону, которым в данном случае является метод, принимающий единственный параметр `IAsyncResult` и ничего не возвращающий:

```
// Целевые методы AsyncCallback должны соответствовать следующему шаблону.
void MyAsyncCallbackMethod(IAsyncResult iar)
```

Предположим, что есть еще один проект консольного приложения (`AsyncCallbackDelegate`), в котором используется делегат `BinaryOp`. Однако на этот раз мы не будем опрашивать делегат с целью выяснения, завершился ли метод `Add()`. Взамен мы определим статический метод по имени `AddComplete()` для получения уведомления о том, что асинхронный вызов завершен. Также в примере применяется булевское статическое поле уровня класса, которое служит для удержания в активном состоянии первичного потока `Main()`, пока не завершится вторичный поток.

На заметку! Строго говоря, использование булевской переменной в данном примере не является безопасным в отношении потоков, поскольку к ее значению имеют доступ два разных потока. В текущем примере подобное допустимо; тем не менее, запомните в качестве очень важного эмпирического правила: вы должны обеспечивать блокировку данных, разделяемых между несколькими потоками. Вы увидите, как это делается, далее в главе.

```
namespace AsyncCallbackDelegate
{
    public delegate int BinaryOp(int x, int y);

    class Program
    {
        private static bool isDone = false;
        static void Main(string[] args)
        {
            Console.WriteLine("***** AsyncCallbackDelegate Example *****");
            Console.WriteLine("Main() invoked on thread {0}.",
                Thread.CurrentThread.ManagedThreadId);

            BinaryOp b = new BinaryOp(Add);
            IAsyncResult ar = b.BeginInvoke(10, 10,
                new AsyncCallback(AddComplete), null);

            // Предположим, что здесь делается какая-то другая работа...
            while (!isDone)
            {
                Console.WriteLine("Working...");
                Thread.Sleep(1000);
            }
            Console.ReadLine();
        }
        static int Add(int x, int y)
        {
            Console.WriteLine("Add() invoked on thread {0}.",
                Thread.CurrentThread.ManagedThreadId);
```

```

        Thread.Sleep(5000);
        return x + y;
    }

    static void AddComplete(IAsyncResult iar)
    {
        Console.WriteLine("AddComplete() invoked on thread {0}.",
            Thread.CurrentThread.ManagedThreadId);
        Console.WriteLine("Your addition is complete");
        isDone = true;
    }
}

```

И снова статический метод `AddComplete()` будет вызван делегатом `AsyncCallback` после завершения метода `Add()`. Запустив программу, можно убедиться, что именно вторичный поток вызывает `AddComplete()`:

```

***** AsyncCallbackDelegate Example *****
Main() invoked on thread 1.
Add() invoked on thread 3.
Working....
Working....
Working....
Working....
Working....
AddComplete() invoked on thread 3.
Your addition is complete

```

Подобно другим примерам, приведенным в настоящей главе, вывод может несколько отличаться. На самом деле вполне возможно, что после завершения работы метода `AddComplete()` появится только одно финальное сообщение `Working....`. Это просто побочный эффект односекундной задержки в `Main()`.

Роль класса `AsyncResult`

В настоящий момент метод `AddComplete()` не выводит на консоль действительный результат операции (сложение двух чисел). Причина в том, что целевой метод делегата `AsyncCallback` (`AsyncResult()` в приведенном примере) не имеет доступа к исходному делегату `BinaryOp`, созданному в контексте `Main()`, а потому вызывать `EndInvoke()` из `AddComplete()` нельзя!

В то время как переменную `BinaryOp` можно было бы просто объявить как статический член класса, чтобы позволить обоим методам обращаться к одному и тому же объекту, более элегантное решение предусматривает применение входного параметра `IAsyncResult`.

Входной параметр `IAsyncResult`, передаваемый целевому методу делегата `AsyncCallback`, в действительности представляет собой экземпляр класса `AsyncResult` (обратите внимание на отсутствие префикса `I` в имени), определенного в пространстве имен `System.Runtime.Remoting.Messaging`. Свойство `AsyncDelegate` возвращает ссылку на исходный асинхронный делегат, который был создан где-то в другом месте.

Следовательно, чтобы получить ссылку на объект делегата `BinaryOp`, размещенный внутри `Main()`, нужно просто привести экземпляр `System.Object`, возвращенный свойством `AsyncDelegate`, к типу `BinaryOp`. В этот момент можно запустить `EndInvoke()`, как и ожидалось:

```
//Не забудьте импортировать пространство имен System.Runtime.Remoting.Messaging!
static void AddComplete(IAsyncResult iar)
{
    Console.WriteLine("AddComplete() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
    Console.WriteLine("Your addition is complete");
    // Теперь получить результат.
    AsyncResult ar = (AsyncResult)iar;
    BinaryOp b = (BinaryOp)ar.AsyncDelegate;
    Console.WriteLine("10 + 10 is {0}.", b.EndInvoke(iar));
    isDone = true;
}
```

Передача и получение специальных данных состояния

Финальным аспектом асинхронных делегатов, который необходимо обсудить, является последний аргумент метода `BeginInvoke()` (который до сих пор был `null`). Он позволяет передавать дополнительную информацию о состоянии методу обратного вызова из первичного потока. Поскольку упомянутый аргумент прототипирован как `System.Object`, в нем можно передавать данные любого типа при условии, что методу обратного вызова известно, чего ожидать. В целях демонстрации предположим, что первичный поток желает передать методу `AddComplete()` специальное текстовое сообщение:

```
static void Main(string[] args)
{
    ...
    IAsyncResult ar = b.BeginInvoke(10, 10, new AsyncCallback(AddComplete),
        "Main() thanks you for adding these numbers.");
    ...
}
```

Для получения таких данных внутри метода `AddComplete()` используется свойство `AsyncState` входного параметра `IAsyncResult`. Обратите внимание, что здесь потребуется явное приведение; следовательно, первичный и вторичный потоки должны согласовать между собой тип, возвращаемый `AsyncState`:

```
static void AddComplete(IAsyncResult iar)
{
    ...
    // Получить информационный объект и привести его к типу string.
    string msg = (string)iar.AsyncState;
    Console.WriteLine(msg);
    isDone = true;
}
```

Ниже показан вывод последней модификации примера:

```
***** AsyncCallbackDelegate Example *****
Main() invoked on thread 1.
Add() invoked on thread 3.
Working....
Working....
Working....
Working....
Working....
AddComplete() invoked on thread 3.
Your addition is complete
10 + 10 is 20.
Main() thanks you for adding these numbers.
```

Теперь, когда вы понимаете, каким образом применять делегат .NET для автоматического запуска вторичного потока выполнения с целью обработки асинхронного вызова метода, внимание можно переключить непосредственно на взаимодействие с потоками, используя пространство имен `System.Threading`. Вспомните, что данное пространство имен было первоначальным API-интерфейсом для реализации многопоточности .NET, который поставлялся еще в версии 1.0.

Исходный код. Проект `AsyncCallbackDelegate` доступен в подкаталоге `Chapter_19`.

Пространство имен `System.Threading`

В рамках платформы .NET пространство имен `System.Threading` предлагает несколько типов, которые дают возможность напрямую конструировать многопоточные приложения. В дополнение к типам, позволяющим взаимодействовать с отдельным потоком CLR, в этом пространстве имен определены типы, которые открывают доступ к пулу потоков, обслуживаемому CLR, простому (не связанному с графическим пользовательским интерфейсом) классу `Timer` и многочисленным типам, применяемым для предоставления синхронизированного доступа к разделяемым ресурсам. В табл. 19.1 перечислены некоторые важные члены пространства имен `System.Threading`. (За полными сведениями обращайтесь в документацию .NET Framework 4.7 SDK.)

Таблица 19.1. Основные типы пространства имен `System.Threading`

Тип	Назначение
<code>Interlocked</code>	Этот тип предоставляет атомарные операции для переменных, разделяемых между несколькими потоками
<code>Monitor</code>	Этот тип обеспечивает синхронизацию потоковых объектов, используя блокировки и ожидания/сигналы. Ключевое слово <code>lock</code> языка C# "за кулисами" применяет объект <code>Monitor</code>
<code>Mutex</code>	Этот примитив синхронизации может использоваться для синхронизации между границами доменов приложений
<code>Parameterized-ThreadStart</code>	Этот делегат позволяет потоку вызывать методы, принимающие произвольное количество аргументов
<code>Semaphore</code>	Этот тип позволяет ограничивать количество потоков, которые могут иметь доступ к ресурсу или к определенному типу ресурсов одновременно
<code>Thread</code>	Этот тип представляет поток, выполняемый в среде CLR. С применением данного типа можно порождать дополнительные потоки в исходном домене приложения
<code>ThreadPool</code>	Этот тип позволяет взаимодействовать с поддерживаемым средой CLR пулом потоков внутри заданного процесса
<code>ThreadPriority</code>	Это перечисление представляет уровень приоритета потока (<code>Highest</code> , <code>Normal</code> и т.д.)
<code>ThreadStart</code>	Этот делегат позволяет указывать метод для вызова в заданном потоке. В отличие от делегата <code>ParametrizedThreadStart</code> целевые методы <code>ThreadStart</code> всегда должны иметь один и тот же прототип
<code>ThreadState</code>	Это перечисление задает допустимые состояния потока (<code>Running</code> , <code>Aborted</code> и т.д.)
<code>Timer</code>	Этот тип предоставляет механизм выполнения метода через указанные интервалы
<code>TimerCallback</code>	Этот тип делегата используется в сочетании с типами <code>Timer</code>

Класс `System.Threading.Thread`

Класс `Thread` является самым элементарным из всех типов в пространстве имен `System.Threading`. Он представляет объектно-ориентированную оболочку вокруг заданного пути выполнения внутри отдельного домена приложения. В этом классе определено несколько методов (статических и уровня экземпляра), которые позволяют создавать новые потоки внутри текущего домена приложения, а также приостанавливать, останавливать и уничтожать указанный поток. Список основных статических членов приведен в табл. 19.2.

Таблица 19.2. Основные статические члены типа `Thread`

Статический член	Назначение
<code>CurrentContext</code>	Это свойство только для чтения возвращает контекст, в котором в текущий момент выполняется поток
<code>CurrentThread</code>	Это свойство только для чтения возвращает ссылку на текущий выполняемый поток
<code>GetDomain()</code> <code>GetDomainID()</code>	Эти методы возвращают ссылку на текущий домен приложения либо идентификатор домена, в котором выполняется текущий поток
<code>Sleep()</code>	Этот метод приостанавливает текущий поток на указанное время

Класс `Thread` также поддерживает члены уровня экземпляра, часть которых описана в табл. 19.3.

Таблица 19.3. Избранные члены уровня экземпляра типа `Thread`

Член уровня экземпляра	Назначение
<code>IsAlive</code>	Возвращает булевское значение, указывающее на то, запущен ли поток (и пока еще не прекращен или не отменен)
<code>IsBackground</code>	Получает или устанавливает значение, которое указывает, является ли данный поток фоновым (что более подробно объясняется далее в главе)
<code>Name</code>	Позволяет установить дружественное текстовое имя потока
<code>Priority</code>	Получает или устанавливает приоритет потока, который может принимать значение из перечисления <code>ThreadPriority</code>
<code>ThreadState</code>	Получает состояние данного потока, которое может принимать значение из перечисления <code>ThreadState</code>
<code>Abort()</code>	Указывает среде CLR на необходимость как можно более скорого прекращения работы потока
<code>Interrupt()</code>	Прерывает (например, приостанавливает) текущий поток на подходящий период ожидания
<code>Join()</code>	Блокирует вызывающий поток до тех пор, пока указанный поток (тот, на котором вызван метод <code>Join()</code>) не завершится
<code>Resume()</code>	Возобновляет выполнение ранее приостановленного потока
<code>Start()</code>	Указывает среде CLR на необходимость как можно более скорого запуска потока
<code>Suspend()</code>	Приостанавливает поток. Если поток уже приостановлен, то вызов <code>Suspend()</code> не оказывает никакого действия

На заметку! Прекращение работы или приостановка активного потока обычно считается плохой идеей. В таком случае есть шанс (хотя и небольшой), что поток может допустить “утечку” своей рабочей нагрузки.

Получение статистики о текущем потоке выполнения

Вспомните, что точка входа исполняемой сборки (т.е. метод `Main()`) запускается в первичном потоке выполнения. Чтобы проиллюстрировать базовое применение типа `Thread`, предположим, что имеется новый проект консольного приложения по имени `ThreadStats`. Как вам известно, статическое свойство `Thread.CurrentThread` извлекает объект `Thread`, который представляет поток, выполняющийся в текущий момент. Получив текущий поток, можно вывести разнообразные статистические сведения о нем:

```
// Не забудьте импортировать пространство имен System.Threading.
static void Main(string[] args)
{
    Console.WriteLine("***** Primary Thread stats *****\n");

    // Получить имя текущего потока.
    Thread primaryThread = Thread.CurrentThread;
    primaryThread.Name = "ThePrimaryThread";

    // Показать детали обслуживающего домена приложения и контекста.
    Console.WriteLine("Name of current AppDomain: {0}",
        Thread.GetDomain().FriendlyName); // Имя текущего домена приложения
    Console.WriteLine("ID of current Context: {0}",
        Thread.CurrentContext.ContextID); // Идентификатор текущего контекста

    // Вывести некоторую статистику о текущем потоке.
    Console.WriteLine("Thread Name: {0}",
        primaryThread.Name); // Имя потока
    Console.WriteLine("Has thread started?: {0}",
        primaryThread.IsAlive); // Запущен ли поток
    Console.WriteLine("Priority Level: {0}",
        primaryThread.Priority); // Приоритет потока
    Console.WriteLine("Thread State: {0}",
        primaryThread.ThreadState); // Состояние потока
    Console.ReadLine();
}
```

Вот как выглядит вывод:

```
***** Primary Thread stats *****
Name of current AppDomain: ThreadStats.exe
ID of current Context: 0
Thread Name: ThePrimaryThread
Has thread started?: True
Priority Level: Normal
Thread State: Running
```

Свойство Name

Хотя показанный выше код более или менее самоочевиден, обратите внимание, что класс `Thread` поддерживает свойство по имени `Name`. Если значение `Name` не было установлено, тогда будет возвращаться пустая строка. Однако назначение конкретному объекту `Thread` дружественного имени может значительно упростить отладку.

Во время сеанса отладки в Visual Studio можно открыть окно Threads (Потоки), выбрав пункт меню Debug Windows Threads (Отладка⇒Окна⇒Потоки). На рис. 19.1 легко заметить, что окно Threads позволяет быстро идентифицировать поток, который нужно диагностировать.

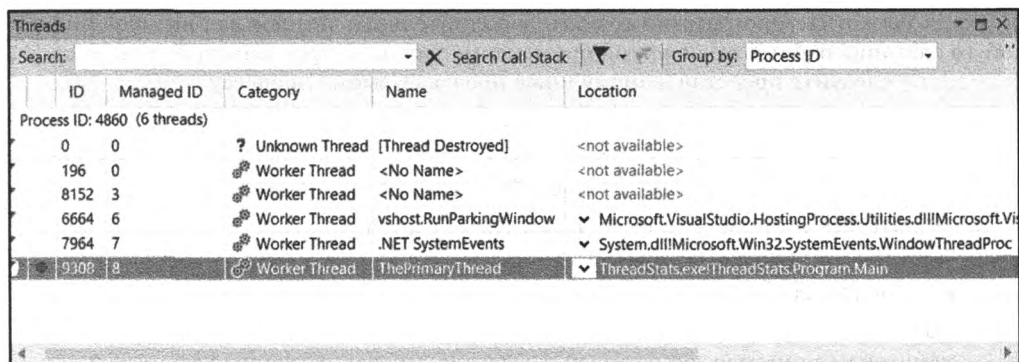


Рис. 19.1. Отладка потока в Visual Studio

Свойство Priority

Далее обратите внимание, что в типе Thread определено свойство по имени Priority. По умолчанию все потоки имеют уровень приоритета Normal. Тем не менее, в любой момент жизненного цикла потока его можно изменить, используя свойство Thread.Priority и связанное с ним перечисление System.Threading.ThreadPriority:

```
public enum ThreadPriority
{
    Lowest,
    BelowNormal,
    Normal, // Стандартное значение.
    AboveNormal,
    Highest
}
```

В случае присваивания уровню приоритета потока значения, отличающегося от стандартного (ThreadPriority.Normal), помните об отсутствии прямого контроля над тем, когда планировщик потоков будет переключать потоки между собой. В действительности уровень приоритета потока предоставляет среде CLR подсказку относительно важности действия потока. Таким образом, поток с уровнем приоритета ThreadPriority.Highest не обязательно гарантированно получит наивысший приоритет.

Опять-таки, если планировщик потоков занят решением определенной задачи (например, синхронизацией объекта, переключением потоков либо их перемещением), то уровень приоритета, скорее всего, будет соответствующим образом изменен. Однако при прочих равных условиях среда CLR прочитает эти значения и проинструктирует планировщик потоков о том, как лучше выделять кванты времени. Потоки с идентичными уровнями приоритета должны получать одинаковое количество времени на выполнение своей работы.

В большинстве случаев необходимость в прямом изменении уровня приоритета потока возникает редко (если вообще когда-либо). Теоретически можно так повысить уровень приоритета набора потоков, что в итоге воспрепятствовать выполнению низкоприоритетных потоков с их запрошенными уровнями (поэтому соблюдайте осторожность).

Ручное создание вторичных потоков

Когда вы хотите программно создать дополнительные потоки для выполнения какой-то единицы работы, то во время применения типов из пространства имен `System.Threading` следуйте представленному ниже предсказуемому процессу.

1. Создать метод, который будет служить точкой входа для нового потока.
2. Создать новый делегат `ParametrizedThreadStart` (или `ThreadStart`), передав его конструктору адрес метода, который был определен на шаге 1.
3. Создать объект `Thread`, передав конструктору делегат `ParametrizedThreadStart/ThreadStart` в качестве аргумента.
4. Установить начальные характеристики потока (имя, приоритет и т.д.).
5. Вызвать метод `Thread.Start()`, что приведет к как можно более быстрому запуску потока для метода, на который ссылается делегат, созданный на шаге 2.

Согласно шагу 2 для указания на метод, который будет выполняться во вторичном потоке, можно использовать два разных типа делегата. Делегат `ThreadStart` может указывать на любой метод, который не принимает аргументов и ничего не возвращает. Такой делегат может быть полезен, когда метод предназначен просто для запуска в фоновом режиме без дальнейшего взаимодействия с ним.

Очевидное ограничение `ThreadStart` связано с невозможностью передавать ему параметры для обработки. Тем не менее, тип делегата `ParametrizedThreadStart` позволяет передать единственный параметр типа `System.Object`. Учитывая, что с помощью `System.Object` представляется все, что угодно, посредством специального класса или структуры можно передавать любое количество параметров. Однако имейте в виду, что делегат `ParametrizedThreadStart` может указывать только на методы, возвращающие `void`.

Работа с делегатом ThreadStart

Чтобы проиллюстрировать процесс построения многопоточного приложения (а также его полезность), создадим проект консольного приложения по имени `SimpleMultiThreadApp`, которое позволяет конечному пользователю выбирать, будет приложение выполнять свою работу в единственном первичном потоке или же разделит рабочую нагрузку с применением двух отдельных потоков выполнения.

После импортирования пространства имен `System.Threading` определяется метод для выполнения работы (возможного) вторичного потока. Чтобы сосредоточиться на механизме построения многопоточных программ, этот метод будет просто выводить на консоль последовательность чисел, делая на каждом шаге паузу примерно в 2 секунды. Ниже показано полное определение класса `Printer`:

```
public class Printer
{
    public void PrintNumbers()
    {
        // Вывести информацию о потоке.
        Console.WriteLine("> {0} is executing PrintNumbers()",
            Thread.CurrentThread.Name);
        // Вывести числа.
```

```
Console.WriteLine("Your numbers: ");
for(int i = 0; i < 10; i++)
{
    Console.WriteLine("{0}, ", i);
    Thread.Sleep(2000);
}
Console.WriteLine();
}
```

Внутри метода `Main()` пользователю сначала предлагается решить, сколько потоков будет использоваться для выполнения работы приложения: один или два. Если пользователь запрашивает один поток, то нужно просто вызвать метод `PrintNumbers()` в первичном потоке. Тем не менее, когда пользователь выбирает два потока, понадобится создать делегат `ThreadStart`, указывающий на `PrintNumbers()`, передать объект делегата конструктору нового объекта `Thread` и вызвать метод `Start()` для информирования среды CLR о том, что данный поток готов к обработке.

Первым делом установим ссылку на сборку `System.Windows.Forms.dll` (а также импортируем пространство имен `System.Windows.Forms`) и отобразим сообщение в `Main()` с применением метода `MessageBox.Show()` (причина такого решения станет ясной после запуска программы). Вот полная реализация `Main()`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** The Amazing Thread App *****\n");
    Console.WriteLine("Do you want [1] or [2] threads? ");
    string threadCount = Console.ReadLine(); // Запрос количества потоков

    // Назначить имя текущему потоку.
    Thread primaryThread = Thread.CurrentThread;
    primaryThread.Name = "Primary";

    // Вывести информацию о потоке.
    Console.WriteLine("-> {0} is executing Main()",
        Thread.CurrentThread.Name);

    // Создать рабочий класс.
    Printer p = new Printer();
    switch(threadCount)
    {
        case "2":
            // Создать поток.
            Thread backgroundThread =
                new Thread(new ThreadStart(p.PrintNumbers));
            backgroundThread.Name = "Secondary";
            backgroundThread.Start();
            break;
        case "1":
            p.PrintNumbers();
            break;
        default:
            Console.WriteLine("I don't know what you want...you get 1 thread.");
            goto case "1"; // Переход к варианту с одним потоком
    }

    // Выполнить некоторую дополнительную работу.
    MessageBox.Show("I'm busy!", "Work on main thread...");
    Console.ReadLine();
}
```

Если теперь запустить программу с одним потоком, то обнаружится, что финальное окно сообщения не будет отображать сообщение, пока вся последовательность чисел не выведется на консоль. Поскольку после вывода каждого числа установлена пауза около 2 секунд, это создаст не особенно приятное впечатление у конечного пользователя. Однако в случае выбора двух потоков окно сообщения отображается немедленно, потому что для вывода чисел на консоль выделен отдельный объект Thread.

Исходный код. Проект SimpleMultiThreadApp доступен в подкаталоге Chapter_19.

Работа с делегатом ParametrizedThreadStart

Вспомните, что делегат ThreadStart может указывать только на методы, которые возвращают void и не принимают аргументов. В некоторых случаях это подходит, но если нужно передать данные методу, выполняющемуся во вторичном потоке, тогда придется использовать тип делегата ParametrizedThreadStart. В целях иллюстрации давайте воссоздадим логику проекта AsyncCallbackDelegate, разработанного ранее в главе, но теперь применим тип делегата ParameterizedThreadStart.

Для начала создадим новый проект консольного приложения по имени AddWithThreads и импортируем пространство имен System.Threading. С учетом того, что делегат ParametrizedThreadStart может указывать на любой метод, принимающий параметр типа System.Object, построим специальный тип, который содержит числа, подлежащие сложению:

```
class AddParams
{
    public int a, b;
    public AddParams(int numb1, int numb2)
    {
        a = numb1;
        b = numb2;
    }
}
```

Далее создадим в классе Program статический метод, который принимает параметр AddParams и выводит на консоль сумму двух чисел:

```
static void Add(object data)
{
    if (data is AddParams)
    {
        Console.WriteLine("ID of thread in Add(): {0}",
            Thread.CurrentThread.ManagedThreadId);

        AddParams ap = (AddParams)data;
        Console.WriteLine("{0} + {1} is {2}",
            ap.a, ap.b, ap.a + ap.b);
    }
}
```

Код внутри Main() прямолинеен. Вместо типа ThreadStart просто используется ParametrizedThreadStart:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Adding with Thread objects *****");
    Console.WriteLine("ID of thread in Main(): {0}",
```

```

    Thread.CurrentThread.ManagedThreadId);
// Создать объект AddParams для передачи вторичному потоку.
AddParams ap = new AddParams(10, 10);
Thread t = new Thread(new ParameterizedThreadStart(Add));
t.Start(ap);

// Подождать, пока другой поток завершится.
Thread.Sleep(5);

Console.ReadLine();
}

```

Класс AutoResetEvent

В нескольких первых примерах для информирования первичного потока о необходимости подождать, пока вторичный поток не завершится, применялся ряд грубых способов. Во время исследования асинхронных делегатов в качестве переключателя использовалась простая булевская переменная; тем не менее, решение подобного рода не является рекомендуемым, т.к. оба потока имеют доступ к одному и тому же элементу данных, что может привести к его повреждению. Более безопасной, хотя все еще неудобной альтернативой будет вызов метода `Thread.Sleep()` с фиксированным периодом времени. Проблема здесь в том, что нет желания ожидать дольше, чем необходимо.

Простой и безопасный к потокам способ заставить один поток ожидать, пока не завершится другой поток, предусматривает применение класса `AutoResetEvent`. В потоке, который должен ожидать (таком как метод `Main()`), создадим экземпляр `AutoResetEvent` и передадим его конструктору значение `false`, указав, что уведомления пока не было. Затем в точке, где требуется ожидать, вызовем метод `WaitOne()`. Ниже приведен модифицированный класс `Program`, который делает все описанное с использованием статической переменной-члена `AutoResetEvent`:

```

class Program
{
    private static AutoResetEvent waitHandle = new AutoResetEvent(false);
    static void Main(string[] args)
    {
        Console.WriteLine("***** Adding with Thread objects *****");
        Console.WriteLine("ID of thread in Main(): {0}",
            Thread.CurrentThread.ManagedThreadId);
        AddParams ap = new AddParams(10, 10);
        Thread t = new Thread(new ParameterizedThreadStart(Add));
        t.Start(ap);

        // Ожидать, пока не поступит уведомление!
        waitHandle.WaitOne();
        Console.WriteLine("Other thread is done!");

        Console.ReadLine();
    }
    ...
}

```

Когда другой поток завершит свою работу, он вызовет метод `Set()` на том же экземпляре типа `AutoResetEvent`:

```

static void Add(object data)
{
    if (data is AddParams)
    {

```

```

Console.WriteLine("ID of thread in Add(): {0}",
    Thread.CurrentThread.ManagedThreadId);
AddParams ap = (AddParams)data;
Console.WriteLine("{0} + {1} is {2}",
    ap.a, ap.b, ap.a + ap.b);

// Сообщить другому потоку о том, что работа завершена.
waitHandle.Set();
}
}

```

Исходный код. Проект AddWithThreads доступен в подкаталоге Chapter_19.

Потоки переднего плана и фоновые потоки

Теперь, когда вы знаете, как программно создавать новые потоки выполнения с применением типов из пространства имен `System.Threading`, давайте формализуем разницу между потоками переднего плана и фоновыми потоками.

- *Потоки переднего плана* имеют возможность предохранять текущее приложение от завершения. Среда CLR не будет прекращать работу приложения (скажем, выгружая текущий домен приложения) до тех пор, пока не будут завершены все потоки переднего плана.
- *Фоновые потоки* (иногда называемые *потоками-демонами*) воспринимаются средой CLR как расширяемые пути выполнения, которые в любой момент времени могут быть проигнорированы (даже если они заняты выполнением некоторой части работы). Таким образом, если при выгрузке домена приложения все потоки переднего плана завершены, то все фоновые потоки автоматически уничтожаются.

Важно отметить, что потоки переднего плана и фоновые потоки — не синонимы первичных и рабочих потоков. По умолчанию каждый поток, создаваемый посредством метода `Thread.Start()`, автоматически становится потоком переднего плана. В итоге домен приложения не выгрузится до тех пор, пока все потоки выполнения не завершат свои единицы работы. В большинстве случаев именно такое поведение и требуется.

Ради доказательства сделанных утверждений предположим, что метод `Printer.PrintNumbers()` необходимо вызвать во вторичном потоке, который должен вести себя как фоновый. Это означает, что метод, указываемый типом `Thread` (через делегат `ThreadStart` или `ParametrizedThreadStart`), должен обладать возможностью безопасного останова, как только все потоки переднего плана закончат свою работу. Конфигурирование такого потока сводится просто к установке свойства `IsBackground` в `true`:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Background Threads *****\n");
    Printer p = new Printer();
    Thread bgroundThread =
        new Thread(new ThreadStart(p.PrintNumbers));

    // Теперь это фоновый поток.
    bgroundThread.IsBackground = true;
    bgroundThread.Start();
}

```

Обратите внимание, что в приведенном выше методе `Main()` не делается вызов `Console.ReadLine()`, чтобы заставить окно консоли оставаться видимым, пока не будет нажата клавиша <Enter>. Таким образом, после запуска приложение немедленно прекращается, потому что объект `Thread` сконфигурирован как фоновый поток. Учитывая, что метод `Main()` инициирует создание первичного потока *переднего плана*, как только логика метода `Main()` завершится, домен приложения будет выгружен, прежде чем вторичный поток сможет закончить свою работу.

Однако если закоментировать строку, которая устанавливает свойство `IsBackground` в `true`, то обнаружится, что на консоль выводятся все числа, поскольку все потоки переднего плана должны завершить свою работу перед тем, как домен приложения будет выгружен из обслуживающего процесса.

По большей части конфигурировать поток для функционирования в фоновом режиме может быть удобно, когда интересующий рабочий поток выполняет не критичную задачу, потребность в которой исчезает после завершения главной задачи программы. Например, можно было бы построить приложение, которое проверяет сервер электронной почты каждые несколько минут на предмет поступления новых сообщений, обновляет текущий прогноз погоды или решает какие-то другие не критичные задачи.

Проблемы параллелизма

При построении многопоточных приложений необходимо гарантировать, что любой фрагмент разделяемых данных защищен от возможности изменения со стороны сразу нескольких потоков. Поскольку все потоки в домене приложения имеют параллельный доступ к разделяемым данным приложения, вообразите, что может произойти, если множество потоков одновременно обратятся к одному и тому же элементу данных. Так как планировщик потоков случайным образом приостанавливает их работу, что если поток А будет вытеснен до завершения своей работы? Тогда поток В прочитает нестабильные данные.

Чтобы проиллюстрировать проблему, связанную с параллелизмом, давайте создадим еще один проект консольного приложения под названием `MultiThreadedPrinting`. В приложении снова будет использоваться построенный ранее класс `Printer`, но на этот раз метод `PrintNumbers()` приостановит текущий поток на случайно сгенерированный период времени.

```
public class Printer
{
    public void PrintNumbers()
    {
        ...
        for (int i = 0; i < 10; i++)
        {
            // Приостановить поток на случайный период времени.
            Random r = new Random();
            Thread.Sleep(1000 * r.Next(5));
            Console.Write("{0}, ", i);
        }
        Console.WriteLine();
    }
}
```

Метод `Main()` отвечает за создание массива из десяти (уникально именованных) объектов `Thread`, каждый из которых производит вызов одного и того же экземпляра класса `Printer`:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("*****Synchronizing Threads *****\n");
        Printer p = new Printer();
        // Создать 10 потоков, которые указывают на один
        // и тот же метод того же самого объекта.
        Thread[] threads = new Thread[10];
        for (int i = 0; i < 10; i++)
        {
            threads[i] = new Thread(new ThreadStart(p.PrintNumbers))
            {
                Name = $"Worker thread #{i}"
            };
        }
        // Теперь запустить их все.
        foreach (Thread t in threads)
            t.Start();
        Console.ReadLine();
    }
}

```

Прежде чем взглянуть на тестовые запуски, кратко повторим проблему. Первичный поток внутри этого домена приложения начинает свое существование с порождения десяти вторичных рабочих потоков. Каждому рабочему потоку указывается на необходимость вызова метода `PrintNumbers()` *того же самого* экземпляра `Printer`. Поскольку никаких мер для блокировки разделяемых ресурсов данного объекта (консоли) не предпринималось, есть неплохой шанс, что текущий поток будет вытеснен до того, как метод `PrintNumbers()` выведет полные результаты. Из-за того, что не известно в точности, когда подобное может произойти (если вообще произойдет), будут получены непредсказуемые результаты. Например, вывод может выглядеть так:

```

*****Synchronizing Threads *****
-> Worker thread #1 is executing PrintNumbers()
Your numbers: -> Worker thread #0 is executing PrintNumbers()
-> Worker thread #2 is executing PrintNumbers()
Your numbers: -> Worker thread #3 is executing PrintNumbers()
Your numbers: -> Worker thread #4 is executing PrintNumbers()
Your numbers: -> Worker thread #6 is executing PrintNumbers()
Your numbers: -> Worker thread #7 is executing PrintNumbers()
Your numbers: -> Worker thread #8 is executing PrintNumbers()
Your numbers: -> Worker thread #9 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #5 is executing PrintNumbers()
Your numbers: 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 2, 1, 0, 0, 4, 3,
4, 1, 2, 4, 5, 5, 5, 6, 6, 6, 2, 7, 7, 7, 3, 4, 0, 8, 4, 5, 1, 5, 8, 8, 9,
2, 6, 1, 0, 9, 1,
6, 2, 7, 9,
2, 1, 7, 8, 3, 2, 3, 3, 9,
8, 4, 4, 5, 9,
4, 3, 5, 5, 6, 3, 6, 7, 4, 7, 6, 8, 7, 4, 8, 5, 5, 6, 6, 8, 7, 7, 9,
8, 9,
8, 9,
9,
9,

```

Запустим приложение еще несколько раз. Вот еще один вариант вывода:

```
*****Synchronizing Threads *****
-> Worker thread #0 is executing PrintNumbers()
-> Worker thread #1 is executing PrintNumbers()
-> Worker thread #2 is executing PrintNumbers()
Your numbers: -> Worker thread #4 is executing PrintNumbers()
Your numbers: -> Worker thread #5 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #6 is executing PrintNumbers()
Your numbers: -> Worker thread #7 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #8 is executing PrintNumbers()
Your numbers: -> Worker thread #9 is executing PrintNumbers()
Your numbers: -> Worker thread #3 is executing PrintNumbers()
Your numbers: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4,
4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7
, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9,
9,
9,
9,
9,
9,
9,
9,
9,
9,
```

На заметку! Если получить непредсказуемый вывод не удастся, увеличьте количество потоков с 10 до 100 (например) или добавьте в код еще один вызов `Thread.Sleep()`. В конце концов, вы столкнетесь с проблемой параллелизма.

Должно быть совершенно ясно, что здесь присутствуют проблемы. В то время как каждый поток сообщает экземпляру `Printer` о необходимости вывода числовых данных, планировщик потоков благополучно переключает потоки в фоновом режиме. В итоге получается несогласованный вывод. Нужен способ программной реализации синхронизированного доступа к разделяемым ресурсам. Как и можно было предположить, пространство имен `System.Threading` предлагает несколько типов, связанных с синхронизацией. В языке `C#` также предусмотрено специальное ключевое слово для синхронизации разделяемых данных в многопоточных приложениях.

Синхронизация с использованием ключевого слова `lock` языка `C#`

Первый прием, который можно применять для синхронизации доступа к разделяемым ресурсам, предполагает использование ключевого слова `lock` языка `C#`. Оно позволяет определять блок операторов, которые должны быть синхронизованными между потоками. В результате входящие потоки не могут прерывать текущий поток, мешая ему завершить свою работу. Ключевое слово `lock` требует указания *маркера* (объектной ссылки), который должен быть получен потоком для входа в область действия блокировки. Чтобы попытаться заблокировать закрытый метод уровня экземпляра, необходимо просто передать ссылку на текущий тип:

```
private void SomePrivateMethod()
{
    // Использовать текущий объект как маркер потока.
    lock(this)
    {
```



```

    // Весь код внутри этого блока является безопасным к потокам.
}
}

```

Тем не менее, если блокируется область кода внутри *открытого* члена, то безопаснее (да и рекомендуется) объявить закрытую переменную-член типа `object` для применения в качестве маркера блокировки:

```

public class Printer
{
    // Маркер блокировки.
    private object threadLock = new object();
    public void PrintNumbers()
    {
        // Использовать маркер блокировки.
        lock (threadLock)
        {
            ...
        }
    }
}

```

В любом случае, если взглянуть на метод `PrintNumbers()`, то можно заметить, что разделяемым ресурсом, за доступ к которому соперничают потоки, является окно консоли. Следовательно, если поместить весь код взаимодействия с типом `Console` внутрь области `lock`, как показано далее, тогда будет построен метод, который позволит текущему потоку завершить свою задачу:

```

public void PrintNumbers()
{
    // Использовать в качестве маркера блокировки закрытый член object.
    lock (threadLock)
    {
        // Вывести информацию о потоке.
        Console.WriteLine("-> {0} is executing PrintNumbers()",
            Thread.CurrentThread.Name);

        // Вывести числа.
        Console.Write("Your numbers: ");
        for (int i = 0; i < 10; i++)
        {
            Random r = new Random();
            Thread.Sleep(1000 * r.Next(5));
            Console.Write("{0}, ", i);
        }
        Console.WriteLine();
    }
}

```

Как только поток входит в область `lock`, маркер блокировки (в данном случае ссылка на текущий объект) становится недоступным другим потокам до тех пор, пока блокировка не будет освобождена после выхода из области `lock`. Таким образом, если поток А получил маркер блокировки, то другие потоки не смогут войти ни в одну из областей, которые используют тот же самый маркер, до тех пор, пока поток А не освободит его.

На заметку! Если необходимо блокировать код в статическом методе, тогда следует просто объявить закрытую статическую переменную-член типа `object`, которая и будет служить маркером блокировки.

Запустив приложение, можно заметить, что каждый поток получил возможность выполнить свою работу до конца:

```
*****Synchronizing Threads *****
-> Worker thread #0 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #1 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #3 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #2 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #4 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #5 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #7 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #6 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #8 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #9 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Исходный код. Проект `MultiThreadedPrinting` доступен в подкаталоге `Chapter_19`.

Синхронизация с использованием типа `System.Threading.Monitor`

Оператор `lock` языка C# на самом деле представляет собой сокращение для работы с классом `System.Threading.Monitor`. При обработке компилятором C# область `lock` преобразуется в следующую конструкцию (в чем легко убедиться с помощью утилиты `ldasm.exe`):

```
public void PrintNumbers()
{
    Monitor.Enter(threadLock);
    try
    {
        // Вывести информацию о потоке.
        Console.WriteLine("-> {0} is executing PrintNumbers()",
            Thread.CurrentThread.Name);
        // Вывести числа.
        Console.Write("Your numbers: ");
        for (int i = 0; i < 10; i++)
        {
            Random r = new Random();
            Thread.Sleep(1000 * r.Next(5));
            Console.Write("{0}, ", i);
        }
        Console.WriteLine();
    }
    finally
    {
        Monitor.Exit(threadLock);
    }
}
```

Первым делом обратите внимание, что конечным получателем маркера потока, который указывается как аргумент ключевого слова `lock`, является метод `Monitor.Enter()`. Весь код внутри области `lock` помещен внутрь блока `try`. Соответствующая конструкция `finally` гарантирует освобождение маркера блокировки (посредством метода `Monitor.Exit()`), даже если возникнут любые исключения времени выполнения. Модифицировав программу `MultiThreadShareData` с целью прямого применения типа `Monitor` (как только что было показано), вы обнаружите, что вывод идентичен.

С учетом того, что ключевое слово `lock` требует написания меньшего объема кода, чем при явной работе с типом `System.Threading.Monitor`, может возникнуть вопрос о преимуществах использования этого типа напрямую. Выражаясь кратко, тип `Monitor` обеспечивает большую степень контроля. Применяя тип `Monitor`, можно заставить активный поток ожидать в течение некоторого периода времени (с помощью статического метода `Monitor.Wait()`), информировать ожидающие потоки о том, что текущий поток завершен (через статические методы `Monitor.Pulse()` и `Monitor.PulseAll()`), и т.д.

Как и можно было ожидать, в значительном числе случаев ключевого слова `lock` будет достаточно. Если вас интересуют дополнительные члены класса `Monitor`, тогда обращайтесь в документацию `.NET Framework 4.7 SDK`.

Синхронизация с использованием типа `System.Threading.Interlocked`

Не заглядывая в код CIL, трудно поверить, что присваивание и простые арифметические операции *не являются атомарными*. По указанной причине в пространстве имен `System.Threading` предоставляется тип, который позволяет атомарно оперировать одиночным элементом данных с меньшими накладными расходами, чем тип `Monitor`. В классе `Interlocked` определены статические члены, часть которых описана в табл. 19.4.

Таблица 19.4. Избранные статические члены типа `System.Threading.Interlocked`

Член	Назначение
<code>CompareExchange()</code>	Безопасно проверяет два значения на равенство и, если они равны, то заменяет одно из значений третьим
<code>Decrement()</code>	Безопасно уменьшает значение на 1
<code>Exchange()</code>	Безопасно меняет два значения местами
<code>Increment()</code>	Безопасно увеличивает значение на 1

Несмотря на то что это не сразу видно, процесс атомарного изменения одиночного значения довольно часто применяется в многопоточной среде. Предположим, что есть метод `AddOne()`, который инкрементирует целочисленную переменную-член по имени `intVal`. Вместо написания кода синхронизации вроде показанного ниже:

```
public void AddOne()
{
    lock(myLockToken)
    {
        intVal++;
    }
}
```

код можно упростить, используя статический метод `Interlocked.Increment()`. Методу потребуется передать инкрементируемую переменную по ссылке.

Обратите внимание, что метод `Increment()` не только изменяет значение входного параметра, но также возвращает полученное новое значение:

```
public void AddOne()
{
    int newVal = Interlocked.Increment(ref intVal);
}
```

В дополнение к методам `Increment()` и `Decrement()` тип `Interlocked` позволяет атомарно присваивать числовые и объектные данные. Например, чтобы присвоить переменной-члену значение 83, можно обойтись без явного оператора `lock` (или явной логики `Monitor`) и применить метод `Interlock.Exchange()`:

```
public void SafeAssignment()
{
    Interlocked.Exchange(ref myInt, 83);
}
```

Наконец, если необходимо проверить два значения на предмет равенства и изменить элемент сравнения в безопасной к потокам манере, тогда допускается использовать метод `Interlocked.CompareExchange()`:

```
public void CompareAndExchange()
{
    // Если значение i равно 83, то изменить его на 99.
    Interlocked.CompareExchange(ref i, 99, 83);
}
```

Синхронизация с использованием атрибута [Synchronization]

Последний из примитивов синхронизации, который мы здесь рассмотрим — атрибут `[Synchronization]`, который определен в пространстве имен `System.Runtime.Remoting.Contexts`. По существу данный атрибут уровня класса блокирует весь код членов экземпляра класса с целью обеспечения безопасности к потокам. Когда среда CLR размещает в памяти объекты, снабженные атрибутами `[Synchronization]`, она помещает объект внутрь контекста синхронизации. Как было показано в главе 17, объекты, которые не должны выходить за границы контекста, являются производными от `ContextBoundObject`. Следовательно, чтобы сделать класс `Printer` безопасным к потокам (без явного написания соответствующего кода внутри членов класса), его определение следует обновить, как показано ниже:

```
using System.Runtime.Remoting.Contexts;
...
// Все методы класса Printer теперь безопасны к потокам!
[Synchronization]
public class Printer : ContextBoundObject
{
    public void PrintNumbers()
    {
        ...
    }
}
```

В некоторых отношениях демонстрируемый подход выглядит как “ленивый” способ написания безопасного к потокам кода, т.к. не приходится углубляться в детали того, какие именно аспекты типа действительно манипулируют чувствительными к пото-

кам данными. Однако главный недостаток подхода в том, что даже если определенный метод не работает с чувствительными к потокам данными, то среда CLR *по-прежнему* будет блокировать вызовы такого метода. Очевидно, что это может привести к снижению общей функциональности типа, а потому применяйте описанный прием с осторожностью.

Программирование с использованием обратных вызовов `Timer`

Многие приложения нуждаются в вызове специфического метода через регулярные интервалы времени. Например, в приложении может существовать необходимость в отображении текущего времени внутри панели состояния с помощью определенной вспомогательной функции. Или, скажем, нужно, чтобы приложение эпизодически вызывало вспомогательную функцию, выполняющую некритичные фоновые задачи, такие как проверка поступления новых сообщений электронной почты. В ситуациях подобного рода можно применять тип `System.Threading.Timer` в сочетании со связанным делегатом по имени `TimerCallback`.

В целях иллюстрации предположим, что имеется проект консольного приложения (`TimerApp`), которое будет выводить текущее время каждую секунду до тех пор, пока пользователь не нажмет клавишу `<Enter>` для прекращения работы приложения. Первый очевидный шаг — написание метода, который будет вызываться типом `Timer` (не забудьте импортировать в файл кода пространство имен `System.Threading`):

```
class Program
{
    static void PrintTime(object state)
    {
        Console.WriteLine("Time is: {0}",
            DateTime.Now.ToLongTimeString());
    }

    static void Main(string[] args)
    {
    }
}
```

Обратите внимание, что метод `PrintTime()` принимает единственный параметр типа `System.Object` и возвращает `void`. Это обязательно, потому что делегат `TimerCallback` может вызывать только методы, которые соответствуют такой сигнатуре. Значение, передаваемое целевому методу делегата `TimerCallback`, может быть объектом любого типа (в случае примера с электронной почтой параметр может представлять имя сервера Microsoft Exchange для взаимодействия в течение процесса). Также обратите внимание, что поскольку параметр на самом деле является экземпляром типа `System.Object`, в нем можно передавать несколько аргументов, используя `System.Array` или специальный класс либо структуру.

Следующий шаг связан с конфигурированием экземпляра делегата `TimerCallback` и передачей его объекту `Timer`. В дополнение к настройке делегата `TimerCallback` конструктор `Timer` позволяет указывать необязательный информационный параметр для передачи целевому методу делегата (определенный как `System.Object`), интервал вызова метода и период ожидания (в миллисекундах), который должен истечь перед первым вызовом.

Вот пример:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Working with Timer type *****\n");
    // Создать делегат для типа Timer.
    TimerCallback timeCB = new TimerCallback(PrintTime);
    // Установить параметры таймера.
    Timer t = new Timer(
        timeCB,          // Объект делегата TimerCallback.
        null,            // Информация для передачи в вызванный метод
                        // (null, если информация отсутствует).
        0,               // Период ожидания перед запуском (в миллисекундах).
        1000);           // Интервал между вызовами (в миллисекундах).

    Console.WriteLine("Hit key to terminate...");
    Console.ReadLine();
}
```

В этом случае метод `PrintTime()` вызывается приблизительно каждую секунду и не получает никакой дополнительной информации. Ниже показан вывод примера:

```
***** Working with Timer type *****

Hit key to terminate...
Time is: 6:51:48 PM
Time is: 6:51:49 PM
Time is: 6:51:50 PM
Time is: 6:51:51 PM
Time is: 6:51:52 PM
Press any key to continue . . .
```

Чтобы передать целевому методу делегата какую-то информацию, необходимо просто заменить значение `null` во втором параметре конструктора подходящей информацией, например:

```
// Установить параметры таймера.
Timer t = new Timer(timeCB, "Hello From Main", 0, 1000);
```

Получить входные данные можно следующим образом:

```
static void PrintTime(object state)
{
    Console.WriteLine("Time is: {0}, Param is: {1}",
        DateTime.Now.ToLongTimeString(), state.ToString());
}
```

Использование автономного отбрасывания

В предыдущем примере переменная `Timer` не применяется в каком-либо пути выполнения и потому может быть заменена отбрасыванием:

```
var _ = new Timer(
    timeCB,          // Объект делегата TimerCallback.
    null,            // Информация для передачи в вызванный метод
                    // (null, если информация отсутствует).
    0,               // Период ожидания перед запуском (в миллисекундах).
    1000);           // Интервал между вызовами (в миллисекундах).
```

Пул потоков CLR

Следующей темой, связанной с потоками, которую мы рассмотрим в главе, будет роль пула потоков CLR. При асинхронном вызове метода с применением типов делегатов (посредством метода `BeginInvoke()`) среда CLR не создает новый поток в прямом смысле слова. В целях эффективности метод `BeginInvoke()` делегата задействует пул рабочих потоков, который поддерживается исполняющей средой. Для взаимодействия с этим пулом ожидающих потоков в пространстве имен `System.Threading` предлагается класс `ThreadPool`.

Чтобы запросить поток из пула для обработки вызова метода, можно использовать метод `ThreadPool.QueueUserWorkItem()`. Он имеет перегруженную версию, которая позволяет в дополнение к экземпляру делегата `WaitCallback` указывать необязательный параметр `System.Object` для передачи специальных данных состояния:

```
public static class ThreadPool
{
    ...
    public static bool QueueUserWorkItem(WaitCallback callBack);
    public static bool QueueUserWorkItem(WaitCallback callBack,
                                         object state);
}
```

Делегат `WaitCallback` может указывать на любой метод, который принимает в качестве единственного параметра экземпляр `System.Object` (представляющий необязательные данные состояния) и ничего не возвращает. Обратите внимание, что если при вызове `QueueUserWorkItem()` не задается экземпляр `System.Object`, то среда CLR автоматически передает значение `null`. Чтобы продемонстрировать работу методов очередей, работающих с пулом потоков CLR, рассмотрим еще раз программу, в которой применяется тип `Printer`. На этот раз массив объектов `Thread` не создается вручную, а метод `PrintNumbers()` будет назначаться членам пула потоков:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with the CLR Thread Pool *****\n");
        Console.WriteLine("Main thread started. ThreadID = {0}",
            Thread.CurrentThread.ManagedThreadId);

        Printer p = new Printer();
        WaitCallback workItem = new WaitCallback(PrintTheNumbers);
        // Поставить в очередь метод десять раз.
        for (int i = 0; i < 10; i++)
        {
            ThreadPool.QueueUserWorkItem(workItem, p);
        }
        Console.WriteLine("All tasks queued");
        Console.ReadLine();
    }
    static void PrintTheNumbers(object state)
    {
        Printer task = (Printer)state;
        task.PrintNumbers();
    }
}
```

У вас может возникнуть вопрос: почему взаимодействовать с пулом потоков, поддерживаемым средой CLR, выгоднее по сравнению с явным созданием объектов Thread? Использование пула потоков обеспечивает следующие преимущества.

- Пул потоков эффективно управляет потоками, сводя к минимуму количество потоков, которые должны создаваться, запускаться и останавливаться.
- За счет применения пула потоков можно сосредоточиться на решении задачи, а не на потоковой инфраструктуре приложения.

Тем не менее, в некоторых случаях ручное управление потоками оказывается более предпочтительным. Ниже приведены примеры.

- Когда требуются потоки переднего плана или должен устанавливаться приоритет потока. Потоки из пула *всегда* являются фоновыми и обладают стандартным приоритетом (`ThreadPriority.Normal`).
- Когда требуется поток с фиксированной идентичностью, чтобы его можно было прерывать, приостанавливать или находить по имени.

Исходный код. Проект `ThreadPoolApp` доступен в подкаталоге `Chapter_19`.

Итак, исследование пространства имен `System.Threading` завершено. Несомненно, понимание вопросов, рассмотренных в настоящей главе до сих пор (особенно в разделе, посвященном проблемам параллелизма), будет чрезвычайно ценным при создании многопоточного приложения. А теперь, опираясь на имеющийся фундамент, мы переклещим внимание на несколько новых аспектов, связанных с потоками, которые присутствуют только в .NET 4.0 и последующих версиях. Для начала мы обратимся к альтернативной потоковой модели, которая называется TPL.

Параллельное программирование с использованием TPL

Вы уже ознакомились с двумя технологиями программирования (асинхронные делегаты и члены пространства имен `System.Threading`), которые позволяют строить многопоточное программное обеспечение. Вспомните, что оба подхода будут работать в любой версии платформы .NET.

Начиная с версии .NET 4.0 в Microsoft ввели новый подход к разработке многопоточных приложений, предусматривающий применение библиотеки параллельного программирования, которая называется TPL. С помощью типов из `System.Threading.Tasks` можно строить мелко модульный масштабируемый параллельный код без необходимости напрямую иметь дело с потоками или пулом потоков.

Однако речь не идет о том, что вы не будете использовать типы из пространства имен `System.Threading` во время применения TPL. В реальности эти два инструментальных набора для создания многопоточных приложений могут вполне естественным образом работать вместе. Сказанное особенно верно в связи с тем, что пространство имен `System.Threading` по-прежнему предоставляет большинство примитивов синхронизации, которые рассматривались ранее (`Monitor`, `Interlocked` и т.д.). В итоге, скорее всего, вы обнаружите, что иметь дело с TPL предпочтительнее, чем с первоначальным пространством имен `System.Threading`, т.к. те же самые задачи могут решаться гораздо проще.

На заметку! Кстати, имейте в виду, что новые ключевые слова `async` и `await` языка C# используют разнообразные члены пространства имен `System.Threading.Tasks`.

Пространство имен `System.Threading.Tasks`

Коллективно типы из пространства `System.Threading.Tasks` называются *библиотекой параллельных задач* (TPL). Библиотека TPL будет автоматически распределять нагрузку приложения между доступными процессорами в динамическом режиме с применением пула потоков CLR. Библиотека TPL поддерживает разбиение работы на части, планирование потоков, управление состоянием и другие низкоуровневые детали. В конечном итоге появляется возможность максимизировать производительность приложений .NET, не сталкиваясь со сложностями прямой работы с потоками (рис. 19.2).

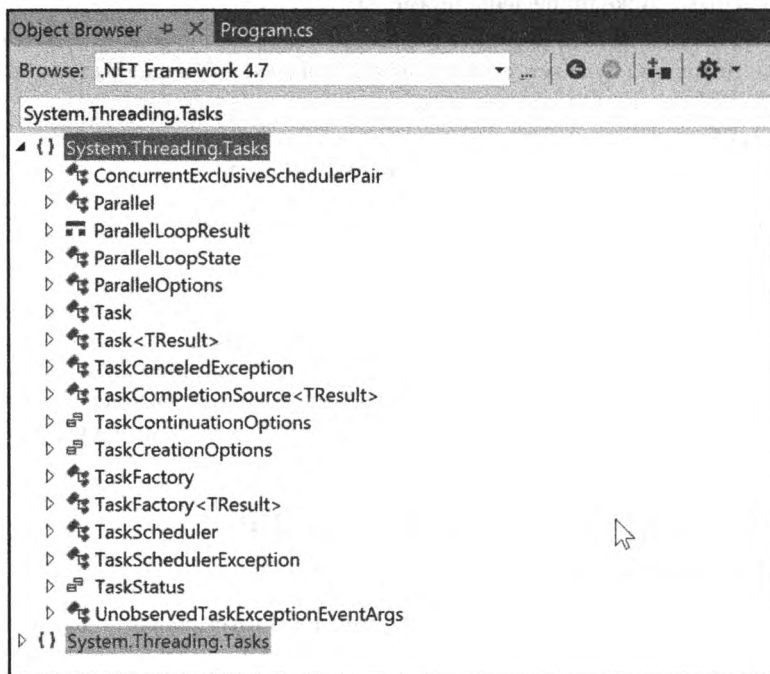


Рис. 19.2. Члены пространства имен `System.Threading.Tasks`

Роль класса `Parallel`

Основным классом в TPL является `System.Threading.Tasks.Parallel`. Он поддерживает набор методов, которые позволяют осуществлять итерацию по коллекции данных (точнее по объекту, реализующему интерфейс `IEnumerable<T>`) в параллельной манере. В документации .NET Framework 4.7 SDK указано, что в этом классе определены два главных статических метода, `Parallel.For()` и `Parallel.ForEach()`, каждый из которых имеет много перегруженных версий.

Упомянутые методы позволяют создавать тело из операторов кода, которое будет выполняться в параллельном режиме. Концептуально такие операторы представляют логику того же рода, которая была бы написана в нормальной циклической конструкции (с использованием ключевых слов `for` и `foreach` языка C#). Преимущество заключается в том, что класс `Parallel` будет самостоятельно извлекать потоки из пула потоков (и управлять параллелизмом).

Оба метода требуют передачи совместимого с `IEnumerable` или `IEnumerable<T>` контейнера, который хранит данные, подлежащие обработке в параллельном ре-

жиме. Контейнер может быть простым массивом, необобщенной коллекцией (вроде `ArrayList`), обобщенной коллекцией (наподобие `List<T>`) или результатами, полученными из запроса LINQ.

Вдобавок понадобится применять делегаты `System.Func<T>` и `System.Action<T>` для указания целевого метода, который будет вызываться при обработке данных. Делегат `Func<T>` уже встречался в главе 12 во время исследования технологии LINQ to Objects. Вспомните, что `Func<T>` представляет метод, который возвращает значение и принимает различное количество аргументов. Делегат `Action<T>` похож на `Func<T>` в том, что позволяет задавать метод, принимающий несколько параметров, но данный метод должен возвращать `void`.

Хотя можно было бы вызывать методы `Parallel.For()` и `Parallel.ForEach()` и передавать им строго типизированный объект делегата `Func<T>` или `Action<T>`, задача программирования упрощается за счет использования подходящих анонимных методов или лямбда-выражений C#.

Обеспечение параллелизма данных с помощью класса `Parallel`

Первое применение библиотеки TPL связано с обеспечением *параллелизма данных*. Таким термином обозначается задача прохода по массиву или коллекции в параллельной манере с помощью метода `Parallel.For()` или `Parallel.ForEach()`. Предположим, что необходимо выполнить некоторые трудоемкие операции файлового ввода-вывода. В частности, требуется загрузить в память большое число файлов *.jpg, повернуть содержащиеся в них изображения и сохранить модифицированные данные изображений в новом месте.

В документации .NET Framework 4.7 SDK приведен пример консольного приложения для указанной часто встречающейся задачи. Тем не менее, мы решим ее с использованием графического пользовательского интерфейса, чтобы взглянуть на применение “анонимных делегатов”, позволяющих вторичным потокам обновлять первичный поток пользовательского интерфейса.

На заметку! При построении многопоточного приложения с графическим пользовательским интерфейсом вторичные потоки никогда не смогут напрямую обращаться к элементам управления пользовательского интерфейса. Причина в том, что элементы управления (кнопки, текстовые поля, метки, индикаторы хода работ и т.п.) привязаны к потоку, в котором они создавались. В следующем примере иллюстрируется один из способов обеспечения для вторичных потоков возможности получать доступ к элементам пользовательского интерфейса в безопасной к потокам манере. Во время рассмотрения ключевых слов `async` и `await` языка C# будет предложен более простой подход.

В целях иллюстрации создадим приложение Windows Presentation Foundation (выбрав шаблон WPF App (.NET Framework)) по имени `DataParallelismWithForEach`.

На заметку! Инфраструктура Windows Presentation Foundation (WPF) будет подробно рассматриваться в главах 24–26. На тот случай, если вы еще не работали с WPF, здесь описано все, что необходимо для данного примера. Полное решение `DataParallelismWithForEach` доступно в подкаталоге `Chapter_19`.

Дважды щелкнув на файле `MainWindow.xaml` в окне Solution Explorer, поместим в него показанное далее содержимое XAML:

```
<Window x:Class="DataParallelismWithForEach.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

```

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:DataParallelismWithForEach"
mc:Ignorable="d"
Title="Fun with TPL" Height="200" Width="400">
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <Label Grid.Row="0" Grid.Column="0">
    Feel free to type here while the images are processed...
  </Label>
  <TextBox Grid.Row="1" Grid.Column="0" Margin="10,10,10,10"/>
  <Grid Grid.Row="2" Grid.Column="0">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="*/>
      <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <Button Name="cmdCancel" Grid.Row="0" Grid.Column="0" Margin="10,10,0,10"
      Click="cmdCancel_Click">
      Cancel
    </Button>
    <Button Name="cmdProcess" Grid.Row="0" Grid.Column="2" Margin="0,10,10,10"
      Click="cmdProcess_Click">
      Click to Flip Your Images!
    </Button>
  </Grid>
</Grid>
</Window>

```

И снова пока не следует задаваться вопросом о том, что означает приведенная разметка или как она работает; вскоре вам придется посвятить немало времени на исследование WPF. Графический пользовательский интерфейс приложения состоит из многострочной текстовой области `TextBox` и одной кнопки `Button` (по имени `cmdProcess`). Текстовая область предназначена для ввода данных во время выполнения работы в фоновом режиме, иллюстрируя тем самым неблокирующую природу параллельной задачи.

Дважды щелкнув на файле `MainWindow.xaml.cs` (может потребоваться развернуть узел `MainWindow.xaml`), добавим в его начало представленные ниже операторы `using`:

```

// Обеспечить доступ к перечисленным ниже пространствам имен!
using System.Drawing;
using System.Threading.Tasks;
using System.Threading;
using System.IO;

```

На заметку! Вы должны обновить строку, передаваемую методу `Directory.GetFiles()`, чтобы она указывала конкретный путь к каталогу на вашей машине, который содержит файлы изображений. Для удобства в каталог `Chapter_19` включено несколько примеров изображений (поставляемых в составе операционной системы Windows). В качестве альтернативы можете скопировать каталог `TestPictures` в каталог `bin\Debug` проекта и оставить строку в том виде как есть.

```

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void cmdCancel_Click(object sender, EventArgs e)
    {
        // Код метода будет вскоре обновлен.
    }

    private void cmdProcess_Click(object sender, EventArgs e)
    {
        ProcessFiles();
    }

    private void ProcessFiles()
    {
        // Загрузить все файлы *.jpg и создать новый каталог
        // для модифицированных данных.
        string[] files = Directory.GetFiles(@"..\TestPictures", "*.jpg",
                                           SearchOption.AllDirectories);
        string newDir = @"..\ModifiedPictures";
        Directory.CreateDirectory(newDir);

        // Обработать данные изображений в блокирующей манере.
        foreach (string currentFile in files)
        {
            string filename = Path.GetFileName(currentFile);
            using (Bitmap bitmap = new Bitmap(currentFile))
            {
                bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
                bitmap.Save(Path.Combine(newDir, filename));

                // Вывести идентификатор потока, обрабатывающего текущее изображение.
                this.Title = $"Processing {filename} on thread
{Thread.CurrentThread.ManagedThreadId}";
            }
        }
    }
}

```

Обратите внимание, что метод `ProcessFiles()` выполнит поворот изображения в каждом файле `*.jpg` из заданного каталога, который в текущий момент содержит 10 файлов (при необходимости укажите другой путь в вызове `Directory.GetFiles()`). В настоящее время вся работа происходит в первичном потоке исполняемой программы. Следовательно, после щелчка на кнопке программа выглядит зависшей. Вдобавок заголовок окна также сообщит о том, что файл обрабатывается тем же самым первичным потоком, т.к. в наличии есть только один поток выполнения.

Чтобы обрабатывать файлы на как можно большем количестве процессоров, текущий цикл `foreach` можно заменить вызовом метода `Parallel.ForEach()`. Вспомните, что этот метод имеет множество перегруженных версий. Простейшая форма метода принимает совместимый с `IEnumerable<T>` объект, который содержит элементы, подлежащие обработке (например, строковый массив `files`), и делегат `Action<T>`, указывающий на метод, который будет выполнять необходимую работу.

Ниже показан модифицированный код, где вместо литерального объекта делегата `Action<T>` применяется лямбда-операция C#. Как видите, в коде закомментированы строки, которые отображают идентификатор потока, обрабатывающего текущий файл изображения. Причина объясняется в следующем разделе.

```
// Обработать данные изображений в параллельном режиме!
Parallel.ForEach(files, currentFile =>
{
    string filename = Path.GetFileName(currentFile);
    using (Bitmap bitmap = new Bitmap(currentFile))
    {
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(Path.Combine(newDir, filename));

        // Этот оператор кода теперь приводит к проблеме! См. следующий раздел.
        // this.Title = $"Processing {filename} on thread
(Thread.CurrentThread.ManagedThreadId)"
        // Thread.CurrentThread.ManagedThreadId);
    }
});
```

Доступ к элементам пользовательского интерфейса во вторичных потоках

Вы заметили, что в показанном выше коде закомментированы строки, которые обновляют заголовок главного окна значением идентификатора текущего выполняющегося потока. Как упоминалось ранее, элементы управления графического пользовательского интерфейса привязаны к потоку, в котором они были созданы. Если вторичные потоки пытаются получить доступ к элементу управления, который они напрямую не создавали, то при отладке программного обеспечения возникают ошибки времени выполнения. С другой стороны, если *запустить* приложение (нажатием <Ctrl+F5>), тогда первоначальный код может и не вызвать каких-либо проблем.

На заметку! Не лишним будет повторить: при отладке (по нажатию <F5>) многопоточного приложения IDE-среда Visual Studio часто способна перехватывать ошибки, когда вторичный поток обращается к элементу управления, созданному в первичном потоке. Однако нередко после запуска (с помощью <Ctrl+F5>) приложение может выглядеть функционирующим корректно (или же ошибка может возникнуть довольно скоро). Если не предпринять меры предосторожности (описанные далее), то приложение в подобных обстоятельствах может потенциально сгенерировать ошибку во время выполнения.

Один из подходов, который можно использовать для предоставления вторичным потокам доступа к элементам управления в безопасной к потокам манере, предусматривает применение другого приема — *анонимного делегата*. Родительский класс `Control` в WPF определяет объект `Dispatcher`, который управляет рабочими элементами для потока. Указанный объект имеет метод по имени `Invoke()`, принимающий на входе `System.Delegate`. Этот метод можно вызывать внутри кода, выполняющегося во вторичных потоках, чтобы обеспечить возможность безопасного в отношении потоков обновления пользовательского интерфейса для заданного элемента управления. В то время как весь требуемый код делегата можно было бы написать напрямую, большинство разработчиков используют в качестве простой альтернативы анонимные делегаты. Вот как выглядит модифицированный код:

```

using (Bitmap bitmap = new Bitmap(currentFile))
{
    bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
    bitmap.Save(Path.Combine(newDir, filename));

    // Это больше не работает!
    // this.Title = $"Processing {filename} on thread
(Thread.CurrentThread.ManagedThreadId)";

    // Вызвать Invoke() на объекте Dispatcher, чтобы позволить вторичным потокам
    // получать доступ к элементам управления в безопасной к потокам манере.
    this.Dispatcher.Invoke((Action)delegate
    {
        this.Title = $"Processing {filename} on thread
(Thread.CurrentThread.ManagedThreadId)";
    });
}

```

Теперь после запуска программы библиотека TPL распределит рабочую нагрузку по множеству потоков из пула, используя столько процессоров, сколько возможно. Тем не менее, имена уникальных потоков в заголовке окна отображаться не будут, а при вводе в текстовой области ничего не будет видно до тех пор, пока не обработаются все файлы изображений! Причина в том, что первичный поток пользовательского интерфейса по-прежнему блокируется, ожидая завершения работы всех остальных потоков.

Класс Task

Класс Task позволяет легко вызывать метод во вторичном потоке и может применяться как простая альтернатива работе с асинхронными делегатами. Изменим обработчик события Click элемента управления Button следующим образом:

```

private void cmdProcess_Click(object sender, EventArgs e)
{
    // Запустить новую "задачу" для обработки файлов.
    Task.Factory.StartNew(() => ProcessFiles());
}

```

Свойство Factory класса Task возвращает объект TaskFactory. Методу StartNew() при вызове передается делегат Action<T> (что здесь скрыто с помощью подходящего лямбда-выражения), указывающий на метод, который подлежит вызову в асинхронной манере. После такой небольшой модификации вы обнаружите, что заголовок окна отображает информацию о потоке из пула, обрабатывающем конкретный файл, а текстовое поле может принимать ввод, поскольку пользовательский интерфейс больше не блокируется.

Обработка запроса на отмену

В текущий пример можно внести еще одно улучшение — предоставить пользователю способ для останова обработки данных изображений путем щелчка на второй кнопке Cancel (Отмена). К счастью, методы Parallel.For() и Parallel.ForEach() поддерживают отмену за счет использования *признаков отмены*. При вызове методов на объекте Parallel им можно передавать объект ParallelOptions, который в свою очередь содержит объект CancellationTokenSource.

Первым делом определим в производном от Window классе закрытую переменную-член cancelToken типа CancellationTokenSource:

```
public partial class MainWindow : Window
{
    // Новая переменная уровня Window.
    private CancellationTokenSource cancelToken = new CancellationTokenSource();
    ...
}
```

Обновим обработчик события Click:

```
private void cmdCancel_Click(object sender, EventArgs e)
{
    // Используется для сообщения всем рабочим потокам о необходимости останова!
    cancelToken.Cancel();
}
```

Теперь можно внести необходимые модификации в метод ProcessFiles(). Вот его финальная реализация:

```
private void ProcessFiles()
{
    // Использовать экземпляр ParallelOptions для хранения CancellationToken.
    ParallelOptions parOpts = new ParallelOptions();
    parOpts.CancellationToken = cancelToken.Token;
    parOpts.MaxDegreeOfParallelism = System.Environment.ProcessorCount;
    //Загрузить все файлы *.jpg и создать новый каталог для модифицированных данных.
    string[] files = Directory.GetFiles(@".\TestPictures", "*.jpg",
                                        SearchOption.AllDirectories);
    string newDir = @".\ModifiedPictures";
    Directory.CreateDirectory(newDir);
    try
    {
        // Обработать данные изображения в параллельном режиме!
        Parallel.ForEach(files, parOpts, currentFile =>
        {
            parOpts.CancellationToken.ThrowIfCancellationRequested();
            string filename = Path.GetFileName(currentFile);
            using (Bitmap bitmap = new Bitmap(currentFile))
            {
                bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
                bitmap.Save(Path.Combine(newDir, filename));
                this.Invoke((Action)delegate
                {
                    this.Title = $"Processing {filename} on thread
{Thread.CurrentThread.ManagedThreadId}";
                });
            }
        });
        this.Invoke((Action)delegate
        {
            this.Title = "Done!"; // Готово!
        });
    }
}
```

```

catch (OperationCanceledException ex)
{
    this.Invoke((Action)delegate
    {
        this.Title = ex.Message;
    });
}
}

```

Обратите внимание, что в начале метода конфигурируется объект `ParallelOptions` с установкой его свойства `CancellationToken` для применения признака `CancellationTokenSource`. Кроме того, этот объект `ParallelOptions` передается во втором параметре методу `Parallel.ForEach()`.

Внутри логики цикла осуществляется вызов `ThrowIfCancellationRequested()` на признаке отмены, гарантируя тем самым, что если пользователь щелкнет на кнопке `Cancel`, то все потоки будут остановлены и в качестве уведомления сгенерируется исключение времени выполнения. Перехватив исключение `OperationCanceledException`, можно добавить в текст главного окна сообщение об ошибке.

Исходный код. Проект `DataParallelismWithForEach` доступен в подкаталоге `Chapter_19`.

Обеспечение параллелизма задач с помощью класса `Parallel`

В дополнение к обеспечению параллелизма данных библиотека TPL также может использоваться для запуска любого количества асинхронных задач с помощью метода `Parallel.Invoke()`. Такой подход немного проще, чем применение делегатов или типов из пространства имен `System.Threading`, но если нужна более высокая степень контроля над выполняемыми задачами, тогда следует отказаться от использования `Parallel.Invoke()` и напрямую работать с классом `Task`, как делалось в предыдущем примере.

Чтобы продемонстрировать параллелизм задач в действии, создадим новый проект консольного приложения по имени `MyEBookReader` и импортируем в начале файла `Program.cs` пространство имен `System.Threading`, `System.Threading.Tasks` и `System.Net` (пример является модификацией полезного примера из документации .NET Framework SDK). Здесь мы будем извлекать публично доступную электронную книгу из сайта проекта Гутенберга (www.gutenberg.org) и затем параллельно выполнять набор длительных задач.

Книга загружается в методе `GetBook()`:

```

static void GetBook()
{
    WebClient wc = new WebClient();
    wc.DownloadStringCompleted += (s, eArgs) =>
    {
        theEBook = eArgs.Result;
        Console.WriteLine("Download complete.");
        GetStats();
    };
    // Загрузить электронную книгу Чарльза Диккенса "A Tale of Two Cities".
    // Может понадобиться двукратное выполнение этого кода, если ранее вы
    // не посещали данный сайт, поскольку при первом его посещении появляется
    // окно с сообщением, предотвращающее нормальное выполнение кода.
    wc.DownloadStringAsync(new Uri("http://www.gutenberg.org/files/98/98-8.txt"));
}

```


Класс `WebClient` определен в пространстве имен `System.Net`. Он предоставляет несколько методов для отправки и получения данных от ресурса, идентифицируемого посредством URL. В свою очередь многие из них имеют асинхронные версии, такие как метод `DownloadStringAsync()`, который автоматически порождает новый поток из пула потоков CLR. Когда объект `WebClient` завершает получение данных, он инициирует событие `DownloadStringCompleted`, которое обрабатывается с применением лямбда-выражения C#. Если вызвать синхронную версию этого метода (`DownloadString()`), то сообщение `Downloading book...` не появится до тех пор, пока загрузка не завершится.

Далее реализуем метод `GetStats()` для извлечения индивидуальных слов, содержащихся в переменной `theEBook`, и передачи строкового массива на обработку несколькими вспомогательными методами:

```
static void GetStats()
{
    // Получить слова из электронной книги.
    string[] words = theEBook.Split(new char[]
    { ' ', '\u000A', ',', '.', ';', ':', '-', '?', '/' },
        StringSplitOptions.RemoveEmptyEntries);

    // Найти 10 наиболее часто встречающихся слов.
    string[] tenMostCommon = FindTenMostCommon(words);

    // Получить самое длинное слово.
    string longestWord = FindLongestWord(words);

    // Когда все задачи завершены, построить строку,
    // показывающую всю статистику в окне сообщений.
    StringBuilder bookStats = new StringBuilder("Ten Most Common Words are:\n");
    foreach (string s in tenMostCommon)
    {
        bookStats.AppendLine(s);
    }
    bookStats.AppendFormat("Longest word is: {0}", longestWord); //Самое
                                                                    //длинное слово

    bookStats.AppendLine();
    Console.WriteLine(bookStats.ToString(), "Book info");        // Информация
                                                                    // о книге
}
```

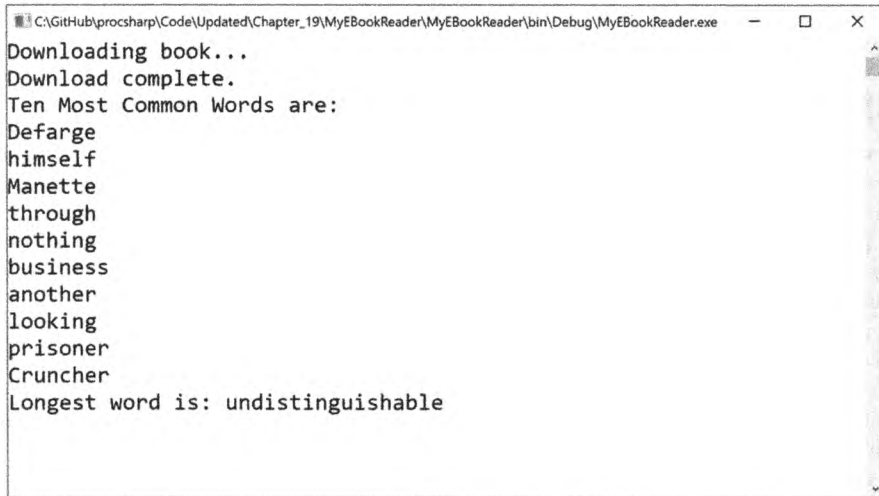
Метод `FindTenMostCommon()` использует запрос LINQ для получения списка объектов `string`, которые наиболее часто встречаются в массиве `string`, а метод `FindLongestWord()` находит самое длинное слово:

```
private string[] FindTenMostCommon(string[] words)
{
    var frequencyOrder = from word in words
                          where word.Length > 6
                          group word by word into g
                          orderby g.Count() descending
                          select g.Key;

    string[] commonWords = (frequencyOrder.Take(10)).ToArray();
    return commonWords;
}

private string FindLongestWord(string[] words)
{
    return (from w in words orderby w.Length descending
            select w).FirstOrDefault();
}
```

После запуска проекта выполнение всех задач может занять внушительный промежуток времени, что зависит от количества процессоров в машине и их тактовой частоты. В конце концов, должен появиться вывод, представленный на рис. 19.3.



```

C:\Github\prochsharp\Code\Updated\Chapter_19\MyEBookReader\MyEBookReader\bin\Debug\MyEBookReader.exe
Downloading book...
Download complete.
Ten Most Common Words are:
Defarge
himself
Manette
through
nothing
business
another
looking
prisoner
Cruncher
Longest word is: undistinguishable

```

Рис. 19.3. Статистика загруженной электронной книги

Помочь удостовериться в том, что приложение задействует все доступные процессоры машины, может параллельный вызов методов `FindTenMostCommon()` и `FindLongestWord()`. Для этого необходимо модифицировать метод `GetStats()` следующим образом:

```

static void GetStats()
{
    // Получить слова из электронной книги.
    string[] words = theEBook.Split(
        new char[] { ' ', '\u000A', ',', '.', '!', ';', ':', '-', '?', '/' },
        StringSplitOptions.RemoveEmptyEntries);
    string[] tenMostCommon = null;
    string longestWord = string.Empty;

    Parallel.Invoke(
        () =>
        {
            // Найти 10 наиболее часто встречающихся слов.
            tenMostCommon = FindTenMostCommon(words);
        },
        () =>
        {
            // Найти самое длинное слово.
            longestWord = FindLongestWord(words);
        });

    // Когда все задачи завершены, построить строку, показывающую всю статистику.
    ...
}

```

Метод `Parallel.Invoke()` ожидает передачи в качестве параметра массива делегатов `Action<>`, который предоставляется косвенно с применением лямбда-выражения.

В то время как вывод идентичен, преимущество заключается в том, что библиотека TPL теперь будет использовать все доступные процессоры машины для вызова каждого метода параллельно, если подобное возможно.

Исходный код. Проект MyEBookReader доступен в подкаталоге Chapter_19.

Запросы Parallel LINQ (PLINQ)

В завершение знакомства с библиотекой TPL следует отметить, что существует еще один способ встраивания параллельных задач в приложения .NET. При желании можно применять набор расширяющих методов, которые позволяют конструировать запрос LINQ, распределяющий свою рабочую нагрузку по параллельным потокам (когда это возможно). Соответственно запросы LINQ, которые спроектированы для параллельного выполнения, называются *запросами Parallel LINQ (PLINQ)*.

Подобно параллельному коду, написанному с использованием класса Parallel, в PLINQ имеется опция игнорирования запроса на обработку коллекции параллельно, если понадобится. Инфраструктура PLINQ оптимизирована во многих отношениях, включая определение того, не будет ли запрос на самом деле более эффективно выполняться в синхронной манере.

Во время выполнения PLINQ анализирует общую структуру запроса, и если есть вероятность, что запрос выиграет от распараллеливания, то он будет выполняться параллельно. Однако если распараллеливание запроса ухудшит производительность, то PLINQ просто запустит запрос последовательно. Когда возникает выбор между потенциально затратным (в плане ресурсов) параллельным алгоритмом и экономным последовательным, предпочтение по умолчанию отдается последовательному алгоритму.

Необходимые расширяющие методы находятся в классе ParallelEnumerable из пространства имен System.Linq. В табл. 19.5 описаны некоторые полезные расширения PLINQ.

Таблица 19.5. Избранные члены класса ParallelEnumerable

Член	Назначение
AsParallel()	Указывает, что остаток запроса должен быть по возможности распараллелен
WithCancellation()	Указывает, что инфраструктура PLINQ должна периодически отслеживать состояние предоставленного признака отмены и при необходимости отменять выполнение
WithDegreeOfParallelism()	Указывает максимальное количество процессоров, которое инфраструктура PLINQ должна задействовать при распараллеливании запроса
ForAll()	Позволяет обрабатывать результаты параллельно без предварительного слияния с потоком потребителя, как происходит при перечислении результата LINQ с применением ключевого слова foreach

Чтобы посмотреть на PLINQ в действии, создадим проект консольного приложения по имени PLINQDataProcessingWithCancellation и импортируем в него пространства имен System.Threading и System.Threading.Tasks (если это еще не сделано). Простая форма подобного рода потребует только двух кнопок с именами btnExecute и btnCancel. После начала обработки запускается новая задача, выполняющая запрос LINQ, кото-

рый просматривает крупный массив целых чисел в поиске элементов, удовлетворяющих условию, что остаток от их деления на 3 дает 0. Вот *непараллельная* версия такого запроса:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Start any key to start processing");
        Console.ReadKey();

        Console.WriteLine("Processing");
        Task.Factory.StartNew(() => ProcessIntData());
        Console.ReadLine();
    }

    static void ProcessIntData()
    {
        // Получить очень большой массив целых чисел.
        int[] source = Enumerable.Range(1, 10_000_000).ToArray();

        // Найти числа, для которых истинно условие num % 3 == 0,
        // и вернуть их в убывающем порядке.
        int[] modThreeIsZero = (from num in source where num % 3 == 0
                                orderby num descending select num).ToArray();

        // Вывести количество найденных чисел.
        Console.WriteLine($"Found { modThreeIsZero.Count() } numbers that match query!");
    }
}
```

Создание запроса PLINQ

Чтобы проинформировать библиотеку TPL о выполнении запроса в параллельном режиме (если такое возможно), необходимо использовать расширяющий метод `AsParallel()`:

```
int[] modThreeIsZero = (from num in source where num % 3 == 0
                        orderby num descending select num).ToArray();
```

Обратите внимание, что общий формат запроса LINQ идентичен тому, что вы видели в предыдущих главах. Тем не менее, за счет включения вызова `AsParallel()` библиотека TPL попытается распределить рабочую нагрузку по доступным процессорам.

Отмена запроса PLINQ

С помощью объекта `CancellationTokenSource` запрос PLINQ можно также информировать о прекращении обработки при определенных условиях (обычно из-за вмешательства пользователя). Объявим на уровне класса `Program` объект `CancellationTokenSource` по имени `cancelToken` и обновим метод `Main()` для принятия ввода от пользователя. Ниже показаны соответствующие изменения в коде:

```
class Program
{
    static CancellationTokenSource cancelToken = new CancellationTokenSource();
    static void Main(string[] args)
    {
```

```

do
{
    Console.WriteLine("Press any key to start processing");
    // Нажмите любую клавишу для начала обработки
    Console.ReadKey();
    Console.WriteLine("Processing");
    Task.Factory.StartNew(() => ProcessIntData());
    Console.Write("Enter Q to quit: ");
    // Введите Q для выхода:
    string answer = Console.ReadLine();
    // Желает ли пользователь выйти?
    if (answer.Equals("Q", StringComparison.OrdinalIgnoreCase))
    {
        cancellationToken.Cancel();
        break;
    }
} while (true);
Console.ReadLine();
}
}

```

Теперь запрос PLINQ необходимо информировать о том, что он должен ожидать входящего запроса на отмену выполнения, добавив в цепочку вызов расширяющего метода `WithCancellation()` с передачей ему признака отмены. Кроме того, этот запрос PLINQ понадобится поместить в подходящий блок `try/catch` и обработать возможные исключения. Финальная версия метода `ProcessIntData()` выглядит следующим образом:

```

static void ProcessIntData()
{
    // Получить очень большой массив целых чисел.
    int[] source = Enumerable.Range(1, 10_000_000).ToArray();
    // Найти числа, для которых истинно условие num % 3 == 0,
    // и вернуть их в убывающем порядке.
    int[] modThreeIsZero = null;
    try
    {
        modThreeIsZero =
            (from num in source.AsParallel().WithCancellation(cancellationToken)
             where num % 3 == 0
             orderby num descending
             select num).ToArray();
        Console.WriteLine();
        // Вывести количество найденных чисел.
        Console.WriteLine($"Found {modThreeIsZero.Count()} numbers that match query!");
    }
    catch (OperationCanceledException ex)
    {
        Console.WriteLine(ex.Message);
    }
}

```

Асинхронные вызовы с помощью ключевого слова `async`

В этой довольно длинной главе было представлено много материала в сжатом виде. Конечно, построение, отладка и понимание сложных многопоточных приложений требует прикладывания усилий в любой инфраструктуре. Хотя TPL, PLINQ и тип делегата могут до некоторой степени упростить решение (особенно по сравнению с другими платформами и языками), разработчики по-прежнему должны хорошо знать детали разнообразных расширенных приемов.

С выходом версии .NET 4.5 в языке программирования C# (и к тому же в VB) появились два новых ключевых слова, которые дополнительно упрощают процесс написания асинхронного кода. По контрасту со всеми примерами, показанными ранее в главе, когда применяются ключевые слова `async` и `await`, компилятор будет самостоятельно генерировать большой объем кода, связанного с потоками, с использованием многочисленных членов из пространств имен `System.Threading` и `System.Threading.Tasks`.

Знакомство с ключевыми словами `async` и `await` языка C#

Ключевое слово `async` языка C# применяется для указания на то, что метод, лямбда-выражение или анонимный метод должен вызываться в асинхронной манере *автоматически*. Да, это правда. Благодаря простой пометке метода модификатором `async` среда CLR будет создавать новый поток выполнения для обработки текущей задачи. Более того, при вызове метода `async` ключевое слово `await` будет *автоматически* приостанавливать текущий поток до тех пор, пока задача не завершится, давая возможность вызывающему потоку продолжать свою работу.

В целях иллюстрации создадим новый проект консольного приложения по имени `FunWithCSharpAsync` и импортируем в файл `Program.cs` пространства имен `System.Threading` и `System.Threading.Tasks`. Добавим метод `DoWork()`, который заставляет вызывающий поток ожидать 5 секунд. Сделаем метод `Main()` асинхронным с возвращаемым типом `Task` (как вскоре будет объяснено). Ниже показан код:

```
class Program
{
    static async Task Main(string[] args)
    {
        Console.WriteLine(" Fun With Async ==>");

        // Для подсказки Visual Studio модернизировать проект до версии C# 7.1.
        List<int> l = default;
        Console.WriteLine(DoWork());
        Console.WriteLine("Completed");
        Console.ReadLine();
    }
    static string DoWork()
    {
        Thread.Sleep(5_000);
        return "Done with work!";
    }
}
```

На заметку! Чтобы все заработало, проект должен быть сконфигурирован для версии C# 7.1. Однако на время написания книги среда Visual Studio не считала такое изменение действительным, и потому придется либо сконфигурировать проект вручную, либо добавить в код еще одно средство версии C# 7.1, чтобы Visual Studio сделала изменение.

Первым делом обратите внимание, что метод `Main()` был снабжен ключевым словом `async`, которое помечает его как член, подлежащий вызову в неблокирующей манере. Вскоре вы узнаете об этом больше.

Вам известно, что после запуска программы придется ожидать 5 секунд, прежде чем сможет произойти что-то еще. В случае графического приложения весь пользовательский интерфейс был бы заблокирован до тех пор, пока работа не завершится.

Если бы мы решили прибегнуть к одному из описанных ранее приемов, чтобы сделать приложение более отзывчивым, тогда пришлось бы немало потрудиться. Тем не менее, начиная с версии .NET 4.5, можно написать следующий код C#:

```
static async Task Main(string[] args)
{
    // Для краткости код не показан.
    string message = await DoWorkAsync();
    Console.WriteLine(message);
    // Для краткости код не показан.
}

static string DoWork()
{
    Thread.Sleep(5_000);
    return "Done with work!";
}

static async Task<string> DoWorkAsync()
{
    return await Task.Run(() =>
    {
        Thread.Sleep(5_000);
        return "Done with work!";
    });
}
```

Обратите внимание на ключевое слово `async` *перед* именем метода, который будет вызываться в неблокирующей манере. Это важно: если метод декорируется ключевым словом `async`, но не имеет хотя бы одного внутреннего вызова метода с использованием `await`, то получится синхронный вызов (на самом деле компилятор выдаст соответствующее предупреждение).

Нам пришлось применить класс `Task` из пространства имен `System.Threading.Tasks`, чтобы превратить метод `DoWork()` в `DoWorkAsync()`. По существу вместо возвращения просто специфического значения (объекта `string` в текущем примере) мы возвращаем объект `Task<T>`, где обобщенный параметр типа `T` представляет собой действительное возвращаемое значение.

Реализация метода `DoWorkAsync()` теперь напрямую возвращает объект `Task<T>`, который является возвращаемым значением `Task.Run()`. Метод `Run()` принимает делегат `Func<>` или `Action<>` и, как вам уже известно, для простоты здесь можно использовать лямбда-выражение. В целом новая версия `DoWorkAsync()` может быть описана следующим образом.

При вызове запускается новая задача, которая заставляет вызывающий поток уснуть на 5 секунд. После завершения вызывающий поток предоставляет строковое возвращаемое значение. Эта строка помещается в новый объект `Task<string>` и возвращается вызывающему коду.

Благодаря новой реализации метода `DoWorkAsync()` мы можем получить некоторое представление о подлинной роли ключевого слова `await`. Оно всегда будет моди-

фицировать метод, который возвращает объект `Task`. Когда поток выполнения достигает `await`, вызывающий поток приостанавливается до тех пор, пока вызов не будет завершен. Запустив эту версию приложения, вы обнаружите, что сообщение `Completed` отображается перед сообщением `Done with work!`. В случае графического приложения можно было бы продолжать работу с пользовательским интерфейсом одновременно с выполнением метода `DoWorkAsync()`.

Соглашения об именовании асинхронных методов

Конечно же, вы заметили, что мы изменили имя метода с `DoWork()` на `DoWorkAsync()`, но по какой причине? Давайте предположим, что новая версия метода по-прежнему называется `DoWork()`, но вызывающий код реализован так:

```
// Отсутствует ключевое слово await!
string message = DoWork();
```

Обратите внимание, что мы действительно поместили метод ключевым словом `async`, но не указали ключевое слово `await` при вызове `DoWork()`. Здесь мы получим ошибки на этапе компиляции, потому что возвращаемым значением `DoWork()` является объект `Task`, который мы пытаемся напрямую присвоить переменной типа `string`. Вспомните, что ключевое слово `await` отвечает за извлечение внутреннего возвращаемого значения, которое содержится в объекте `Task`. Поскольку `await` отсутствует, возникает несоответствие типов.

На заметку! Метод, поддерживающий `await` — это просто метод, который возвращает `Task<T>`.

С учетом того, что методы, которые возвращают объекты `Task`, теперь могут вызываться в неблокирующей манере посредством конструкций `async` и `await`, в Microsoft рекомендуют (в качестве установившейся практики) снабжать имя любого метода, возвращающего `Task`, суффиксом `Async`. В таком случае разработчики, которым известно данное соглашение об именовании, получают визуальное напоминание о том, что ключевое слово `await` является обязательным, если они намерены вызывать метод внутри асинхронного контекста.

На заметку! Обработчики событий для элементов управления графического пользовательского интерфейса (вроде обработчика события `Click` кнопки), к которым применяются ключевые слова `async` и `await`, не следуют указанному соглашению об именовании.

Асинхронные методы, возвращающие `void`

В настоящий момент наш метод `DoWorkAsync()` возвращает объект `Task`, содержащий “реальные данные” для вызывающего кода, которые будут получены прозрачным образом через ключевое слово `await`. Однако что если требуется построить асинхронный метод, возвращающий `void`? В таком случае мы используем необобщенный класс `Task` и опускаем любые операторы `return`:

```
static async Task MethodReturningVoidAsync()
{
    await Task.Run(() => { /* Выполнить какую-то работу... */
        Thread.Sleep(4_000);
    });
    Console.WriteLine("Void method completed");
}
```


Затем в коде, вызывающем метод `MethodReturningVoidAsync()`, будут применяться ключевые слова `await` и `async`:

```
await MethodReturningVoidAsync();
Console.WriteLine("Void method complete");
```

Асинхронные методы с множеством контекстов `await`

Внутри реализации асинхронного метода разрешено иметь множество контекстов `await`. Следующий код является вполне допустимым:

```
static async Task MultiAwaits()
{
    await Task.Run(() => { Thread.Sleep(2_000); });
    Console.WriteLine("Done with first task!");

    await Task.Run(() => { Thread.Sleep(2_000); });
    Console.WriteLine("Done with second task!");

    await Task.Run(() => { Thread.Sleep(2_000); });
    Console.WriteLine("Done with third task!");
}
```

Здесь каждая задача всего лишь приостанавливает текущий поток на некоторый период времени; тем не менее, посредством таких задач может быть представлена любая единица работы (обращение к веб-службе, чтение базы данных или что-нибудь еще).

Вызов асинхронных методов из неасинхронных методов

В каждом из предшествующих примеров ключевое слово `async` использовалось для возвращения в поток вызывающего кода, пока выполняется асинхронный метод. В целом ключевое слово `await` может применяться только в методе, помеченном как `async`. А что если вы не можете (или не хотите) помечать метод с помощью `async`?

К счастью, существуют другие способы вызова асинхронных методов. Если вы просто не используете ключевое слово `await`, тогда код продолжает работу после асинхронного метода, не возвращая управление вызывающему коду. Если вам необходимо действительно ожидать завершения асинхронного метода (что происходит, когда применяется ключевое слово `await`), то можете обратиться к свойству `Result` на методе. Оно является свойством объекта `Task` и ожидает, пока выполнение завершится, и затем возвращает лежащие в основе данные `Task`. Например, вы могли бы вызвать метод `DoWorkAsync()` следующим образом:

```
Console.WriteLine(DoWorkAsync().Result);
```

Чтобы остановить выполнение до тех пор, пока не произойдет возврат из метода с возвращаемым типом `void`, просто вызовите метод `Wait()` на объекте `Task`:

```
MethodReturningVoidAsync().Wait();
```

Ожидание с помощью `await` в блоках `catch` и `finally`

В версии C# 6 появилась возможность помещения вызовов `await` в блоки `catch` и `finally`. Для этого сам метод обязан быть `async`. Указанная возможность демонстрируется в следующем примере кода:

```
static async Task<string> MethodWithTryCatch()
{
    try
    {
        // Выполнить некоторую работу.
```

```
        return "Hello";
    }
    catch (Exception ex)
    {
        await LogTheErrors();
        throw;
    }
    finally
    {
        await DoMagicCleanUp();
    }
}
```

Обобщенные возвращаемые типы в асинхронных методах (нововведение)

До выхода версии C# 7 возвращаемыми типами методов `async` были только `Task`, `Task<T>` и `void`. В версии C# 7 доступны дополнительные возвращаемые типы при условии, что они следуют паттерну с ключевым словом `async`. В качестве конкретного примера можно назвать `ValueTask` из NuGet-пакета `System.Threading.Tasks.Extensions`. Чтобы установить пакет, понадобится открыть консоль диспетчера пакетов (`Package Manager Console`), выбрав в меню `View` (Вид) пункт `Other Windows` (Другие окна), и ввести такую команду:

```
install-package System.Threading.Tasks.Extensions
```

После установки пакета можно создавать код, подобный показанному ниже:

```
static async ValueTask<int> ReturnAnInt()
{
    await Task.Delay(1_000);
    return 5;
}
```

Здесь применяются все те же самые правила: это просто объект `Task` для типов значений вместо принудительного размещения объекта в куче.

Локальные функции (нововведение)

Локальные функции были представлены в главе 4 и использовались в главе 8 с итераторами. Они могут оказаться полезными для асинхронных методов. Чтобы продемонстрировать преимущество, сначала нужно взглянуть на проблему. Добавим новый метод по имени `MethodWithProblems()` со следующим кодом:

```
static async Task MethodWithProblems(int firstParam, int secondParam)
{
    Console.WriteLine("Enter");
    await Task.Run(() =>
    {
        // Вызвать длительно выполняющийся метод.
        Thread.Sleep(4_000);
        Console.WriteLine("First Complete");
        // Вызвать еще один длительно выполняющийся метод, который терпит
        // неудачу из-за того, что значение второго параметра выходит
        // за пределы допустимого диапазона.
        Console.WriteLine("Something bad happened");
    });
}
```

Сценарий заключается в том, что вторая длительно выполняющаяся задача терпит неудачу из-за недопустимых входных данных. Вы можете (и должны) добавить в начало метода проверки, но поскольку весь метод является асинхронным, нет никаких гарантий, что такие проверки выполнятся. Было бы лучше, чтобы проверки происходили непосредственно перед выполнением вызываемого кода. В приведенном далее обновленном коде проверки делаются в синхронной манере, и затем закрытая функция выполняется асинхронным образом.

```
static async Task MethodWithProblemsFixed(int firstParam, int secondParam)
{
    Console.WriteLine("Enter");
    if (secondParam < 0)
    {
        Console.WriteLine("Bad data");
        return;
    }
    actualImplementation();
    async Task actualImplementation()
    {
        await Task.Run(() =>
        {
            // Вызвать длительно выполняющийся метод.
            Thread.Sleep(4_000);
            Console.WriteLine("First Complete");
            // Вызвать еще один длительно выполняющийся метод, который терпит
            // неудачу из-за того, что значение второго параметра выходит
            // за пределы допустимого диапазона.
            Console.WriteLine("Something bad happened");
        });
    }
}
```

Итоговые сведения о ключевых словах `async` и `await`

Настоящий раздел содержал много примеров; ниже перечислены ключевые моменты, которые в нем рассматривались.

- Методы (а также лямбда-выражения или анонимные методы) могут быть помечены ключевым словом `async`, что позволяет им работать в неблокирующей манере.
- Методы (а также лямбда-выражения или анонимные методы), помеченные ключевым словом `async`, будут выполняться синхронно до тех пор, пока не встретится ключевое слово `await`.
- Один метод `async` может иметь множество контекстов `await`.
- Когда встречается выражение `await`, вызывающий поток приостанавливается до тех пор, пока ожидаемая задача не завершится. Тем временем управление возвращается коду, вызвавшему метод.
- Ключевое слово `await` будет скрывать с глаз возвращаемый объект `Task`, что выглядит как прямой возврат лежащего в основе возвращаемого значения. Методы, не имеющие возвращаемого значения, просто возвращают `void`.
- Проверка параметров и другая обработка ошибок должна делаться в главной части метода с переносом фактической порции `async` в закрытую функцию.
- Для переменных, находящихся в стеке, объект `ValueTask` более эффективен, чем объект `Task`, который может стать причиной упаковки и распаковки.

- По соглашению об именовании методы, которые могут вызываться асинхронно, должны быть помечены с помощью суффикса `Async`.

Исходный код. Проект `FunWithCSharpAsync` доступен в подкаталоге `Chapter_19`.

Резюме

Глава начиналась с исследования особенностей конфигурирования типов делегатов `.NET` для выполнения метода в асинхронной манере. Вы видели, что методы `BeginInvoke()` и `EndInvoke()` позволяют косвенно манипулировать вторичным потоком с минимальными усилиями. В ходе обсуждения были представлены интерфейс `IAsyncResult` и класс `AsyncResult`. Вы узнали, что эти типы предлагают разнообразные способы синхронизации вызывающего потока и получения возможных возвращаемых значений методов.

Следующая порция главы была посвящена выяснению роли пространства имен `System.Threading`. Как было показано, когда приложение создает дополнительные потоки выполнения, в результате появляется возможность выполнять множество задач (по внешнему виду) одновременно. Также было продемонстрировано несколько способов защиты чувствительных к потокам блоков кода, чтобы предотвратить повреждение разделяемых ресурсов.

Затем в главе исследовались новые модели для разработки многопоточных приложений, введенные в `.NET 4.0`, в частности `Task Parallel Library` и `PLINQ`. В завершение главы была раскрыта роль ключевых слов `async` и `await`. Вы видели, что эти ключевые слова используются многими типами в библиотеке `TPL`; однако большинство работ по созданию сложного кода для многопоточной обработки и синхронизации компилятор выполняет самостоятельно.

ГЛАВА 20

Файловый ввод-вывод и сериализация объектов

При создании настольных приложений возможность сохранения информации между пользовательскими сеансами является привычным делом. В настоящей главе рассматривается несколько тем, касающихся ввода-вывода, с точки зрения платформы .NET Framework. Первая задача связана с исследованием основных типов, определенных в пространстве имен `System.IO`, с помощью которых можно программно модифицировать структуру каталогов и файлов. Вторая задача предусматривает изучение разнообразных способов чтения и записи символьных, двоичных, строковых и находящихся в памяти структур данных.

После изучения способов манипулирования файлами и каталогами с использованием основных типов ввода-вывода вы ознакомитесь со связанной темой — *сериализацией объектов*. Сериализацию объектов можно применять для сохранения и извлечения состояния объекта с помощью любого типа, производного от `System.IO.Stream`. Возможность сериализации объектов критична, когда объект необходимо копировать на удаленную машину, используя различные технологии удаленного взаимодействия, такие как Windows Communication Foundation (WCF). Однако сериализация довольно полезна сама по себе и с высокой вероятностью пригодится во многих разрабатываемых приложениях .NET (распределенных или нет).

На заметку! Чтобы можно было успешно выполнять примеры в главе, IDE-среда Visual Studio должна быть запущена с правами администратора (для этого нужно просто щелкнуть правой кнопкой мыши на значке Visual Studio и выбрать в контекстном меню пункт Запуск от имени администратора). В противном случае при доступе к файловой системе компьютера могут возникать исключения, связанные с безопасностью.

Исследование пространства имен `System.IO`

В рамках платформы .NET пространство имен `System.IO` представляет собой раздел библиотек базовых классов, выделенный службам файлового ввода и вывода, а также ввода и вывода в памяти. Подобно любому пространству имен внутри `System.IO` определен набор классов, интерфейсов, перечислений, структур и делегатов, большинство из которых находятся в сборке `mscorlib.dll`. В дополнение к типам, содержащимся внутри `mscorlib.dll`, в сборке `System.dll` определены дополнительные члены пространства имен `System.IO`. Обратите внимание, что во всех проектах Visual Studio автоматически устанавливаются ссылки на обе сборки.

Многие типы из пространства имен `System.IO` сосредоточены на программной манипуляции физическими каталогами и файлами. Тем не менее, дополнительные типы предоставляют поддержку чтения и записи данных в строковые буферы, а также в области памяти. В табл. 20.1 кратко описаны основные (неабстрактные) классы, которые дают понятие о функциональности, доступной в пространстве имен `System.IO`.

Таблица 20.1. Основные члены пространства имен `System.IO`

Неабстрактные классы ввода-вывода	Описание
<code>BinaryReader</code> <code>BinaryWriter</code>	Эти классы позволяют сохранять и извлекать данные элементарных типов (целочисленные, булевские, строковые и т.д.) как двоичные значения
<code>BufferedStream</code>	Этот класс предоставляет временное хранилище для потока байтов, который может быть зафиксирован в постоянном хранилище в более позднее время
<code>Directory</code> <code>DirectoryInfo</code>	Эти классы применяются для манипулирования структурой каталогов машины. Тип <code>Directory</code> открывает функциональность с использованием <i>статических членов</i> . Тип <code>DirectoryInfo</code> обеспечивает аналогичную функциональность через действительную <i>объектную ссылку</i>
<code>DriveInfo</code>	Этот класс предоставляет детальную информацию о дисковых устройствах, присутствующих на заданной машине
<code>File</code> <code>FileInfo</code>	Эти классы служат для манипулирования набором файлов на машине. Тип <code>File</code> открывает функциональность через <i>статические члены</i> . Тип <code>FileInfo</code> обеспечивает аналогичную функциональность через действительную <i>объектную ссылку</i>
<code>FileStream</code>	Этот класс предоставляет произвольный доступ к файлу (например, с возможностями поиска) с данными, представленными в виде потока байтов
<code>FileSystemWatcher</code>	Этот класс позволяет отслеживать модификацию внешних файлов в указанном каталоге
<code>MemoryStream</code>	Этот класс обеспечивает произвольный доступ к данным, хранящимся в памяти, а не в физическом файле
<code>Path</code>	Этот класс выполняет операции над типами <code>System.String</code> , которые содержат информацию о пути к файлу или каталогу, в независимой от платформы манере
<code>StreamWriter</code> <code>StreamReader</code>	Эти классы применяются для хранения (и извлечения) текстовой информации в файле. Они не поддерживают произвольный доступ к файлу
<code>StringWriter</code> <code>StringReader</code>	Подобно <code>StreamWriter/StreamReader</code> эти классы также работают с текстовой информацией. Однако лежащим в основе хранилищем является строковый буфер, а не физический файл

В дополнение к описанным конкретным классам в `System.IO` определено несколько перечислений, а также набор абстрактных классов (скажем, `Stream`, `TextReader` и `TextWriter`), которые формируют разделяемый полиморфный интерфейс для всех наследников. В главе вы узнаете о многих типах пространства имен `System.IO`.

Классы Directory (DirectoryInfo) и File (FileInfo)

Пространство имен `System.IO` предлагает четыре класса, которые позволяют манипулировать индивидуальными файлами, а также взаимодействовать со структурой каталогов машины. Первые два класса, `Directory` и `File`, открывают доступ к операциям создания, удаления, копирования и перемещения через разнообразные статические члены. Тесно связанные с ними классы `FileInfo` и `DirectoryInfo` обеспечивают похожую функциональность в виде методов уровня экземпляра (следовательно, придется создавать их экземпляры с помощью ключевого слова `new`). На рис. 20.1 видно, что `Directory` и `File` расширяют непосредственно класс `System.Object`, в то время как `DirectoryInfo` и `FileInfo` являются производными от абстрактного класса `FileSystemInfo`.

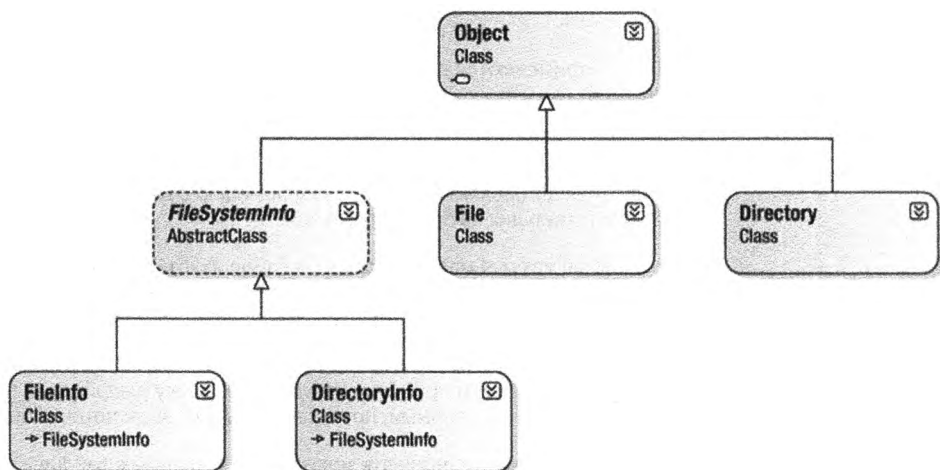


Рис. 20.1. Классы для работы с файлами и каталогами

Обычно классы `FileInfo` и `DirectoryInfo` считаются лучшим выбором для получения полных сведений о файле или каталоге (например, время создания или возможности чтения/записи), т.к. их члены возвращают строго типизированные объекты. В отличие от них члены классов `Directory` и `File`, как правило, возвращают простые строковые значения, а не строго типизированные объекты. Тем не менее, это всего лишь рекомендация; во многих случаях одну и ту же работу можно делать с использованием `File/FileInfo` или `Directory/DirectoryInfo`.

Абстрактный базовый класс FileSystemInfo

Классы `DirectoryInfo` и `FileInfo` получают многие линии поведения от абстрактного базового класса `FileSystemInfo`. По большей части члены класса `FileSystemInfo` применяются для выяснения общих характеристик (таких как время создания, разнообразные атрибуты и т.д.) заданного файла или каталога. В табл. 20.2 перечислены некоторые основные свойства, представляющие интерес.

В классе `FileSystemInfo` также определен метод `Delete()`. Он реализуется производными типами для удаления заданного файла или каталога с жесткого диска. Кроме того, перед получением информации об атрибутах можно вызвать метод `Refresh()`, чтобы обеспечить актуальность статистики о текущем файле или каталоге.

Таблица 20.2. Избранные свойства класса `FileSystemInfo`

Свойство	Описание
<code>Attributes</code>	Получает или устанавливает ассоциированные с текущим файлом атрибуты, которые представлены перечислением <code>FileAttributes</code> (например, доступный только для чтения, зашифрованный, скрытый или сжатый)
<code>CreationTime</code>	Получает или устанавливает время создания текущего файла или каталога
<code>Exists</code>	Определяет, существует ли данный файл или каталог
<code>Extension</code>	Извлекает расширение файла
<code>FullName</code>	Получает полный путь к файлу или каталогу
<code>LastAccessTime</code>	Получает или устанавливает время последнего доступа к текущему файлу или каталогу
<code>LastWriteTime</code>	Получает или устанавливает время последней записи в текущий файл или каталог
<code>Name</code>	Получает имя текущего файла или каталога

Работа с типом `DirectoryInfo`

Первый неабстрактный тип, связанный с вводом-выводом, который мы исследуем здесь — `DirectoryInfo`. Этот класс содержит набор членов, используемых для создания, перемещения, удаления и перечисления каталогов и подкаталогов. В дополнение к функциональности, предоставленной его базовым классом (`FileSystemInfo`), класс `DirectoryInfo` предлагает ключевые члены, описанные в табл. 20.3.

Таблица 20.3. Основные члены типа `DirectoryInfo`

Член	Описание
<code>Create()</code>	Создает каталог (или набор подкаталогов) по заданному путевому имени
<code>CreateSubdirectory()</code>	
<code>Delete()</code>	Удаляет каталог и все его содержимое
<code>GetDirectories()</code>	Возвращает массив объектов <code>DirectoryInfo</code> , которые представляют все подкаталоги в текущем каталоге
<code>GetFiles()</code>	Извлекает массив объектов <code>FileInfo</code> , представляющий набор файлов в заданном каталоге
<code>MoveTo()</code>	Перемещает каталог со всем содержимым в новый путь
<code>Parent</code>	Извлекает родительский каталог данного каталога
<code>Root</code>	Получает корневую часть пути

Работа с типом `DirectoryInfo` начинается с указания отдельного пути в параметре конструктора. Если требуется получить доступ к текущему рабочему каталогу (каталогу выполняющегося приложения), то следует применять обозначение в виде точки (`.`). Вот некоторые примеры:

```
// Привязаться к текущему рабочему каталогу.
DirectoryInfo dir1 = new DirectoryInfo(".");
// Привязаться к C:\Windows, используя дословную строку.
DirectoryInfo dir2 = new DirectoryInfo(@"C:\Windows");
```


Во втором примере предполагается, что путь, передаваемый конструктору (C:\Windows), уже существует на физической машине. Однако при попытке взаимодействия с несуществующим каталогом генерируется исключение `System.IO.DirectoryNotFoundException`. Таким образом, чтобы указать каталог, который пока еще не создан, перед работой с ним понадобится вызвать метод `Create()`:

```
// Привязаться к несуществующему каталогу, затем создать его.
DirectoryInfo dir3 = new DirectoryInfo(@"C:\MyCode\Testing");
dir3.Create();
```

После создания объекта `DirectoryInfo` можно исследовать содержимое лежащего в основе каталога с помощью любого свойства, унаследованного от `FileSystemInfo`. В целях иллюстрации создадим новый проект консольного приложения по имени `DirectoryApp` и импортируем в файл кода C# пространство имен `System.IO`. Далее модифицируем класс `Program`, добавив показанный ниже новый статический метод, который создает объект `DirectoryInfo`, отображенный на C:\Windows (при необходимости подкорректируйте путь), и выводит интересные статистические данные:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Directory(Info) *****\n");
        ShowWindowsDirectoryInfo();
        Console.ReadLine();
    }

    static void ShowWindowsDirectoryInfo()
    {
        // Вывести информацию о каталоге.
        DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");
        Console.WriteLine("***** Directory Info *****");
        Console.WriteLine("FullName: {0}", dir.FullName); // полное имя
        Console.WriteLine("Name: {0}", dir.Name);          // имя каталога
        Console.WriteLine("Parent: {0}", dir.Parent);      // родительский каталог
        Console.WriteLine("Creation: {0}", dir.CreationTime); // время создания
        Console.WriteLine("Attributes: {0}", dir.Attributes); // атрибуты
        Console.WriteLine("Root: {0}", dir.Root);          // корневой каталог
        Console.WriteLine("*****\n");
    }
}
```

Вывод у вас может быть другим, но похожим:

```
***** Fun with Directory(Info) *****
***** Directory Info *****
FullName: C:\Windows
Name: Windows
Parent:
Creation: 10/10/2015 10:22:32 PM
Attributes: Directory
Root: C:\
*****
```

Перечисление файлов с помощью типа `DirectoryInfo`

В дополнение к получению базовых сведений о существующем каталоге текущий пример можно расширить, чтобы задействовать некоторые методы типа `DirectoryInfo`.

Первым делом мы используем метод `GetFiles()` для получения информации обо всех файлах `*.jpg`, расположенных в каталоге `C:\Windows\Web\Wallpaper`.

На заметку! Если на вашей машине нет каталога `C:\Windows\Web\Wallpaper`, тогда скорректируйте код так, чтобы в нем читались файлы из какого-то существующего каталога (например, все файлы `*.bmp` из каталога `C:\Windows`).

Метод `GetFiles()` возвращает массив объектов `FileInfo`, каждый из которых открывает доступ к детальной информации о конкретном файле (тип `FileInfo` будет подробно описан далее в главе). Предположим, что в методе `Main()` вызывается следующий статический метод класса `Program`:

```
static void DisplayImageFiles()
{
    DirectoryInfo dir = new DirectoryInfo(@"C:\Windows\Web\Wallpaper");
    // Получить все файлы с расширением *.jpg.
    FileInfo[] imageFiles = dir.GetFiles("*.jpg", SearchOption.AllDirectories);
    // Сколько файлов найдено?
    Console.WriteLine("Found {0} *.jpg files\n", imageFiles.Length);
    // Вывести информацию о каждом файле.
    foreach (FileInfo f in imageFiles)
    {
        Console.WriteLine("*****");
        Console.WriteLine("File name: {0}", f.Name);           // имя файла
        Console.WriteLine("File size: {0}", f.Length);         // размер
        Console.WriteLine("Creation: {0}", f.CreationTime);     // время создания
        Console.WriteLine("Attributes: {0}", f.Attributes);     // атрибуты
        Console.WriteLine("*****\n");
    }
}
```

Обратите внимание на указание в вызове `GetFiles()` варианта поиска: `SearchOption.AllDirectories` обеспечивает просмотр всех подкаталогов корня. В результате запуска приложения выводится список файлов, которые соответствуют поисковому шаблону.

Создание подкаталогов с помощью типа `DirectoryInfo`

Посредством метода `DirectoryInfo.CreateSubdirectory()` можно программно расширять структуру каталогов. Он позволяет создавать одиночный подкаталог, а также множество вложенных подкаталогов в единственном вызове. В приведенном далее методе демонстрируется расширение структуры диска `C:` несколькими специальными подкаталогами:

```
static void ModifyAppDirectory()
{
    DirectoryInfo dir = new DirectoryInfo(@"C:\");
    // Создать \MyFolder в каталоге приложения.
    dir.CreateSubdirectory("MyFolder");
    // Создать \MyFolder2\Data в каталоге приложения.
    dir.CreateSubdirectory(@"MyFolder2\Data");
}
```

Получать возвращаемое значение метода `CreateSubdirectory()` не обязательно, но важно знать, что в случае его успешного выполнения возвращается объект `DirectoryInfo`, представляющий вновь созданный элемент. Взгляните на следующую модификацию предыдущего метода. Обратите внимание на указание строки "." в конструкторе `DirectoryInfo`, что дает доступ к месту установки приложения.

```
static void ModifyAppDirectory()
{
    DirectoryInfo dir = new DirectoryInfo(".");
    // Создать \MyFolder в начальном каталоге.
    dir.CreateSubdirectory("MyFolder");
    // Получить возвращенный объект DirectoryInfo.
    DirectoryInfo myDataFolder = dir.CreateSubdirectory(@"MyFolder2\Data");
    // Выводит путь к ..\MyFolder2\Data.
    Console.WriteLine("New Folder is: {0}", myDataFolder);
}
```

Вызвав метод `ModifyAppDirectory()` в `Main()` и запустив программу, в проводнике Windows можно будет увидеть новые подкаталоги.

Работа с типом Directory

Вы видели тип `DirectoryInfo` в действии и теперь готовы к изучению типа `Directory`. По большей части статические члены типа `Directory` воспроизводят функциональность, которая предоставляется членами уровня экземпляра, определенными в `DirectoryInfo`. Тем не менее, вспомните, что члены типа `Directory` обычно возвращают строковые данные, а не строго типизированные объекты `FileInfo`/`DirectoryInfo`.

Давайте взглянем на функциональность типа `Directory`; показанный ниже вспомогательный метод отображает имена всех логических устройств на текущем компьютере (с помощью метода `Directory.GetLogicalDrives()`) и применяет статический метод `Directory.Delete()` для удаления созданных ранее подкаталогов `\MyFolder` и `\MyFolder2\Data`:

```
static void FunWithDirectoryType()
{
    // Вывести список всех логических устройств на текущем компьютере.
    string[] drives = Directory.GetLogicalDrives();
    Console.WriteLine("Here are your drives:");
    foreach (string s in drives)
        Console.WriteLine("--> {0} ", s);
    // Удалить ранее созданные подкаталоги.
    Console.WriteLine("Press Enter to delete directories");
    Console.ReadLine();
    try
    {
        Directory.Delete(@"C:\MyFolder");
        // Второй параметр указывает, нужно ли удалять внутренние подкаталоги.
        Directory.Delete(@"C:\MyFolder2", true);
    }
    catch (IOException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Исходный код. Проект DirectoryApp доступен в подкаталоге Chapter_20.

Работа с типом DriveInfo

Пространство имен System.IO содержит класс по имени DriveInfo. Подобно Directory.GetLogicalDrives() статический метод DriveInfo.GetDrives() позволяет выяснить имена устройств на машине. Однако в отличие от Directory.GetLogicalDrives() метод DriveInfo.GetDrives() предоставляет множество дополнительных деталей (например, тип устройства, доступное свободное пространство и метка тома). Рассмотрим следующий класс Program, определенный в новом проекте консольного приложения DriveInfoApp (не забудьте импортировать пространство имен System.IO):

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with DriveInfo *****\n");
        // Получить информацию обо всех устройствах.
        DriveInfo[] myDrives = DriveInfo.GetDrives();
        // Вывести сведения об устройствах.
        foreach(DriveInfo d in myDrives)
        {
            Console.WriteLine("Name: {0}", d.Name);           // имя
            Console.WriteLine("Type: {0}", d.DriveType);       // тип
            // Проверить, смонтировано ли устройство.
            if(d.IsReady)
            {
                Console.WriteLine("Free space: {0}", d.TotalFreeSpace);
                // свободное пространство
                Console.WriteLine("Format: {0}", d.DriveFormat); // формат устройства
                Console.WriteLine("Label: {0}", d.VolumeLabel); // метка тома
            }
            Console.WriteLine();
        }
        Console.ReadLine();
    }
}
```

Вот возможный вывод:

```
***** Fun with DriveInfo *****

Name: C:\
Type: Fixed
Free space: 791699763200
Format: NTFS
Label: Windows10_OS

Name: D:\
Type: Fixed
Free space: 23804067840
Format: NTFS
Label: LENOVO

Press any key to continue . . .
```

К этому моменту вы изучили несколько основных линий поведения классов `Directory`, `DirectoryInfo` и `DriveInfo`. Далее вы ознакомитесь с тем, как создавать, открывать, закрывать и удалять файлы, находящиеся в заданном каталоге.

Исходный код. Проект `DriveInfoApp` доступен в подкаталоге `Chapter_20`.

Работа с классом `FileInfo`

Как было показано в предыдущем примере `DirectoryApp`, класс `FileInfo` позволяет получать сведения о существующих файлах на жестком диске (такие как время создания, размер и атрибуты) и помогает создавать, копировать, перемещать и удалять файлы. В дополнение к набору функциональности, унаследованной от `FileSystemInfo`, класс `FileInfo` имеет ряд уникальных членов, которые описаны в табл. 20.4.

Таблица 20.4. Основные члены `FileInfo`

Член	Описание
<code>AppendText()</code>	Создает объект <code>StreamWriter</code> (описанный далее в главе) и добавляет текст в файл
<code>CopyTo()</code>	Копирует существующий файл в новый файл
<code>Create()</code>	Создает новый файл и возвращает объект <code>FileStream</code> (описанный далее в главе) для взаимодействия с вновь созданным файлом
<code>CreateText()</code>	Создает объект <code>StreamWriter</code> , который производит запись в новый текстовый файл
<code>Delete()</code>	Удаляет файл, к которому привязан экземпляр <code>FileInfo</code>
<code>Directory</code>	Получает экземпляр родительского каталога
<code>DirectoryName</code>	Получает полный путь к родительскому каталогу
<code>Length</code>	Получает размер текущего файла
<code>MoveTo()</code>	Перемещает указанный файл в новое местоположение, предоставляя возможность указания нового имени для файла
<code>Name</code>	Получает имя файла
<code>Open()</code>	Открывает файл с разнообразными привилегиями чтения/записи и совместного доступа
<code>OpenRead()</code>	Создает объект <code>FileStream</code> , доступный только для чтения
<code>OpenText()</code>	Создает объект <code>StreamReader</code> (описанный далее в главе), который производит чтение из существующего текстового файла
<code>OpenWrite()</code>	Создает объект <code>FileStream</code> , доступный только для записи

Обратите внимание, что большинство методов класса `FileInfo` возвращают специфический объект ввода-вывода (например, `FileStream` и `StreamWriter`), который позволяет начать чтение и запись данных в ассоциированный файл во множестве форматов. Вскоре мы исследуем указанные типы, но прежде чем рассмотреть работающий пример, давайте изучим различные способы получения дескриптора файла с использованием класса `FileInfo`.

Метод `FileInfo.Create()`

Один из способов создания дескриптора файла предусматривает применение метода `FileInfo.Create()`:

```
static void Main(string[] args)
{
    // Создать новый файл на диске C:.
    FileInfo f = new FileInfo(@"C:\Test.dat");
    FileStream fs = f.Create();

    // Использовать объект FileStream...

    // Закрыть файловый поток.
    fs.Close();
}
```

Метод `FileInfo.Create()` возвращает тип `FileStream`, который предоставляет синхронную и асинхронную операции записи/чтения лежащего в его основе файла. Имейте в виду, что объект `FileStream`, возвращаемый `FileInfo.Create()`, открывает полный доступ по чтению и записи всем пользователям.

Также обратите внимание, что после окончания работы с текущим объектом `FileStream` необходимо обеспечить закрытие его дескриптора для освобождения внутренних неуправляемых ресурсов потока. Учитывая, что `FileStream` реализует интерфейс `IDisposable`, можно использовать блок `using` и позволить компилятору сгенерировать логику завершения (подробности ищите в главе 8):

```
static void Main(string[] args)
{
    // Определение блока using для типов файлового ввода-вывода.
    FileInfo f = new FileInfo(@"C:\Test.dat");
    using (FileStream fs = f.Create())
    {
        // Использовать объект FileStream...
    }
}
```

Метод `FileInfo.Open()`

С помощью метода `FileInfo.Open()` можно открывать существующие файлы, а также создавать новые файлы с более высокой точностью представления, чем обеспечивает метод `FileInfo.Create()`, поскольку `Open()` обычно принимает несколько параметров для описания общей структуры файла, с которым будет производиться работа. В результате вызова `Open()` возвращается объект `FileStream`. Взгляните на следующий код:

```
static void Main(string[] args)
{
    // Создать новый файл посредством FileInfo.Open().
    FileInfo f2 = new FileInfo(@"C:\Test2.dat");
    using (FileStream fs2 = f2.Open(FileMode.OpenOrCreate,
        FileAccess.ReadWrite, FileShare.None))
    {
        // Использовать объект FileStream...
    }
}
```

Эта версия перегруженного метода `Open()` требует передачи трех параметров. Первый параметр указывает общий тип запроса ввода-вывода (например, создать но-

вый файл, открыть существующий файл или дописать в файл), который представлен в виде перечисления `FileMode` (описание его членов приведено в табл. 20.5):

```
public enum FileMode
{
    CreateNew,
    Create,
    Open,
    OpenOrCreate,
    Truncate,
    Append
}
```

Таблица 20.5. Члены перечисления `FileMode`

Член	Описание
CreateNew	Информирует операционную систему о необходимости создания нового файла. Если файл уже существует, то генерируется исключение <code>IOException</code>
Create	Информирует операционную систему о необходимости создания нового файла. Если файл уже существует, тогда он будет перезаписан
Open	Открывает существующий файл. Если файл не существует, то генерируется исключение <code>FileNotFoundException</code>
OpenOrCreate	Открывает файл, если он существует; в противном случае создает новый
Truncate	Открывает файл и усекает его до нулевой длины
Append	Открывает файл, переходит в его конец и начинает операции записи (этот флаг может применяться лишь с потоками только для записи). Если файл не существует, тогда создается новый файл

Второй параметр метода `Open()` — значение перечисления `FileAccess` — служит для определения поведения чтения/записи лежащего в основе потока:

```
public enum FileAccess
{
    Read,
    Write,
    ReadWrite
}
```

Наконец, третий параметр метода `Open()` — значение перечисления `FileShare` — указывает, каким образом файл может совместно использоваться другими файловыми дескрипторами:

```
public enum FileShare
{
    Delete,
    Inheritable,
    None,
    Read,
    ReadWrite,
    Write
}
```

Методы `FileInfo.OpenRead()` и `FileInfo.OpenWrite()`

Метод `FileOpen.Open()` позволяет получить дескриптор файла в гибкой манере, но класс `FileInfo` также предлагает методы `OpenRead()` и `OpenWrite()`. Как и можно было ожидать, указанные методы возвращают подходящим образом сконфигурированный только для чтения или только для записи объект `FileStream` без необходимости в предоставлении значений разных перечислений. Подобно `FileInfo.Create()` и `FileInfo.Open()` методы `OpenRead()` и `OpenWrite()` возвращают объект `FileStream` (обратите внимание, что в следующем коде предполагается наличие на диске C: файлов `Test3.dat` и `Test4.dat`):

```
static void Main(string[] args)
{
    // Получить объект FileStream с правами только для чтения.
    FileInfo f3 = new FileInfo(@"C:\Test3.dat");
    using(FileStream readOnlyStream = f3.OpenRead())
    {
        // Использовать объект FileStream...
    }

    // Теперь получить объект FileStream с правами только для записи.
    FileInfo f4 = new FileInfo(@"C:\Test4.dat");
    using(FileStream writeOnlyStream = f4.OpenWrite())
    {
        // Использовать объект FileStream...
    }
}
```

Метод `FileInfo.OpenText()`

Еще одним членом типа `FileInfo`, связанным с открытием файлов, является `OpenText()`. В отличие от `Create()`, `Open()`, `OpenRead()` и `OpenWrite()` метод `OpenText()` возвращает экземпляр типа `StreamReader`, а не `FileStream`. Исходя из того, что на диске C: имеется файл по имени `boot.ini`, вот как получить доступ к его содержимому:

```
static void Main(string[] args)
{
    // Получить объект StreamReader.
    FileInfo f5 = new FileInfo(@"C:\boot.ini");
    using(StreamReader sreader = f5.OpenText())
    {
        // Использовать объект StreamReader...
    }
}
```

Вскоре вы увидите, что тип `StreamReader` предоставляет способ чтения символьных данных из лежащего в основе файла.

Методы `FileInfo.CreateText()` и `FileInfo.AppendText()`

Последними двумя методами, представляющими интерес в данный момент, являются `CreateText()` и `AppendText()`. Оба они возвращают объект `StreamWriter`:


```
static void Main(string[] args)
{
    FileInfo f6 = new FileInfo(@"C:\Test6.txt");
    using(StreamWriter swriter = f6.CreateText())
    {
        // Использовать объект StreamWriter...
    }

    FileInfo f7 = new FileInfo(@"C:\FinalTest.txt");
    using(StreamWriter swriterAppend = f7.AppendText())
    {
        // Использовать объект StreamWriter...
    }
}
```

Как и можно было ожидать, тип `StreamWriter` предлагает способ записи данных в связанный с ним файл.

Работа с типом `File`

В типе `File` определено несколько статических методов для предоставления функциональности, почти идентичной той, которая доступна в типе `FileInfo`. Подобно `FileInfo` тип `File` поддерживает методы `AppendText()`, `Create()`, `CreateText()`, `Open()`, `OpenRead()`, `OpenWrite()` и `OpenText()`. Во многих случаях типы `File` и `FileInfo` могут применяться взаимозаменяемо. Чтобы увидеть тип `File` в действии, упростим каждый из приведенных ранее примеров использования типа `FileStream`, применив вместо него `File`:

```
static void Main(string[] args)
{
    // Получить объект FileStream через File.Create().
    using(FileStream fs = File.Create(@"C:\Test.dat"))
    {}

    // Получить объект FileStream посредством File.Open().
    using(FileStream fs2 = File.Open(@"C:\Test2.dat",
        FileMode.OpenOrCreate,
        FileAccess.ReadWrite, FileShare.None))
    {}

    // Получить объект FileStream с правами только для чтения.
    using(FileStream readOnlyStream = File.OpenRead(@"Test3.dat"))
    {}

    // Получить объект FileStream с правами только для записи.
    using(FileStream writeOnlyStream = File.OpenWrite(@"Test4.dat"))
    {}

    // Получить объект StreamReader.
    using(StreamReader reader = File.OpenText(@"C:\boot.ini"))
    {}

    // Получить несколько объектов StreamWriter.
    using(StreamWriter swriter = File.CreateText(@"C:\Test6.txt"))
    {}

    using(StreamWriter swriterAppend = File.AppendText(@"C:\FinalTest.txt"))
    {}
}
```

Дополнительные члены File

Тип `File` также поддерживает несколько членов, описанных в табл. 20.6, которые могут значительно упростить процессы чтения и записи текстовых данных.

Таблица 20.6. Методы типа `File`

Метод	Описание
<code>ReadAllBytes()</code>	Открывает указанный файл, возвращает двоичные данные в виде массива байтов и закрывает файл
<code>ReadAllLines()</code>	Открывает указанный файл, возвращает символьные данные в виде массива строк и закрывает файл
<code>ReadAllText()</code>	Открывает указанный файл, возвращает символьные данные в виде объекта <code>System.String</code> и закрывает файл
<code>WriteAllBytes()</code>	Открывает указанный файл, записывает в него массив байтов и закрывает файл
<code>WriteAllLines()</code>	Открывает указанный файл, записывает в него массив строк и закрывает файл
<code>WriteAllText()</code>	Открывает указанный файл, записывает в него символьные данные из заданной строки и закрывает файл

Приведенные в табл. 20.6 методы типа `File` можно использовать для реализации чтения и записи пакетов данных посредством всего нескольких строк кода. Еще лучше то, что эти методы автоматически закрывают лежащий в основе файловый дескриптор. Например, следующий проект консольного приложения (по имени `SimpleFileIO`) сохраняет строковые данные в новом файле на диске C: (и читает их в память) с минимальными усилиями (здесь предполагается, что было импортировано пространство имен `System.IO`):

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Simple I/O with the File Type *****\n");
        string[] myTasks = {
            "Fix bathroom sink", "Call Dave",
            "Call Mom and Dad", "Play Xbox One"};

        // Записать все данные в файл на диске C:.
        File.WriteAllLines(@"tasks.txt", myTasks);

        // Прочитать все данные и вывести на консоль.
        foreach (string task in File.ReadAllLines(@"tasks.txt"))
        {
            Console.WriteLine("TODO: {0}", task);
        }
        Console.ReadLine();
    }
}
```

Из продемонстрированного примера можно сделать вывод: когда необходимо быстро получить файловый дескриптор, тип `File` позволит сэкономить на объеме кодирования. Тем не менее, преимущество предварительного создания объекта `FileInfo` заключается в возможности сбора сведений о файле с применением членов абстрактного базового класса `FileSystemInfo`.

Исходный код. Проект SimpleFileIO доступен в подкаталоге Chapter_20.

Абстрактный класс Stream

Вы уже видели много способов получения объектов FileStream, StreamReader и StreamWriter, но нужно еще читать данные или записывать их в файл, используя упомянутые типы. Чтобы понять, как это делается, необходимо освоить концепцию потока. В мире манипуляций вводом-выводом *поток* (stream) представляет порцию данных, протекающую между источником и приемником. Потоки предоставляют общий способ взаимодействия с *последовательностью байтов* независимо от того, какого рода устройство (скажем, файл, сетевое подключение или принтер) хранит или отображает байты.

Абстрактный класс System.IO.Stream определяет набор членов, которые обеспечивают поддержку синхронного и асинхронного взаимодействия с хранилищем (например, файлом или областью памяти).

На заметку! Концепция потока не ограничена файловым вводом-выводом. Конечно, библиотеки .NET предоставляют потоковый доступ к сетям, областям памяти и прочим абстракциям, связанным с потоками.

Потомки класса Stream представляют данные в виде низкоуровневых потоков байтов; следовательно, работа непосредственно с низкоуровневыми потоками может оказаться не особенно понятной. Некоторые типы, производные от Stream, поддерживают *позиционирование*, которое означает процесс получения и корректировки текущей позиции в потоке. В табл. 20.7 приведено описание основных членов класса Stream, что помогает понять его функциональность.

Таблица 20.7. Члены абстрактного класса Stream

Член	Описание
CanRead CanWrite CanSeek	Определяют, поддерживает ли текущий поток чтение, поиск и/или запись
Close()	Закрывает текущий поток и освобождает все ресурсы (такие как сокеты и файловые дескрипторы), ассоциированные с текущим потоком. Внутренне этот метод является псевдонимом Dispose(), поэтому <i>закрытие потока</i> функционально эквивалентно <i>освобождению потока</i>
Flush()	Обновляет лежащий в основе источник данных или хранилище текущим состоянием буфера и затем очищает буфер. Если поток не реализует буфер, то метод ничего не делает
Length	Возвращает длину потока в байтах
Position	Определяет текущую позицию в потоке
Read() ReadByte() ReadAsync()	Читают последовательность байтов (или одиночный байт) из текущего потока и перемещают текущую позицию потока вперед на количество прочитанных байтов
Seek()	Устанавливает позицию в текущем потоке
SetLength()	Устанавливает длину текущего потока
Write() WriteByte() WriteAsync()	Записывают последовательность байтов (или одиночный байт) в текущий поток и перемещают текущую позицию вперед на количество записанных байтов

Работа с классом `FileStream`

Класс `FileStream` предоставляет реализацию абстрактных членов `Stream` в манере, подходящей для потоковой работы с файлами. Это элементарный поток: он может записывать или читать только одиночный байт или массив байтов. Однако напрямую взаимодействовать с членами типа `FileStream` вам придется нечасто. Взамен вы, скорее всего, будете применять разнообразные *оболочки потоков*, которые облегчают работу с текстовыми данными или типами `.NET`. Тем не менее, полезно поэкспериментировать с возможностями синхронного чтения/записи типа `FileStream`.

Пусть имеется новый проект консольного приложения под названием `FileStreamApp` (и в файле кода `C#` импортировано пространство имен `System.IO` и `System.Text`). Целью будет запись простого текстового сообщения в новый файл по имени `myMessage.dat`. Однако с учетом того, что `FileStream` может оперировать только с низкоуровневыми байтами, объект типа `System.String` придется закодировать в соответствующий байтовый массив. К счастью, в пространстве имен `System.Text` определен тип `Encoding`, предоставляющий члены, которые кодируют и декодируют строки в массивы байтов (подробное описание типа `Encoding` ищите в документации `.NET Framework SDK`).

После кодирования байтовый массив сохраняется в файле с помощью метода `FileStream.Write()`. Чтобы прочесть байты обратно в память, понадобится сбросить внутреннюю позицию потока (посредством свойства `Position`) и вызвать метод `ReadByte()`. Наконец, на консоль выводится содержимое низкоуровневого байтового массива и декодированная строка. Ниже приведен полный код метода `Main()`.

```
// Не забудьте импортировать пространства имен System.Text и System.IO.
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with FileStreams *****\n");

    // Получить объект FileStream.
    using(FileStream fStream = File.Open(@"C:\myMessage.dat",
        FileMode.Create))
    {
        // Закодировать строку в виде массива байтов.
        string msg = "Hello!";
        byte[] msgAsByteArray = Encoding.Default.GetBytes(msg);

        // Записать byte[] в файл.
        fStream.Write(msgAsByteArray, 0, msgAsByteArray.Length);

        // Сбросить внутреннюю позицию потока.
        fStream.Position = 0;

        // Прочитать byte[] из файла и вывести на консоль.
        Console.Write("Your message as an array of bytes: ");
        byte[] bytesFromFile = new byte[msgAsByteArray.Length];
        for (int i = 0; i < msgAsByteArray.Length; i++)
        {
            bytesFromFile[i] = (byte)fStream.ReadByte();
            Console.Write(bytesFromFile[i]);
        }

        // Вывести декодированное сообщение.
        Console.Write("\nDecoded Message: ");
        Console.WriteLine(Encoding.Default.GetString(bytesFromFile));
    }
    Console.ReadLine();
}
```

В приведенном примере не только производится наполнение файла данными, но также демонстрируется основной недостаток прямой работы с типом `FileStream`: необходимость оперирования низкоуровневыми байтами. Другие производные от `Stream` типы работают в похожей манере. Например, чтобы записать последовательность байтов в область памяти, понадобится создать объект `MemoryStream`. Аналогично для передачи массива байтов через сетевое подключение используется класс `NetworkStream` (из пространства имен `System.Net.Sockets`).

Как упоминалось ранее, в пространстве имен `System.IO` доступно несколько типов для средств чтения и записи, которые инкапсулируют детали работы с типами, производными от `Stream`.

Исходный код. Проект `FileStreamApp` доступен в подкаталоге `Chapter_20`.

Работа с классами `StreamWriter` и `StreamReader`

Классы `StreamWriter` и `StreamReader` удобны всякий раз, когда нужно читать или записывать символьные данные (например, строки). Оба типа по умолчанию работают с символами `Unicode`; тем не менее, это можно изменить за счет предоставления должным образом сконфигурированной ссылки на объект `System.Text.Encoding`. Чтобы не усложнять пример, предположим, что стандартная кодировка `Unicode` вполне устраивает.

Класс `StreamReader` является производным от абстрактного класса по имени `TextReader`, как и связанный с ним тип `StringReader` (обсуждается далее в главе). Базовый класс `TextReader` предоставляет каждому из своих наследников ограниченный набор функциональности, в частности возможность читать и “заглядывать” в символьный поток.

Класс `StreamWriter` (а также `StringWriter`, который будет рассматриваться позже) порожден от абстрактного базового класса по имени `TextWriter`, в котором определены члены, позволяющие производным типам записывать текстовые данные в текущий символьный поток.

Чтобы содействовать пониманию основных возможностей записи в классах `StreamWriter` и `StringWriter`, в табл. 20.8 перечислены основные члены абстрактного базового класса `TextWriter`.

Таблица 20.8. Основные члены `TextWriter`

Член	Описание
<code>Close()</code>	Этот метод закрывает средство записи и освобождает все связанные с ним ресурсы. В процессе буфер автоматически сбрасывается (и снова этот член функционально эквивалентен методу <code>Dispose()</code>)
<code>Flush()</code>	Этот метод очищает все буферы текущего средства записи и записывает все буферизованные данные на лежащее в основе устройство; однако он не закрывает средство записи
<code>NewLine</code>	Это свойство задает константу новой строки для унаследованного класса средства записи. По умолчанию ограничителем строки в <code>Windows</code> является возврат каретки, за которым следует перевод строки (<code>\r\n</code>)
<code>Write()</code>	Этот перегруженный метод записывает данные в текстовый поток без добавления константы новой строки
<code>WriteLine()</code>	Этот перегруженный метод записывает данные в текстовый поток с добавлением константы новой строки

На заметку! Вероятно, последние два члена класса `TextWriter` покажутся знакомыми. Вспомните, что тип `System.Console` имеет члены `Write()` и `WriteLine()`, которые выталкивают текстовые данные на стандартное устройство вывода. В действительности свойство `Console.In` является оболочкой для объекта `TextWriter`, а `Console.Out` — для `TextWriter`.

Производный класс `StreamWriter` предоставляет подходящую реализацию методов `Write()`, `Close()` и `Flush()`, а также определяет дополнительное свойство `AutoFlush`. Установка этого свойства в `true` заставляет `StreamWriter` выталкивать данные при каждой операции записи. Имейте в виду, что за счет установки `AutoFlush` в `false` можно достичь более высокой производительности, но по завершении работы с объектом `StreamWriter` должен быть вызван метод `Close()`.

Запись в текстовый файл

Чтобы увидеть класс `StreamWriter` в действии, создадим новый проект консольного приложения по имени `StreamWriterReaderApp` и импортируем пространство имен `System.IO`. В показанном ниже методе `Main()` с помощью метода `File.CreateText()` создается новый файл `reminders.txt` внутри текущего каталога выполнения. С применением полученного объекта `StreamWriter` в новый файл будут добавляться текстовые данные.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
    // Получить объект StreamWriter и записать строковые данные.
    using(StreamWriter writer = File.CreateText("reminders.txt"))
    {
        writer.WriteLine("Don't forget Mother's Day this year...");
        writer.WriteLine("Don't forget Father's Day this year...");
        writer.WriteLine("Don't forget these numbers:");
        for(int i = 0; i < 10; i++)
            writer.Write(i + " ");

        // Вставить новую строку.
        writer.Write(writer.NewLine);
    }

    Console.WriteLine("Created file and wrote some thoughts...");
    Console.ReadLine();
}
```

После выполнения программы можно просмотреть содержимое созданного файла (рис. 20.2), который находится в папке `bin\Debug` текущего приложения, т.к. при вызове `CreateText()` абсолютный путь не указывался.

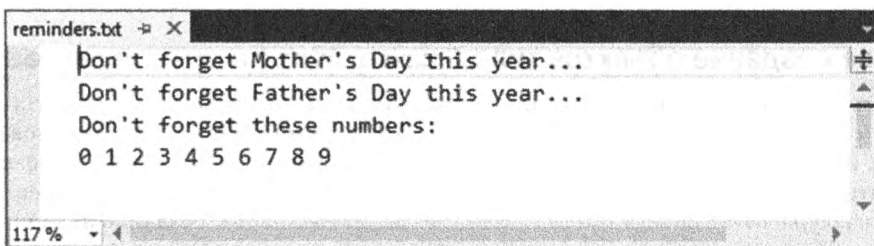


Рис. 20.2. Содержимое созданного текстового файла

Чтение из текстового файла

Далее вы научитесь программно читать данные из файла, используя соответствующий тип `StreamReader`. Вспомните, что `StreamReader` является производным от абстрактного класса `TextReader`, который предлагает функциональность, описанную в табл. 20.9.

Таблица 20.9. Основные члены `TextReader`

Член	Описание
<code>Peek()</code>	Возвращает следующий доступный символ (выраженный в виде целого числа), не изменяя текущей позиции средства чтения. Значение <code>-1</code> указывает на достижение конца потока
<code>Read()</code>	Читает данные из входного потока
<code>ReadBlock()</code>	Читает указанное максимальное количество символов из текущего потока и записывает данные в буфер, начиная с заданного индекса
<code>ReadLine()</code>	Читает строку символов из текущего потока и возвращает данные в виде строки (строка <code>null</code> указывает на признак конца файла)
<code>ReadToEnd()</code>	Читает все символы от текущей позиции до конца потока и возвращает их в виде единственной строки

Расширим текущий пример приложения с целью применения класса `StreamReader`, чтобы в нем можно было читать текстовые данные из файла `reminders.txt`:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
    ...
    // Прочитать данные из файла.
    Console.WriteLine("Here are your thoughts:\n");
    using(StreamReader sr = File.OpenText("reminders.txt"))
    {
        string input = null;
        while ((input = sr.ReadLine()) != null)
        {
            Console.WriteLine (input);
        }
    }
    Console.ReadLine();
}
```

После запуска программы в окне консоли отобразятся символьные данные из файла `reminders.txt`.

Прямое создание объектов типа `StreamWriter/StreamReader`

Один из запутывающих аспектов работы с типами пространства имен `System.IO` связан с тем, что идентичных результатов часто можно добиться с использованием разных подходов. Например, ранее вы уже видели, что метод `CreateText()` позволяет получить объект `StreamWriter` с типом `File` или `FileInfo`. Вообще говоря, есть еще один способ работы с объектами `StreamWriter` и `StreamReader`: создание их напрямую. Скажем, текущее приложение можно было бы переделать следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
    // Получить объект StreamWriter и записать строковые данные.
    using(StreamWriter writer = new StreamWriter("reminders.txt"))
    {
        ...
    }
    // Прочитать данные из файла.
    using(StreamReader sr = new StreamReader("reminders.txt"))
    {
        ...
    }
}
```

Несмотря на то что существование такого количества на первый взгляд одинаковых подходов к файловому вводу-выводу может сбивать с толку, имейте в виду, что конечным результатом является высокая гибкость. Теперь, когда вам известно, как перемещать символьные данные в файл и из файла с применением классов `StreamWriter` и `StreamReader`, давайте займемся исследованием роли классов `StringWriter` и `StringReader`.

Исходный код. Проект `StreamWriterReaderApp` доступен в подкаталоге `Chapter_20`.

Работа с классами `StringWriter` и `StringReader`

Классы `StringWriter` и `StringReader` можно использовать для трактовки текстовой информации как потока символов в памяти. Это определенно может быть полезно, когда нужно добавить символьную информацию к лежащему в основе буферу. Для иллюстрации в следующем проекте консольного приложения (`StringReaderWriterApp`) блок строковых данных записывается в объект `StringWriter` вместо файла на локальном жестком диске (не забудьте импортировать пространство имен `System.IO`):

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with StringWriter / StringReader *****\n");
    // Создать объект StringWriter и записать символьные данные в память.
    using(StringWriter strWriter = new StringWriter())
    {
        strWriter.WriteLine("Don't forget Mother's Day this year...");
        // Получить копию содержимого (хранящегося в строке)
        // и вывести на консоль.
        Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
    }
    Console.ReadLine();
}
```

Классы `StringWriter` и `StreamWriter` порождены от одного и того же базового класса (`TextWriter`), поэтому логика записи более или менее похожа. Тем не менее, с учетом природы `StringWriter` вы должны также знать, что данный класс позволяет применять метод `GetStringBuilder()` для извлечения объекта `System.Text.StringBuilder`:


```
using (StringWriter strWriter = new StringWriter())
{
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);

    // Получить внутренний объект StringBuilder.
    StringBuilder sb = strWriter.GetStringBuilder();
    sb.Insert(0, "Hey!! ");
    Console.WriteLine("-> {0}", sb.ToString());
    sb.Remove(0, "Hey!! ".Length);
    Console.WriteLine("-> {0}", sb.ToString());
}
```

Когда необходимо прочитать из потока строковые данные, можно использовать соответствующий тип `StringReader`, который (вполне ожидаемо) функционирует идентично `StreamReader`. Фактически класс `StringReader` лишь переопределяет унаследованные члены, чтобы выполнять чтение из блока символьных данных, а не из файла:

```
using (StringWriter strWriter = new StringWriter())
{
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);

    // Читать данные из объекта StringWriter.
    using (StringReader strReader = new StringReader(strWriter.ToString()))
    {
        string input = null;
        while ((input = strReader.ReadLine()) != null)
        {
            Console.WriteLine(input);
        }
    }
}
```

Исходный код. Проект `StringReaderWriterApp` доступен в подкаталоге `Chapter_20`.

Работа с классами `BinaryWriter` и `BinaryReader`

Последним набором классов средств чтения и записи, которые рассматриваются в настоящем разделе, являются `BinaryWriter` и `BinaryReader`; они оба унаследованы прямо от `System.Object`.

Типы `BinaryWriter` и `BinaryReader` позволяют читать и записывать в поток дискретные типы данных в компактном двоичном формате. В классе `BinaryWriter` определен многократно перегруженный метод `Write()`, предназначенный для помещения некоторого типа данных в поток. Помимо `Write()` класс `BinaryWriter` предоставляет дополнительные члены, которые позволяют получать или устанавливать объекты производных от `Stream` типов; кроме того, класс `BinaryWriter` также предлагает поддержку произвольного доступа к данным (табл. 20.10).

Класс `BinaryReader` дополняет функциональность класса `BinaryWriter` членами, описанными в табл. 20.11.

Таблица 20.10. Основные члены BinaryWriter

Член	Описание
BaseStream	Это свойство только для чтения обеспечивает доступ к лежащему в основе потоку, используемому с объектом BinaryWriter
Close()	Этот метод закрывает двоичный поток
Flush()	Этот метод выталкивает буфер двоичного потока
Seek()	Этот метод устанавливает позицию в текущем потоке
Write()	Этот метод записывает значение в текущий поток

Таблица 20.11. Основные члены BinaryReader

Член	Описание
BaseStream	Это свойство только для чтения обеспечивает доступ к лежащему в основе потоку, используемому с объектом BinaryReader
Close()	Этот метод закрывает двоичный поток
PeekChar()	Этот метод возвращает следующий доступный символ без перемещения текущей позиции потока
Read()	Этот метод читает заданный набор байтов или символов и сохраняет их во входном массиве
ReadXXXX()	В классе BinaryReader определены многочисленные методы чтения, которые извлекают из потока объекты различных типов (ReadBoolean(), ReadByte(), ReadInt32() и т.д.)

В показанном далее примере (проект консольного приложения по имени BinaryWriterReader) в файл *.dat записываются данные нескольких типов:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Binary Writers / Readers *****\n");
    // Открыть средство двоичной записи в файл.
    FileInfo f = new FileInfo("BinFile.dat");
    using(BinaryWriter bw = new BinaryWriter(f.OpenWrite()))
    {
        // Вывести на консоль тип BaseStream
        // (System.IO.FileStream в этом случае).
        Console.WriteLine("Base stream is: {0}", bw.BaseStream);
        // Создать некоторые данные для сохранения в файле.
        double aDouble = 1234.67;
        int anInt = 34567;
        string aString = "A, B, C";
        // Записать данные.
        bw.Write(aDouble);
        bw.Write(anInt);
        bw.Write(aString);
    }
    Console.WriteLine("Done!");
    Console.ReadLine();
}
```

Обратите внимание, что объект `FileStream`, возвращенный методом `FileInfo.OpenWrite()`, передается конструктору типа `BinaryWriter`. Применение такого приема облегчает организацию потока *по уровням* перед записью данных. Конструктор класса `BinaryWriter` принимает любой тип, производный от `Stream` (например, `FileStream`, `MemoryStream` или `BufferedStream`). Таким образом, запись двоичных данных в память сводится просто к использованию допустимого объекта `MemoryStream`.

Для чтения данных из файла `BinFile.dat` в классе `BinaryReader` предлагается несколько способов. Ниже для извлечения каждой порции данных из файлового потока вызываются разнообразные члены, связанные с чтением:

```
static void Main(string[] args)
{
    ...
    FileInfo f = new FileInfo("BinFile.dat");
    ...
    // Читать двоичные данные из потока.
    using(BinaryReader br = new BinaryReader(f.OpenRead()))
    {
        Console.WriteLine(br.ReadDouble());
        Console.WriteLine(br.ReadInt32());
        Console.WriteLine(br.ReadString());
    }
    Console.ReadLine();
}
```

Исходный код. Проект `BinaryWriterReader` доступен в подкаталоге `Chapter_20`.

Программное слежение за файлами

Теперь, когда вы знаете, как применять различные средства чтения и записи, займемся исследованием роли класса `FileSystemWatcher`, который полезен, когда требуется программно отслеживать состояние файлов в системе. В частности, с помощью `FileSystemWatcher` можно организовать мониторинг файлов на предмет любых действий, указываемых значениями перечисления `System.IO.NotifyFilters` (более подробные сведения об данном перечислении приведены в документации .NET Framework 4.7 SDK):

```
public enum NotifyFilters
{
    Attributes, CreationTime,
    DirectoryName, FileName,
    LastAccess, LastWrite,
    Security, Size
}
```

Чтобы начать работу с типом `FileSystemWatcher`, в свойстве `Path` понадобится указать имя (и местоположение) каталога, содержащего файлы, которые нужно отслеживать, а в свойстве `Filter` — расширения отслеживаемых файлов.

В настоящий момент можно выбрать обработку событий `Changed`, `Created` и `Deleted`, которые функционируют в сочетании с делегатом `FileSystemEventHandler`. Этот делегат может вызывать любой метод, соответствующий следующей сигнатуре:

```
// Делегат FileSystemEventHandler должен указывать
// на методы, соответствующие следующей сигнатуре.
void MyNotificationHandler(object source, FileSystemEventArgs e)
```

Событие `Renamed` может быть также обработано с использованием делегата `RenamedEventHandler`, который позволяет вызывать методы с такой сигнатурой:

```
// Делегат RenamedEventHandler должен указывать
// на методы, соответствующие следующей сигнатуре.
void MyRenamedHandler(object source, RenamedEventArgs e)
```

В то время как для обработки каждого события можно применять традиционный синтаксис делегатов/событий, вы определенно будете использовать синтаксис лямбда-выражений (как делается в загружаемом коде проекта).

Давайте взглянем на процесс слежения за файлом. Предположим, что на диске `C:` создан новый каталог по имени `MyFolder`, который содержит разнообразные файлы `*.txt` (с произвольными именами). Показанный ниже проект консольного приложения (`MyDirectoryWatcher`) наблюдает за файлами `*.txt` в каталоге `MyFolder` и выводит на консоль сообщения, когда происходит их создание, удаление, модификация и переименование:

```
static void Main(string[] args)
{
    Console.WriteLine("***** The Amazing File Watcher App *****\n");
    // Установить путь к каталогу, за которым нужно наблюдать.
    FileSystemWatcher watcher = new FileSystemWatcher();
    try
    {
        watcher.Path = @"C:\MyFolder";
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
        return;
    }

    // Указать цели наблюдения.
    watcher.NotifyFilter = NotifyFilters.LastAccess
        | NotifyFilters.LastWrite
        | NotifyFilters.FileName
        | NotifyFilters.DirectoryName;

    // Следить только за текстовыми файлами.
    watcher.Filter = "*.txt";

    // Добавить обработчики событий.
    watcher.Changed += new FileSystemEventHandler(OnChanged);
    watcher.Created += new FileSystemEventHandler(OnChanged);
    watcher.Deleted += new FileSystemEventHandler(OnChanged);
    watcher.Renamed += new RenamedEventHandler(OnRenamed);

    // Начать наблюдение за каталогом.
    watcher.EnableRaisingEvents = true;

    // Ожидать от пользователя команду завершения программы.
    Console.WriteLine(@"Press 'q' to quit app.");
    while (Console.Read() != 'q')
    ;
}
```

Следующие два обработчика событий просто выводят сообщения о модификации текущего файла:

```
static void OnChanged(object source, FileSystemEventArgs e)
{
    // Сообщить о действии изменения, создания или удаления файла.
    Console.WriteLine("File: {0} {1}!", e.FullPath, e.ChangeType);
}

static void OnRenamed(object source, RenamedEventArgs e)
{
    // Сообщить о действии переименования файла.
    Console.WriteLine("File: {0} renamed to {1}", e.OldFullPath, e.FullPath);
}
```

Запустите программу и откройте проводник Windows. Попробуйте переименовать файлы, создать файл *.txt, удалить файл *.txt и т.д. Вы увидите различные сообщения о состоянии текстовых файлов внутри MyFolder, например:

```
***** The Amazing File Watcher App *****

Press 'q' to quit app.
File: C:\MyFolder\New Text Document.txt Created!
File: C:\MyFolder\New Text Document.txt renamed to C:\MyFolder\Hello.txt
File: C:\MyFolder\Hello.txt Changed!
File: C:\MyFolder\Hello.txt Changed!
File: C:\MyFolder\Hello.txt Deleted!
```

Исходный код. Проект MyDirectoryWatcher доступен в подкаталоге Chapter_20.

Итак, знакомство с фундаментальными операциями ввода-вывода, предлагаемыми платформой .NET, завершено. Вы наверняка будете применять все продемонстрированные приемы во многих приложениях. Вдобавок вы обнаружите, что службы *сериализации объектов* способны значительно упростить задачу сохранения больших объемов данных.

Понятие сериализации объектов

Термин *сериализация* описывает процесс сохранения (и возможно передачи) состояния объекта в потоке (например, файловом потоке или потоке в памяти). Сохраненная последовательность данных содержит всю информацию, необходимую для воссоздания (или *десериализации*) состояния объекта с целью последующего использования. Применение такой технологии делает тривиальным сохранение крупных объемов данных (в разнообразных форматах). Во многих случаях сохранение данных приложения с использованием служб сериализации дает в результате меньше кода, чем с применением средств чтения/записи из пространства имен System.IO.

Например, пусть требуется создать настольное приложение с графическим пользовательским интерфейсом, которое должно предоставлять конечным пользователям возможность сохранения их предпочтений (цвета окон, размер шрифта и т.д.). Для этого можно определить класс по имени UserPrefs и инкапсулировать в нем около двадцати полей данных. В случае использования типа System.IO.BinaryWriter пришлось бы *вручную* сохранять каждое поле объекта UserPrefs. Подобным же образом при загрузке данных из файла обратно в память понадобилось бы применять класс System.IO.BinaryReader и снова *вручную* читать каждое значение, чтобы повторно сконфигурировать новый объект UserPrefs.

Хотя поступать так вполне допустимо, можно сэкономить значительное время, поместив класс UserPrefs атрибуту [Serializable]:

```
[Serializable]
public class UserPrefs
{
    public string WindowColor;
    public int FontSize;
}
```

В итоге полное состояние объекта может быть сохранено с помощью лишь нескольких строк кода. Не вдаваясь пока в детали, взгляните на показанный ниже метод Main():

```
static void Main(string[] args)
{
    UserPrefs userData= new UserPrefs();
    userData.WindowColor = "Yellow";
    userData.FontSize = 50;

    // BinaryFormatter сохраняет данные в двоичном формате.
    // Чтобы получить доступ к BinaryFormatter, необходимо импортировать
    // пространство имен System.Runtime.Serialization.Formatters.Binary.
    BinaryFormatter binFormat = new BinaryFormatter();

    // Сохранить объект в локальном файле.
    using(Stream fStream = new FileStream("user.dat",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, userData);
    }
    Console.ReadLine();
}
```

Сериализация объектов .NET упрощает сохранение объектов, но ее внутренний процесс довольно сложен. Например, когда объект сохраняется в потоке, все ассоциированные с ним данные (т.е. данные базового класса и содержащиеся в нем объекты) также автоматически сериализируются. Следовательно, при попытке сериализации производного класса в игру вступают также все данные по цепочке наследования. Вы увидите, что для представления множества взаимосвязанных объектов используется граф объектов.

Службы сериализации .NET также позволяют сохранять граф объектов в разных форматах. В предыдущем примере кода применялся тип `BinaryFormatter`, поэтому состояние объекта `UserPrefs` сохранялось в компактном двоичном формате. Граф объектов можно также сохранить в формате SOAP или XML, используя другие типы форматов. Такие форматы могут быть очень полезными, когда необходимо гарантировать возможность передачи сохраненных объектов между операционными системами, языками и архитектурами.

На заметку! Инфраструктура WCF предлагает слегка отличающийся механизм для сериализации объектов в и из операций служб WCF; он предусматривает применение атрибутов `[DataContract]` и `[DataMember]`. Данная тема подробно раскрывается в главе 23.

Наконец, имейте в виду, что граф объектов может быть сохранен в любом типе, производном от `System.IO.Stream`. В предыдущем примере для сохранения объекта `UserPrefs` в локальном файле использовался тип `FileStream`. Однако если необходимо сохранить объект в заданной области памяти, тогда взамен можно применить тип `MemoryStream`. Главное, чтобы последовательность данных корректно представляла состояние объектов внутри графа.

Роль графов объектов

Как упоминалось ранее, среда CLR будет учитывать все связанные объекты, чтобы обеспечить корректное сохранение данных, когда объект сериализуется. Такой набор связанных объектов называется *графом объектов*. Графы объектов предоставляют простой способ документирования взаимосвязи между множеством элементов. Следует отметить, что графы объектов не обозначают отношения “является” и “имеет” объектно-ориентированного программирования. Взаимен стрелки в графе объектов можно трактовать как “требует” или “зависит от”.

Каждый объект в графе получает уникальное числовое значение. Важно помнить, что числа, присвоенные объектам в графе, являются произвольными и не имеют никакого смысла для внешнего мира.

После того как всем объектам назначены числовые значения, граф объектов может записывать набор зависимостей для каждого объекта.

В качестве примера предположим, что создано множество классов, которые моделируют автомобили. Существует базовый класс по имени *Car*, который “имеет” класс *Radio*. Другой класс по имени *JamesBondCar* расширяет базовый тип *Car*. На рис. 20.3 показан возможный граф объектов, моделирующий такие отношения.

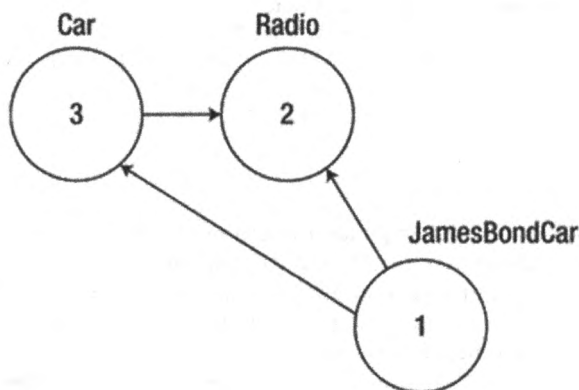


Рис. 20.3. Простой граф объектов

При чтении графов объектов для описания соединяющих стрелок можно использовать выражение “зависит от” или “ссылается на”. Таким образом, на рис. 20.3 видно, что класс *Car* ссылается на класс *Radio* (учитывая отношение “имеет”), *JamesBondCar* ссылается на *Car* (из-за отношения “является”), а также на *Radio* (поскольку наследует эту защищенную переменную-член).

Разумеется, среда CLR не рисует картинки в памяти для представления графа связанных объектов. Взаимен отношение, показанное на рис. 20.3, представляется математической формулой, которая выглядит следующим образом:

```
[Car 3, ref 2], [Radio 2], [JamesBondCar 1, ref 3, ref 2]
```

Проанализировав формулу, вы заметите, что объект 3 (*Car*) имеет зависимость от объекта 2 (*Radio*). Объект 2 (*Radio*) — “одинокий волк”, которому никто не нужен. Наконец, объект 1 (*JamesBondCar*) имеет зависимость от объекта 3, а также от объекта 2. В любом случае при сериализации или десериализации экземпляра *JamesBondCar* граф объектов гарантируется, что типы *Radio* и *Car* также примут участие в процессе.

Привлекательность процесса сериализации заключается в том, что граф, представляющий отношения между объектами, устанавливается автоматически “за кулисами”.

Как будет показано позже в главе, при желании в конструирование графа объектов можно вмешиваться, настраивая процесс сериализации с применением атрибутов и интерфейсов.

На заметку! Строго говоря, тип `XmlSerializer` (описанный далее в главе) не сохраняет состояние с использованием графа объектов; тем не менее, этот тип сериализует и десериализует связанные объекты в предсказуемой манере.

Конфигурирование объектов для сериализации

Чтобы сделать объект доступным службам сериализации .NET, понадобится только декорировать каждый связанный класс (или структуру) атрибутом `[Serializable]`. Если выясняется, что какой-то тип имеет члены-данные, которые не должны (или не могут) участвовать в сериализации, тогда их необходимо пометить атрибутом `[NonSerialized]`. Это помогает сократить размер хранимых данных, когда в сериализуемом классе есть переменные-члены, которые запоминать не нужно (например, фиксированные значения, случайные значения и кратковременные данные).

Определение сериализуемых типов

Для начала создадим новый проект консольного приложения под названием `SimpleSerialize`. Добавим в него новый класс по имени `Radio`, помеченный атрибутом `[Serializable]`, в котором исключается одна переменная-член (`radioID`), помеченная атрибутом `[NonSerialized]` и потому не сохраняемая в указанном потоке данных:

```
[Serializable]
public class Radio
{
    public bool hasTweeters;
    public bool hasSubWoofers;
    public double[] stationPresets;

    [NonSerialized]
    public string radioID = "XF-552RR6";
}
```

Затем добавим два дополнительных типа для представления классов `JamesBondCar` и `Car`, которые также помечены атрибутом `[Serializable]`, и определим в них следующие поля данных:

```
[Serializable]
public class Car
{
    public Radio theRadio = new Radio();
    public bool isHatchBack;
}

[Serializable]
public class JamesBondCar : Car
{
    public bool canFly;
    public bool canSubmerge;
}
```

Имейте в виду, что атрибут `[Serializable]` не наследуется от родительского класса. Таким образом, если вы порождаете класс от типа, помеченного `[Serializable]`, то

дочерний класс также должен быть снабжен атрибутом [Serializable], иначе он не сможет сохраняться в потоке. В действительности все объекты внутри графа объектов должны быть помечены атрибутом [Serializable]. Попытка сериализации несериализуемого объекта с применением класса BinaryFormatter или SoapFormatter приведет к генерации исключения SerializationException во время выполнения.

Открытые поля, закрытые поля и открытые свойства

Обратите внимание, что в каждом классе поля данных определялись как открытые, что способствовало упрощению примера. Несомненно, с точки зрения объектно-ориентированного программирования закрытые данные с доступом через открытые свойства были бы предпочтительнее. Кроме того, ради простоты в типах не определялись какие-либо специальные конструкторы; следовательно, все неинициализированные поля данных получали ожидаемые стандартные значения.

Оставив в стороне принципы объектно-ориентированного программирования, вас может интересовать вопрос: какое определение полей данных внутри типа ожидают форматы, чтобы сериализовать их в поток? Ответ: в зависимости от обстоятельств. Если вы сохраняете состояние объекта с использованием типа BinaryFormatter или SoapFormatter, то это совершенно безразлично. Указанные типы запрограммированы на сериализацию *всех* сериализуемых полей типа независимо от того, являются они открытыми полями, закрытыми полями или закрытыми полями, доступными через открытые свойства. Однако вспомните, что при наличии элементов данных, которые не должны сохраняться в графе объектов, можно выборочно пометить открытые или закрытые поля атрибутом [NonSerialized], как было сделано со строковым полем в типе Radio.

Тем не менее, в случае применения типа XmlSerializer ситуация совсем другая. Он будет сериализовать *только* открытые поля данных или закрытые данные, доступные посредством открытых свойств. Закрытые данные, которые не доступны через свойства, будут игнорироваться. Например, рассмотрим следующий сериализуемый тип Person:

```
[Serializable]
public class Person
{
    // Открытое поле.
    public bool isAlive = true;

    // Закрытое поле.
    private int personAge = 21;

    // Открытое свойство/закрытые данные.
    private string fName = string.Empty;
    public string FirstName
    {
        get { return fName; }
        set { fName = value; }
    }
}
```

При обработке типа Person с помощью BinaryFormatter или SoapFormatter обнаружится, что поля isAlive, personAge и fName сохраняются в выбранном потоке. Однако тип XmlSerializer не сохранит значение personAge, потому что эта часть закрытых данных не инкапсулирована в открытом свойстве. Чтобы сохранять personAge с помощью XmlSerializer, поле personAge понадобится определить как открытое либо инкапсулировать закрытый член в открытом свойстве.

Выбор формatera сериализации

После конфигурирования типов для участия в схеме сериализации .NET путем применения необходимых атрибутов потребуется выбрать формат (двоичный, SOAP или XML), в котором будет храниться состояние объектов. Перечисленные возможности представлены следующими классами:

- BinaryFormatter
- SoapFormatter
- XmlSerializer

На заметку! Вас может интересовать, почему в приведенный выше список не включен формат JSON. Причина в том, что JSON рассматривается в главе 27.

Тип BinaryFormatter сериализует состояние объекта в поток, используя компактный двоичный формат, и определен внутри пространства имен System.Runtime.Serialization.Formatters.Binary, которое входит в сборку mscorlib.dll. Для получения доступа к типу BinaryFormatter понадобится указать следующую директиву using:

```
// Получить доступ к BinaryFormatter из сборки mscorlib.dll.
using System.Runtime.Serialization.Formatters.Binary;
```

Тип SoapFormatter сохраняет состояние объекта в виде сообщения SOAP (стандартный XML-формат для передачи и приема сообщений от веб-служб, основанных на SOAP) и определен в пространстве имен System.Runtime.Serialization.Formatters.Soap, находящемся в *отдельной сборке*. Таким образом, для форматирования графа объектов в сообщение SOAP необходимо сначала установить ссылку на System.Runtime.Serialization.Formatters.Soap.dll с применением диалогового окна Add Reference (Добавить ссылку) в Visual Studio и затем указать приведенную ниже директиву using:

```
// Требуется ссылка на сборку System.Runtime.Serialization.Formatters.Soap.dll.
using System.Runtime.Serialization.Formatters.Soap;
```

И, наконец, для сохранения дерева объектов в документе XML предусмотрен тип XmlSerializer. Чтобы его использовать, нужно указать директиву using для пространства имен System.Xml.Serialization и установить ссылку на сборку System.Xml.dll. К счастью, шаблоны проектов Visual Studio автоматически ссылаются на System.Xml.dll, так что достаточно просто поступить следующим образом:

```
// Определено внутри сборки System.Xml.dll.
using System.Xml.Serialization;
```

Интерфейсы IFormatter и IRemotingFormatter

Независимо от того, какой формater выбран, имейте в виду, что все они унаследованы непосредственно от System.Object, а потому не разделяют общий набор членов от какого-то базового класса сериализации. Тем не менее, типы BinaryFormatter и SoapFormatter поддерживают общие члены за счет реализации интерфейсов IFormatter и IRemotingFormatter (как ни странно, тип XmlSerializer не реализует ни одного из них).

Интерфейс System.Runtime.Serialization.IFormatter определяет основные методы Serialize() и Deserialize(), которые делают всю черновую работу по перемещению графов объектов в специфический поток и обратно. Помимо них в IFormatter определено несколько свойств, которые реализующий тип применяет “за кулисами”:

```
public interface IFormatter
{
    SerializationBinder Binder { get; set; }
    StreamingContext Context { get; set; }
    ISurrogateSelector SurrogateSelector { get; set; }
    object Deserialize(Stream serializationStream);
    void Serialize(Stream serializationStream, object graph);
}
```

Интерфейс `System.Runtime.Remoting.Messaging.IRemotingFormatter` (который внутренне используется уровнем удаленного взаимодействия `.NET Remoting`) перегружает методы `Serialize()` и `Deserialize()` в манере, больше подходящей для распределенного сохранения. Обратите внимание, что интерфейс `IRemotingFormatter` является производным от более общего интерфейса `IFormatter`:

```
public interface IRemotingFormatter : IFormatter
{
    object Deserialize(Stream serializationStream, HeaderHandler handler);
    void Serialize(Stream serializationStream, object graph, Header[] headers);
}
```

Несмотря на то что взаимодействие с этими интерфейсами в большинстве сценариев сериализации может не потребоваться, вспомните, что полиморфизм на основе интерфейсов позволяет указывать экземпляры `BinaryFormatter` или `SoapFormatter` с применением ссылки на `IFormatter`. Следовательно, если необходимо построить метод, способный сериализовать граф объектов с использованием любого из этих классов, то можно написать такой код:

```
static void SerializeObjectGraph(IFormatter itfFormat,
                                Stream destStream, object graph)
{
    itfFormat.Serialize(destStream, graph);
}
```

Точность воспроизведения типов среди форматов

Наиболее очевидное отличие между тремя формateraми касается того, каким образом граф объектов сохраняется в потоке (двоичном, SOAP или XML). Вы должны также знать о нескольких более тонких аспектах отличия, в частности, как форматы справляются с *точностью воспроизведения типов*. Когда применяется тип `BinaryFormatter`, он сохраняет не только данные полей объектов из графа, но также полностью заданное имя каждого типа и полное имя определяющей его сборки (имя, версию, маркер открытого ключа и культуру). Такие дополнительные элементы данных делают `BinaryFormatter` идеальным выбором, когда необходимо передавать объекты по значению (например, полные копии) между границами машин для использования в приложениях .NET.

Форматер `SoapFormatter` сохраняет следы сборки источника за счет применения пространства имен XML. Для примера вспомните тип `Person`, определенный ранее в главе. Если бы данный тип был сохранен как сообщение SOAP, то вы обнаружили бы, что открывающий элемент `Person` уточнен сгенерированным параметром `xmlns`. Взгляните на следующее частичное определение, обратив особое внимание на пространство имен XML под названием `al`:

```
<al:Person id="ref-1" xmlns:al=
"http://schemas.microsoft.com/clr/nsassem/SimpleSerialize/MyApp%2C%20
Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<isAlive>true</isAlive>
```

```
<personAge>21</personAge>
<fName id="ref-3">Mel</fName>
</al:Person>
```

Однако `XmlSerializer` не пытается сберечь точную информацию о типе; следовательно, он не записывает полностью заданное имя типа или сборку, где тип определен. На первый взгляд это может показаться ограничением, но сериализация XML используется классическими веб-службами .NET, которые могут вызываться клиентами из любой платформы. Другими словами, нет никакого смысла сериализовать полные метаданные типа .NET. Вот возможное XML-представление типа `Person`:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <isAlive>true</isAlive>
  <PersonAge>21</PersonAge>
  <FirstName>Frank</FirstName>
</Person>
```

Если необходимо сохранить состояние объекта так, чтобы с ним можно было работать в любой операционной системе (скажем, в Windows, macOS и разнообразных дистрибутивах Linux), платформе приложений (например, .NET, Java Enterprise Edition и COM) или языке программирования, тогда поддерживать полную точность воспроизведения типов не нужно, поскольку нельзя рассчитывать на то, что все возможные получатели способны воспринимать типы данных, специфичные для .NET. С учетом сказанного типы `SoapFormatter` и `XmlSerializer` являются идеальным выбором, когда требуется обеспечить как можно более широкое распространение сохраненного дерева объектов.

Сериализация объектов с использованием `BinaryFormatter`

Чтобы проиллюстрировать, насколько просто сохранять экземпляр `JamesBondCar` в физическом файле, можно применить тип `BinaryFormatter`. Двумя ключевыми методами типа `BinaryFormatter`, о которых следует знать, являются `Serialize()` и `Deserialize()`:

- `Serialize()` сохраняет граф объектов в указанном потоке как последовательность байтов;
- `Deserialize()` преобразует сохраненную последовательность байтов в граф объектов.

Предположим, что после создания экземпляра `JamesBondCar` и модификации некоторых данных состояния требуется сохранить его в файле `*.dat`. Начать следует с создания самого файла `*.dat`. Достичь этого можно путем создания экземпляра типа `System.IO.FileStream`. Затем можно создать экземпляр `BinaryFormatter` и передать ему экземпляр `FileStream` и граф объектов для сохранения. Взгляните на следующий метод `Main()`:

```
// Не забудьте импортировать пространства имен
// System.Runtime.Serialization.Formatters.Binary и System.IO.
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Serialization *****\n");
    // Создать объект JamesBondCar и установить состояние.
```

```

JamesBondCar jbc = new JamesBondCar();
jbc.canFly = true;
jbc.canSubmerge = false;
jbc.theRadio.stationPresets = new double[]{89.3, 105.1, 97.1};
jbc.theRadio.hasTweeters = true;

// Сохранить объект JamesBondCar в указанном файле в двоичном формате.
SaveAsBinaryFormat(jbc, "CarData.dat");
Console.ReadLine();
}

```

Метод `SaveAsBinaryFormat()` реализован так, как показано ниже:

```

static void SaveAsBinaryFormat(object objGraph, string fileName)
{
    // Сохранить граф объектов в файл CarData.dat в двоичном виде.
    BinaryFormatter binFormat = new BinaryFormatter();
    using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in binary format!");
}

```

Как видите, метод `BinaryFormatter.Serialize()` — это член, ответственный за построение графа объектов и передачу последовательности байтов объекту производного от `Stream` типа. В рассматриваемом случае поток представляет физический файл. Сериализовать объекты можно было бы также в любой тип, производный от `Stream`, такой как область памяти или сетевой поток.

После выполнения программы можно просмотреть содержимое файла `CarData.dat`, которое представляет данный экземпляр `JamesBondCar`, перейдя в папку `bin\Debug` текущего проекта. На рис. 20.4 показано содержимое данного файла, открытого в Visual Studio.

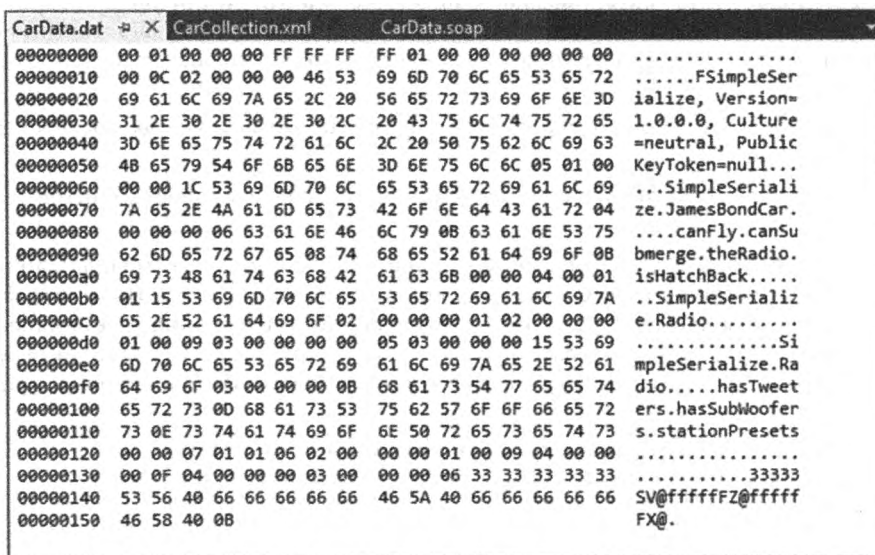


Рис. 20.4. Объект `JamesBondCar`, сериализованный с использованием `BinaryFormatter`

Десериализация объектов с использованием BinaryFormatter

Теперь предположим, что необходимо прочитать сохраненный объект `JamesBondCar` из двоичного файла обратно в объектную переменную. После открытия файла `CataData.dat` (с помощью метода `File.OpenRead()`) можно вызвать метод `Deserialize()` класса `BinaryFormatter`. Имейте в виду, что `Deserialize()` возвращает объект общего типа `System.Object`, так что понадобится применить явное приведение:

```
static void LoadFromBinaryFile(string fileName)
{
    BinaryFormatter binFormat = new BinaryFormatter();
    // Прочитать объект JamesBondCar из двоичного файла.
    using(Stream fStream = File.OpenRead(fileName))
    {
        JamesBondCar carFromDisk =
            (JamesBondCar)binFormat.Deserialize(fStream);
        Console.WriteLine("Can this car fly? : {0}", carFromDisk.canFly);
    }
}
```

Обратите внимание, что при вызове метода `Deserialize()` передается объект производного от `Stream` типа, который представляет местоположение сохраненного графа объектов. После приведения возвращенного объекта к корректному типу вы обнаружите, что данные состояния восстановлены на момент, когда объект сохранялся.

Сериализация объектов с использованием SoapFormatter

Следующим рассматриваемым форматером будет `SoapFormatter`, который сериализует данные в подходящем конверте SOAP. Выражаясь кратко, протокол SOAP (Simple Object Access Protocol — простой протокол доступа к объектам) описывает стандартный процесс вызова методов в независимой от платформы и операционной системы манере.

Предполагая, что была добавлена ссылка на сборку `System.Runtime.Serialization.Formatters.Soap.dll` (и импортировано пространство имен `System.Runtime.Serialization.Formatters.Soap`), для сохранения и извлечения объекта `JamesBondCar` как сообщения SOAP в предыдущем примере можно просто заменить все вхождения `BinaryFormatter` типом `SoapFormatter`. Взгляните на следующий новый метод класса `Program`, который сериализует объект в локальный файл в формате SOAP:

```
// Не забудьте импортировать пространства имен
// System.Runtime.Serialization.Formatters.Soap и установить ссылку
// на сборку System.Runtime.Serialization.Formatters.Soap.dll.
static void SaveAsSoapFormat(object objGraph, string fileName)
{
    // Сохранить граф объектов в файле CarData.soap в формате SOAP.
    SoapFormatter soapFormat = new SoapFormatter();
    using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        soapFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in SOAP format!");
}
```

Как и ранее, для перемещения графа объектов в поток и обратно применяются методы `Serialize()` и `Deserialize()`. После вызова метода `SaveAsSoapFormat()` внутри `Main()` и запуска приложения можно открыть результирующий файл *.soap. В нем находятся XML-элементы, которые описывают значения состояния текущего объекта `JamesBondCar`, а также отношения между объектами в графе, представленные с использованием лексем `#ref` (рис. 20.5).

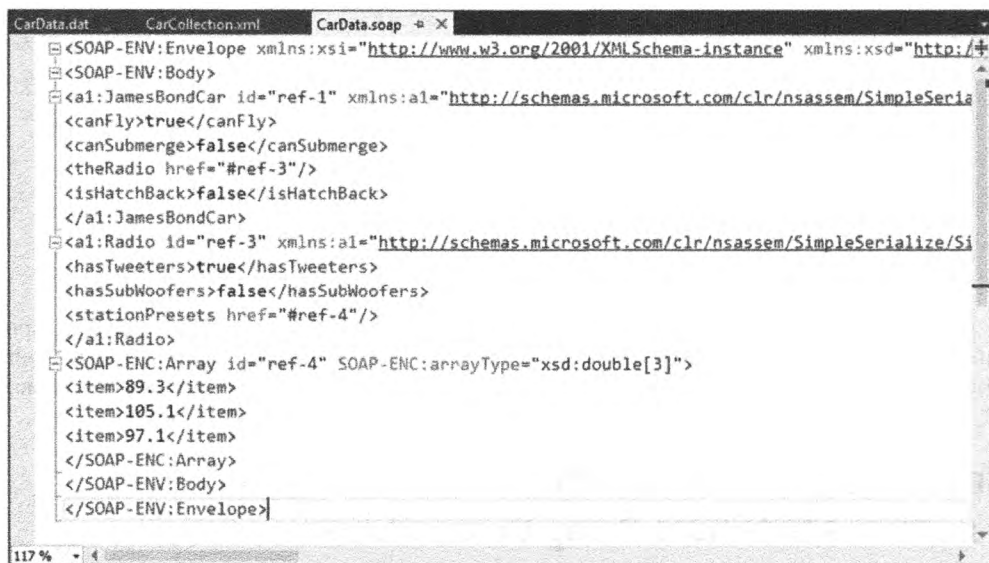


Рис. 20.5. Объект `JamesBondCar`, сериализованный с применением `SoapFormatter`

Сериализация объектов с использованием `XmlSerializer`

В дополнение к двоичному формату и формату SOAP сборка `System.Xml.dll` предлагает третий класс формatera — `System.Xml.Serialization.XmlSerializer`. Его можно применять для сохранения *открытого* состояния заданного объекта в виде чистой XML-разметки как противоположность XML-данным, помещенным внутрь сообщения SOAP. Работа с типом `XmlSerializer` несколько отличается от работы с типами `SoapFormatter` или `BinaryFormatter`. Рассмотрим показанный ниже код, в котором предполагается, что было импортировано пространство имен `System.Xml.Serialization`:

```
static void SaveAsXmlFormat(object objGraph, string fileName)
{
    // Сохранить объект в файле CarData.xml в формате XML.
    XmlSerializer xmlFormat = new XmlSerializer(typeof(JamesBondCar));
    using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        xmlFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in XML format!");
}
```

Основное отличие здесь в том, что класс `XmlSerializer` требует указания информации о типе, которая представляет класс, подлежащий сериализации. В сгенерированном XML-файле (в случае вызова метода `SaveAsXmlFormat()` внутри `Main()`) находятся следующие данные XML:

```
<?xml version="1.0"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <theRadio>
    <hasTweeters>true</hasTweeters>
    <hasSubWoofers>false</hasSubWoofers>
    <stationPresets>
      <double>89.3</double>
      <double>105.1</double>
      <double>97.1</double>
    </stationPresets>
    <radioID>XF-552RR6</radioID>
  </theRadio>
  <isHatchBack>false</isHatchBack>
  <canFly>true</canFly>
  <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

На заметку! Класс `XmlSerializer` требует, чтобы все сериализованные типы в графе объектов поддерживали стандартный конструктор (а потому не забудьте его добавить, если определяли специальные конструкторы). В противном случае во время выполнения сгенерируется исключение `InvalidOperationException`.

Управление генерацией данных XML

Если у вас есть опыт работы с технологиями XML, то вы знаете, что часто важно гарантировать соответствие данных внутри документа XML набору правил, которые устанавливают *действительность* данных.

Понятие *действительного* документа XML не имеет никакого отношения к синтаксической правильности элементов XML (вроде того, что все открывающие элементы должны иметь закрывающие элементы).

Действительные документы отвечают согласованным правилам форматирования (например, поле `X` должно быть выражено в виде атрибута, но не подэлемента), которые обычно задаются посредством схемы XML или файла определения типа документа (Document-Type Definition — DTD).

По умолчанию класс `XmlSerializer` сериализирует все открытые поля/свойства как элементы XML, а не как атрибуты XML. Чтобы управлять генерацией результирующего документа XML с помощью класса `XmlSerializer`, необходимо декорировать типы любым количеством дополнительных атрибутов из пространства имен `System.Xml.Serialization`.

В табл. 20.12 описаны некоторые (но не все) атрибуты, влияющие на способ кодирования данных XML в потоке.

Таблица 20.12. Избранные атрибуты из пространства имен System.Xml.Serialization

Атрибут .NET	Описание
[XmlAttribute]	Этот атрибут .NET можно применять к полю или свойству для сообщения XmlSerializer о необходимости сериализовать данные как атрибут XML (а не как подэлемент)
[XmlElement]	Поле или свойство будет сериализовано как элемент XML, именованный по вашему выбору
[XmlEnum]	Этот атрибут предоставляет имя элемента члена перечисления
[XmlRoot]	Этот атрибут управляет тем, как будет сконструирован корневой элемент (пространство имен и имя элемента)
[XmlText]	Свойство или поле будет сериализовано как текст XML (т.е. содержимое, находящееся между начальным и конечным дескрипторами корневого элемента)
[XmlType]	Этот атрибут предоставляет имя и пространство имен типа XML

В следующем простом примере показано текущее представление данных полей объекта JamesBondCar в XML:

```
<?xml version="1.0" encoding="utf-8"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ...
    <canFly>true</canFly>
    <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

Если необходимо указать специальное пространство имен XML, которое уточняет JamesBondCar и кодирует значения canFly и canSubmerge в виде атрибутов XML, тогда определение класса JamesBondCar можно модифицировать, как показано ниже:

```
[Serializable, XmlRoot(Namespace = "http://www.MyCompany.com")]
public class JamesBondCar : Car
{
    [XmlAttribute]
    public bool canFly;
    [XmlAttribute]
    public bool canSubmerge;
}
```

Результатом будет следующий документ XML (обратите внимание на открывающий элемент <JamesBondCar>):

```
<?xml version="1.0" ""?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               canFly="true" canSubmerge="false"
               xmlns="http://www.MyCompany.com">
    ...
</JamesBondCar>
```

Разумеется, для управления тем, как XmlSerializer генерирует результирующий документ XML, можно использовать многие другие атрибуты .NET. За подробной информацией обращайтесь к описанию пространства имен System.Xml.Serialization в документации .NET Framework 4.7 SDK.

Сериализация коллекций объектов

Теперь, когда вы видели, каким образом сохранять одиночный объект в потоке, давайте посмотрим, как сохранить набор объектов. Вы наверняка заметили, что метод `Serialize()` интерфейса `IFormatter` не предлагает способа указания произвольного количества объектов в качестве входных данных (а только единственного объекта типа `System.Object`). Вдобавок возвращаемое значение `Deserialize()` также является одиночным объектом `System.Object` (то же самое базовое ограничение касается и `XmlSerializer`):

```
public interface IFormatter
{
    ...
    object Deserialize(Stream serializationStream);
    void Serialize(Stream serializationStream, object graph);
}
```

Вспомните, что с помощью `System.Object` можно представить целое дерево объектов. Поэтому если передать объект, который помечен атрибутом `[Serializable]` и содержит в себе другие объекты `[Serializable]`, то с помощью единственного вызова данного метода сохранится весь набор объектов. К счастью, большинство типов в пространствах имен `System.Collections` и `System.Collections.Generic` уже помечено атрибутом `[Serializable]`. Следовательно, для сохранения набора объектов нужно просто добавить его в контейнер (такой как обычный массив, `ArrayList` или `List<T>`) и сериализовать контейнер в выбранный поток.

Предположим, что класс `JamesBondCar` дополнен конструктором, принимающим два аргумента, который позволяет устанавливать несколько фрагментов данных состояния (также должен быть добавлен стандартный конструктор, как того требует `XmlSerializer`):

```
[Serializable,
 XmlRoot(Namespace = "http://www.MyCompany.com")]
public class JamesBondCar : Car
{
    public JamesBondCar(bool skyWorthy, bool seaWorthy)
    {
        canFly = skyWorthy;
        canSubmerge = seaWorthy;
    }
    // XmlSerializer требует стандартного конструктора!
    public JamesBondCar() {}
    ...
}
```

Теперь можно сохранять любое количество объектов `JamesBondCar`:

```
static void SaveListOfCars()
{
    // Сохранить список List<T> объектов JamesBondCar.
    List<JamesBondCar> myCars = new List<JamesBondCar>();
    myCars.Add(new JamesBondCar(true, true));
    myCars.Add(new JamesBondCar(true, false));
    myCars.Add(new JamesBondCar(false, true));
    myCars.Add(new JamesBondCar(false, false));

    using(Stream fStream = new FileStream("CarCollection.xml",
        FileMode.Create, FileAccess.Write, FileShare.None))
```

```

{
    XmlSerializer xmlFormat = new XmlSerializer(typeof(List<JamesBondCar>));
    xmlFormat.Serialize(fStream, myCars);
}
Console.WriteLine("=> Saved list of cars!");
}

```

Здесь применяется класс `XmlSerializer`, так что для каждого из подобъектов внутри корневого объекта (`List<JamesBondCar>` в данном случае) должна быть указана информация о типе. Но если взамен использовать тип `BinaryFormatter` или `SoapFormatter`, то логика станет еще проще:

```

static void SaveListOfCarsAsBinary()
{
    // Сохранить объект ArrayList (myCars) в двоичном виде.
    List<JamesBondCar> myCars = new List<JamesBondCar>();
    BinaryFormatter binFormat = new BinaryFormatter();
    using(Stream fStream = new FileStream("AllMyCars.dat",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, myCars);
    }
    Console.WriteLine("=> Saved list of cars in binary!");
}

```

Исходный код. Проект `SimpleSerialize` доступен в подкаталоге `Chapter_20`.

Настройка процесса сериализации SOAP и двоичной сериализации

В большинстве ситуаций стандартная схема сериализации, предлагаемая платформой .NET, будет в точности тем, что требуется. Нужно лишь применить атрибут `[Serializable]` к связанным типам и передать дерево объектов выбранному формату для обработки. Тем не менее, в некоторых случаях может понадобиться вмешательство в процессы конструирования дерева и сериализации. Например, пусть существует бизнес-правило, которое гласит, что все поля данных должны сохраняться в определенном формате, или же в поток необходимо добавить дополнительные данные, которые не отображаются напрямую на поля сохраняемого объекта (скажем, временные метки и уникальные идентификаторы).

В пространстве имен `System.Runtime.Serialization` предусмотрено несколько типов, которые позволяют вмешиваться в процесс сериализации объектов.

В табл. 20.13 описаны основные типы, о которых следует знать.

Углубленный взгляд на сериализацию объектов

Прежде чем приступить к исследованию различных способов настройки процесса сериализации, полезно более внимательно присмотреться к тому, что происходит "за кулисами". Когда тип `BinaryFormatter` сериализует граф объектов, он отвечает за передачу в указанный поток следующей информации:

- полностью заданное имя объекта в графе (например, `MyApp.JamesBondCar`);
- имя сборки, определяющей граф объектов (скажем, `MyApp.exe`);
- экземпляр класса `SerializationInfo`, который содержит все данные состояния, поддерживаемого членами графа объектов.

Таблица 20.13. Основные типы пространства имен `System.Runtime.Serialization`

Тип	Описание
<code>ISerializable</code>	Этот интерфейс может быть реализован типом <code>[Serializable]</code> для управления его сериализацией и десериализацией
<code>ObjectIDGenerator</code>	Этот тип генерирует идентификаторы для членов в графе объектов
<code>[OnDeserialized]</code>	Этот атрибут позволяет указать метод, который будет вызван немедленно после десериализации объекта
<code>[OnDeserializing]</code>	Этот атрибут позволяет указать метод, который будет вызван перед началом процесса десериализации
<code>[OnSerialized]</code>	Этот атрибут позволяет указать метод, который будет вызван немедленно после того, как объект сериализирован
<code>[OnSerializing]</code>	Этот атрибут позволяет указать метод, который будет вызван перед началом процесса сериализации
<code>[OptionalField]</code>	Этот атрибут позволяет определить поле типа, которое может быть пропущено в указанном потоке
<code>SerializationInfo</code>	Этот класс является <i>пакетом свойств</i> , который поддерживает пары "имя-значение", представляющие состояние объекта во время процесса сериализации

Во время процесса десериализации `BinaryFormatter` использует ту же самую информацию для построения идентичной копии объекта с применением данных, извлеченных из лежащего в основе потока. Процесс, выполняемый `SoapFormatter`, очень похож.

На заметку! Вспомните, что для обеспечения максимальной мобильности объекта формater `XmlSerializer` не сохраняет полностью заданное имя типа либо имя определяющей его сборки. Тип `XmlSerializer` позволяет сохранять только открытые данные.

Помимо перемещения необходимых данных в поток и обратно формaterы также анализируют члены графа объектов на предмет перечисленных ниже частей инфраструктуры.

- Выполняется проверка, помечен ли объект атрибутом `[Serializable]`. Если объект не помечен, то генерируется исключение `SerializationException`.
- Если объект помечен атрибутом `[Serializable]`, тогда производится проверка, реализует ли он интерфейс `ISerializable`. Если да, то на этом объекте вызывается метод `GetObjectData()`.
- Если объект не реализует интерфейс `ISerializable`, тогда используется стандартный процесс сериализации, который обрабатывает все поля, не помеченные атрибутом `[NonSerialized]`.

В дополнение к определению того, поддерживает ли тип интерфейс `ISerializable`, формaterы также отвечают за исследование типов на предмет поддержки членов, которые оснащены атрибутами `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]` или `[OnDeserialized]`. Мы рассмотрим роль указанных атрибутов чуть позже, а сначала выясним предназначение интерфейса `ISerializable`.

Настройка сериализации с использованием `ISerializable`

У объектов, помеченных атрибутом `[Serializable]`, имеется возможность реализации интерфейса `ISerializable`, что позволяет вмешиваться в процесс сериализации и выполнять любое форматирование данных до или после сериализации.

Интерфейс `ISerializable` довольно прост; в нем определен единственный метод `GetObjectData()`:

```
// Когда необходимо настраивать процесс сериализации,
// следует реализовать интерфейс ISerializable.
public interface ISerializable
{
    void GetObjectData(SerializationInfo info, StreamingContext context);
}
```

Метод `GetObjectData()` вызывается автоматически заданным форматером в течение процесса сериализации. Реализация этого метода заполняет входной параметр типа `SerializationInfo` последовательностью пар "имя-значение", которые (обычно) отображаются на данные полей сохраняемого объекта. В классе `SerializationInfo` определены многочисленные вариации перегруженного метода `AddValue()`, а также небольшой набор свойств, которые позволяют устанавливать и получать имя, определяющую сборку и количество членов типа. Вот частичное определение `SerializationInfo`:

```
public sealed class SerializationInfo
{
    public SerializationInfo(Type type, IFormatterConverter converter);
    public string AssemblyName { get; set; }
    public string FullTypeName { get; set; }
    public int MemberCount { get; }
    public void AddValue(string name, short value);
    public void AddValue(string name, ushort value);
    public void AddValue(string name, int value);
    ...
}
```

Типы, которые реализуют интерфейс `ISerializable`, также должны определять специальный конструктор со следующей сигнатурой:

```
// Чтобы позволить исполняющей среде устанавливать состояние объекта,
// вы должны предоставить специальный конструктор с такой сигнатурой:
[Serializable]
class SomeClass : ISerializable
{
    protected SomeClass (SerializationInfo si, StreamingContext ctx) {...}
    ...
}
```

Обратите внимание, что видимость конструктора указана как `protected`. Это вполне допустимо, поскольку форматер будет иметь доступ к члену независимо от его видимости. Такие специальные конструкторы обычно определяются как `protected` (или `private`), гарантируя тем самым, что небрежный пользователь объекта никогда не создаст объект с их помощью. Первым параметром конструктора является экземпляр типа `SerializationInfo` (который был показан ранее).

Второй параметр специального конструктора имеет тип `StreamingContext` и содержит информацию относительно источника данных. Наиболее информативным членом `StreamingContext` является свойство `State`, которое хранит значение из перечисления `StreamingContextStates`. Значения данного перечисления представляют базовую композицию текущего потока.

Если только вы не намерены реализовывать какие-то низкоуровневые специальные службы удаленного взаимодействия, то иметь дело с перечислением `StreamingContextStates` напрямую придется редко. Тем не менее, ниже приведены члены перечисления `StreamingContextStates` (за более подробной информацией обращайтесь в документацию .NET Framework 4.7 SDK):

```
public enum StreamingContextStates
{
    CrossProcess,
    CrossMachine,
    File,
    Persistence,
    Remoting,
    Other,
    Clone,
    CrossAppDomain,
    All
}
```

Давайте посмотрим, каким образом настраивать процесс сериализации с применением `ISerializable`. Предположим, что имеется новый проект консольного приложения (по имени `CustomSerialization`), в котором определен тип класса, содержащий два элемента данных `string`. Также представим, что для этих объектов `string` требуется обеспечить сериализацию в поток в верхнем регистре, а десериализацию — в нижнем. Чтобы обеспечить соблюдение упомянутых правил, можно реализовать интерфейс `ISerializable`, как показано ниже (не забыв импортировать пространство имен `System.Runtime.Serialization`):

```
[Serializable]
class StringData : ISerializable
{
    private string dataItemOne = "First data block";
    private string dataItemTwo = "More data";
    public StringData() {}
    protected StringData(SerializationInfo si, StreamingContext ctx)
    {
        // Восстановить переменные-члены из потока.
        dataItemOne = si.GetString("First_Item").ToLower();
        dataItemTwo = si.GetString("dataItemTwo").ToLower();
    }
    void ISerializable.GetObjectData(SerializationInfo info, StreamingContext ctx)
    {
        // Наполнить объект SerializationInfo форматированными данными.
        info.AddValue("First_Item", dataItemOne.ToUpper());
        info.AddValue("dataItemTwo", dataItemTwo.ToUpper());
    }
}
```

Обратите внимание, что при наполнении объекта типа `SerializationInfo` внутри метода `GetObjectData()` именовать элементы данных идентично именам внутренних переменных-членов типа не обязательно. Очевидно, такой подход может быть полезным, если требуется отвязать данные типа от формата хранения. Однако имейте в виду, что получать значения в специальном защищенном конструкторе понадобится с указанием тех же самых имен, которые были назначены внутри `GetObjectData()`.

Чтобы протестировать результаты настройки, сохраним экземпляр `MyStringData` с использованием `SoapFormatter` (соответствующим образом обновив ссылки на сборки и директивы `using`):

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Custom Serialization *****");
    // Вспомните, что этот тип реализует интерфейс ISerializable.
    StringData myData = new StringData();
```

```
// Сохранить экземпляр в локальный файл в формате SOAP.
SoapFormatter soapFormat = new SoapFormatter();
using(Stream fStream = new FileStream("MyData.soap",
    FileMode.Create, FileAccess.Write, FileShare.None))
{
    soapFormat.Serialize(fStream, myData);
}
Console.ReadLine();
}
```

Заглянув внутрь полученного файла *.soap, вы увидите, что строковые поля действительно сохранены в верхнем регистре:

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:clr="http://schemas.microsoft.com/soap/encoding clr/1.0"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>

    <a1:StringData id="ref-1" ...>
        <First_Item id="ref-3">FIRST DATA BLOCK</First_Item>
        <dataItemTwo id="ref-4">MORE DATA</dataItemTwo>
    </a1:StringData>
</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Настройка сериализации с использованием атрибутов

Несмотря на то что реализация интерфейса `ISerializable` представляет собой один из приемов настройки процесса сериализации, предпочтительным способом такой настройки является определение методов, которые оснащены любыми атрибутами, связанными с сериализацией: `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]` или `[OnDeserialized]`. Применение этих атрибутов менее обременительно, чем реализация интерфейса `ISerializable`, т.к. не приходится вручную взаимодействовать с входным параметром `SerializationInfo`. Взамен можно модифицировать данные состояния напрямую, в то время как формater оперирует с типом.

На заметку! Указанные атрибуты сериализации определены в пространстве имен `System.Runtime.Serialization`.

Методы, декорированные указанными атрибутами, должны определяться так, чтобы принимать параметр `StreamingContext` и ничего не возвращать (иначе во время выполнения сгенерируется исключение). Обратите внимание, что использовать каждый атрибут сериализации необязательно; можно просто вмешиваться в те стадии процесса сериализации, которые желательно перехватывать. Сказанное иллюстрируется в следующем фрагменте кода. Здесь новый тип `[Serializable]` устанавливает те же самые требования, что и `StringData`, но на этот раз применяются атрибуты `[OnSerializing]` и `[OnDeserialized]`:

```
[Serializable]
class MoreData
{
    private string dataItemOne = "First data block";
    private string dataItemTwo = "More data";
```

```

[OnSerializing]
private void OnSerializing(StreamingContext context)
{
    // Вызывается в течение процесса сериализации.
    dataItemOne = dataItemOne.ToUpper();
    dataItemTwo = dataItemTwo.ToUpper();
}
[OnDeserialized]
private void OnDeserialized(StreamingContext context)
{
    // Вызывается, когда процесс десериализации завершен.
    dataItemOne = dataItemOne.ToLower();
    dataItemTwo = dataItemTwo.ToLower();
}
}

```

Если вы сериализуете новый тип `MoreData`, то снова обнаружите, что данные сохраняются в верхнем регистре, а восстанавливаются — в нижнем.

Исходный код. Проект `CustomSerialization` доступен в подкаталоге `Chapter_20`.

Приведенным выше примером завершается исследование основных деталей служб сериализации объектов, в том числе разнообразных способов настройки процесса сериализации. Вы видели, что процесс сериализации и десериализации упрощает задачу сохранения крупных объемов данных и может оказаться менее трудоемким, чем работа с различными классами средств чтения/записи из пространства имен `System.IO`.

Резюме

Глава начиналась с демонстрации использования типов `Directory` (`DirectoryInfo`) и `File` (`FileInfo`). Вы узнали, что эти классы позволяют манипулировать физическими файлами и каталогами на жестком диске. Затем вы ознакомились с несколькими классами, производными от абстрактного класса `Stream`. Поскольку производные от `Stream` типы оперируют с низкоуровневым потоком байтов, пространство имен `System.IO` предлагает многочисленные типы средств чтения/записи (например, `StreamWriter`, `StringWriter` и `BinaryWriter`), которые упрощают процесс. Попутно вы взглянули на функциональность типа `DriveType`, научились наблюдать за файлами с применением типа `FileSystemWatcher` и выяснили, каким образом взаимодействовать с потоками в асинхронной манере.

В главе также рассматривались службы сериализации объектов. Вы видели, что платформа .NET использует граф объектов, чтобы учесть полный набор связанных объектов, которые должны сохраняться в потоке. До тех пор, пока каждый член в графе объектов помечен атрибутом `[Serializable]`, данные сохраняются в выбранном формате (двоичном или SOAP).

Вы также ознакомились с возможностями настройки готового процесса сериализации посредством двух подходов. Во-первых, вы узнали, как реализовать интерфейс `ISerializable` (и поддерживать специальный закрытый конструктор), что позволяет вмешиваться в процесс сохранения формateraми предоставленных данных. Во-вторых, вы изучили набор атрибутов .NET, которые упрощают процесс специальной сериализации. Все, что для этого нужно — применить атрибут `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]` или `[OnDeserialized]` к членам, которые принимают параметр `StreamingContext`, после чего формaterы будут вызывать их соответствующим образом.

ГЛАВА 21

Доступ к данным с помощью ADO.NET

Внутри платформы .NET определено несколько пространств имен, которые позволяют взаимодействовать с реляционными базами данных. Все вместе эти пространства имен известны как *ADO.NET*. В настоящей главе вы сначала узнаете об общей роли инфраструктуры ADO.NET, а также основных типов и пространств имен, после чего будет обсуждаться тема поставщиков данных ADO.NET. Платформа .NET поддерживает многочисленные поставщики данных (как являющиеся частью .NET Framework, так и доступные от независимых разработчиков), каждый из которых оптимизирован для взаимодействия с конкретной системой управления базами данных (СУБД), например, Microsoft SQL Server, Oracle, MySQL и т.д.

После освоения общей функциональности, предлагаемой различными поставщиками данных, мы рассмотрим паттерн фабрики поставщиков данных. Вы увидите, что с использованием типов из пространства имен *System.Data.Common* (и связанного файла *App.config*) можно строить единственную кодовую базу, которая способна динамически выбирать поставщик данных без необходимости в повторной компиляции или развертывании кодовой базы приложения.

Далее вы научитесь работать напрямую с поставщиком баз данных SQL Server, создавая и открывая подключения для извлечения данных и затем вставляя, обновляя и удаляя данные, и ознакомитесь с темой транзакций базы данных. Наконец, вы запустите средство массового копирования SQL Server с применением ADO.NET для загрузки списка записей внутрь базы данных.

На заметку! Внимание в настоящей главе концентрируется на низкоуровневой инфраструктуре ADO.NET. В главе 22 раскрывается инфраструктура объектно-реляционного отображения (object-relational mapping — ORM) производства Microsoft под названием Entity Framework (EF). Инфраструктуры ORM вроде Entity Framework значительно упрощают (и ускоряют) процесс создания кода доступа к данным, но для доступа к данным внутренне они по-прежнему задействуют ADO.NET. Хорошее понимание принципов работы ADO.NET жизненно важно при поиске и устранении проблем с доступом к данным, особенно когда код был создан какой-то инфраструктурой, а не написан вами самостоятельно. К тому же вы будете встречать задачи, которые решить с помощью EF не удастся (например, массовое копирование данных в SQL), и для их решения потребуется знание ADO.NET.

Высокоуровневое определение ADO.NET

Если у вас есть опыт работы с предшествующей моделью доступа к данным на основе COM от Microsoft (Active Data Objects — ADO) и вы только начинаете использовать платформу .NET, то имейте в виду, что инфраструктура ADO.NET имеет мало общего с ADO помимо наличия букв *A*, *D* и *O* в своем названии. В то время как определенная взаимосвязь между двумя системами действительно существует (скажем, в обеих присутствует концепция объектов подключений и объектов команд), некоторые знакомые по ADO типы (например, *Recordset*) больше не доступны. Вдобавок вы обнаружите в ADO.NET много новых типов, которые не имеют прямых эквивалентов в классической технологии ADO (скажем, адаптер данных).

На заметку! Предшествующие издания книги содержали главу, посвященную наборам данных ADO.NET, таблицам данных и связанным с ними технологиям. Глава по-прежнему доступна в виде приложения A.

С точки зрения программирования большая часть ADO.NET представлена основной сборкой по имени *System.Data.dll*. Внутри этого двоичного файла находится порядочное количество пространств имен (рис. 21.1), многие из которых представляют типы отдельного поставщика данных ADO.NET.

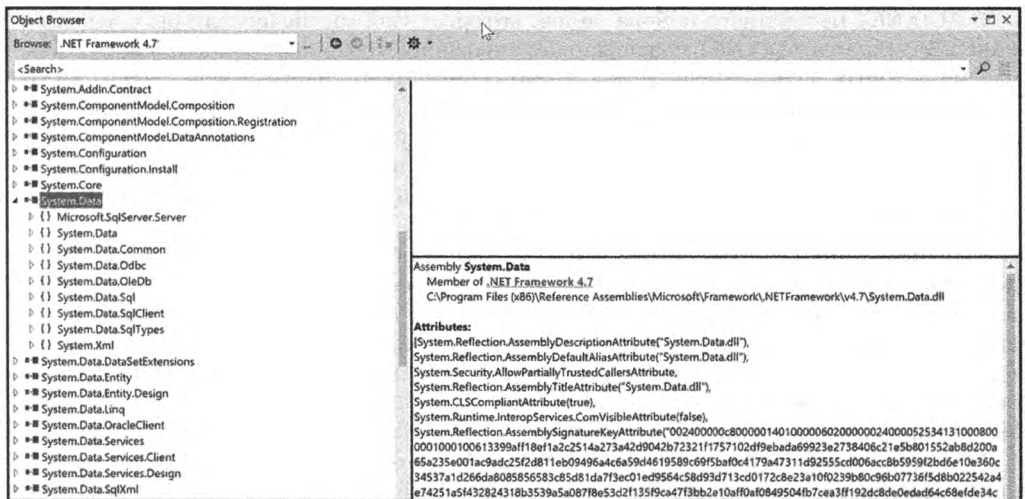


Рис. 21.1. Основной сборкой ADO.NET является *System.Data.dll*

На самом деле большинство шаблонов проектов Visual Studio автоматически ссылаются на указанную ключевую сборку доступа к данным. Вы должны также уяснить, что кроме *System.Data.dll* есть и другие ориентированные на ADO.NET сборки, на которые необходимо вручную ссылаться в текстовом проекте посредством диалогового окна Add Reference (Добавление ссылки).

Три грани ADO.NET

Библиотеки ADO.NET можно применять тремя концептуально отличающимися способами: в подключенном режиме, в автономном режиме и через инфраструктуру ORM, такую как Entity Framework. Когда используется *подключенный уровень* (тема настоя-

щей главы), кодовая база явно подключается к лежащему в основе хранилищу данных и отключается от него. В случае применения ADO.NET в такой манере вы обычно взаимодействуете с хранилищем данных с использованием объектов подключений, объектов команд и объектов чтения данных.

Автономный уровень (рассматриваемый в приложении А) позволяет манипулировать набором объектов DataTable (содержащихся внутри объекта DataSet), который функционирует как копия внешних данных на стороне клиента. Получив объект DataSet, вызывающий код может просматривать и манипулировать его содержимым. Если вызывающему коду нужно отправить изменения в хранилище данных, тогда для обновления источника данных применяется адаптер данных (в сочетании с набором операторов SQL).

Третий способ (раскрываемый в главе 22) предусматривает использование средства ORM, подобного NHibernate или Entity Framework. Средства ORM применяют объекты C#, представляющие данные в ориентированной на приложение манере, и позволяют разработчикам абстрагироваться от большого объема кода доступа к данным. Инфраструктура EF также предоставляет возможность взаимодействия с реляционными базами данных, используя строго типизированные запросы LINQ, которые динамически создают запросы, специфические для баз данных.

Поставщики данных ADO.NET

В ADO.NET нет единого набора типов, которые взаимодействовали бы с множеством СУБД. Взамен ADO.NET поддерживает многочисленные *поставщики данных*, каждый из которых оптимизирован для взаимодействия со специфичной СУБД. Первое преимущество такого подхода в том, что вы можете запрограммировать специализированный поставщик данных для доступа к любым уникальным средствам отдельной СУБД. Второе преимущество связано с тем, что поставщик данных может подключаться непосредственно к механизму интересующей СУБД без какого-либо промежуточного уровня отображения.

Выражаясь просто, поставщик данных — это набор типов, определенных в отдельном пространстве имен, которым известно, как взаимодействовать с конкретным источником данных. Безотносительно к тому, какой поставщик данных применяется, каждый из них определяет набор классов, предоставляющих основную функциональность. В табл. 21.1 описаны некоторые распространенные основные типы, их базовые классы (определенные в пространстве имен System.Data.Common) и ключевые интерфейсы (определенные в пространстве имен System.Data), которые они реализуют.

Хотя конкретные имена основных классов будут отличаться между поставщиками данных (например, SqlConnection в сравнении с OdbcConnection), все они являются производными от того же самого базового класса (DbConnection в случае объектов подключения), реализующего идентичные интерфейсы (вроде IDbConnection). С учетом сказанного вполне корректно предположить, что после освоения работы с одним поставщиком данных остальные поставщики покажутся довольно простыми.

На заметку! Когда речь идет об объекте подключения в ADO.NET, то на самом деле имеется в виду специфичный тип, производный от DbConnection; не существует класса с буквальным именем *Connection*. Та же идея остается справедливой в отношении объекта команды, объекта адаптера данных и т.д. По соглашению об именовании объекты в конкретном поставщике данных снабжаются префиксом в форме названия связанной СУБД (например, SqlConnection, SqlCommand и SqlDataReader).

Таблица 21.1. Основные типы поставщиков данных ADO.NET

Тип	Базовый класс	Реализуемые интерфейсы	Описание
Connection	DbConnection	IDbConnection	Предоставляет возможность подключения и отключения от хранилища данных. Объекты подключения также обеспечивают доступ к связанному объекту транзакции
Command	DbCommand	IDbCommand	Представляет запрос SQL или хранимую процедуру. Объекты команд также предоставляют доступ к объекту чтения данных поставщика
DataReader	DbDataReader	IDataReader, IDataRecord	Предоставляет доступ к данным в направлении только вперед, допускающий только чтение, с использованием курсора на стороне сервера
DataAdapter	DbDataAdapter	IDataAdapter, IDbDataAdapter	Передаёт объекты DataSet между вызывающим кодом и хранилищем данных. Адаптеры данных содержат объект подключения и набор из четырех внутренних объектов команд для выборки, вставки, изменения и удаления информации из хранилища данных
Parameter	DbParameter	IDataParameter, IDbDataParameter	Представляет именованный параметр внутри параметризованного запроса
Transaction	DbTransaction	IDbTransaction	Инкапсулирует транзакцию базы данных

На рис. 21.2 иллюстрируется место поставщиков данных в инфраструктуре ADO.NET. На показанной диаграмме клиентская сборка может быть приложением .NET любого типа: консольной программой, приложением Windows Forms, приложением WPF, веб-страницей ASP.NET, службой WCF, службой Web API, библиотекой кода .NET и т.д.

Кроме типов, показанных на рис. 21.2, поставщики данных будут предоставлять и другие типы; однако эти основные объекты определяют общие характеристики для всех поставщиков данных.

Поставщики данных ADO.NET производства Microsoft

В состав дистрибутива .NET от Microsoft входят многочисленные поставщики данных, включая поставщики для SQL Server и подключений в стиле OLE DB/ODBC. В табл. 21.2 описаны пространства имен и содержащие их сборки для всех поставщиков данных Microsoft ADO.NET.

Таблица 21.2. Поставщики данных Microsoft ADO.NET

Поставщик данных	Пространство имен	Сборка
OLE DB	System.Data.OleDb	System.Data.dll
Microsoft SQL Server LocalDb	System.Data.SqlClient	System.Data.dll
ODBC	System.Data.Odbc	System.Data.dll

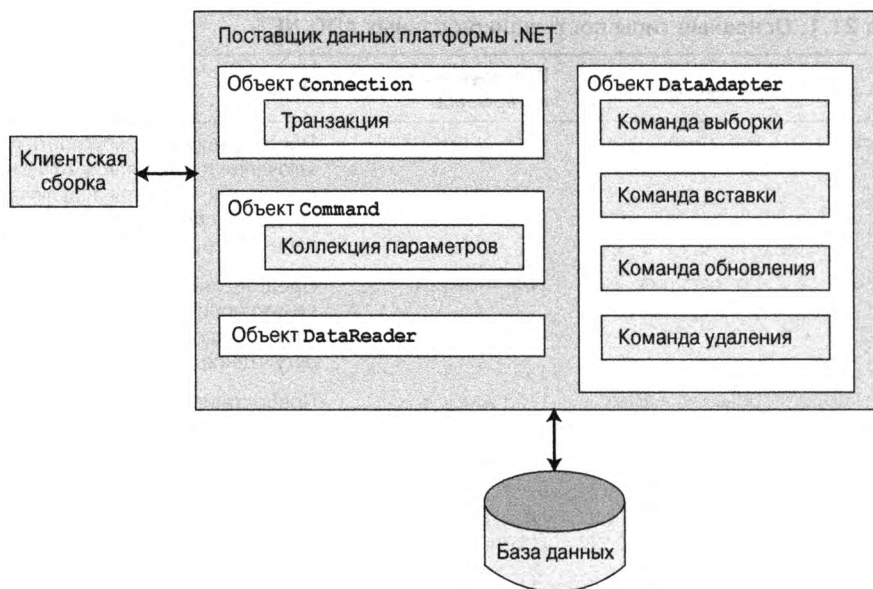


Рис. 21.2. Поставщики данных ADO.NET предоставляют доступ к конкретным СУБД

На заметку! В случае необходимости подключения к базе данных Oracle придется применять инструменты разработчика Oracle для Visual Studio (Oracle Developer Tools for Visual Studio), поставляемые компанией Oracle. Если вы откроете окно проводника серверов (Server Explorer) и выберете элемент New Connection (Новое подключение), а затем Oracle Database (База данных Oracle), то Visual Studio предоставит ссылку, по которой эти инструменты можно загрузить. В текущий момент ссылка выглядит как <http://www.oracle.com/technetwork/topics/dotnet/whatsnew/vs2012welcome-1835382.html>.

Не существует специального поставщика данных, который бы отображался прямо на механизм Jet (используемый Microsoft Access). Если нужно взаимодействовать с файлом данных Access, тогда можно применять поставщик данных OLE DB или ODBC.

Поставщик данных OLE DB, который образован из типов, определенных в пространстве имен `System.Data.OleDb`, позволяет обращаться к данным, находящимся в любом хранилище данных, если оно поддерживает классический основанный на COM протокол OLE DB. Этот поставщик можно использовать для взаимодействия с любой базой данных, совместимой с OLE DB, просто подстраивая сегмент `Provider` строки подключения.

Тем не менее, "за кулисами" поставщик OLE DB взаимодействует с разнообразными объектами COM, что может повлиять на производительность приложения. В общем и целом поставщик данных OLE DB удобен, только если вы взаимодействуете с СУБД, для которой не определен специфичный поставщик данных .NET. Однако, с учетом того, что в наши дни любая стоящая СУБД должна предлагать для загрузки специальный поставщик данных ADO.NET, пространство имен `System.Data.OleDb` следует считать устаревшим и малопригодным в мире .NET 4.7. (Сказанное стало еще более справедливым с появлением в .NET 2.0 модели фабрики поставщиков данных, о которой вы вскоре узнаете.)

На заметку! Есть один случай, когда необходимо применять типы из `System.Data.OleDb`: если нужно взаимодействовать с базами данных Microsoft SQL Server 6.5 или более ранней версии. Пространство имен `System.Data.SqlClient` допускает взаимодействие только с Microsoft SQL Server 7.0 и последующих версий.

Поставщик данных Microsoft SQL Server предлагает прямой доступ к хранилищам данных Microsoft SQL Server (7.0 и последующих версий) — и только к ним. Пространство имен `System.Data.SqlClient` содержит типы, используемые поставщиком SQL Server, и обеспечивает ту же базовую функциональность, что и поставщик OLE DB. Главная разница между ними заключается в том, что поставщик SQL Server обходит уровень OLE DB и дает существенный выигрыш в производительности. Поставщик данных Microsoft SQL Server также позволяет получить доступ к уникальным возможностям указанной конкретной СУБД.

Оставшийся поставщик от Microsoft (`System.Data.Odbc`) обеспечивает доступ к подключениям ODBC. Типы ODBC, определенные в пространстве имен `System.Data.Odbc`, обычно полезны, только если требуется взаимодействие с СУБД, для которой отсутствует специальный поставщик данных .NET. Причина в том, что ODBC является широко распространенной моделью, которая предоставляет доступ к нескольким хранилищам данных.

Получение сторонних поставщиков данных ADO.NET

Кроме поставщиков данных, предлагаемых Microsoft (а также специальной библиотеки .NET от Oracle), существуют многочисленные сторонние поставщики данных для разнообразных баз данных с открытым кодом и коммерческих баз данных. Хотя у вас наверняка будет возможность получить поставщик данных ADO.NET непосредственно у разработчика СУБД, вы должны знать о следующем веб-сайте:

<https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/ado-net-overview>

Это один из множества веб-сайтов, где документируются все известные поставщики данных ADO.NET и указываются ссылки для получения дополнительных сведений и загрузки. Здесь вы найдете множество поставщиков ADO.NET, в том числе для SQLite, IBM DB2, MySQL, Postgres, Sybase и т.д.

Несмотря на наличие большого количества поставщиков данных ADO.NET, в примерах, приводимых в книге, будет использоваться поставщик данных Microsoft SQL Server (`System.Data.SqlClient.dll`). Вспомните, что он позволяет взаимодействовать с Microsoft SQL Server 7.0 и последующих версий, включая SQL Server Express и LocalDb. Если вы намерены применять ADO.NET для работы с другой СУБД, то проблем возникнуть не должно при условии, что вы уясните изложенный ниже материал.

Дополнительные пространства имен ADO.NET

В дополнение к пространствам имен .NET, где определены типы специфических поставщиков данных, библиотеки базовых классов .NET предлагают несколько добавочных пространств имен, относящихся к ADO.NET; некоторые из них описаны в табл. 21.3.

Мы не собираемся исследовать каждый отдельный тип внутри абсолютно всех пространств имен ADO.NET (такая задача потребовала бы написания полноценной книги), но очень важно, чтобы вы понимали типы в пространстве имен `System.Data`.

Таблица 21.3. Избранные дополнительные пространства имен, связанные с ADO.NET

Пространство имен	Описание
Microsoft.SqlServer.Server	Предоставляет типы, которые содействуют работе со службами интеграции CLR и SQL Server 2005 и последующих версий
System.Data	Определяет основные типы ADO.NET, используемые всеми поставщиками данных, в том числе общие интерфейсы и многочисленные типы, которые представляют автономный уровень (например, DataSet и DataTable)
System.Data.Common	Содержит типы, которые разделяются всеми поставщиками ADO.NET, включая общие абстрактные базовые классы
System.Data.Sql	Предоставляет типы, которые позволяют обнаружить экземпляры Microsoft SQL Server, установленные в текущей локальной сети
System.Data.SqlTypes	Определяет собственные типы данных, применяемые Microsoft SQL Server. Вы всегда можете использовать соответствующие типы данных CLR, но типы в пространстве имен SqlTypes оптимизированы для работы с SQL Server (скажем, если база данных SQL Server содержит целочисленное значение, то его можно представить с применением либо int, либо SqlTypes.SqlInt32)

Типы из пространства имен System.Data

Из всех пространств имен, относящихся к ADO.NET, System.Data является "наименьшим общим знаменателем". Не указав это пространство имен, вы не сумеете построить приложение ADO.NET с доступом к данным. Оно содержит типы, которые совместно используются всеми поставщиками данных ADO.NET вне зависимости от лежащего в основе хранилища данных. В дополнение к нескольким исключениям, связанным с базами данных (например, NonNullAllowedException, RowNotInTableException и MissingPrimaryKeyException), пространство имен System.Data содержит типы, которые представляют разнообразные примитивы баз данных (вроде таблиц, строк, столбцов и ограничений), а также общие интерфейсы, реализуемые классами поставщиков данных. В табл. 21.4 описаны основные типы, о которых следует знать.

Таблица 21.4. Основные типы пространства имен System.Data

Тип	Описание
Constraint	Представляет ограничение для заданного объекта DataColumn
DataColumn	Представляет одиночный столбец внутри объекта DataTable
DataRelation	Представляет отношение "родительский-дочерний" между двумя объектами DataTable
DataRow	Представляет одиночную строку внутри объекта DataTable
DataSet	Представляет находящийся в памяти кеш данных, который состоит из любого количества взаимосвязанных объектов DataTable
DataTable	Представляет табличный блок данных, находящийся в памяти
DataTableReader	Позволяет трактовать объект DataTable как допускающий только чтение курсор для доступа к данным в прямом направлении

Тип	Описание
DataView	Определяет настраиваемое представление DataTable для сортировки, фильтрации, поиска, редактирования и навигации
IDataAdapter	Определяет основное поведение объекта адаптера данных
IDataParameter	Определяет основное поведение объекта параметра
IDataReader	Определяет основное поведение объекта чтения данных
IDbCommand	Определяет основное поведение объекта команды
IDbDataAdapter	Расширяет интерфейс IDataAdapter для обеспечения дополнительной функциональности объекту адаптера данных
IDbTransaction	Определяет основное поведение объекта транзакции

Следующей задачей будет исследование основных интерфейсов System.Data на высоком уровне, что поможет лучше понять общую функциональность, предлагаемую любым поставщиком данных. В ходе чтения настоящей главы вы также ознакомитесь с конкретными деталями, а пока лучше сосредоточить внимание на общем поведении каждого типа интерфейса.

Роль интерфейса IDbConnection

Интерфейс IDbConnection реализован *объектом подключения* поставщика данных. В нем определен набор членов, применяемых для конфигурирования подключения к специфичному хранилищу данных. Он также позволяет получить объект транзакции поставщика данных. Вот формальное определение IDbConnection:

```
public interface IDbConnection : IDisposable
{
    string ConnectionString { get; set; }
    int ConnectionTimeout { get; }
    string Database { get; }
    ConnectionState State { get; }

    IDbTransaction BeginTransaction();
    IDbTransaction BeginTransaction(IsolationLevel il);
    void ChangeDatabase(string databaseName);
    void Close();
    IDbCommand CreateCommand();
    void Open();
}
```

На заметку! Как и во многих других типах из библиотек базовых классов .NET, вызов метода Close() функционально эквивалентен прямому или косвенному вызову метода Dispose() внутри области using (см. главу 13).

Роль интерфейса IDbTransaction

Перегруженный метод BeginTransaction(), определенный в интерфейсе IDbConnection, предоставляет доступ к *объекту транзакции* поставщика. Члены, определенные интерфейсом IDbTransaction, позволяют программно взаимодействовать с транзакционным сеансом и лежащим в основе хранилищем данных:


```
public interface IDbTransaction : IDisposable
{
    IDbConnection Connection { get; }
    IsolationLevel IsolationLevel { get; }

    void Commit();
    void Rollback();
}
```

Роль интерфейса IDbCommand

Интерфейс `IDbCommand` будет реализован объектом команды поставщика данных. Подобно другим объектным моделям доступа к данным объекты команд позволяют программно манипулировать с операторами SQL, хранимыми процедурами и параметризованными запросами. Объекты команд также обеспечивают доступ к типу чтения данных поставщика данных посредством перегруженного метода `ExecuteReader()`:

```
public interface IDbCommand : IDisposable
{
    IDbConnection Connection { get; set; }
    IDbTransaction Transaction { get; set; }
    string CommandText { get; set; }
    int CommandTimeout { get; set; }
    CommandType CommandType { get; set; }
    IDataParameterCollection Parameters { get; }
    UpdateRowSource UpdatedRowSource { get; set; }

    void Prepare();
    void Cancel();
    IDbDataParameter CreateParameter();
    int ExecuteNonQuery();
    IDataReader ExecuteReader();
    IDataReader ExecuteReader(CommandBehavior behavior);
    object ExecuteScalar();
}
```

Роль интерфейсов IDbDataParameter и IDataParameter

Обратите внимание, что свойство `Parameters` интерфейса `IDbCommand` возвращает строго типизированную коллекцию, реализующую интерфейс `IDataParameterCollection`, который предоставляет доступ к набору классов, совместимых с `IDbDataParameter` (например, объектам параметров):

```
public interface IDbDataParameter : IDataParameter
{
    byte Precision { get; set; }
    byte Scale { get; set; }
    int Size { get; set; }
}
```

Интерфейс `IDbDataParameter` расширяет `IDataParameter` с целью обеспечения дополнительных линий поведения:

```
public interface IDataParameter
{
    DbType DbType { get; set; }
    ParameterDirection Direction { get; set; }
    bool IsNullable { get; }
    string ParameterName { get; set; }
```

```

string SourceColumn { get; set; }
DataRowVersion SourceVersion { get; set; }
object Value { get; set; }
}

```

Вы увидите, что функциональность интерфейсов `IDbDataParameter` и `IDataParameter` позволяет представлять параметры внутри команды SQL (включая хранимые процедуры) с помощью специфических объектов параметров ADO.NET вместо жестко закодированных строковых литералов.

Роль интерфейсов `IDbDataAdapter` и `IDataAdapter`

Адаптеры данных будут использоваться для помещения и извлечения объектов `DataSet` в и из хранилища данных. Интерфейс `IDbDataAdapter` определяет следующий набор свойств, которые можно применять для поддержки операторов SQL, выполняющих связанные операции выборки, вставки, обновления и удаления:

```

public interface IDbDataAdapter : IDataAdapter
{
    IDbCommand SelectCommand { get; set; }
    IDbCommand InsertCommand { get; set; }
    IDbCommand UpdateCommand { get; set; }
    IDbCommand DeleteCommand { get; set; }
}

```

В дополнение к показанным четырем свойствам адаптер данных ADO.NET также получает поведение, определенное базовым интерфейсом, т.е. `IDataAdapter`. Интерфейс `IDataAdapter` определяет ключевую функцию типа адаптера данных: способность передавать объекты `DataSet` между вызывающим кодом и внутренним хранилищем данных, используя методы `Fill()` и `Update()`. Кроме того, интерфейс `IDataAdapter` позволяет с помощью свойства `TableMappings` сопоставлять имена столбцов базы данных с более дружественными к пользователю отображаемыми именами:

```

public interface IDataAdapter
{
    MissingMappingAction MissingMappingAction { get; set; }
    MissingSchemaAction MissingSchemaAction { get; set; }
    ITableMappingCollection TableMappings { get; }

    DataTable[] FillSchema(DataSet dataSet, SchemaType schemaType);
    int Fill(DataSet dataSet);
    IDataParameter[] GetFillParameters();
    int Update(DataSet dataSet);
}

```

Роль интерфейсов `IDataReader` и `IDataRecord`

Следующим основным интерфейсом является `IDataReader`, который представляет общие линии поведения, поддерживаемые отдельно взятым объектом чтения данных. После получения от поставщика данных ADO.NET объекта совместимого с `IDataReader` типа можно выполнять проход по результирующему набору в прямом направлении с поддержкой только чтения.

```

public interface IDataReader : IDisposable, IDataRecord
{
    int Depth { get; }
    bool IsClosed { get; }
    int RecordsAffected { get; }
}

```

```

void Close();
DataTable GetSchemaTable();
bool NextResult();
bool Read();
}

```

Наконец, интерфейс `IDataReader` расширяет `IDataRecord`. В интерфейсе `IDataRecord` определено много членов, которые позволяют извлекать из потока строго типизированное значение, а не приводить к нужному типу экземпляр `System.Object`, полученный из перегруженного метода индексируемого объекта чтения данных. Вот определение интерфейса `IDataRecord`:

```

public interface IDataRecord
{
    int FieldCount { get; }
    object this[ int i ] { get; }
    object this[ string name ] { get; }
    string GetName(int i);
    string GetDataTypeName(int i);
    Type GetFieldType(int i);
    object GetValue(int i);
    int GetValues(object[] values);
    int GetOrdinal(string name);
    bool GetBoolean(int i);
    byte GetByte(int i);
    long GetBytes(int i, long fieldOffset, byte[] buffer, int bufferoffset, int length);
    char GetChar(int i);
    long GetChars(int i, long fieldoffset, char[] buffer, int bufferoffset, int length);
    Guid GetGuid(int i);
    short GetInt16(int i);
    int GetInt32(int i);
    long GetInt64(int i);
    float GetFloat(int i);
    double GetDouble(int i);
    string GetString(int i);
    Decimal GetDecimal(int i);
    DateTime GetDateTime(int i);
    IDataReader GetData(int i);
    bool IsDBNull(int i);
}

```

На заметку! Метод `IDataReader.IsDBNull()` можно применять для программного выяснения, установлено ли указанное поле в `null`, прежде чем получать значение из объекта чтения данных (во избежание генерации исключения во время выполнения). Также вспомните, что язык C# поддерживает типы данных, допускающие `null` (см. главу 4), идеально подходящие для взаимодействия со столбцами, которые могут иметь значение `null` в таблице базы данных.

Абстрагирование поставщиков данных с использованием интерфейсов

К настоящему моменту вы должны лучше понимать общую функциональность, присущую всем поставщикам данных .NET. Вспомните, что хотя точные имена реализуемых типов будут отличаться между поставщиками данных, в коде такие типы применяются в схожей манере — в том и заключается преимущество полиморфизма на основе интер-

фейсов. Скажем, если определить метод, который принимает параметр `IDbConnection`, то ему можно передавать любой объект подключения ADO.NET:

```
public static void OpenConnection(IDbConnection connection)
{
    // Открыть входное подключение для вызывающего кода.
    connection.Open();
}
```

На заметку! Использовать интерфейсы вовсе не обязательно; аналогичного уровня абстракции можно достичь путем применения абстрактных базовых классов (таких как `DbConnection`) в качестве параметров или возвращаемых значений.

То же самое справедливо для возвращаемых значений. Например, рассмотрим простой проект консольного приложения C# (по имени `MyConnectionFactory`), которое позволяет выбирать специфический объект подключения на основе значения из специального перечисления. В целях диагностики мы просто выводим лежащий в основе объект подключения с использованием служб рефлексии:

```
using System;
using static System.Console;
// Эти пространства имен нужны для получения определений
// общих интерфейсов и разнообразных объектов подключений.
using System.Data;
using System.Data.SqlClient;
using System.Data.Odbc;
using System.Data.OleDb;

namespace MyConnectionFactory
{
    // Список возможных поставщиков.
    enum DataProvider
    {
        SqlServer, OleDb, Odbc, None
    }

    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("**** Very Simple Connection Factory *****\n");
            // Получить конкретное подключение.
            IDbConnection myConnection = GetConnection(DataProvider.SqlServer);
            WriteLine($"Your connection is a {myConnection.GetType().Name}");
            // Открыть, использовать и закрыть подключение...
            ReadLine();
        }

        // Этот метод возвращает конкретный объект подключения
        // на основе значения перечисления DataProvider.
        static IDbConnection GetConnection(DataProvider dataProvider)
        {
            IDbConnection connection = null;
            switch (dataProvider)
            {
                case DataProvider.SqlServer:
                    connection = new SqlConnection();
                    break;
            }
        }
    }
}
```

```

        case DataProvider.OleDb:
            connection = new OleDbConnection();
            break;
        case DataProvider.Odbc:
            connection = new OdbcConnection();
            break;
    }
    return connection;
}
}
}

```

На заметку! В Visual Studio 2015 появилась директива `using static`. После добавления `using static System.Console;` к другим операторам `using` можно записывать `WriteLine("some text")` вместо `Console.WriteLine("some text")`. Для всех консольных проектов в этой и последующих главах будет применяться такая более короткая версия за счет помещения в начало файлов кода оператора `using static System.Console;`.

Преимущество работы с общими интерфейсами из пространства имен `System.Data` (или на самом деле с абстрактными базовыми классами из пространства имен `System.Data.Common`) связано с более высокими шансами построить гибкую кодовую базу, которую со временем можно развивать. Например, в настоящий момент вы можете разрабатывать приложение, предназначенное для Microsoft SQL Server; тем не менее, вполне возможно, что спустя несколько месяцев ваша компания перейдет на другую СУБД. Если вы строите решение с жестко закодированными типами из пространства имен `System.Data.SqlClient`, которые специфичны для Microsoft SQL Server, тогда вполне очевидно, что в случае смены серверной СУБД сборку придется редактировать, заново компилировать и развертывать.

Повышение гибкости с использованием конфигурационных файлов приложения

Для повышения гибкости приложений ADO.NET на стороне клиента можно предусмотреть файл `*.config`, в котором имеется элемент `<appSettings>`, содержащий специальные пары “ключ-значение”. Вспомните из главы 14, что специальные данные, хранящиеся в файле `*.config`, можно получать программно с применением типов из пространства имен `System.Configuration`. Предположим, что внутри конфигурационного файла указано следующее значение поставщика данных:

```

<configuration>
  <appSettings>
    <!-- Это значение ключа отображается на одно из значений перечисления. -->
    <add key="provider" value="SqlServer"/>
  </appSettings>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6"/>
  </startup>
</configuration>

```

Теперь можно обновить метод `Main()`, чтобы программно получить объект поставщика данных. По существу тем самым создается *фабрика объектов подключений*, которая позволяет изменять объект поставщика без необходимости в повторной компиляции кодовой базы (просто модифицируется файл `*.config`). Ниже показаны необходимые изменения в `Main()`:

```

static void Main(string[] args)
{
    WriteLine("**** Very Simple Connection Factory ****\n");
    // Прочитать ключ provider.
    string dataProviderString = ConfigurationManager.AppSettings["provider"];
    // Преобразовать строку в перечисление.
    DataProvider dataProvider = DataProvider.None;
    if (Enum.IsDefined(typeof (DataProvider), dataProviderString))
    {
        dataProvider =
            (DataProvider) Enum.Parse(typeof (DataProvider), dataProviderString);
    }
    else
    {
        WriteLine("Sorry, no provider exists!"); // Поставщики отсутствуют
        ReadLine();
        return;
    }
    // Получить конкретное подключение.
    IDbConnection myConnection = GetConnection(dataProvider);
    WriteLine($"Your connection is a {myConnection?.GetType().Name}
        ?? "unrecognized type"");
    // Открыть, использовать и закрыть подключение...
    ReadLine();
}

```

На заметку! Чтобы можно было использовать тип `ConfigurationManager`, необходимо установить ссылку на сборку `System.Configuration.dll` и импортировать пространство имен `System.Configuration`.

К настоящему моменту вы построили код ADO.NET, который позволяет динамически указывать лежащее в основе подключение. Однако здесь присутствует одна очевидная проблема: такая абстракция применяется только внутри приложения `MyConnectionFactory.exe`. Если переделать пример в виде библиотеки кода .NET (скажем, `MyConnectionFactory.dll`), то появилась бы возможность создавать любое количество клиентов, которые могли бы получать разнообразные объекты подключений, используя уровни абстракции.

Тем не менее, получение объекта подключения — лишь один аспект работы с ADO.NET. Чтобы построить полезную библиотеку фабрики поставщиков данных, необходимо также учитывать объекты команд, объекты чтения данных, адаптеры данных, объекты транзакций и другие типы, связанные с данными. Создание подобной библиотеки кода не обязательно будет трудным, но все-таки потребует написания значительного объема кода и затрат времени.

Начиная с версии .NET 2.0, такая функциональность встроена прямо в библиотеки базовых классов .NET. Вскоре мы исследуем этот формальный API-интерфейс, но сначала понадобится создать специальную базу данных для применения в настоящей главе (и во многих последующих главах).

Исходный код. Проект `MyConnectionFactory` доступен в подкаталоге `Chapter_21`.

Создание базы данных AutoLot

На протяжении оставшегося материала главы мы будем выполнять запросы в отношении простой тестовой базы данных SQL Server по имени AutoLot. В продолжение автомобильной темы, затрагиваемой повсеместно в книге, база данных AutoLot будет содержать три взаимосвязанных таблицы (Inventory, Orders и Customers), которые хранят различные данные о заказах гипотетической компании по продаже автомобилей.

На заметку! В предшествующих изданиях книги для проектирования и создания таблиц и хранимых процедур, а также для других действий с базой данных использовалась IDE-среда Visual Studio. Хотя ее можно по-прежнему применять, бесплатный инструмент SQL Server Management Studio (SSMS) более эффективен, поэтому в настоящем издании он будет использоваться вместо Visual Studio.

Установка SQL Server 2016 и SQL Server Management Studio

В книге предполагается наличие у вас копии Microsoft SQL Server 2016 (любой редакции). Вместе с Visual Studio 2017 устанавливается специальный экземпляр по имени (localdb)\mssqllocaldb, который будет применяться далее в книге. При наличии установленной копии другой редакции (скажем, Express) можете использовать ее, но только соответствующим образом измените строку подключения. Также рекомендуется загрузить и установить инструмент SQL Server Management Studio (SSMS) 17 (или последующей версии), доступный по адресу:

<https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms>

Продукт Microsoft SQL Server Express 2016 является бесплатным при проработке примеров, приведенных в этой книге, и доступен для загрузки по следующему адресу:

<https://www.microsoft.com/en-us/sql-server/sql-server-editions-express>

Исходный код. Резервная копия базы данных SQL Server (autolot.bak) доступна в подкаталоге Chapter_21.

Создание таблицы Inventory

Чтобы приступить к построению тестовой базы данных, необходимо запустить инструмент SSMS. Он предложит ввести имя сервера. Введем (localdb)\mssqllocaldb (вы можете ввести свою строку подключения) и щелкнем на кнопке Connect (Подключиться), как показано на рис. 21.3.

В открывшемся окне Object Explorer (Проводник объектов) инструмента SSMS щелкнем правой кнопкой мыши на узле Databases (Базы данных) и выберем в контекстном меню пункт New Database (Создать базу данных). В открывшемся диалоговом окне введем AutoLot в качестве имени базы данных и оставим без изменений все остальные настройки (рис. 21.4). Щелкнем на кнопке ОК для создания базы данных.

Пока что база данных AutoLot не содержит какие-либо объекты (таблицы, хранимые процедуры и т.д.). Чтобы добавить новую таблицу, развернем узел Databases, затем узел AutoLot, щелкнем правой кнопкой мыши на узле Tables (Таблицы) и выберем в контекстном меню пункт Table (Таблица), как демонстрируется на рис. 21.5.

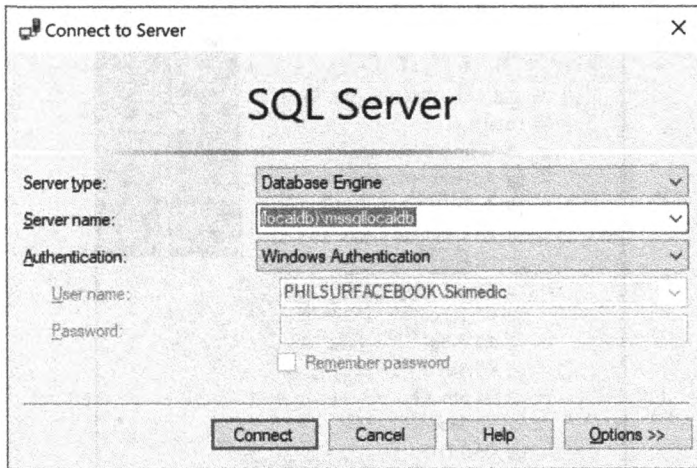


Рис. 21.3. Подключение к серверу с помощью SQL Server Management Studio 17

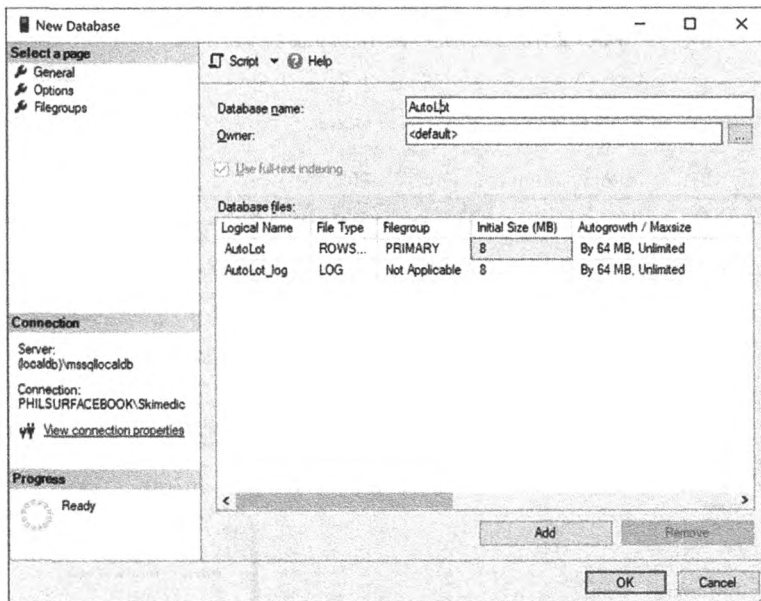


Рис. 21.4. Создание новой базы данных SQL Server

Посредством редактора таблиц добавим четыре столбца (CarId (идентификатор автомобиля), Make (модель), Color (цвет) и PetName (дружественное имя)). Укажем для столбца CarId тип int, а для остальных — тип nvarchar(50). Необходимо обеспечить установку столбца CarId в качестве первичного ключа (щелкнув правой кнопкой мыши на CarId и выбрав в контекстном меню пункт Set Primary Key (Установить первичный ключ)) и описания идентичности (указав CarId в поле Identity Column (Столбец идентичности) на вкладке Properties (Свойства)). Изменим имя таблицы на Inventory. Наконец, щелкнем на значке Save (Сохранить) для сохранения таблицы в базе данных. Финальные параметры таблицы представлены на рис. 21.6.

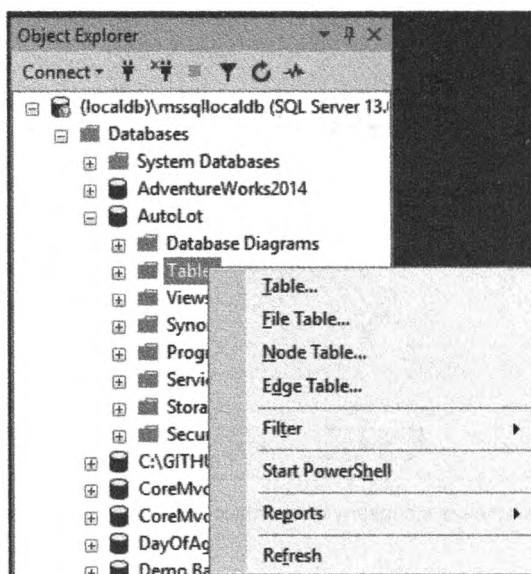


Рис. 21.5. Добавление таблицы Inventory

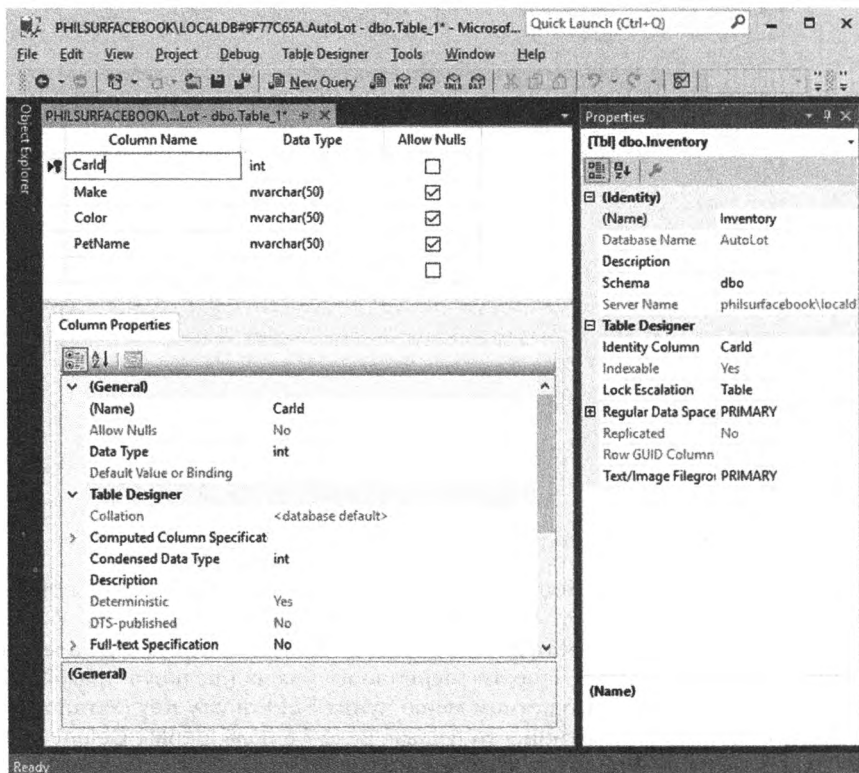
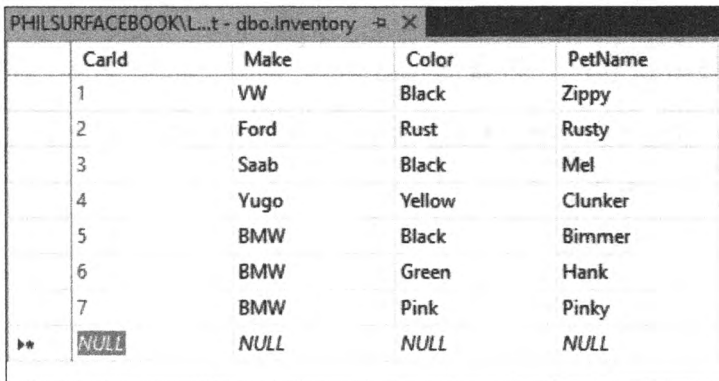


Рис. 21.6. Проектирование таблицы Inventory

Добавление тестовых записей в таблицу Inventory

Чтобы добавить записи в нашу первую таблицу, развернем узел Tables, щелкнем правой кнопкой мыши на таблице Inventory и выберем в контекстном меню пункт Edit Top 200 Rows (Редактировать первые 200 строк). Если таблица Inventory не видна, тогда нужно щелкнуть на значке Refresh (Обновить) в окне Object Explorer. Введем данные о нескольких новых автомобилях (для интереса можно сделать так, чтобы у некоторых автомобилей были одинаковые цвета и модели). Не следует забывать, что поле CarId является столбцом идентичности, поэтому база данных позаботится о создании для него уникальных значений. На рис. 21.7 приведен возможный список инвентарных записей.



	CarId	Make	Color	PetName
	1	VW	Black	Zippy
	2	Ford	Rust	Rusty
	3	Saab	Black	Mel
	4	Yugo	Yellow	Clunker
	5	BMW	Black	Bimmer
	6	BMW	Green	Hank
	7	BMW	Pink	Pinky
»	NULL	NULL	NULL	NULL

Рис. 21.7. Наполнение таблицы Inventory

Создание хранимой процедуры GetPetName ()

Позже в главе вы научитесь применять ADO.NET для вызова хранимых процедур. Как вам уже может быть известно, хранимые процедуры представляют собой подпрограммы, находящиеся внутри базы данных, которые что-то делают. Подобно методам C# хранимые процедуры могут возвращать данные или просто выполнять операции над данными, ничего не возвращая. Мы добавим в базу данных единственную хранимую процедуру, которая будет возвращать дружественное имя автомобиля на основе предоставленного значения CarId. Развернем узел Programmability (Программируемость) в базе данных AutoLot, щелкнем правой кнопкой мыши на узле Stored Procedures (Хранимые процедуры) и выберем в контекстном меню пункт Stored Procedure (Хранимая процедура). Очистим загруженный шаблон и введем следующий код:

```
CREATE PROCEDURE GetPetName
    @carID int,
    @petName char(10) output
AS
SELECT @petName = PetName from Inventory where CarId = @carID
```

На заметку! Хранимые процедуры не обязаны возвращать данные, используя выходные параметры, как сделано здесь; однако, поступая подобным образом, мы формируем почву для обсуждения материалов, приводимых далее в главе.

Щелкнем на значке Execute (Выполнить) или нажмем клавишу <F5>, чтобы создать хранимую процедуру.

Создание таблиц Customers и Orders

База данных AutoLot нуждается в двух дополнительных таблицах: Customers (клиенты) и Orders (заказы). Таблица Customers будет хранить список клиентов и содержать три столбца: CustId (идентификатор клиента; должен быть установлен в качестве первичного ключа), FirstName (имя) и LastName (фамилия). Создать таблицу Customers можно, следуя тем же самым шагам, которые выполнялись при создании таблицы Inventory, или применив T-SQL (язык “программирования” для SQL Server). Создадим новый запрос T-SQL, щелкнув на кнопке New Query (Создать запрос). Удостоверимся в том, что указана база данных AutoLot (рис. 21.8).

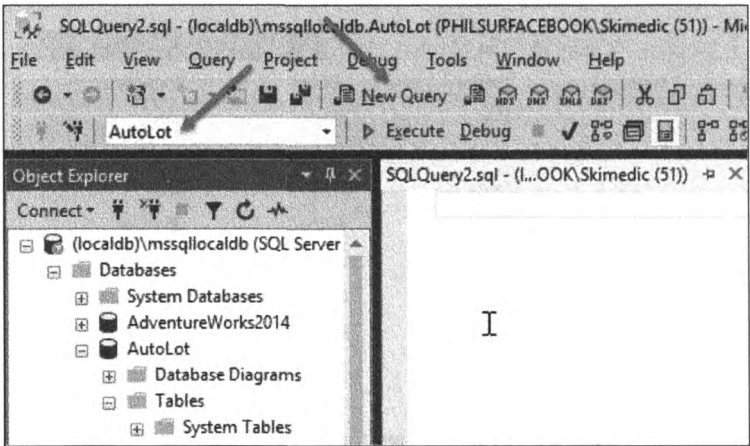


Рис. 21.8. Создание нового запроса для построения таблицы Customers

Введем показанный ниже текст и щелкнем на значке Execute, чтобы создать таблицу:

```
CREATE TABLE [dbo].[Customers]
(
    [CustID] INT IDENTITY(1, 1) NOT NULL ,
    [FirstName] NVARCHAR(50) NULL ,
    [LastName] NVARCHAR(50) NULL ,
    PRIMARY KEY CLUSTERED ( [CustID] ASC )
);
```

После создания таблицы добавим в нее несколько записей о клиентах (рис. 21.9).

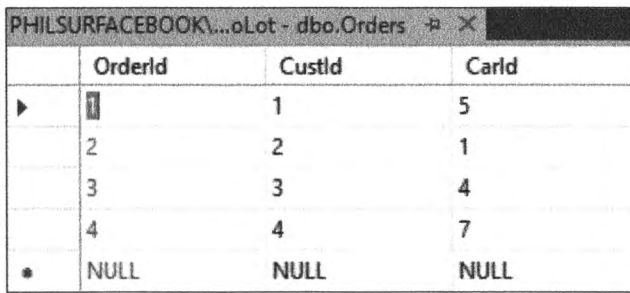
	CustID	FirstName	LastName
▶	1	Dave	Brenner
	2	Matt	Walton
	3	Steve	Hagen
	4	Pat	Walton
*	NULL	NULL	NULL

Рис. 21.9. Наполнение таблицы Customers

Финальная таблица, `Orders`, будет применяться для представления автомобиля, который клиент планирует приобрести. Создадим новый запрос со следующим кодом и щелкнем на значке `Execute`:

```
CREATE TABLE [dbo].[Orders]
(
    [OrderId] INT NOT NULL PRIMARY KEY IDENTITY,
    [CustId] INT NOT NULL,
    [CarId] INT NOT NULL
);
```

Теперь добавим данные в таблицу `Orders`. Ни одного отношения между таблицами пока еще не создано, и потому вводить значения в таблицы придется вручную. Для каждого значения `CustId` потребуется выбрать уникальное значение `CarId` (на рис. 21.10 показаны записи, которые основаны на ранее показанном примере данных).



	OrderId	CustId	CarId
▶	1	1	5
	2	2	1
	3	3	4
	4	4	7
*	NULL	NULL	NULL

Рис. 21.10. Наполнение таблицы `Orders`

Например, записи здесь указывают на то, что Дэйв Бреннер (Dave Brenner; `CustId` = 1) интересуется автомобилем марки BMW черного цвета (`CarId` = 5), а Пэт Уолтон (Pat Walton; `CustId` = 4) остановила свой выбор на BMW розового цвета (`CarId` = 7).

Создание отношений между таблицами

Установить отношения между таблицами проще всего, создав диаграмму базы данных. Дважды щелкнем на узле `Database Diagrams` (Диаграммы баз данных); будет предложено добавить обязательные поддерживающие объекты (рис. 21.11).

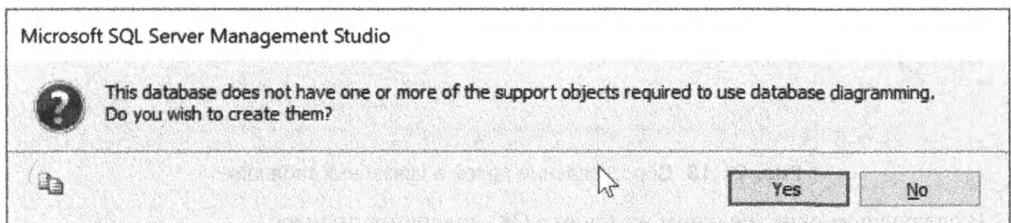


Рис. 21.11. Добавление поддерживающих объектов для построения диаграммы базы данных

Щелкнем правой кнопкой мыши на узле `Database Diagrams` и выберем в контекстном меню пункт `New Database Diagram` (Создать диаграмму базы данных). В открывшемся диалоговом окне `Add Table` (Добавление таблицы) выберем все таблицы и щелкнем на кнопке `Add` (Добавить), как показано на рис. 21.12.

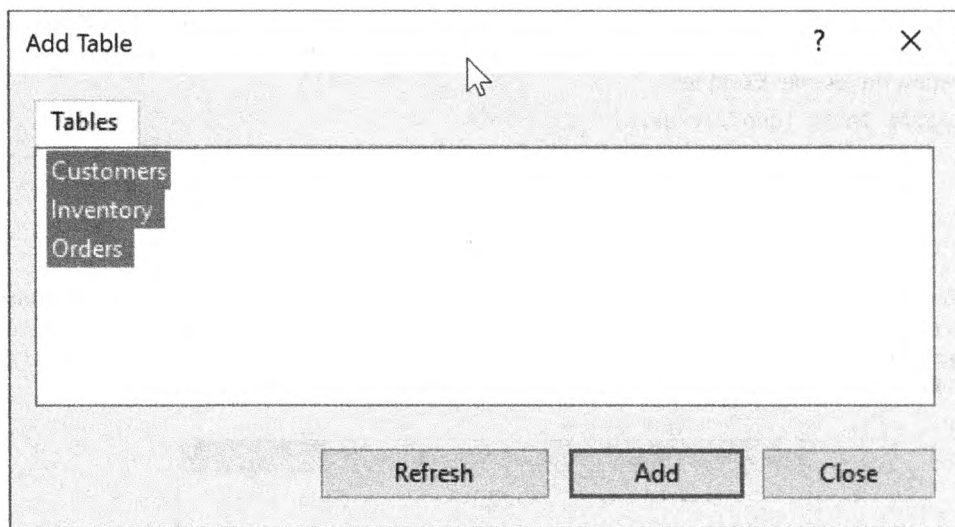


Рис. 21.12. Добавление таблиц в диаграмму базы данных

Чтобы создать внешний ключ, щелкнем на столбце CarId в таблице Inventory и перетащим его в таблицу Orders. В результате инструмент SSMS сопоставит столбцы CarId в обеих таблицах и отобразит диалоговое окно, предлагающее подтвердить выбор (рис. 21.13). Щелкнем на кнопке ОК для принятия выбора.

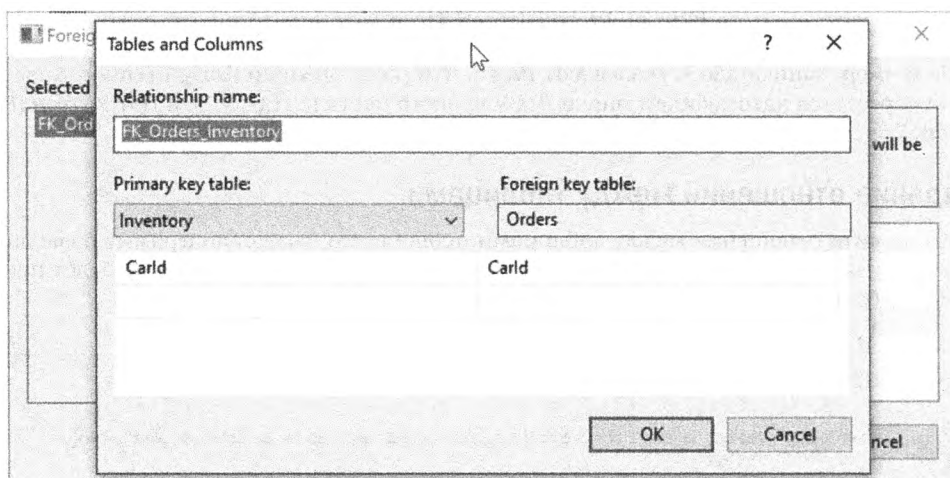


Рис. 21.13. Сопоставление полей в связанных таблицах

В следующем окне щелкнем на кнопке ОК, завершив процесс.

Перетащим столбец CustId из таблицы Customers на столбец CustId в таблице Orders. Удостоверимся, что поля сопоставлены (CustId из Customers и CustId из Orders). В открывшемся далее окне развернем узел INSERT And UPDATE Specification (Спецификация INSERT и UPDATE) и установим параметры Delete Rule (Правило удаления) и Update Rule (Правило обновления) в Cascade (Каскадное), как показано на рис. 21.14.

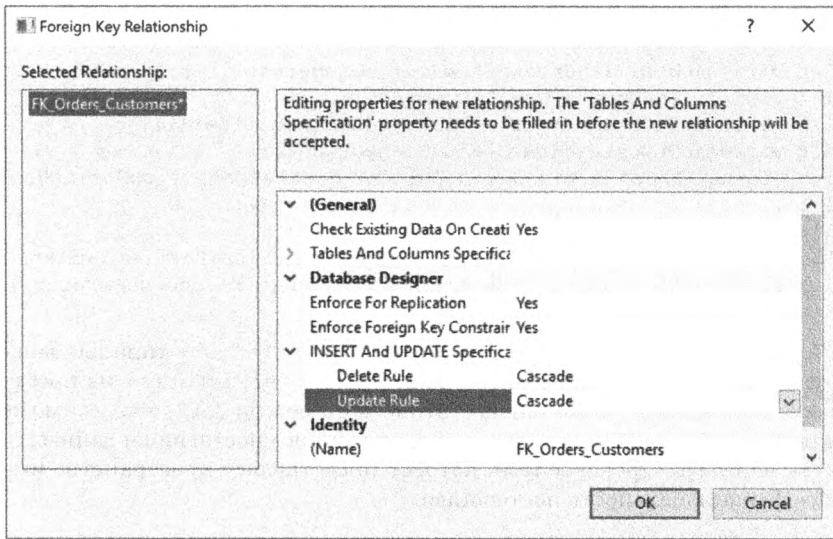


Рис. 21.14. Установка правил удаления и обновления в Cascade

Щелчком на кнопке Save (Сохранить) и назовем диаграмме имя (вроде AutoLotDiagram).

На этом создание базы данных AutoLot завершено. Разумеется, она очень далека от реальной корпоративной базы данных, но будет успешно удовлетворять всем нуждам текущей главы. Располагая тестовой базой данных, можно приступить к погружению в детали, касающиеся модели фабрики поставщиков данных ADO.NET.

Модель фабрики поставщиков данных ADO.NET

Модель фабрики поставщиков данных .NET позволяет строить единую кодовую базу, используя обобщенные типы доступа к данным. Кроме того, с применением конфигурационных файлов приложения (и подэлемента <connectionStrings>) можно получать поставщики и строки подключения декларативным образом без необходимости в повторной компиляции или развертывания сборки, в которой применяются API-интерфейсы ADO.NET.

Чтобы разобраться в реализации фабрики поставщиков данных, вспомните из табл. 21.1, что все классы внутри поставщика данных являются производными от тех же самых базовых классов, определенных внутри пространства имен System.Data.Common:

- DbCommand — абстрактный базовый класс для всех классов команд;
- DbConnection — абстрактный базовый класс для всех классов подключений;
- DbDataAdapter — абстрактный базовый класс для всех классов адаптеров данных;
- DbDataReader — абстрактный базовый класс для всех классов чтения данных;
- DbParameter — абстрактный базовый класс для всех классов параметров;
- DbTransaction — абстрактный базовый класс для всех классов транзакций.

Каждый поставщик данных, предлагаемый Microsoft, содержит класс, производный от System.Data.Common.DbProviderFactory. В этом базовом классе определено несколько методов, которые извлекают объекты данных, специфичные для поставщика. Вот члены класса DbProviderFactory:

```
public abstract class DbProviderFactory
{
    public virtual bool CanCreateDataSourceEnumerator { get; };
    public virtual DbCommand CreateCommand();
    public virtual DbCommandBuilder CreateCommandBuilder();
    public virtual DbConnection CreateConnection();
    public virtual DbConnectionStringBuilder CreateConnectionStringBuilder();
    public virtual DbDataAdapter CreateDataAdapter();
    public virtual DbParameter CreateParameter();
    public virtual CodeAccessPermission CreatePermission(PermissionState state);
    public virtual DbDataSourceEnumerator CreateDataSourceEnumerator();
}
```

Чтобы получить объект производного от `DbProviderFactory` типа для вашего поставщика данных, нужно использовать класс `DbProviderFactories` из пространства имен `System.Data.Common`. С помощью статического метода `GetFactory()` можно получить конкретный объект `DbProviderFactory` указанного поставщика данных, для чего понадобится передать строковое имя, которое представляет пространство имен .NET, содержащее функциональность поставщика:

```
static void Main(string[] args)
{
    // Получить фабрику для поставщика данных SQL.
    DbProviderFactory sqlFactory =
        DbProviderFactories.GetFactory("System.Data.SqlClient");
    ...
}
```

Конечно, вместо применения жестко закодированного строкового литерала получить фабрику можно было бы, прочитав такую информацию из файла `*.config` на стороне клиента (во многом аналогично ранее рассмотренному примеру `MyConnectionFactory`). Вскоре вы узнаете, как это делать, а пока после получения фабрики для поставщика данных можно получить связанные с ним объекты данных (такие как объекты подключений, объекты команд и объекты чтения данных).

На заметку! Для всех практических целей передаваемый методу `DbProviderFactories.GetFactory()` аргумент можно рассматривать как название пространства имен .NET поставщика данных. В реальности такое строковое значение используется в файле `machine.config` для динамической загрузки корректной библиотеки из глобального кеша сборок.

Полный пример фабрики поставщиков данных

Для демонстрации завершенного примера мы создадим новый проект консольного приложения C# (по имени `DataProviderFactory`), которое выводит инвентарный список автомобилей из базы данных `AutoLot`. В начальном примере мы жестко закодируем логику доступа к данным прямо в сборке `DataProviderFactory.exe` (чтобы излишне не усложнять код). По мере изучения материалов настоящей главы (и следующей, посвященной `Entity Framework`) вы узнаете более удачные способы решения задачи.

Первым делом добавим ссылку на сборку `System.Configuration.dll` и импортируем пространство имен `System.Configuration`. Затем модифицируем файл `App.config`, включив в него элемент `<appSettings>` с приведенным ниже содержимым (при необходимости измените должным образом строку подключения):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
```

```

<appSettings>
  <!-- Какой поставщик? -->
  <add key="provider" value="System.Data.SqlClient" />

  <!-- Какая строка подключения? -->
  <add key="connectionString" value="Data Source=(localdb)\mssqllocaldb;
    Initial Catalog=AutoLot;Integrated Security=True"/>
</appSettings>
<startup>
  <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7"/>
</startup>
</configuration>

```

Теперь, когда имеется подходящий файл *.config, можно прочитать значения provider и connectionString с использованием индексатора ConfigurationManager.AppSettings. Значение provider будет передано методу DbProviderFactories.GetFactory() с целью получения объекта типа фабрики, специфичного для поставщика данных. Значение connectionString будет применяться для установки свойства ConnectionString объекта производного от DbConnection типа.

Предполагая, что были импортированы пространства имен System.Data и System.Data.Common, а также добавлен оператор using static System.Console;, метод Main() можно обновить следующим образом:

```

static void Main(string[] args)
{
    WriteLine("***** Fun with Data Provider Factories *****\n");
    // Получить строку подключения и поставщик из файла *.config.
    string dataProvider =
        ConfigurationManager.AppSettings["provider"];
    string connectionString =
        ConfigurationManager.AppSettings["connectionString"];
    // Получить фабрику поставщиков.
    DbProviderFactory factory = DbProviderFactories.GetFactory(dataProvider);
    // Получить объект подключения.
    using (DbConnection connection = factory.CreateConnection())
    {
        if (connection == null)
        {
            ShowError("Connection");
            return;
        }
        WriteLine($"Your connection object is a: {connection.GetType().Name}");
        connection.ConnectionString = connectionString;
        connection.Open();

        // Создать объект команды.
        DbCommand command = factory.CreateCommand();
        if (command == null)
        {
            ShowError("Command");
            return;
        }
        WriteLine($"Your command object is a: {command.GetType().Name}");
        command.Connection = connection;
        command.CommandText = "Select * From Inventory";
        // Вывести данные с помощью объекта чтения данных.
    }
}

```



```

using (DbDataReader dataReader = command.ExecuteReader())
{
    WriteLine($"Your data reader object is a: {dataReader.GetType().Name}");
    WriteLine("\n***** Current Inventory *****");
    while (dataReader.Read())
        WriteLine($"-> Car #{dataReader["CarId"]} is a {dataReader["Make"]}.");
}
}
ReadLine();
}

private static void ShowError(string objectName)
{
    WriteLine($"There was an issue creating the {objectName}");
    // Возникла проблема с созданием объекта
    ReadLine();
}

```

Обратите внимание, что в целях диагностики с помощью служб рефлексии выводятся имена лежащих в основе объектов подключения, команды и чтения данных. В результате запуска приложения в окне консоли отобразятся текущие данные из таблицы Inventory базы данных AutoLot:

```

***** Fun with Data Provider Factories *****

Your connection object is a: SqlConnection
Your command object is a: SqlCommand
Your data reader object is a: SqlDataReader

***** Current Inventory *****
-> Car #1 is a VW.
-> Car #2 is a Ford.
-> Car #3 is a Saab.
-> Car #4 is a Yugo.
-> Car #5 is a BMW.
-> Car #6 is a BMW.
-> Car #7 is a BMW.

```

Изменим содержимое файла *.config, указав System.Data.OleDb в качестве поставщика данных, модифицируем строку подключения на Provider=SQLNCLI11 и сменим значение Integrated Security с true на SSPI:

```

<configuration>
  <appSettings>
    <!-- Какой поставщик? -->
    <add key="provider" value="System.Data.OleDb" />

    <!-- Какая строка подключения? -->
    <add key="connectionString" value=
      "Provider=SQLNCLI11;Data Source=(localdb)\mssqllocaldb;
      Integrated Security=SSPI;Initial Catalog=AutoLot"/>
  </appSettings>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7"/>
  </startup>
</configuration>

```

Обнаружится, что "за кулисами" использовались типы из пространства имен System.Data.OleDb; ниже показан вывод:

```

***** Fun with Data Provider Factories *****
Your connection object is a: OleDbConnection
Your command object is a: OleDbCommand
Your data reader object is a: OleDbDataReader

***** Current Inventory *****
-> Car #1 is a VW.
-> Car #2 is a Ford.
-> Car #3 is a Saab.
-> Car #4 is a Yugo.
-> Car #5 is a BMW.
-> Car #6 is a BMW.
-> Car #7 is a BMW.

```

Конечно, в зависимости от опыта работы с ADO.NET у вас может не быть полного понимания того, что в действительности *делают* объекты подключений, команд и чтения данных. Не вдаваясь в детали, пока просто запомните, что модель фабрики поставщиков данных ADO.NET позволяет строить единственную кодовую базу, которая способна потреблять разнообразные поставщики данных в декларативной манере.

Потенциальный недостаток модели фабрики поставщиков данных

Хотя модель фабрики поставщиков данных характеризуется высокой мощностью, вы должны обеспечить применение в кодовой базе только типов и методов, общих для всех поставщиков, посредством членов абстрактных базовых классов. Следовательно, при разработке кодовой базы вы ограничены членами `DbConnection`, `DbCommand` и других типов из пространства имен `System.Data.Common`.

С учетом сказанного вы можете прийти к заключению, что такой обобщенный подход предотвращает прямой доступ к дополнительным возможностям отдельной СУБД. Если вы должны быть в состоянии обращаться к специфическим членам лежащего в основе поставщика (например, `SqlConnection`), то можете воспользоваться явным приведением:

```

using (DbConnection connection = factory.CreateConnection())
{
    if (connection == null)
    {
        ShowError("Connection");
        return;
    }
    WriteLine($"Your connection object is a: {connection.GetType().Name}");
    connection.ConnectionString = connectionString;
    connection.Open();
    var sqlConnection = connection as SqlConnection;
    if (sqlConnection != null)
    {
        // Вывести информацию об используемой версии SQL Server.
        WriteLine(sqlConnection.ServerVersion);
    }
    // Ради краткости оставшийся код удален.
}

```

Однако в таком случае кодовая база становится чуть труднее в сопровождении (и менее гибкой), потому что придется добавить некоторое количество проверок времени выполнения. Тем не менее, если необходимо строить библиотеки доступа к данным наиболее гибким способом из числа возможных, тогда модель фабрики поставщиков данных предлагает замечательный механизм для решения такой задачи.

Элемент <connectionStrings>

В настоящий момент данные строки подключения находятся в элементе <appSettings> внутри файла *.config. В конфигурационных файлах приложений допускается указывать элемент <connectionStrings>. Внутри него можно определять любое количество пар "имя-значение", которые будут программно читаться в память с применением индексатора ConfigurationManager.ConnectionStrings. Преимущества такого подхода (в противоположность использованию элемента <appSettings> и индексатора ConfigurationManager.AppSettings) связано с возможностью определения нескольких строк подключения для одного приложения в согласованной манере.

Чтобы посмотреть на это в действии, модифицируем текущий файл App.config следующим образом (обратите внимание, что каждая строка подключения описана с применением атрибутов name и connectionString, а не key и value, как в <appSettings>):

```
<configuration>
  <appSettings>
    <!-- Какой поставщик? -->
    <add key="provider" value="System.Data.SqlClient" />
  </appSettings>

  <!-- Несколько строк подключения. -->
  <connectionStrings>
    <add name="AutoLotSqlProvider" connectionString =
      "Data Source=(localdb)\mssqllocaldb;
      Integrated Security=true;Initial Catalog=AutoLot"/>
    <add name="AutoLotOleDbProvider" connectionString =
      "Provider=SQLNCLI11;Data Source=(localdb)\mssqllocaldb;
      Integrated Security=SSPI;Initial Catalog=AutoLot"/>
  </connectionStrings>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7"/>
  </startup>
</configuration>
```

Далее обновим метод Main():

```
static void Main(string[] args)
{
    WriteLine("***** Fun with Data Provider Factories *****\n");
    string dataProvider =
        ConfigurationManager.AppSettings["provider"];
    string connectionString = ConfigurationManager.ConnectionStrings[
        "AutoLotSqlProvider"].ConnectionString;
    ...
}
```

К настоящему моменту мы располагаем приложением, которое способно отображать содержимое таблицы Inventory базы данных AutoLot, используя нейтральную кодовую базу. Вынесение имени поставщика и строки подключения во внешний файл *.config означает, что модель фабрики поставщиков данных может динамически загружать корректный поставщик на заднем плане. Первый пример завершен, и теперь можно углубиться в детали работы с ADO.NET. В оставшемся материале главы (и книги) внимание будет сосредоточено на пространстве имен System.Data.SqlClient, поскольку серверная база данных функционирует под управлением SQL Server.

Понятие подключенного уровня ADO.NET

Вспомните, что *подключенный уровень* ADO.NET позволяет взаимодействовать с базой данных с помощью объектов подключения, чтения данных и команд имеющегося поставщика данных. Упомянутые объекты уже применялись в предыдущем приложении `DataProviderFactory`, и мы пройдем через процесс снова, используя на этот раз расширенный пример. Чтобы подключиться к базе данных и прочитать записи посредством объекта чтения данных, необходимо выполнить следующие шаги.

1. Создать, сконфигурировать и открыть объект подключения.
2. Создать и сконфигурировать объект команды, указав объект подключения в аргументе конструктора или через свойство `Connection`.
3. Вызвать метод `ExecuteReader()` на сконфигурированном объекте команды.
4. Обработать каждую запись с применением метода `Read()` объекта чтения данных.

Для начала создадим новый проект консольного приложения по имени `AutoLotDataReader` и импортируем пространства имен `System.Data` и `System.Data.SqlClient`. Ниже приведен полный код метода `Main()` с последующим анализом:

```
class Program
{
    static void Main(string[] args)
    {
        WriteLine("***** Fun with Data Readers *****\n");

        // Создать и открыть подключение.
        using (SqlConnection connection = new SqlConnection())
        {
            connection.ConnectionString =
                @"Data Source=(localdb)\mssqllocaldb;Integrated Security=true;
Initial Catalog=AutoLot";
            connection.Open();

            // Создать объект команды SQL.
            string sql = "Select * From Inventory";
            SqlCommand myCommand = new SqlCommand(sql, connection);

            // Получить объект чтения данных с помощью ExecuteReader().
            using (SqlDataReader myDataReader = myCommand.ExecuteReader())
            {
                // Пройти в цикле по результатам.
                while (myDataReader.Read())
                {
                    WriteLine($"-> Make: {myDataReader["Make"]},
                        PetName: {myDataReader["PetName"]},
                        Color: {myDataReader["Color"]}");
                }
            }
            ReadLine();
        }
    }
}
```

Работа с объектами подключений

При работе с поставщиком данных первым делом понадобится установить сеанс с источником данных, используя объект подключения (производного от `DbConnection`

типа). Объекты подключений .NET обеспечиваются форматированной строкой подключения, которая содержит несколько пар "имя-значение", разделенных точками с запятой. Такая информация идентифицирует имя машины, к которой нужно подключиться, требуемые настройки безопасности, имя базы данных на машине и другие специфичные для поставщика сведения.

Из приведенного выше кода можно сделать вывод, что имя Initial Catalog относится к базе данных, с которой необходимо установить сеанс. Имя Data Source идентифицирует имя машины, где находится база данных. Мы применяем строку (localdb)\mssqllocaldb, которая относится к версии SQL Server Express, установленной вместе с Visual Studio 2017. В случае использования другого экземпляра свойство определяется как имя_машины\экземпляр. Например, MYSERVER\SQLSERVER2016 обозначает сервер по имени MYSERVER с экземпляром по имени SQLSERVER2016. Если применяется локальная машина разработки, тогда в качестве имени сервера указывается точка (.) или конструкция (local). В случае стандартного экземпляра SQL Server имя экземпляра опускается. Скажем, если база данных AutoLot была создана в стандартном экземпляре Microsoft SQL Server на локальном компьютере, то следует использовать Data Source=(local).

Кроме того, можно указать любое количество конструкций, которые представляют учетные данные безопасности. Здесь Integrated Security устанавливается в true, что предусматривает применение для аутентификации текущей учетной записи Windows.

На заметку! Для получения дополнительных сведений по каждой паре "имя-значение", связанной со специфичной СУБД, ищите описание свойства ConnectionString объекта подключения поставщика данных в документации .NET Framework 4.7 SDK.

Когда строка подключения готова, можно вызывать метод Open() для установления подключения к базе данных. В дополнение к членам ConnectionString, Open() и Close() объект подключения предоставляет несколько членов, которые позволяют конфигурировать дополнительные настройки подключения, такие как таймаут и транзакционная информация. В табл. 21.5 кратко описаны избранные члены базового класса DbConnection.

Таблица 21.5. Избранные члены типа DbConnection

Член	Описание
BeginTransaction()	Этот метод позволяет начать транзакцию базы данных
ChangeDatabase()	Этот метод изменяет базу данных, связанную с открытым подключением
ConnectionTimeout	Это свойство только для чтения возвращает промежуток времени, в течение которого происходит ожидание установления подключения, прежде чем будет сгенерирована ошибка (стандартное значение составляет 15 секунд). Чтобы изменить стандартное значение, необходимо указать в строке подключения конструкцию Connect Timeout (например, Connect Timeout=30)
Database	Это свойство только для чтения возвращает имя базы данных, обслуживаемой объектом подключения
DataSource	Это свойство только для чтения возвращает местоположение базы данных, обслуживаемой объектом подключения
GetSchema()	Этот метод возвращает объект DataTable, содержащий информацию схемы из источника данных
State	Это свойство только для чтения возвращает текущее состояние подключения, представленное перечислением ConnectionState

Свойства типа `DbConnection` обычно по своей природе допускают только чтение и полезны, только если требуется получить характеристики подключения во время выполнения. Когда необходимо переопределить стандартные настройки, придется изменить саму строку подключения. Например, в следующей строке подключения время таймута устанавливается равным 30 секундам вместо 15:

```
static void Main(string[] args)
{
    WriteLine("***** Fun with Data Readers *****\n");
    using(SqlConnection connection = new SqlConnection())
    {
        connection.ConnectionString =
            @"Data Source=(localdb)\mssqllocaldb;" +
            "Integrated Security=SSPI;Initial Catalog=AutoLot;Connect Timeout=30";
        connection.Open();

        // Новая вспомогательная функция (см. ниже) .
        ShowConnectionStatus(connection);
        ...
    }
}
```

В приведенном выше коде объект подключения передается в виде параметра новому статическому вспомогательному методу класса `Program` по имени `ShowConnectionStatus()` с такой реализацией:

```
static void ShowConnectionStatus(SqlConnection connection)
{
    // Вывести различные сведения о текущем объекте подключения.
    WriteLine("***** Info about your connection *****");
    WriteLine($"Database location: {connection.DataSource}");
        // Местоположение базы данных
    WriteLine($"Database name: {connection.Database}"); // Имя базы данных
    WriteLine($"Timeout: {connection.ConnectionTimeout}"); // Таймаут
    WriteLine($"Connection state: {connection.State}\n"); // Состояние
}
```

Большинство этих свойств понятно без объяснений, но свойство `State` требует дополнительного упоминания. Ему можно присвоить любое значение из перечисления `ConnectionState`:

```
public enum ConnectionState
{
    Broken, Closed,
    Connecting, Executing,
    Fetching, Open
}
```

Однако допустимыми значениями `ConnectionState` будут только `ConnectionState.Open`, `ConnectionState.Connecting` и `ConnectionState.Closed` (остальные члены перечисления зарезервированы для будущего использования). Кроме того, закрывать подключение всегда безопасно, даже если его состоянием в текущий момент является `ConnectionState.Closed`.

Работа с объектами `ConnectionStringBuilder`

Работа со строками подключения в коде может быть утомительной, т.к. они часто представлены в виде строковых литералов, которые в лучшем случае трудно обрабатывать и контролировать на предмет ошибок. Предлагаемые Microsoft поставщики

данных ADO.NET поддерживают *объекты построителей строк подключения*, которые позволяют устанавливать пары “имя-значение” с применением строго типизированных свойств. Взгляните на следующую модификацию текущего метода Main():

```
static void Main(string[] args)
{
    WriteLine("***** Fun with Data Readers *****\n");
    // Создать строку подключения с помощью объекта построителя.
    var cnStringBuilder = new SqlConnectionStringBuilder
    {
        InitialCatalog = "AutoLot",
        DataSource = @"(localdb)\mssqllocaldb",
        ConnectTimeout = 30,
        IntegratedSecurity = true
    };
    using(SqlConnection connection = new SqlConnection())
    {
        connection.ConnectionString = cnStringBuilder.ConnectionString;
        connection.Open();
        ShowConnectionStatus(connection);
        ...
    }
    ReadLine();
}
```

В этой версии метода Main() создается экземпляр класса SqlConnectionStringBuilder, соответствующим образом устанавливаются его свойства, после чего с использованием свойства ConnectionString получается внутренняя строка. Обратите внимание, что здесь применяется стандартный конструктор типа. При желании объект построителя строки подключения для поставщика данных можно также создать, передав в качестве отправной точки существующую строку подключения (что может быть удобно, когда значения динамически читаются из файла App.config). После наполнения объекта начальными строковыми данными отдельные пары “имя-значение” можно изменять с помощью связанных свойств, как демонстрируется далее:

```
static void Main(string[] args)
{
    WriteLine("***** Fun with Data Readers *****\n");
    // Предположим, что значение connectionString на самом деле получено
    // из файла *.config.
    string connectionString = @"Data Source=(localdb)\mssqllocaldb;" +
        "Integrated Security=true;Initial Catalog=AutoLot";
    SqlConnectionStringBuilder cnStringBuilder =
        new SqlConnectionStringBuilder(connectionString);
    // Изменить значение таймаута.
    cnStringBuilder.ConnectTimeout = 5;
    ...
}
```

Работа с объектами команд

Теперь, когда вы лучше понимаете роль объекта подключения, следующей задачей будет выяснение, каким образом отправлять SQL-запросы базе данных. Тип SqlCommand (производный от DbCommand) является объектно-ориентированным представлением SQL-

запроса, имени таблицы или хранимой процедуры. Тип команды указывается с использованием свойства `CommandType`, которое принимает любое значение из перечисления `CommandType`:

```
public enum CommandType
{
    StoredProcedure,
    TableDirect,
    Text // Стандартное значение.
}
```

При создании объекта команды SQL-запрос можно указывать как параметр конструктора или устанавливать свойство `CommandText` напрямую. Кроме того, когда создается объект команды, необходимо задать желаемое подключение. Его также можно указать как параметр конструктора либо с применением свойства `Connection`. Взгляните на следующий фрагмент кода:

```
// Создать объект команды посредством аргументов конструктора.
string sql = "Select * From Inventory";
SqlCommand myCommand = new SqlCommand(sql, connection);

// Создать еще один объект команды через свойства.
SqlCommand testCommand = new SqlCommand();
testCommand.Connection = connection;
testCommand.CommandText = sql;
```

Учтите, что в текущий момент вы еще не отправили SQL-запрос базе данных AutoLot, а только подготовили состояние объекта команды для будущего использования.

В табл. 21.6 описаны некоторые дополнительные члены типа `DbCommand`.

Таблица 21.6. Члены типа `DbCommand`

Член	Описание
<code>CommandTimeout</code>	Получает или устанавливает время ожидания, пока не завершится попытка выполнить команду и сгенерируется ошибка. Стандартное значение составляет 30 секунд
<code>Connection</code>	Получает или устанавливает объект <code>DbConnection</code> , применяемый текущим объектом <code>DbCommand</code>
<code>Parameters</code>	Получает коллекцию типов <code>DbParameter</code> , используемых для параметризованного запроса
<code>Cancel()</code>	Отменяет выполнение команды
<code>ExecuteReader()</code>	Выполняет запрос SQL и возвращает объект <code>DbDataReader</code> поставщика данных, который предоставляет допускающий только чтение доступ к результату запроса в прямом направлении
<code>ExecuteNonQuery()</code>	Выполняет оператор SQL, отличающийся от запроса (например, вставку, обновление, удаление или создание таблицы)
<code>ExecuteScalar()</code>	Легковесная версия метода <code>ExecuteReader()</code> , которая спроектирована специально для одноэлементных запросов (например, получение количества записей)
<code>Prepare()</code>	Создает подготовленную (или скомпилированную) версию команды для источника данных. Как вам может быть известно, <i>подготовленный запрос</i> выполняется несколько быстрее и удобен, когда один и тот же запрос необходимо выполнить многократно (обычно каждый раз с разными параметрами)

Работа с объектами чтения данных

После установления активного подключения и объекта команды SQL следующим действием будет отправка запроса источнику данных. Как вы наверняка догадались, это можно делать несколькими путями. Самый простой и быстрый способ получения информации из хранилища данных предлагает тип `DbDataReader` (реализующий интерфейс `IDataReader`). Вспомните, что объекты чтения данных представляют поток данных, допускающий только чтение в прямом направлении, который возвращает по одной записи за раз. Таким образом, объекты чтения данных полезны, только когда лежащему в основе хранилищу данных отправляются SQL-операторы выборки.

Объекты чтения данных удобны, если нужно быстро пройти по большому объему данных без необходимости иметь их представление в памяти. Например, в случае запрашивания 20 000 записей из таблицы с целью их сохранения в текстовом файле помещение такой информации в объект `DataSet` приведет к значительному расходу памяти (поскольку `DataSet` хранит полный результат запроса в памяти).

Более эффективный подход предусматривает создание объекта чтения данных, который максимально быстро проходит по всем записям. Тем не менее, имейте в виду, что объекты чтения данных (в отличие от объектов адаптеров данных, которые рассматриваются позже) удерживают подключение к источнику данных открытым до тех пор, пока вы его явно не закроете.

Объекты чтения данных получают из объекта команды с применением вызова `ExecuteReader()`. Объект чтения данных представляет текущую запись, прочитанную из базы данных. Он имеет метод индексатора (например, синтаксис `[]` в языке C#), который позволяет обращаться к столбцам текущей записи. Доступ к конкретному столбцу возможен либо по имени, либо по целочисленному индексу, начинающемуся с нуля.

В приведенном ниже примере использования объекта чтения данных задействован метод `Read()`, с помощью которого выясняется, когда достигнут конец записей (в случае чего он возвращает `false`). Для каждой прочитанной из базы данных записи с применением индексатора типа выводится модель, дружественное имя и цвет каждого автомобиля. Обратите внимание, что сразу после завершения обработки записей вызывается метод `Close()`, которые освобождает объект подключения.

```
static void Main(string[] args)
{
    ...
    // Получить объект чтения данных посредством ExecuteReader().
    using(SqlDataReader myDataReader = myCommand.ExecuteReader())
    {
        // Пройти в цикле по результатам.
        while (myDataReader.Read())
        {
            WriteLine($"-> Make: { myDataReader["Make"]},
                        PetName: { myDataReader["PetName"]},
                        Color: { myDataReader["Color"]}");
        }
    }
    ReadLine();
}
```

Индексатор объекта чтения данных перегружен для приема либо значения `string` (имя столбца), либо значения `int` (порядковый номер столбца). Таким образом, текущую логику объекта чтения можно сделать яснее (и избежать жестко закодированных строковых имен), внося следующие изменения (обратите внимание на использование свойства `FieldCount`):

```

while (myDataReader.Read())
{
    WriteLine("***** Record *****");
    for (int i = 0; i < myDataReader.FieldCount; i++)
    {
        WriteLine($"{myDataReader.GetName(i)} = { myDataReader.GetValue(i)} ");
    }
    WriteLine();
}

```

Если в настоящий момент скомпилировать проект и запустить его на выполнение, то должен отобразиться список всех автомобилей из таблицы Inventory базы данных AutoLot. В следующем выводе показано несколько начальных записей:

```

***** Fun with Data Readers *****
***** Info about your connection *****
Database location: (localdb)\mssqllocaldb
Database name: AutoLot
Timeout: 30
Connection state: Open
***** Record *****
CarId = 1
Make = VW
Color = Black
PetName = Zippy
***** Record *****
CarId = 2
Make = Ford
Color = Rust
PetName = Rusty

```

Получение множества результирующих наборов с использованием объекта чтения данных

Объекты чтения данных могут получать несколько результирующих наборов с применением одиночного объекта команды. Например, если вы хотите получить все строки из таблицы Inventory, а также все строки из таблицы Customers, тогда можете указать два SQL-оператора Select, разделив их точкой с запятой:

```
string sql = "Select * From Inventory;Select * from Customers";
```

После получения объекта чтения данных можно выполнить проход по каждому результирующему набору, используя метод `NextResult()`. Обратите внимание, что автоматически возвращается первый результирующий набор. Таким образом, если нужно прочитать все строки каждой таблицы, то понадобится построить показанную ниже итерационную конструкцию:

```

do
{
    while (myDataReader.Read())
    {
        WriteLine("***** Record *****");
        for (int i = 0; i < myDataReader.FieldCount; i++)
        {
            WriteLine($"{myDataReader.GetName(i)} = {myDataReader.GetValue(i)}");
        }
    }
}

```

```

        WriteLine();
    }
} while (myDataReader.NextResult());

```

К этому моменту вы уже должны лучше понимать функциональность, предлагаемую объектами чтения данных. Не забывайте, что объект чтения данных способен обрабатывать только SQL-операторы `Select`; его нельзя применять для изменения существующей таблицы базы данных с использованием запросов `Insert`, `Update` или `Delete`. Модификация существующей базы данных требует дальнейшего исследования объектов команд.

Исходный код. Проект `AutoLotDataReader` доступен в подкаталоге `Chapter_21`.

Работа с запросами создания, обновления и удаления

Метод `ExecuteReader()` извлекает объект чтения данных, который позволяет просматривать результаты SQL-оператора `Select` с помощью потока информации, допускающего только чтение в прямом направлении. Однако если необходимо отправить операторы SQL, которые в итоге модифицируют таблицу данных (или любой другой отличающийся от запроса оператор SQL, такой как создание таблицы либо выдача разрешений), то потребуется вызов метода `ExecuteNonQuery()` объекта команды. В зависимости от формата текста команды указанный единственный метод выполняет вставки, обновления и удаления.

На заметку! Говоря формально, “отличающийся от запроса” означает оператор SQL, который не возвращает результирующий набор. Таким образом, операторы `Select` являются запросами, тогда как `Insert`, `Update` и `Delete` — нет. С учетом сказанного метод `ExecuteNonQuery()` возвращает значение `int`, которое представляет количество строк, затронутых операторами, а не новый набор записей.

Все примеры взаимодействия с базами данных, рассмотренные в настоящей главе до сих пор, располагали только открытыми подключениями и применяли их для извлечения данных. Это лишь одна часть работы с базами данных; инфраструктура доступа к данным не приносила бы так много пользы, если бы полностью не поддерживала также функциональность создания, чтения, обновления и удаления (`create`, `read`, `update`, `delete` — `CRUD`). Далее вы научитесь пользоваться такой функциональностью, применяя вызовы `ExecuteNonQuery()`.

Начнем с создания нового проекта библиотеки классов C# по имени `AutoLotDAL` (сокращение от *AutoLot Data Access Layer* — уровень доступа к данным `AutoLot`) и удалим стандартный файл класса. Добавим новую папку с помощью пункта меню `Project` → `New Folder` (Проект → Новая папка), удостоверившись в том, что в окне `Solution Explorer` выбран проект, и назовем ее `DataOperations`. Затем поместим в новую папку файл класса по имени `InventoryDAL.cs` и изменим его на `public`. В классе `InventoryDAL` будут определены разнообразные члены, предназначенные для взаимодействия с таблицей `Inventory` базы данных `AutoLot`. Наконец, импортируем следующие пространства имен .NET:

```

// Будет использоваться поставщик SQL Server; однако
// для обеспечения более высокой гибкости разрешено
// также применять паттерн фабрики поставщиков ADO.NET.
using System.Data;
using System.Data.SqlClient;

```

На заметку! Вы можете вспомнить из главы 13, что когда объекты используют типы, управляющие низкоуровневыми ресурсами (например, подключением к базе данных), рекомендуется реализовать интерфейс `IDisposable` и написать подходящий финализатор. В производственной среде классы, подобные `InventoryDAL`, делают то же самое, но здесь мы поступать так не будем, чтобы сосредоточиться на особенностях инфраструктуры ADO.NET.

Добавление конструкторов

Создадим конструктор, который принимает строковый параметр (`connectionString`) и присваивает его значение переменной уровня класса. Затем создадим стандартный конструктор, передающий стандартную строку подключения другому конструктору. В итоге вызывающий код получит возможность изменения строки подключения, если стандартный вариант не подходит. Ниже показан соответствующий код.

```
private readonly string _connectionString;
public InventoryDAL() : this(@"Data Source = (localdb)\mssqllocaldb;Integrated
Security=true;Initial Catalog=AutoLot")
{
}
public InventoryDAL(string connectionString)
=> _connectionString = connectionString;
```

Открытие и закрытие подключения

Добавим переменную уровня класса, которая будет хранить подключение, применяемое кодом доступа к данным. Добавим также два метода: один для открытия подключения (`OpenConnection()`) и еще один для закрытия подключения (`CloseConnection()`). В методе `CloseConnection()` проверим состояние подключения и если оно не закрыто, тогда вызовем метод `Close()` на объекте подключения. Вот листинг кода:

```
private SqlConnection _sqlConnection = null;
private void OpenConnection()
{
    _sqlConnection = new SqlConnection { ConnectionString = _connectionString };
    _sqlConnection.Open();
}
private void CloseConnection()
{
    if (_sqlConnection?.State != ConnectionState.Closed)
    {
        _sqlConnection?.Close();
    }
}
```

Ради краткости в большинстве методов класса `InventoryDAL` не будет предприниматься проверка на предмет возникновения возможных исключений, равно как не будут генерироваться специальные исключения для сообщения о разнообразных проблемах при выполнении (скажем, неправильно сформированная строка подключения). Если бы строилась библиотека доступа к данным производственного уровня, то определенно пришлось бы использовать приемы структурированной обработки исключений (как объяснялось в главе 7), чтобы учесть любые аномалии времени выполнения.

Создание модели автомобиля

Современные библиотеки доступа к данным применяют классы (обычно называемые *моделями*) для представления и перемещения данных из базы данных. В следующей главе вы увидите, что такая концепция является основой инфраструктур ORM вроде Entity Framework, но пока мы просто собираемся создать одну модель для представления строки в таблице Inventory. Добавим в проект новую папку под названием Models и поместим в нее новый открытый класс по имени Car со следующим кодом:

```
public class Car
{
    public int CarId { get; set; }
    public string Color { get; set; }
    public string Make { get; set; }
    public string PetName { get; set; }
}
```

Добавление методов выборки

Для начала объединим имеющиеся сведения об объектах команд, чтения данных и обобщенных коллекциях, чтобы получить все записи из таблицы Inventory. Как было показано в начале главы, объект чтения данных в поставщике делает возможной выборку записей с использованием курсора на стороне сервера, который допускает только чтение в прямом направлении и доступен через метод Read(). Свойство CommandBehavior класса SqlDataReader настроено на автоматическое закрытие подключения, когда закрывается объект чтения данных. Метод GetAllInventory() возвращает экземпляр List<Car>, представляющий все данные в таблице Inventory:

```
using AutoLotDAL.Models;
public List<Car> GetAllInventory()
{
    OpenConnection();
    // Здесь будут храниться записи.
    List<Car> inventory = new List<Car>();

    // Подготовить объект команды.
    string sql = "Select * From Inventory";
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        command.CommandType = CommandType.Text;
        SqlDataReader dataReader =
            command.ExecuteReader(CommandBehavior.CloseConnection);
        while (dataReader.Read())
        {
            inventory.Add(new Car
            {
                CarId = (int)dataReader["CarId"],
                Color = (string)dataReader["Color"],
                Make = (string)dataReader["Make"],
                PetName = (string)dataReader["PetName"]
            });
        }
        dataReader.Close();
    }
    return inventory;
}
```

Следующий метод выборки получает одиночную запись об автомобиле на основе значения CarId:

```
public Car GetCar(int id)
{
    OpenConnection();
    Car car = null;
    string sql = $"Select * From Inventory where CarId = {id}";
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        command.CommandType = CommandType.Text;
        SqlDataReader dataReader =
            command.ExecuteReader(CommandBehavior.CloseConnection);
        while (dataReader.Read())
        {
            car = new Car
            {
                CarId = (int)dataReader["CarId"],
                Color = (string)dataReader["Color"],
                Make = (string)dataReader["Make"],
                PetName = (string)dataReader["PetName"]
            };
        }
        dataReader.Close();
    }
    return car;
}
```

Вставка новой записи об автомобиле

Вставка новой записи в таблицу Inventory сводится к построению SQL-оператора Insert (на основе пользовательского ввода), открытию подключения, вызову метода ExecuteNonQuery() с применением объекта команды и закрытию подключения. Увидеть вставку в действии можно, добавив к типу InventoryDAL открытый метод по имени InsertAuto(), который принимает четыре параметра, отображаемые на четыре столбца таблицы Inventory (CarId, Color, Make и PetName). Указанные аргументы используются при форматировании строки для вставки новой записи. И, наконец, для выполнения итогового оператора SQL применяется объект SqlConnection.

```
public void InsertAuto(string color, string make, string petName)
{
    OpenConnection();
    // Сформатировать и выполнить оператор SQL.
    string sql = $"Insert Into Inventory (Make, Color, PetName)
        Values ('{make}', '{color}', '{petName}')";

    // Выполнить, используя наше подключение.
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        command.CommandType = CommandType.Text;
        command.ExecuteNonQuery();
    }
    CloseConnection();
}
```

Приведенный выше метод принимает три строковых значения и работает при условии, что вызывающий код передает строки в правильном порядке. Более совершенный

метод использует модель `Car`, становясь строго типизированным, и гарантирует корректность порядка передачи свойств.

Создание строго типизированного метода `InsertCar()`

Добавим в класс `InventoryDAL` еще одну версию метода `InsertAuto()`, которая принимает в качестве параметра модель `Car`, не забыв об операторе `using` для пространства имен `AutoLotDAL.Models`:

```
public void InsertAuto(Car car)
{
    OpenConnection();
    // Сформатировать и выполнить оператор SQL.
    string sql = "Insert Into Inventory (Make, Color, PetName) Values " +
        $"('{car.Make}', '{car.Color}', '{car.PetName}')"";
    // Выполнить, используя наше подключение.
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        command.CommandType = CommandType.Text;
        command.ExecuteNonQuery();
    }
    CloseConnection();
}
```

Определение классов, представляющих записи в реляционной базе данных, является распространенным способом построения библиотеки доступа к данным, а также основой `Entity Framework`, как будет показано в следующей главе.

На заметку! Создание оператора SQL с применением конкатенации строк может быть рискованным с точки зрения безопасности (вспомните об атаках внедрением в SQL). Текст команды предпочтительнее формировать с использованием параметризованного запроса, с которым вы ознакомитесь позже в главе.

Добавление логики удаления

Удаление существующей записи не сложнее вставки новой записи. В отличие от метода `InsertAuto()` на этот раз вы узнаете о важном блоке `try/catch`, который обрабатывает возможную попытку удалить запись об автомобиле, уже заказанном кем-то из таблицы `Customers`. Стандартные параметры `INSERT` и `UPDATE` для внешних ключей по умолчанию предотвращают удаление зависимых записей в связанных таблицах. Когда предпринимается попытка подобного удаления, генерируется исключение `SqlException`. В реальной программе была бы предусмотрена логика обработки такой ошибки, но в рассматриваемом примере мы просто генерируем новое исключение. Добавим в класс `InventoryDAL` следующий метод:

```
public void DeleteCar(int id)
{
    OpenConnection();
    // Получить идентификатор автомобиля, подлежащего удалению,
    // и удалить запись о нем.
    string sql = $"Delete from Inventory where CarId = '{id}'";
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        try
        {

```

```

        command.CommandType = CommandType.Text;
        command.ExecuteNonQuery();
    }
    catch (SqlException ex)
    {
        Exception error = new Exception("Sorry! That car is on order!", ex);
        // Этот автомобиль заказан!
        throw error;
    }
}
CloseConnection();
}

```

Добавление логики обновления

Когда речь идет об обновлении существующей записи в таблице Inventory, первым делом потребуется решить, какие характеристики будет позволено изменять вызывающему коду: цвет автомобиля, его дружественное имя, модель или все перечисленное? Один из способов предоставления вызывающему коду полной гибкости заключается в определении метода, принимающего параметр типа string, который представляет любой оператор SQL, но в лучшем случае это сопряжено с риском.

В идеале лучше иметь набор методов, которые позволяют вызывающему коду обновлять запись разнообразными способами. Тем не менее, для такой простой библиотеки доступа к данным мы определим единственный метод, который дает вызывающему коду возможность обновить дружественное имя указанного автомобиля:

```

public void UpdateCarPetName(int id, string newPetName)
{
    OpenConnection();
    // Получить идентификатор автомобиля для модификации дружественного имени.
    string sql = $"Update Inventory Set PetName = '{newPetName}'
Where CarId = '{id}'";
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        command.ExecuteNonQuery();
    }
    CloseConnection();
}

```

Работа с параметризованными объектами команд

В настоящий момент внутри логики вставки, обновления и удаления для типа InventoryDAL используются жестко закодированные строковые литералы, представляющие каждый запрос SQL. В *параметризованных запросах* параметры SQL являются объектами, а не простыми порциями текста. Трактовка запросов SQL в более объектно-ориентированной манере помогает сократить количество опечаток (учитывая, что свойства строго типизированы). Вдобавок параметризованные запросы обычно выполняются значительно быстрее запросов в виде строковых литералов, т.к. они подвергаются разбору только однажды (а не каждый раз, когда строка с запросом SQL присваивается свойству CommandText). Параметризованные запросы также содействуют в защите против атак внедрением в SQL (хорошо известная проблема безопасности доступа к данным).

Для поддержки параметризованных запросов объекты команд ADO.NET содержат коллекцию индивидуальных объектов параметров. По умолчанию коллекция пуста, но в нее можно вставить любое количество объектов параметров, которые отображаются

на *параметры-заполнители* в запросе SQL. Чтобы ассоциировать параметр внутри запроса SQL с членом коллекции параметров в объекте команды, параметр запроса SQL необходимо снабдить префиксом в виде символа @ (во всяком случае, когда применяется Microsoft SQL Server; не все СУБД поддерживают такую систему обозначений).

Указание параметров с использованием типа DbParameter

Перед построением параметризованного запроса вы должны ознакомиться с типом DbParameter (который является базовым классом для объекта параметра поставщика). Класс DbParameter поддерживает несколько свойств, которые позволяют конфигурировать имя, размер и тип параметра, а также другие характеристики, включая направление движения параметра. Некоторые основные свойства типа DbParameter описаны в табл. 21.7.

Таблица 21.7. Основные свойства типа DbParameter

Свойство	Описание
DbType	Получает или устанавливает собственный тип данных параметра, представленный как тип данных CLR
Direction	Получает или устанавливает направление движения параметра: только для ввода, только для вывода, для ввода и для вывода или для возвращаемого значения
IsNullable	Получает или устанавливает признак, может ли параметр принимать значения null
ParameterName	Получает или устанавливает имя DbParameter
Size	Получает или устанавливает максимальный размер данных в байтах для параметра (полезно только для текстовых данных)
Value	Получает или устанавливает значение параметра

Давайте теперь посмотрим, как заполнять коллекцию совместимых с DbParameter объектов, содержащуюся в объекте команды, для чего переделаем метод InsertAuto() так, чтобы в нем были задействованы объекты параметров (аналогичным образом можно переписать остальные методы, но в настоящем примере это не обязательно):

```
public void InsertAuto(Car car)
{
    OpenConnection();
    // Обратите внимание на "заполнители" в запросе SQL.
    string sql = "Insert Into Inventory" +
        "(Make, Color, PetName) Values" +
        "(@Make, @Color, @PetName)";
    // Эта команда будет иметь внутренние параметры.
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        // Заполнить коллекцию параметров.
        SqlParameter parameter = new SqlParameter
        {
            ParameterName = "@Make",
            Value = car.Make,
            SqlDbType = SqlDbType.Char,
            Size = 10
        };
    }
}
```

```

command.Parameters.Add(parameter);
parameter = new SqlParameter
{
    ParameterName = "@Color",
    Value = car.Color,
    SqlDbType = SqlDbType.Char,
    Size = 10
};
command.Parameters.Add(parameter);
parameter = new SqlParameter
{
    ParameterName = "@PetName",
    Value = car.PetName,
    SqlDbType = SqlDbType.Char,
    Size = 10
};
command.Parameters.Add(parameter);
command.ExecuteNonQuery();
CloseConnection();
}
}

```

Снова обратите внимание, что запрос SQL содержит четыре символа-заполнителя, снабженные префиксами @. Тип SqlParameter применяется для сопоставления с каждым заполнителем за счет указания в свойстве ParameterName его имени и предоставления других деталей (например, значения, типа данных и размера) в строго типизированной манере. После того, как объект параметра готов, вызовом метода Add() он добавляется в коллекцию объекта команды.

На заметку! В приведенном примере для установки объектов параметров используются различные свойства. Тем не менее, следует отметить, что объекты параметров поддерживают несколько перегруженных конструкторов, которые позволяют устанавливать значения разнообразных свойств (в результате давая более компактную кодовую базу).

В то время как построение параметризованного запроса часто требует большего объема кода, в результате получается более удобный способ для программной настройки операторов SQL и достигается лучшая производительность. Параметризованные запросы также чрезвычайно удобны, когда нужно запускать хранимые процедуры.

Выполнение хранимой процедуры

Вспомните, что *хранимая процедура* представляет собой именованный блок кода SQL, сохраненный в базе данных. Хранимые процедуры можно конструировать так, чтобы они возвращали набор строк либо скалярных типов данных или выполняли еще какие-то осмысленные действия (например, вставку, обновление или удаление записей); в них также можно предусмотреть любое количество необязательных параметров. Конечным результатом будет единица работы, которая ведет себя подобно типичной функции, но только находится в хранилище данных, а не в двоичном бизнес-объекте. В текущий момент в базе данных AutoLot определена единственная хранимая процедура по имени GetPetName.

Рассмотрим следующий финальный метод типа InventoryDAL, в котором вызывает-ся хранимая процедура GetPetName:

```

public string LookUpPetName(int carId)
{
    OpenConnection();
    string carPetName;

    // Установить имя хранимой процедуры.
    using (SqlCommand command = new SqlCommand("GetPetName", _sqlConnection))
    {
        command.CommandType = CommandType.StoredProcedure;

        // Входной параметр.
        SqlParameter param = new SqlParameter
        {
            ParameterName = "@carId",
            SqlDbType = SqlDbType.Int,
            Value = carId,
            Direction = ParameterDirection.Input
        };
        command.Parameters.Add(param);

        // Выходной параметр.
        param = new SqlParameter
        {
            ParameterName = "@petName",
            SqlDbType = SqlDbType.Char,
            Size = 10,
            Direction = ParameterDirection.Output
        };
        command.Parameters.Add(param);

        // Выполнить хранимую процедуру.
        command.ExecuteNonQuery();

        // Возвратить выходной параметр.
        carPetName = (string)command.Parameters["@petName"].Value;
        CloseConnection();
    }
    return carPetName;
}

```

С вызовом хранимых процедур связан один важный аспект: объект команды может представлять оператор SQL (по умолчанию) либо имя хранимой процедуры. Когда объекту команды необходимо сообщить о том, что он будет вызывать хранимую процедуру, потребуется указать имя этой процедуры (в аргументе конструктора или в свойстве `CommandText`) и установить свойство `CommandType` в `CommandType.StoredProcedure`. (В противном случае возникнет исключение времени выполнения, т.к. по умолчанию объект команды ожидает оператор SQL.)

```

SqlCommand command = new SqlCommand("GetPetName", _sqlConnection);
command.CommandType = CommandType.StoredProcedure;

```

Далее обратите внимание, что свойство `Direction` объекта параметра позволяет указать направление движения каждого параметра, передаваемого хранимой процедуре (например, входной параметр, выходной параметр, входной/выходной параметр или возвращаемое значение). Как и ранее, все объекты параметров добавляются в коллекцию параметров объекта команды:

```

// Входной параметр.
SqlParameter param = new SqlParameter
{

```

```

ParameterName = "@carId",
SqlDbType = SqlDbType.Int,
Value = carId,
Direction = ParameterDirection.Input
};
command.Parameters.Add(param);

```

После того, как хранимая процедура, запущенная вызовом метода `ExecuteNonQuery()`, завершила работу, можно получить значение выходного параметра, просмотрев коллекцию параметров объекта команды и применив соответствующее приведение:

```

// Возвратить выходной параметр.
carPetName = (string)command.Parameters["@petName"].Value;

```

Итак, мы располагаем простейшей библиотекой доступа к данным `AutoLotDAL`, которую можно задействовать при построении клиента для отображения и редактирования данных (например, в консольном, настольном или основанном на HTML веб-приложении). Вопросы создания графических пользовательских интерфейсов пока не обсуждались, поэтому мы протестируем полученную библиотеку доступа к данным с помощью нового консольного приложения.

Создание консольного клиентского приложения

Добавим в решение `AutoLotDAL` новый проект консольного приложения (по имени `AutoLotClient`). Установим новый проект как запускаемый (щелкнув правой кнопкой мыши на проекте в окне `Solution Explorer` и выбрав в контекстном меню пункт `Set as StartUp Project` (Установить как запускаемый проект)) и добавим ссылку на проект `AutoLotDAL`. Поместим в начало файла `Program.cs` следующие операции `using`:

```

using AutoLotDAL.DataOperations;
using AutoLotDAL.Models;

```

Заменяем метод `Main()` показанным ниже кодом для взаимодействия с `AutoLotDAL`:

```

class Program
{
    static void Main(string[] args)
    {
        InventoryDAL dal = new InventoryDAL();
        var list = dal.GetAllInventory();
        Console.WriteLine(" ***** All Cars ***** ");
        Console.WriteLine("CarId\tMake\tColor\tPet Name");
        foreach (var itm in list)
        {
            Console.WriteLine($"{itm.CarId}\t{itm.Make}\t{itm.Color}\t{itm.PetName}");
        }
        Console.WriteLine();
        var car = dal.GetCar(list.OrderBy(x=>x.Color).Select(x => x.CarId).First());
        Console.WriteLine(" ***** First Car By Color ***** ");
        Console.WriteLine("CarId\tMake\tColor\tPet Name");
        Console.WriteLine($"{car.CarId}\t{car.Make}\t{car.Color}\t{car.PetName}");

        try
        {
            dal.DeleteCar(5);
            Console.WriteLine("Car deleted.");
            // Запись об автомобиле удалена
        }
    }
}

```

```

catch (Exception ex)
{
    Console.WriteLine($"An exception occurred: {ex.Message}");
    // Возникло исключение
}
dal.InsertAuto(new Car {Color="Blue",Make="Pilot",PetName = "TowMonster"});
list = dal.GetAllInventory();
var newCar = list.First(x => x.PetName == "TowMonster");
Console.WriteLine(" ***** New Car ***** ");
Console.WriteLine("CarId\tMake\tColor\tPet Name");
Console.WriteLine($"{newCar.CarId}\t{newCar.Make}\t{newCar.Color}
\t{newCar.PetName}");
dal.DeleteCar(newCar.CarId);
var petName = dal.LookUpPetName(car.CarId);
Console.WriteLine(" ***** New Car ***** ");
Console.WriteLine($"Car pet name: {petName}");
Console.Write("Press enter to continue...");
// Для продолжения нажмите <Enter>...
Console.ReadLine();
}
}

```

Понятие транзакций базы данных

Давайте завершим исследование ADO.NET рассмотрением концепции транзакции базы данных. Выражаясь просто, *транзакция* — это набор операций в базе данных, которые должны быть либо *все* успешно выполнены, либо *все* потерпеть неудачу как единая группа. Несложно предположить, что транзакции по-настоящему важны для обеспечения безопасности, достоверности и согласованности табличных данных.

Транзакции также важны в ситуациях, когда операция базы данных включает в себя взаимодействие с множеством таблиц или хранимых процедур (либо с комбинацией атомарных элементов базы данных). Классическим примером транзакции может служить процесс перевода денежных средств с одного банковского счета на другой. Например, если вам понадобилось перевести \$500 с депозитного счета на текущий чековый счет, то следующие шаги должны быть выполнены в транзакционной манере.

1. Банк должен снять \$500 с вашего депозитного счета.
2. Банк должен добавить \$500 на ваш текущий чековый счет.

Вряд ли бы вам понравилось, если бы деньги были сняты с депозитного счета, но не переведены (из-за какой-то ошибки со стороны банка) на текущий чековый счет, потому что вы попросту лишились бы \$500. Однако если поместить указанные шаги внутрь транзакции базы данных, тогда СУБД гарантирует, что все взаимосвязанные шаги будут выполнены как единое целое. Если любая часть транзакции откажет, то будет произведен *откат* всей операции в исходное состояние. С другой стороны, если все шаги выполняются успешно, то транзакция будет *зафиксирована*.

На заметку! Из литературы, посвященной транзакциям, вам может быть известно сокращение ACID. Оно обозначает четыре ключевых характеристики транзакции: *атомарность* (Atomic; все или ничего), *согласованность* (Consistent; данные остаются устойчивыми на протяжении транзакции), *изоляция* (Isolated; транзакции не влияют друг на друга) и *постоянство* (Durable; транзакции сохраняются и протоколируются в журнале).

В свою очередь платформа .NET поддерживает транзакции различными способами. Здесь мы рассмотрим объект транзакции поставщика данных ADO.NET (`SqlTransaction` в случае `System.Data.SqlClient`). Библиотеки базовых классов ADO.NET также предлагают поддержку транзакций внутри многочисленных API-интерфейсов, включая перечисленные ниже.

- `System.EnterpriseServices`. Это пространство имен (находящееся в сборке `System.EnterpriseServices.dll`) содержит типы, которые позволяют интегрироваться с уровнем исполняющей среды COM+, в том числе с ее поддержкой распределенных транзакций.
- `System.Transactions`. Это пространство имен (находящееся в сборке `System.Transactions.dll`) содержит классы, позволяющие писать собственные транзакционные приложения и диспетчеры ресурсов для разнообразных служб (скажем, `MSMQ`, `ADO.NET` и `COM+`).
- *Windows Communication Foundation (WCF)*. API-интерфейс WCF предоставляет службы для упрощения организации транзакций с помощью различных классов распределенной привязки.

В дополнение к готовой поддержке транзакций внутри библиотек базовых классов .NET можно также использовать язык SQL имеющейся СУБД. Например, вы могли бы написать хранимую процедуру, в которой применяются операторы `BEGIN TRANSACTION`, `ROLLBACK` и `COMMIT`.

Основные члены объекта транзакции ADO.NET

Несмотря на то что транзакционные типы существуют повсюду в библиотеках базовых классов, мы сосредоточим внимание на объектах транзакций, находящихся в поставщике данных ADO.NET; все они являются производными от `DBTransaction` и реализуют интерфейс `IdbTransaction`. Как упоминалось в начале главы, интерфейс `IdbTransaction` определяет несколько членов:

```
public interface IDbTransaction : IDisposable
{
    IDbConnection Connection { get; }
    IsolationLevel IsolationLevel { get; }

    void Commit();
    void Rollback();
}
```

Обратите внимание на свойство `Connection`, возвращающее ссылку на объект подключения, который инициировал текущую транзакцию (как вы вскоре увидите, объект транзакции получается из заданного объекта подключения). Метод `Commit()` вызывает, если все операции в базе данных завершились успешно, что приводит к сохранению в хранилище данных всех ожидающих изменений. И наоборот, метод `Rollback()` можно вызвать в случае генерации исключения времени выполнения, что информирует СУБД о необходимости проигнорировать все ожидающие изменения и оставить первоначальные данные незатронутыми.

На заметку! Свойство `IsolationLevel` объекта транзакции позволяет указать, насколько активно транзакция должна защищаться от действий со стороны других параллельно выполняющихся транзакций. По умолчанию транзакции полностью изолируются вплоть до их фиксации. Подробное описание значений перечисления `IsolationLevel` ищите в документации .NET Framework 4.7 SDK.

Помимо членов, определенных в интерфейсе `IDbTransaction`, тип `SqlTransaction` определяет дополнительный член под названием `Save()`, который предназначен для определения точек сохранения. Такая концепция позволяет откатить отказавшую транзакцию до именованной точки вместо того, чтобы осуществлять откат всей транзакции. При вызове метода `Save()` с использованием объекта `SqlTransaction` можно задавать удобный строковый псевдоним. А при вызове `Rollback()` этот псевдоним можно указывать в качестве аргумента для выполнения *частичного отката*. Вызов `Rollback()` без аргументов приводит к отмене всех ожидающих изменений.

Добавление таблицы `CreditRisks` в базу данных `AutoLot`

Давайте посмотрим, как применять транзакции ADO.NET. Начнем с использования инструмента `SQL Server Management Studio` для добавления в базу данных `AutoLot` новой таблицы по имени `CreditRisks`, которая содержит такие же столбцы, как и созданная ранее таблица `Customers`: `CustID` (первичный ключ), `FirstName` и `LastName`. Таблица `CreditRisks` предназначена для отсеивания нежелательных клиентов с плохой кредитной историей (рис. 21.15). Вот код SQL для ее создания:

```
CREATE TABLE [dbo].[CreditRisks] (
    [CustID] [INT] IDENTITY(1,1) NOT NULL,
    [FirstName] [NVARCHAR] (50) NULL,
    [LastName] [NVARCHAR] (50) NULL,
    PRIMARY KEY CLUSTERED ( [CustID] ASC )
)
```

Как и в предыдущем примере, где переводились деньги с депозитного на чековый счет, перемещение рискованного клиента из таблицы `Customers` в таблицу `CreditRisks` должно происходить под недремлющим оком транзакционного контекста (в конце концов, вы ведь хотите запомнить имена некредитоспособных клиентов). В частности, вам необходимо гарантировать, что либо текущие кредитные риски будут успешно удалены из таблицы `Customers` и добавлены в таблицу `CreditRisks`, либо ни одна из упомянутых операций базы данных не выполнится.

На заметку! С точки зрения проектирования баз данных вам не придется строить совершенно новую таблицу базы данных для хранения рискованных клиентов; взамен в существующей таблице `Customers` можно предусмотреть булевский столбец по имени `IsCreditRisk`. Тем не менее, наличие новой таблицы даст возможность поработать с простой транзакцией.

Добавление метода транзакции в `InventoryDAL`

Давайте посмотрим, как работать с транзакциями ADO.NET программным образом. Для начала откроем созданный ранее проект библиотеки кода `AutoLotDAL` и добавим в класс `InventoryDAL` новый открытый метод по имени `processCreditRisk()`, предназначенный для работы с кредитными рисками. (Обратите внимание, что ради простоты в примере не применяется параметризованный запрос, но в аналогичном методе производственного уровня он должен быть задействован.)

```
public void ProcessCreditRisk(bool throwEx, int custId)
{
    OpenConnection();
    // Первым делом найти текущее имя по идентификатору клиента.
    string fName;
    string lName;
    var cmdSelect = new SqlCommand($"Select * from Customers where CustId = {custId}",
        _sqlConnection);
```

```

using (var dataReader = cmdSelect.ExecuteReader())
{
    if (dataReader.HasRows)
    {
        dataReader.Read();
        fName = (string)dataReader["FirstName"];
        lName = (string)dataReader["LastName"];
    }
    else
    {
        CloseConnection();
        return;
    }
}

// Создать объекты команд, которые представляют каждый шаг операции.
var cmdRemove =
    new SqlCommand($"Delete from Customers where CustId = {custId}",
        _sqlConnection);
var cmdInsert =
    new SqlCommand("Insert Into CreditRisks" +
        $"(FirstName, LastName) Values('{fName}', '{lName}')" ,
        _sqlConnection);

// Это будет получено из объекта подключения.
SqlTransaction tx = null;
try
{
    tx = _sqlConnection.BeginTransaction();
    // Включить команды в транзакцию.
    cmdInsert.Transaction = tx;
    cmdRemove.Transaction = tx;

    // Выполнить команды.
    cmdInsert.ExecuteNonQuery();
    cmdRemove.ExecuteNonQuery();

    // Эмулировать ошибку.
    if (throwEx)
    {
        // Возникла ошибка, связанная с базой данных! Отказ транзакции...
        throw new Exception("Sorry! Database error! Tx failed...");
    }
    // Зафиксировать транзакцию!
    tx.Commit();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    // Любая ошибка приведет к откату транзакции.
    // Использовать условную операцию для проверки на предмет null.
    tx?.Rollback();
}
finally
{
    CloseConnection();
}
}

```


Здесь используется входной параметр типа `bool`, который указывает, нужно ли генерировать произвольное исключение при попытке обработки проблемного клиента. Такой прием позволяет эмулировать непредвиденные обстоятельства, которые могут привести к неудачному завершению транзакции. Понятно, что это делается лишь в демонстрационных целях; настоящий метод транзакции не должен позволять вызывающему процессу нарушать работу логики по своему усмотрению!

Обратите внимание на применение двух объектов `SqlCommand` для представления каждого шага транзакции, которая будет запущена. После получения имени и фамилии клиента на основе входного параметра `custId` с помощью метода `BeginTransaction()` объекта подключения можно получить допустимый объект `SqlTransaction`. Затем (что очень важно) потребуется *привлечь к участию каждый объект команды*, присвоив его свойству `Transaction` полученного объекта транзакции. Если этого не сделать, то логика вставки и удаления не будет находиться в транзакционном контексте.

После вызова метода `ExecuteNonQuery()` на каждой команде генерируется исключение, если (и только если) значение параметра `bool` равно `true`. В таком случае происходит откат всех ожидающих операций базы данных. Если исключение не было сгенерировано, тогда оба шага будут зафиксированы в таблицах базы данных в результате вызова `Commit()`. Теперь нужно скомпилировать модифицированный проект `AutoLotDAL`, чтобы удостовериться в отсутствии ошибок.

Тестирование транзакции базы данных

В окне инструмента SSMS выберем одного из клиентов и запомним его `CustId`. В имеющейся базе данных таким клиентом будет Дэйв Бреннер с `CustId`, равным 1. Добавим в файл `Program.cs` проекта `AutoLotClient` новый метод по имени `MoveCustomer()`.

```
public static void MoveCustomer()
{
    Console.WriteLine("***** Simple Transaction Example *****\n");
    // Простой способ позволить транзакции успешно завершиться или отказать.
    bool throwEx = true;

    Console.Write("Do you want to throw an exception (Y or N): ");
    // Хотите ли вы сгенерировать исключение?
    var userAnswer = Console.ReadLine();
    if (userAnswer?.ToLower() == "n")
    {
        throwEx = false;
    }

    var dal = new InventoryDAL();
    // Обработать клиента 1 - ввести идентификатор клиента,
    // подлежащего перемещению.
    dal.ProcessCreditRisk(throwEx, 1);
    Console.WriteLine("Check CreditRisk table for results");
    // Результаты ищите в таблице CreditRisk
    Console.ReadLine();
}
```

Если запустить программу и указать на необходимость генерации исключения, то запись о клиенте с идентификатором 1 не будет удалена из таблицы `Customers`, т.к. произойдет откат всей транзакции. Однако если исключение не генерируется, тогда окажется, что клиент с идентификатором 1 больше не находится в таблице `Customers`, а перемещен в таблицу `CreditRisks`.

Выполнение массового копирования с помощью ADO.NET

В случае, когда необходимо загрузить много записей в базу данных, показанные до сих пор методы будут довольно неэффективными. В SQL Server имеется средство, называемое *массовым копированием*, которое предназначено специально для таких сценариев, и в ADO.NET для него предусмотрена оболочка в виде класса `SqlBulkCopy`. В настоящем разделе главы мы объясним, как выполнять массовое копирование с помощью ADO.NET.

Исследование класса `SqlBulkCopy`

Класс `SqlBulkCopy` имеет один метод, `WriteToServer()` (и его асинхронную версию `WriteToServerAsync()`), который обрабатывает список записей и помещает данные в базу более эффективно, чем последовательность операторов `Insert`, выполненная с помощью объектов команд. Метод `WriteToServer()` перегружен, чтобы принимать объект `DataTable`, объект `DataReader` или массив объектов `DataRow` (рис. 21.15).

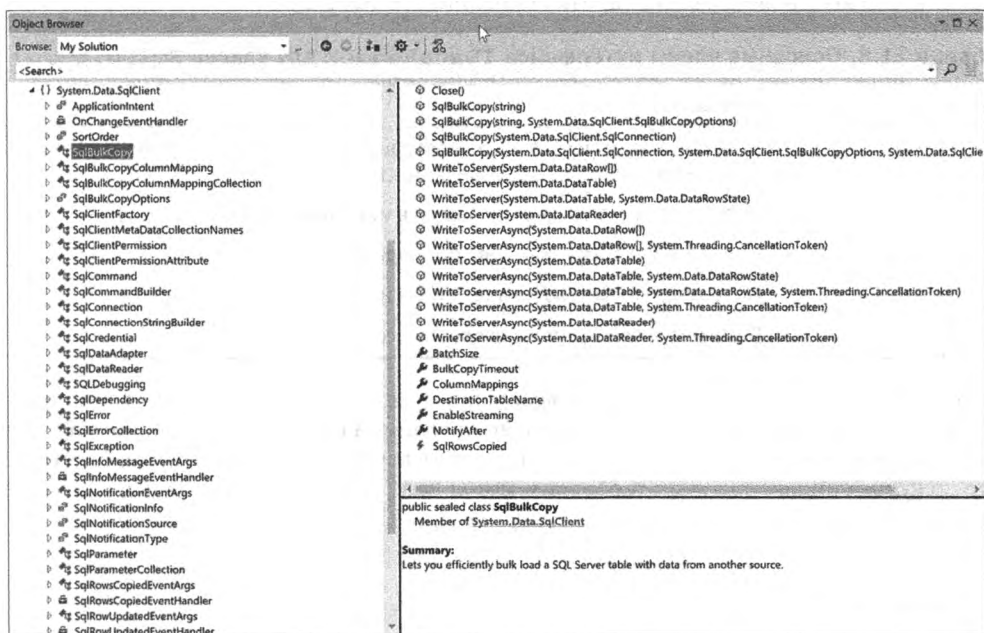


Рис. 21.15. Исследование класса `SqlBulkCopy`

Придерживаясь тематики главы, мы собираемся использовать версию `WriteToServer()`, которая принимает `DataReader`, так что нам необходимо создать специальный класс чтения данных.

Создание специального класса чтения данных

Мы хотим, чтобы специальный класс чтения данных был обобщенным и содержал список моделей, которые нужно импортировать. Начнем с создания в проекте `AutoLotDAL` новой папки по имени `BulkImport`, а в ней — нового обобщенного интерфейса `IMyDataReader`, реализующего `IDataReader`:

```
public interface IMyDataReader<T> : IDataReader
{
    List<T> Records { get; set; }
}
```

Задача теперь заключается в реализации специального класса чтения данных. Как вы уже видели, классы чтения данных содержат много частей, отвечающих за перемещение данных. Хорошая новость в том, что для `SqlBulkCopy` понадобится реализовать лишь несколько из них. Создадим новый обобщенный класс по имени `MyDataReader` и обеспечим реализацию им интерфейса `IMyDataReader`. Кроме того, модифицируем операторы `using` следующим образом:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Reflection;
```

Позволим Visual Studio самостоятельно реализовать все методы, что даст нам отправную точку для нашего специального класса. В рассматриваемом сценарии потребуется реализовать лишь члены, кратко описанные в табл. 21.8.

Таблица 21.8. Основные члены интерфейса `IDataReader` для класса `SqlBulkCopy`

Член	Описание
<code>Read()</code>	Получает следующую запись; возвращает <code>true</code> , если существует еще одна запись, или <code>false</code> , если достигнут конец списка
<code>FieldCount</code>	Получает общее количество полей в источнике данных
<code>GetName()</code>	Получает имя поля на основе его порядковой позиции
<code>GetOrdinal()</code>	Получает порядковую позицию поля на основе его имени поля
<code>GetValue()</code>	Получает значение поля на основе его порядковой позиции

Начнем с метода `Read()`, который возвращает `false`, если класс для чтения находится в конце списка, или `true` (с инкрементированием счетчика уровня класса), если конец списка еще не достигнут. Добавим переменную уровня класса, которая будет хранить текущий индекс `List<T>`, и обновим метод `Read()`, как показано ниже:

```
public class MyDataReader<T> : IMyDataReader<T>
{
    private int _currentIndex = -1;
    public bool Read()
    {
        if ((_currentIndex + 1) >= Records.Count) return false;
        _currentIndex++;
        return true;
    }
}
```

Методы `GetXXX()` и свойство `FieldCount` требуют знания специфической модели, подлежащей загрузке. Можно было бы жестко закодировать детали для каждого типа, но тогда пришлось бы выполнить немало работы. Вот как выглядел бы метод `GetName()`, использующий модель `Car`:

```
public object GetValue(int i)
{

```

```

Car currentRecord = Records[_currentIndex] as Car;
switch (i)
{
    case 0: return currentRecord.CarId;
    case 1: return currentRecord.Color;
    case 2: return currentRecord.Make;
    case 3: return currentRecord.PetName;
    default: return string.Empty;
}
}

```

База данных содержит только четыре таблицы, но это означает необходимость в наличии четырех вариаций класса чтения данных. А подумайте о реальной производственной базе данных, в которой таблиц гораздо больше! Решить проблему можно более эффективно с применением рефлексии (см. главу 15) и LINQ to Objects (см. главу 12). Создадим стандартный конструктор, который будет извлекать все свойства обобщенного типа и создавать из полученных в результате рефлексии свойств объект Dictionary, связывающий имя свойства с его индексом. Также добавим две переменные уровня класса для хранения значений, собранных в конструкторе. Ниже представлен итоговый код.

```

private readonly PropertyInfo[] _propertyInfos;
private readonly Dictionary<string, int> _nameDictionary;

public MyDataReader()
{
    _propertyInfos = typeof(T).GetProperties();
    _nameDictionary = _propertyInfos
        .Select((x, index) => new { x.Name, index })
        .ToDictionary(pair => pair.Name, pair => pair.index);
}

```

Теперь показанные далее методы могут быть реализованы обобщенным образом, используя полученную посредством рефлексии информацию. Вот обновленные методы:

```

public int FieldCount => _propertyInfos.Length;
public string GetName(int i)
    => i >= 0 && i < FieldCount ? _propertyInfos[i].Name : string.Empty;
public int GetOrdinal(string name)
    => _nameDictionary.ContainsKey(name) ? _nameDictionary[name] : -1;
public object GetValue(int i)
    => _propertyInfos[i].GetValue(Records[_currentIndex]);

```

Выполнение массового копирования

Добавим в папку BulkImport новый файл класса public static по имени ProcessBulkImport.cs. Напишем код для обработки открытия и закрытия подключений (подобный коду в классе InventoryDAL):

```

private static SqlConnection _sqlConnection = null;
private static readonly string _connectionString =
    @"Data Source = (localdb)\mssqllocaldb;Integrated Security=true;Initial
    Catalog=AutoLot";

private static void OpenConnection()
{
    _sqlConnection = new SqlConnection {ConnectionString = _connectionString};
    _sqlConnection.Open();
}

```

```
private static void CloseConnection()
{
    if (_sqlConnection?.State != ConnectionState.Closed)
    {
        _sqlConnection?.Close();
    }
}
```

Для обработки записей классу `SqlBulkCopy` требуется имя (и схема, если она отличается от `dbo`). После создания нового экземпляра `SqlBulkCopy` (с передачей объекта подключения) установим свойство `DestinationTableName`. Затем создадим новый экземпляр специального класса чтения данных, который хранит список объектов, подлежащих массовому копированию, и вызовем метод `WriteToServer()`. Ниже приведен код метода `ExecuteBulkImport()`:

```
public static void ExecuteBulkImport<T>(IEnumerable<T> records, string tableName)
{
    OpenConnection();
    using (SqlConnection conn = _sqlConnection)
    {
        SqlBulkCopy bc = new SqlBulkCopy(conn)
        {
            DestinationTableName = tableName
        };
        var dataReader = new MyDataReader<T>(records.ToList());
        try
        {
            bc.WriteToServer(dataReader);
        }
        catch (Exception ex)
        {
            // Здесь должно что-то делаться.
        }
        finally
        {
            CloseConnection();
        }
    }
}
```

Тестирование массового копирования

Добавим в файл `Program.cs` новый метод по имени `DoBulkCopy()`. Создадим список объектов `Car` и передадим его вместе с именем таблицы методу `ExecuteBulkImport()`. Оставшаяся часть кода отображает результаты массового копирования.

```
public static void DoBulkCopy()
{
    Console.WriteLine(" ***** Do Bulk Copy ***** ");
    var cars = new List<Car>
    {
        new Car() {Color = "Blue", Make = "Honda", PetName = "MyCar1"},
        new Car() {Color = "Red", Make = "Volvo", PetName = "MyCar2"},
        new Car() {Color = "White", Make = "VW", PetName = "MyCar3"},
        new Car() {Color = "Yellow", Make = "Toyota", PetName = "MyCar4"}
    };
}
```

```

ProcessBulkImport.ExecuteBulkImport(cars, "Inventory");
InventoryDAL dal = new InventoryDAL();
var list = dal.GetAllInventory();
Console.WriteLine(" ***** All Cars ***** ");
Console.WriteLine("CarId\tMake\tColor\tPet Name");
foreach (var itm in list)
{
    Console.WriteLine($"{itm.CarId}\t{itm.Make}\t{itm.Color}\t{itm.PetName}");
}
Console.WriteLine();
}

```

Хотя добавление четырех новых объектов Car не показывает достоинства работы, связанной с применением класса `SqlBulkCopy`, вообразите себе попытку загрузки тысяч записей. Мы проделывали подобное с таблицей `Customers`, и время загрузки составляло считанные секунды, тогда как проход в цикле по всем записям занимал часы! Как и все в .NET, класс `SqlBulkCopy` — просто еще один инструмент, который должен находиться в вашем инструментальном наборе и использоваться в ситуациях, когда в этом есть смысл.

Исходный код. Решение `AutoLotDAL` доступно в подкаталоге `Chapter_21`.

Резюме

Инфраструктура ADO.NET представляет собой собственную технологию доступа к данным платформы .NET, которую можно использовать тремя отдельными способами: подключенным, автономным и через инфраструктуры ORM вроде `Entity Framework`. В настоящей главе было начато исследование роли поставщиков данных, которые по существу являются конкретными реализациями нескольких абстрактных базовых классов (из пространства имен `System.Data.Common`) и интерфейсных типов (из пространства имен `System.Data`). Вы видели, что с применением модели фабрики поставщиков данных ADO.NET можно построить кодовую базу, не зависящую от поставщика.

Вы также узнали, что с помощью объектов подключений, объектов транзакций, объектов команд и объектов чтения данных из подключенного уровня можно выбирать, обновлять, вставлять и удалять записи. Кроме того, было показано, что объекты команд поддерживают внутреннюю коллекцию параметров, которые можно использовать для обеспечения безопасности к типам в запросах SQL; они также удобны при запуске хранимых процедур.

Наконец, вы научились защищать код манипулирования данными с помощью транзакций и ознакомились с применением класса `SqlBulkCopy` для загрузки крупных объемов данных в базы данных `SQL Server`, используя .NET.

ГЛАВА 22

Введение в Entity Framework 6

В предыдущей главе исследовались фундаментальные основы ADO.NET. Инфраструктура ADO.NET позволяет разработчикам приложений .NET взаимодействовать с реляционными данными (в относительно прямолинейной манере) с самого первого выпуска платформы .NET. Однако в версии .NET 3.5 Service Pack 1 был введен новый компонент API-интерфейса ADO.NET под названием *Entity Framework (EF)*.

На заметку! Хотя первая версия EF была подвергнута большой критике, команда разработчиков EF в Microsoft усердно трудилась над выпуском новых версий. Текущей версией EF для полной инфраструктуры .NET Framework является 6.1.3 (на момент написания главы), которая предлагает множество средств и улучшений производительности по сравнению с более ранними версиями. Вдобавок доступна версия Entity Framework Core (ранее называемая EF 7), которая рассматривается в части IX книги вместе с .NET Core.

Главная цель EF — предоставить возможность взаимодействия с данными из реляционных баз данных с использованием объектной модели, которая отображается прямо на бизнес-объекты (или объекты предметной области) в приложении. Например, вместо трактовки пакета данных как коллекции строк и столбцов можно оперировать коллекцией строго типизированных объектов, которые называются *сущностями* (entity). Такие сущности естественным образом поддерживают LINQ, и к ним можно применять запросы, используя ту же самую грамматику LINQ, которая была представлена в главе 12. Исполняющая среда EF самостоятельно транслирует запросы LINQ в подходящие запросы SQL.

В настоящей главе вы узнаете о доступе к данным с применением Entity Framework. Вы научитесь создавать модель предметной области и отображать классы модели на базу данных, а также ознакомитесь с ролью класса DbContext. Кроме того, будет сказано о навигационных свойствах, транзакциях и проверке параллелизма.

К концу главы мы построим финальную версию сборки AutoLotDAL.dll. Эта версия AutoLotDAL.dll используется в оставшихся главах книги (до тех пор, пока не будет переделана с применением EF Core).

На заметку! Все версии Entity Framework (включая EF 6.x) поддерживают визуальный конструктор сущностей, предназначенный для создания XML-файла модели сущностных данных (entity data model XML — EDMX). Начиная с версии 4.1, в инфраструктуре EF доступна поддержка *простых старых объектов CLR* (plain old CLR object — POCO) с использованием приема, называемого Code First (Сначала код). Версия EF Core поддерживает только парадигму Code First, отбросив всю поддержку EDMX. По указанной причине (и многочисленным проблемам, присущим

парадигме EDMX) в главе применяется только парадигма Code First. На самом деле название Code First внушает страх, т.к. оно дает ощущение, что эту парадигму нельзя использовать с существующей базой данных. Мы предпочли бы термин Code Centric (Направленная на код), но в Microsoft не заинтересовались нашим мнением!

Роль Entity Framework

Инфраструктура ADO.NET предоставляет фабрику, которая позволяет выбирать, вставлять, обновлять и удалять данные с помощью объектов подключений, команд и чтения данных. Хотя все это замечательно, такие аспекты ADO.NET заставляют трактовать полученные данные в манере, которая тесно связана со схемой физической базы данных. Например, при работе с подключенным уровнем обычно производится итерация по всем записям за счет указания объекту чтения данных имен столбцов.

Во время применения ADO.NET вы всегда обязаны помнить о физической структуре серверной базы данных. Вы должны знать схему каждой таблицы данных, писать потенциально сложные запросы SQL для взаимодействия с табличными данными и т.д. Такие действия могут требовать написания довольно громоздкого кода C#, поскольку сам язык C# не позволяет взаимодействовать напрямую со схемой базы данных.

Хуже того, обычный способ создания физической базы данных полностью сосредоточен на таких конструкциях базы данных, как внешние ключи, представления, хранимые процедуры и нормализация данных, а не на объектно-ориентированном программировании.

Инфраструктура Entity Framework сокращает разрыв между целями и оптимизацией базы данных и целями и оптимизацией объектно-ориентированного программирования. Используя EF, можно взаимодействовать с реляционными базами данных, (при желании) не сталкиваясь ни с одной строкой кода SQL. Когда запросы LINQ применяются к строго типизированным классам, исполняющая среда EF самостоятельно генерирует подходящие операторы SQL.

На заметку! *LINQ to Entities* — термин, описывающий действие по применению запросов LINQ к сущностным объектам ADO.NET.

Инфраструктура EF — просто другой способ реализации API-интерфейсов доступа к данным, который вовсе не призывает полностью отказаться от непосредственного использования ADO.NET в коде C#. Однако, поработав какое-то время с EF, вы быстро обнаружите, что иметь дело с этой развитой объектной моделью предпочтительнее, чем создавать весь код для доступа к данным с нуля. Инфраструктура EF является еще одним инструментом в вашем инструментальном наборе, и только вы вправе решать, какой подход более пригоден в разрабатываемом проекте.

На заметку! Вы можете вспомнить, что в версии .NET 3.5 появился API-интерфейс программирования для баз данных под названием LINQ to SQL. Концептуально он близок (а в плане конструкций программирования очень близок) к EF. Инфраструктура LINQ to SQL поддерживается в режиме сопровождения, т.е. будет получать только исправления критических ошибок. Располагая приложением, в котором используется LINQ to SQL, имейте в виду, что официальная политика Microsoft предусматривает поддержку всего программного обеспечения на протяжении минимум десяти лет после прекращения его существования. Таким образом, хотя инфраструктура LINQ to SQL не будет удалена из машины какими-то средствами защиты программного обеспечения, официальная позиция в Microsoft состоит в том, что вы должны сконцентрировать усилия на EF, а не на LINQ to SQL.

Роль сущностей

Упомянутые ранее (и продемонстрированные на примере класса `Car` в предыдущей главе) строго типизированные классы официально называются *сущностями*. Сущности представляют собой концептуальную модель физической базы данных, отображаемую на предметную область. Формально такая модель называется *моделью сущностных данных* (Entity Data Model — EDM). Модель EDM выглядит как набор классов клиентской стороны, которые отображаются на физическую базу данных посредством соглашений и конфигурации Entity Framework. Вы должны понимать, что сущности не обязаны напрямую отображаться на схему базы данных и обычно они не отображаются. Вы вольны структурировать сущностные классы для удовлетворения нуждам приложения и затем отображать уникальные сущности на имеющуюся схему базы данных.

На заметку! В мире Code First большинство разработчиков ссылаются на классы POCO как на *модели*, а на коллекцию таких классов — как на *граф объектов*. После создания экземпляров классов моделей с помощью данных из хранилища на них ссылаются как на *сущности*. В действительности перечисленные термины очень часто применяются взаимозаменяемо, что и делается повсеместно в настоящей книге.

Например, возьмем простую таблицу `Inventory` из базы данных `AutoLot` и класс модели `Car` из предыдущей главы. Посредством конфигурации модели инфраструктура EF информируется о том, что модель `Car` представляет таблицу `Inventory`. Подобное слабое связывание означает возможность формирования сущностей так, чтобы они близко моделировали предметную область.

На заметку! Во многих случаях классы моделей будут именоваться идентично связанным с ними таблицам базы данных. Тем не менее, не забывайте, что вы всегда можете изменить форму моделей для соответствия конкретной ситуации.

Вскоре мы построим полноценный пример с использованием EF. Однако пока что взгляните на следующий класс `Program`, в котором применяется класс модели `Car` и связанный контекстный класс (мы рассмотрим его очень скоро) по имени `AutoLotEntities` для добавления новой строки в таблицу `Inventory` базы данных `AutoLot`.

```
class Program
{
    static void Main(string[] args)
    {
        // Строка подключения автоматически читается из конфигурационного файла.
        using (AutoLotEntities context = new AutoLotEntities())
        {
            // Добавить новую строку в таблицу Inventory, используя нашу модель.
            context.Cars.Add(new Car() { Color = "Black",
                                         Make = "Pinto",
                                         PetName = "Pete" });
            context.SaveChanges();
        }
    }
}
```

Инфраструктура EF исследует конфигурацию моделей и контекстного класса, чтобы получить клиентское представление таблицы `Inventory` (здесь класс по имени `Car`) и отобразить его обратно на корректные столбцы таблицы `Inventory`. Обратите внимание на отсутствие каких-либо следов SQL-оператора `INSERT`. Мы просто добавляем

новый объект `Car` в коллекцию, поддерживаемую надлежаще именованным свойством `Cars` контекстного объекта, и затем сохраняем изменения.

В приведенном примере нет никакой магии. “За кулисами” создается и открывается подключение к базе данных, генерируется и выполняется подходящий оператор SQL и подключение освобождается и закрывается. Детали такого рода обрабатываются инфраструктурой без нашего участия. Теперь давайте рассмотрим основные службы EF, которые делают это возможным.

Строительные блоки Entity Framework

Инфраструктура EF (и ее ядро) использует инфраструктуру ADO.NET, которая была описана в предыдущей главе. Подобно любому взаимодействию ADO.NET для взаимодействия с хранилищем данных в EF применяется поставщик данных ADO.NET. Тем не менее, поставщик данных должен быть модернизирован, чтобы поддерживать новый набор служб, прежде чем он сможет работать с API-интерфейсом EF. Как и можно было ожидать, поставщик данных Microsoft SQL Server был обновлен необходимой инфраструктурой, которая учитывается при использовании сборки `System.Data.Entity.dll`.

На заметку! Многие СУБД от сторонних производителей (скажем, Oracle и MySQL) предлагают совместимые с EF поставщики данных. Если вы работаете не с SQL Server, тогда выясните детали у производителя СУБД или просмотрите список известных поставщиков данных ADO.NET по адресу <https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/ado-net-overview>.

Помимо добавления необходимых аспектов к поставщику данных Microsoft SQL Server сборка `System.Data.Entity.dll` содержит разнообразные пространства имен, которые предназначены для самих служб EF. Первая основная порция API-интерфейса EF, на которой мы сосредоточим внимание — класс `DbContext`. В случае применения EF в библиотеках доступа к данным будет создаваться производный контекст, специфичный для модели.

Роль класса DbContext

Класс `DbContext` представляет комбинацию паттернов проектирования “Единица работы” (Unit of Work) и “Хранилище” (Repository), которые могут использоваться для запрашивания базы данных и объединения изменений, записываемых обратно в базу данных в виде одиночной единицы работы. Класс `DbContext` предоставляет дочерним классам набор основных служб, включая возможность сохранения всех изменений (результатом будет обновление базы данных), настройку строки подключения, удаление объектов, вызов хранимых процедур и поддержку других фундаментальных деталей. В табл. 22.1 описаны некоторые часто применяемые члены класса `DbContext`.

Таблица 22.1. Часто используемые члены DbContext

Член	Описание
<code>DbContext()</code>	Конструктор, применяемый по умолчанию в производном контекстном классе. В строковом параметре указывается либо имя базы данных, либо строка подключения, хранящаяся в файле <code>*.config</code>
<code>Entry()</code> <code>Entry<TEntity>()</code>	Извлекает объект <code>System.Data.Entity.Infrastructure.DbEntityEntry</code> , который предоставляет доступ к информации и возможность выполнения действий над сущностью

Член	Описание
GetValidationErrors()	Проверяет достоверность обработанных сущностей и возвращает коллекцию объектов System.Data.Entity.Validation.DbEntityValidationResult
SaveChanges()	Сохраняет в базе данных все изменения, внесенные в этот контекст.
SaveChangesAsync()	Возвращает количество затронутых сущностей
Configuration	Предоставляет доступ к свойствам конфигурации контекста
Database	Предлагает механизм для создания/удаления/проверки существования лежащей в основе базы данных, для выполнения хранимых процедур и низкоуровневых операторов SQL в отношении внутреннего хранилища данных, а также для доступа к функциональности транзакций

Класс DbContext также реализует интерфейс IObjectContextAdapter, поэтому доступна любая функциональность, имеющаяся в классеObjectContext. Несмотря на то что класс DbContext позаботится о большинстве ваших потребностей, есть два события, которые могут оказаться исключительно удобными, как вы увидите позже в главе. События класса DbContext описаны в табл. 22.2.

Таблица 22.2. События DbContext

Событие	Описание
ObjectMaterialized	Иницируется при создании нового сущностного объекта из хранилища данных как части операции запроса или загрузки
SavingChanges	Происходит, когда изменения сохраняются в хранилище данных, но перед тем, как данные станут постоянными

Роль производных контекстных классов

Как упоминалось ранее, класс DbContext предоставляет основную функциональность во время работы со средством Code First инфраструктуры EF. В наших проектах будет создаваться производный от DbContext класс для специфической предметной области. В его конструкторе понадобится передать конструктору базового класса имя строки подключения для этого контекстного класса, как показано ниже:

```
public class AutoLotEntities : DbContext
{
    public AutoLotEntities() : base("name=AutoLotConnection")
    {
    }
    protected override void Dispose(bool disposing)
    {
    }
}
```

Роль DbSet<T>

Чтобы добавить таблицы в контекст, для каждой таблицы в объектной модели понадобится предусмотреть свойство типа DbSet<T>. Для включения ленивой загрузки свойства в контексте должны быть виртуальными, например:

```
public virtual DbSet<CreditRisk> CreditRisks { get; set; }
public virtual DbSet<Customer> Customers { get; set; }
public virtual DbSet<Inventory> Inventory { get; set; }
public virtual DbSet<Order> Orders { get; set; }
```

Каждый объект `DbSet<T>` предлагает для каждой коллекции несколько основных служб, таких как создание, удаление и нахождение записей в представленной таблице. Некоторые основные члены класса `DbSet<T>` описаны в табл. 22.3.

Таблица 22.3. Избранные основные члены `DbSet<T>`

Член	Описание
<code>Add()</code> <code>AddRange()</code>	Позволяют вставлять в коллекцию новый объект (или диапазон объектов). Объекты получают состояние <code>EntityState.Added</code> и будут вставлены в базу данных в результате вызова метода <code>SaveChanges()</code> (или <code>SaveChangesAsync()</code>) на объекте <code>DbContext</code>
<code>Attach()</code>	Ассоциирует объект с <code>DbContext</code> . Широко применяется в автономных приложениях вроде ASP.NET/MVC
<code>Create()</code> <code>Create<T>()</code>	Создает новый экземпляр указанного сущностного типа
<code>Find()</code> <code>FindAsync()</code>	Находит строку данных по первичному ключу и возвращает объект, представляющий эту строку
<code>Remove()</code> <code>RemoveRange()</code>	Помечает объект (или диапазон объектов) для последующего удаления
<code>SqlQuery()</code>	Создает низкоуровневый запрос SQL, который возвратит сущности в этом наборе

Добравшись до нужного свойства контекста, вы можете обратиться к любому члену `DbSet<T>`. Еще раз взгляните на фрагмент кода, приведенный в начале главы:

```
using (AutoLotEntities context = new AutoLotEntities())
{
    // Добавить новую строку в таблицу Inventory, используя нашу модель.
    context.Cars.Add(new Car() { Color = "Black",
                                Make = "Pinto",
                                PetName = "Pete" });

    context.SaveChanges();
}
```

Здесь `AutoLotEntities` — производный контекстный класс. Свойство `Cars` предоставляет доступ к переменной `DbSet<Car>`. Эта ссылка используется для вставки нового сущностного объекта `Car` и сообщения экземпляру `DbContext` о необходимости сохранения всех изменений в базе данных.

Целью запросов LINQ to Entities обычно является объект `DbSet<T>`; по существу класс `DbSet<T>` поддерживает те же самые расширяющие методы, о которых вы узнали в главе 12, такие как `ForEach()`, `Select()` и `All()`. Более того, `DbSet<T>` получает приличный объем функциональности от своего непосредственного родительского класса `DbQuery<T>`, который представляет строго типизированный запрос LINQ (или Entity SQL).

Парадигма Code First

Ранее упоминалось, что парадигма Code First вовсе не означает невозможность применения EF с существующей базой данных. В действительность она просто означает отсутствие модели EDMX. Мы предпочитаем термин Code Centric, т.к. он отражает фактическое положение дел. Вы можете использовать Code First с существующей базой данных или создать новую базу данных из сущностей с помощью миграций EF.

Поддержка транзакций

Все версии EF помещают каждый вызов `SaveChanges()` / `SaveChangesAsync()` внутрь транзакции. Уровень изоляции результирующих автоматических транзакций будет таким же, как стандартный уровень изоляции для базы данных (READ COMMITTED в случае SQL Server). При необходимости можно обеспечить более высокий контроль над поддержкой транзакций в EF. Дополнительные сведения ищите по адресу <https://msdn.microsoft.com/en-us/data/dn456843.aspx?f=255&MSPPErr=-2147217396>.

На заметку! Несмотря на что в книге указанный факт не раскрывается, теперь вызовы метода `ExecuteSqlCommand()` объекта базы данных `DbContext` для выполнения операторов SQL помещаются внутрь неявной транзакции. Это нововведение версии EF 6.

Состояние сущности и отслеживание изменений

Класс `DbChangeTracker` автоматически отслеживает состояние любого объекта, загруженного в `DbSet<T>`, внутри `DbContext`. Хотя в предшествующих примерах модификации производились внутри оператора `using`, любые изменения в данных будут отслеживаться и сохраняться при вызове метода `SaveChanges()` класса `AutoLotEntities`. Возможные значения состояния объекта описаны в табл. 22.4.

Таблица 22.4. Значения перечисления `EntityState`

Значение	Описание
Detached	Объект существует, но не отслеживается. Сущность попадает в это состояние немедленно после создания и перед добавлением к объектному контексту
Unchanged	Объект не изменялся с момента присоединения к контексту или с момента последнего вызова метода <code>SaveChanges()</code>
Added	Объект является новым и добавлен в объектный контекст, но метод <code>SaveChanges()</code> не вызывался
Deleted	Объект был удален из объектного контекста, но пока еще не удален из хранилища данных
Modified	Одно из скалярных свойств объекта было модифицировано, но метод <code>SaveChanges()</code> не вызывался

Для проверки состояния объекта используется следующий код:

```
EntityState state = context.Entry(entity).State;
```

Обычно переживать по поводу состояния объектов нет никакой необходимости. Однако в случае удаления объекта можно установить состояние объекта в `EntityState.Deleted` и выполнить обращение к базе данных. Позже в главе будет показано, как реализовать такое действие.

Аннотации данных Entity Framework

Аннотации данных — это атрибуты C#, которые применяются для придания формы сущностям. В табл. 22.5 перечислены некоторые распространенные аннотации данных для определения того, как сущностные классы и свойства отображаются на таблицы и поля базы данных. В оставшихся материалах главы и книги вы увидите, что доступно много других аннотаций, которые можно использовать для уточнения модели и добавления проверок достоверности.

Таблица 22.5. Аннотации данных, поддерживаемые Entity Framework

Аннотация данных	Описание
Key	Определяет первичный ключ для модели. Это не будет обязательным, если свойство ключа именовано как <code>Id</code> или как комбинация имени класса с <code>Id</code> , например, <code>OrderId</code> . В случае составного ключа потребуется добавить атрибут <code>Column</code> с параметром <code>Order</code> , такой как <code>Column[Order=1]</code> и <code>Column[Order=2]</code> . Поля ключа также неявно являются <code>[Required]</code>
Required	Объявляет свойство как не допускающее значения <code>null</code>
ForeignKey	Объявляет свойство, которое применяется как внешний ключ для навигационного свойства
StringLength	Указывает минимальную и максимальную длины для строкового свойства
NotMapped	Объявляет свойство, которое не отображается на поле базы данных
ConcurrencyCheck	Помечает поле, которое должно использоваться при проверке параллелизма, когда сервер баз данных выполняет обновления, вставки либо удаления
TimeStamp	Объявляет тип как версию строки или отметку времени (в зависимости от поставщика СУБД)
Table Column	Позволяет именовать классы и поля моделей не так, как они объявлены в базе данных. Атрибут <code>Table</code> делает также возможной спецификацию схемы (при условии, что хранилище данных поддерживает схемы)
DatabaseGenerated	Указывает, что поле сгенерировано базой данных. Может быть <code>Computed</code> , <code>Identity</code> или <code>None</code>
NotMapped	Указывает, что инфраструктура EF должна игнорировать это свойство относительно полей базы данных
Index	Указывает, что столбец должен иметь индекс, созданный для него. Можно задавать кластеризацию, уникальность, имя и порядок

Парадигма Code First из существующей базы данных

Теперь, когда вы лучше понимаете, что собой представляет инфраструктура ADO.NET Entity Framework и как она работает на высоком уровне, самое время рассмотреть первый полноценный пример. Мы разработаем простое консольное приложение с применением приема Code First из существующей базы данных для создания классов моделей, представляющих существующую базу данных AutoLot, которая была построена в главе 21.

Генерация модели

Начнем с создания нового проекта консольного приложения по имени `AutoLotConsoleApp`. С помощью пункта меню `Project⇒New Folder` (Проект⇒Новая папка) добавим в проект папку и назовем ее `EF`. При выбранной папке `EF` выберем пункт меню `Project⇒Add New Item` (Проект⇒Добавить новый элемент), удостоверимся в том, что выделен узел `Data (Данные)`, и вставим элемент `ADO.NET Entity Data Model (Модель сущностных данных ADO.NET)` по имени `AutoLotEntities` (рис. 22.1).

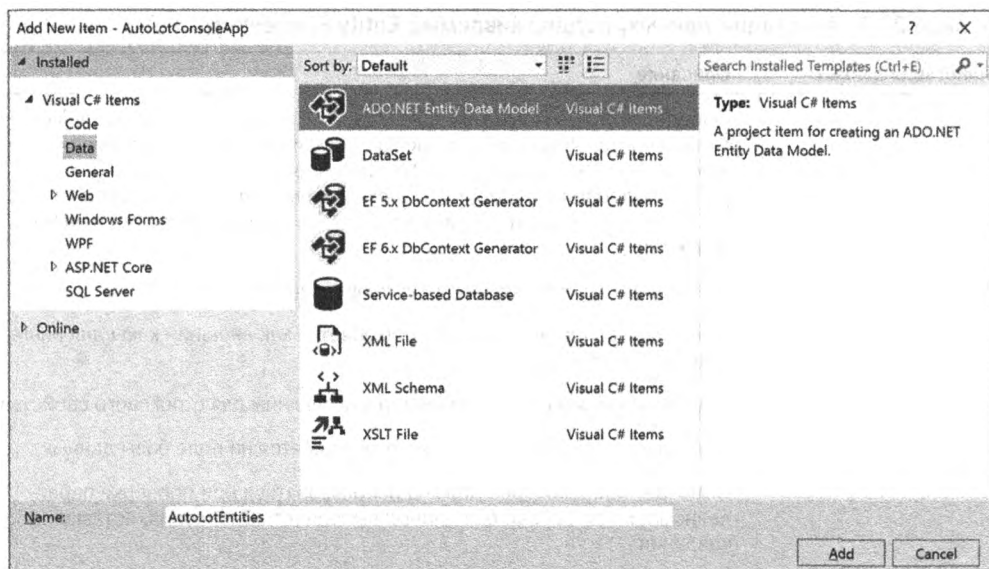


Рис. 22.1. Вставка в проект нового элемента ADO.NET EDM

Щелчок на кнопке Add (Добавить) приводит к запуску мастера создания модели сущностных данных (Entity Data Model Wizard). На первом экране мастер позволяет выбрать вариант генерации модели EDM с использованием визуального конструктора Entity Framework (из существующей базы данных или с созданием пустой поверхности конструктора) либо с применением приема Code First (из существующей базы данных или с созданием пустого объекта DbContext). Выберите вариант Code First from database (Code First из базы данных) и щелкните на кнопке Next (Далее), как показано на рис. 22.2.

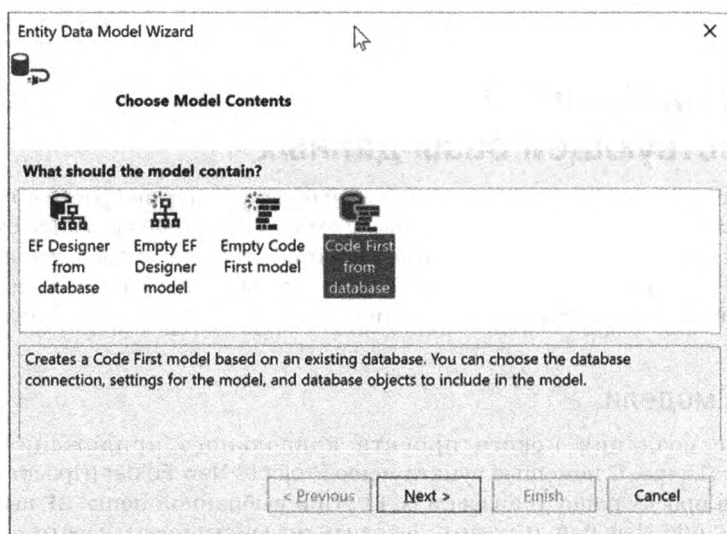


Рис. 22.2. Генерация модели EDM из существующей базы данных

На экране Choose Your Data Connection (Выберите свое подключение к данным) будут автоматически заполнены любые строки подключения, хранящиеся в Visual Studio. При наличии подключения к базе данных в окне Server Explorer среды Visual Studio оно будет присутствовать в раскрывающемся списке (что видно на рис. 22.5 далее в главе). В противном случае понадобится щелкнуть на кнопке New Connection (Создать подключение).

Откроется экран Choose Data Source (Выберите источник данных). Выберем Microsoft SQL Server и щелкнем на кнопке Continue (Продолжить), как показано на рис. 22.3.

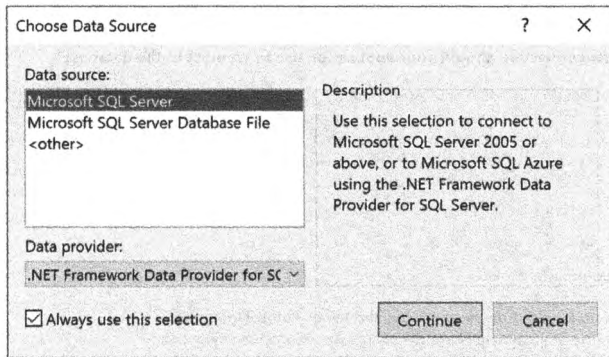


Рис. 22.3. Выбор источника данных SQL Server для новой строки подключения

В открывшемся диалоговом окне Connection Properties (Свойства подключения) выберем (localdb)\mssqllocaldb для сервера и AutoLot для базы данных (рис. 22.4).

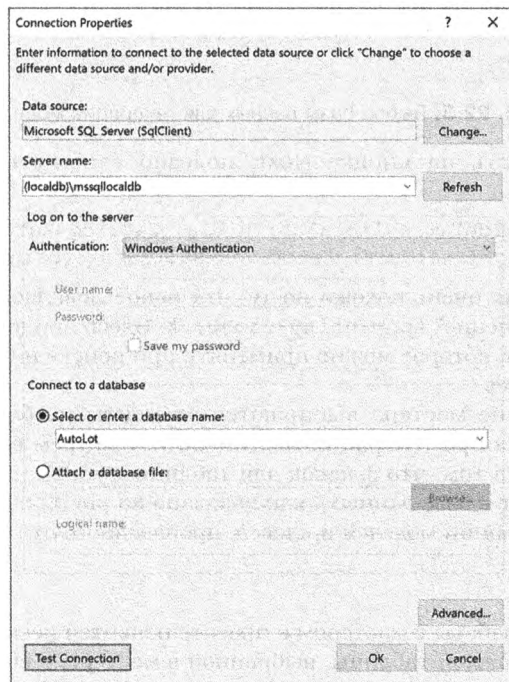


Рис. 22.4. Создание строки подключения к AutoLot

После щелчка на кнопке ОК новая строка подключения будет создана и выбрана на экране Choose Your Data Connection. Удостоверимся в том, что флажок Save connection settings in App.config as (Сохранить настройки подключения в App.config как) отмечен, и в поле под флажком укажем AutoLotConnection (рис. 22.5).

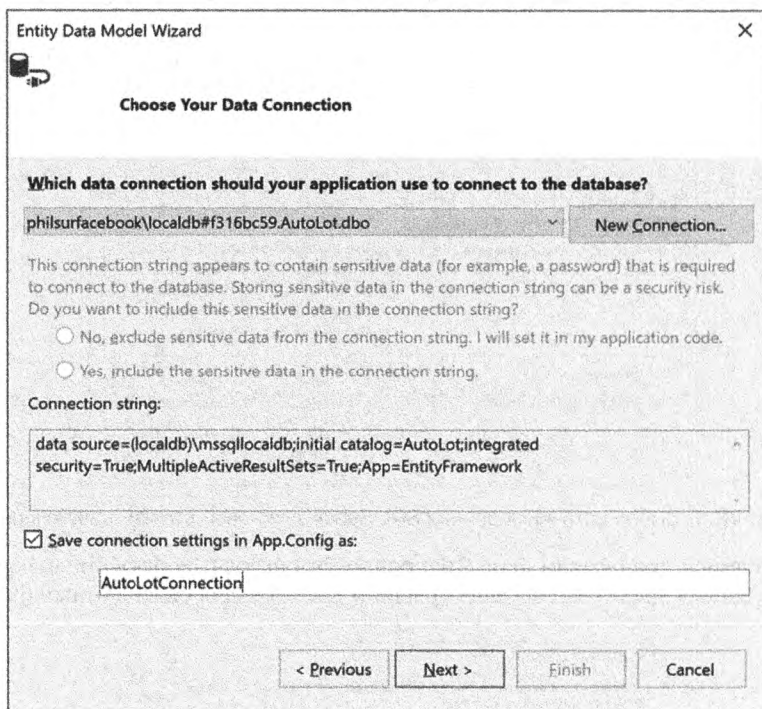


Рис. 22.5. Выбор базы данных для генерации модели

Прежде чем щелкать на кнопке Next, полезно взглянуть на формат строки подключения:

```
Data source= (localdb)\mssqllocaldb;Initial Catalog=AutoLot;
Integrated Security=True;MultipleActiveResultSets=true;App=EntityFramework
```

Строка подключения очень похожа на ту, что использовалась в главе 21, с добавлением пары “имя-значение” App=EntityFramework. Здесь App является сокращением для имени приложения, которое можно применять при поиске и устранении проблем с базой данных SQL Server.

На последнем экране мастера выбираются элементы из базы данных, которые должны использоваться при генерации модели EDM. Выберем все таблицы приложения, удостоверившись в том, что флажок для таблицы sysdiagrams не отмечен (если эта таблица существует в базе данных), как показано на рис. 22.6. Щелкнем на кнопке Finish (Готово) для генерации моделей и класса, производного от DbContext.

Что мы получили?

После завершения работы с мастером в проекте появится несколько новых классов: по одному классу для каждой таблицы, выбранной в мастере, и еще один класс по имени AutoLotEntities (имя, которое вводилось на первом экране мастера).

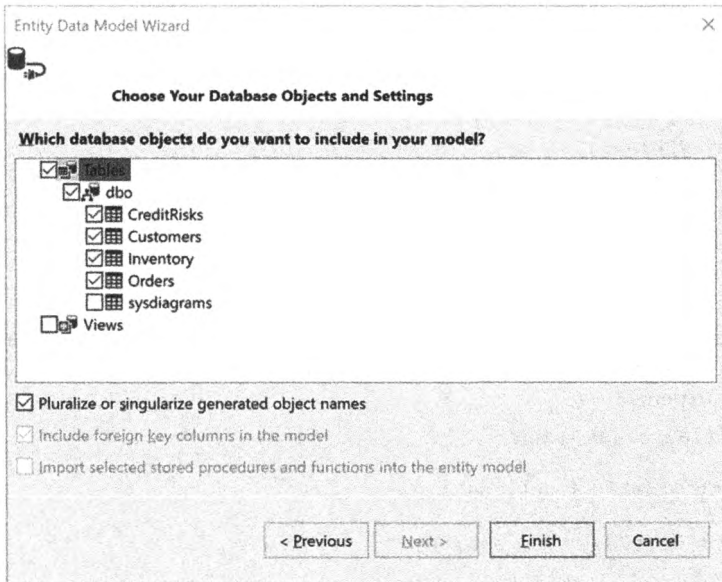


Рис. 22.6. Выбор элементов базы данных

По умолчанию имена сущностных классов будут совпадать с именами исходных объектов базы данных. С применением аннотаций данных, перечисленных в табл. 21.5, при необходимости (или по желанию) можно изменять имена сущностей, равно как имена свойств, на что-то другое. Позже в главе аннотации данных будут использоваться для внесения ряда модификаций в сущностные классы.

На заметку! Другой способ конфигурирования классов и свойств моделей для отображения на базу данных предлагает Fluent API. Все, что можно делать с помощью аннотаций данных, также возможно реализовать в коде посредством Fluent API. Из-за ограничений, касающихся объема, в главе подробно раскрываются только аннотации данных, а Fluent API будет упоминаться лишь кратко.

Откроем код класса `Inventory` и исследуем атрибуты, декорирующие класс и свойства. На уровне класса атрибут `Table` указывает, на какую таблицу базы данных отображается этот класс. На уровне свойств применяются два атрибута. Первый из них — атрибут `Key`, который устанавливает первичный ключ для таблицы. Второй атрибут, `StringLength`, указывает длину строки для поля базы данных.

На заметку! Имеются также два атрибута `SuppressMessage`. Они инструктируют статические анализаторы, такие как `FXCop` и новые анализаторы кода `Roslyn`, о необходимости отключения специфических правил, перечисленных в конструкторе. В главе они не рассматриваются и ради ясности из примера кода удалены.

```
[Table("Inventory")]
public partial class Inventory
{
    public Inventory()
    {
        Orders = new HashSet<Order>();
    }
}
```

```

[Key]
public int CarId { get; set; }

[StringLength(50)]
public string Make { get; set; }

[StringLength(50)]
public string Color { get; set; }

[StringLength(50)]
public string PetName { get; set; }

public virtual ICollection<Order> Orders { get; set; }
}

```

Можно также заметить, что класс `Inventory` содержит коллекцию объектов `Order` и что класс `Order` содержит свойство `CarId`. Это навигационные свойства, которые вскоре будут рассмотрены.

```

public partial class Order
{
    public int OrderId { get; set; }

    public int CustId { get; set; }

    public int CarId { get; set; }

    public virtual Customer Customer { get; set; }

    public virtual Inventory Inventory { get; set; }
}

```

Теперь откроем код класса `AutoLotEntities`. Он является производным от класса `DbContext` и содержит свойство типа `DbSet<TEntity>` для каждой таблицы, которая была выбрана в мастере. В нем переопределен метод `OnModelCreating()`, чтобы с помощью `Fluent API` установить отношения между таблицами `Orders` и `Inventory`.

```

public partial class AutoLotEntities : DbContext
{
    public AutoLotEntities()
        : base("name=AutoLotConnection")
    {
    }

    public virtual DbSet<CreditRisk> CreditRisks { get; set; }
    public virtual DbSet<Customer> Customers { get; set; }
    public virtual DbSet<Inventory> Inventories { get; set; }
    public virtual DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Inventory>()
            .HasMany(e => e.Orders)
            .WithRequired(e => e.Inventory)
            .WillCascadeOnDelete(false);
    }
}

```

Наконец, откроем файл `App.config`. В нем легко заметить новый раздел конфигурации (по имени `entityFramework`) и строку подключения, сгенерированную мастером. Большую ее часть можно проигнорировать, но в случае изменения местоположения

базы данных строку подключения понадобится модифицировать, указав в ней новое местоположение.

```
<configuration>
  <configSections>
    <!-- Дополнительные сведения о конфигурации Entity Framework доступны -->
    <!-- по адресу http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
      type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
      EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c56
      1934e089"
      requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />
  </startup>
  <entityFramework>
    <defaultConnectionFactory
      type="System.Data.Entity.Infrastructure.SqlConnectionFactory,
      EntityFramework" />
    <providers>
      <provider invariantName="System.Data.SqlClient"
        type="System.Data.Entity.SqlServer.SqlProviderServices,
        EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
  <connectionStrings>
    <add name="AutoLotConnection"
      connectionString="data source=(localdb)\mssqllocaldb;
      initial catalog=AutoLot;integrated security=True;MultipleActiveResult
      Sets=True;App=EntityFramework" providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Изменение стандартных отображений

Аннотация данных `[Table("Inventory")]` уровня класса отображает класс на таблицу `Inventory` в базе данных безотносительно к фактическому имени класса. Изменим имя файла и имя класса (а также конструктора) на `Car`. Атрибут `[Column("PetName")]` отображает декорированное свойство `C#` на поле `PetName` в таблице, позволяя изменять имя свойства `C#` на любое желаемое, скажем, на `CarNickName`. Обновленный класс выглядит следующим образом:

```
[Table("Inventory")]
public partial class Car
{
    public Car()
    {
        Orders = new HashSet<Order>();
    }

    [StringLength(50), Column("PetName")]
    public string CarNickName { get; set; }

    // Для краткости оставшийся код не показан.
}
```

Обратите внимание, что в классе `Order` придется также изменить тип свойства `Inventory` на `Car`. Вдобавок можно изменять имя свойства, как показано ниже:

```
public partial class Order
{
    public virtual Car Car { get; set; }
    // Для краткости оставшийся код не показан.
}
```

Финальное изменение потребуется внести в класс `AutoLotEntities`. Откроем файл с его кодом и изменим два вхождения `Inventory` на `Car`, а `DbSet<Car>` — на `Cars`. Вот обновленный код:

```
public partial class AutoLotEntities : DbContext
{
    public AutoLotEntities()
        : base("name=AutoLotConnection")
    {
    }
    // Для краткости оставшийся код не показан.
    public virtual DbSet<Car> Cars { get; set; }
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Car>()
            .HasMany(e => e.Orders)
            .WithRequired(e => e.Car
                .WillCascadeOnDelete(false));
        // Для краткости оставшийся код не показан.
    }
}
```

Добавление возможностей в сгенерированные классы моделей

Все классы, сгенерированные визуальным конструктором, объявлены с ключевым словом `partial`, что особенно удобно при работе с программной моделью EF, т.к. в сущностные классы можно добавлять дополнительные методы, которые помогают лучше моделировать предметную область.

Например, можно переопределить метод `ToString()` сущностного класса `Car`, чтобы он возвращал состояние сущности в виде сформатированной строки. Если добавить код переопределенного метода `ToString()` к сгенерированному классу, то возникнет риск утери этого специального кода каждый раз, когда классы моделей будут генерироваться повторно. Взамен определим следующий частичный класс в новом файле по имени `CarPartial.cs`:

```
public partial class Car
{
    public override string ToString()
    {
        // Поскольку столбец PetName может быть пустым,
        // предоставить стандартное имя **No Name**.
        return $"{this.CarNickName ?? "***No Name**"} is a {this.Color}
            {this.Make} with ID {this.CarId}.";
    }
}
```

Использование классов моделей в коде

Теперь, располагая классами моделей, можно написать код, который взаимодействует с ними и, следовательно, с базой данных. Начнем с добавления к классу Program следующих операторов using:

```
using AutoLotConsoleApp.EF;
using AutoLotConsoleApp.Models;
using static System.Console;
```

Вставка данных

Поскольку сущностные классы и свойства отображаются на таблицы и поля базы данных, когда изменения в коллекциях DbSet<T> необходимо сохранить, инфраструктура EF сгенерирует код SQL для внесения изменений. Добавление новых записей сводится к добавлению записей в DbSet<T> и вызову метода SaveChanges(). Можно добавлять либо по одной записи за раз, либо целый диапазон записей.

Вставка одной записи

Чтобы добавить новую запись, создадим экземпляр класса модели, добавим его в соответствующее свойство типа DbSet<T> контекстного класса AutoLotEntities и вызовем метод SaveChanges(). Определим в классе Program вспомогательный метод по имени AddNewRecord() с таким кодом:

```
private static int AddNewRecord()
{
    // Добавить запись в таблицу Inventory базы данных AutoLot.
    using (var context = new AutoLotEntities())
    {
        try
        {
            // В целях тестирования жестко закодировать данные для новой записи.
            var car = new Car() { Make = "Yugo", Color = "Brown", CarNickName="Brownie"};
            context.Cars.Add(car);
            context.SaveChanges();
            // В случае успешного сохранения EF заполняет поле идентификатора
            // значением, сгенерированным базой данных.
            return car.CarId;
        }
        catch (Exception ex)
        {
            WriteLine(ex.InnerException?.Message);
            return 0;
        }
    }
}
```

В коде применяется метод Add() класса DbSet<Car>. Метод Add() принимает объект типа Car и добавляет его в коллекцию Cars контекстного класса AutoLotEntities. Когда объект добавляется в DbSet<T>, экземпляр класса DbChangeTracker устанавливает состояние нового объекта в EntityState.Added. В результате вызова метода SaveChanges() на DbContext для всех ожидающих изменений, отслеживаемых экземпляром DbChangeTracker, генерируются операторы SQL (в этом случае оператор Insert) и выполняются в отношении базы данных. При наличии более одного изменения операторы будут выполняться внутри транзакции. Если ошибок не возникло, тогда изме-

нения сохраняются в базе данных, а любые свойства, сгенерированные базой данных (в рассматриваемом случае свойство `CarId`), обновляются значениями из базы данных.

Чтобы увидеть метод `AddNewRecord()` в действии, модифицируем метод `Main()`, как показано ниже:

```
static void Main(string[] args)
{
    WriteLine("***** Fun with ADO.NET EF *****\n");
    int carId = AddNewRecord();
    WriteLine(carId);
    ReadLine();
}
```

На консоль действительно выводится значение поля `CarId` новой записи. Когда добавляется новая запись (или обновляется существующая), инфраструктура EF выполняется оператор `SELECT` для свойств, генерируемых базой данных, чтобы получить их значения и заполнить свойства сущности.

Вставка множества записей

Помимо добавления одной записи за раз с помощью метода `AddRange()` можно добавлять сразу множество записей. Метод `AddRange()` принимает `IEnumerable<T>` и добавляет все элементы в списке к `DbSet<T>`.

```
private static void AddNewRecords(IEnumerable<Car> carsToAdd)
{
    using (var context = new AutoLotEntities())
    {
        context.Cars.AddRange(carsToAdd);
        context.SaveChanges();
    }
}
```

Несмотря на то что все записи теперь находятся внутри `DbSet<T>`, как и при добавлении одной записи, в базе данных ничего не произойдет, пока не будет вызван метод `SaveChanges()`. Метод `AddRange()` предназначен для удобства. Можно было бы организовать проход по коллекции, вызывая метод `Add()` для каждого элемента в ней, и затем вызвать `SaveChanges()`, что приведет к таким же результатам, как в случае вызова `AddRange()` и `SaveChanges()`.

Выборка записей

Существует несколько способов получения записей из базы данных с использованием EF. Простейший способ предусматривает проход по коллекции `DbSet<Car>`, что эквивалентно выполнению следующей команды SQL:

```
Select * from Inventory
```

Для его демонстрации добавим новый метод по имени `PrintAllInventory()`. Реализуем в нем цикл `foreach` для свойства `Cars` объекта `DbContext` и выведем сведения о каждом объекте автомобиля (с применением переопределенного метода `ToString()`):

```
private static void PrintAllInventory()
{
    // Выбрать все элементы из таблицы Inventory базы данных AutoLot
    // и вывести данные с применением специального метода ToString()
    // сущностного класса Car.
    using (var context = new AutoLotEntities())
    {
```

```

        foreach (Car c in context.Cars)
        {
            WriteLine(c);
        }
    }
}

```

“За кулисами” инфраструктура EF извлекает все записи Inventory из базы данных и с использованием объекта `DataReader` создает новый экземпляр класса `Car` для каждой строки, возвращенной из базы данных.

Запрашивание данных с помощью SQL из `DbSet<T>`

Для извлечения сущностей из базы данных можно также применять встраиваемый запрос SQL или хранимые процедуры. Хитрость в том, что поля запроса должны совпадать со свойствами заполняемого класса. Чтобы проверить такой прием, модифицируем метод `PrintInventory()`, как показано ниже:

```

private static void PrintAllInventory()
{
    using (var context = new AutoLotEntities())
    {
        foreach (Car c in context.Cars.SqlQuery(
            "Select CarId,Make,Color,PetName as CarNickName from Inventory where Make=@p0", "BMW"))
        {
            WriteLine(c);
        }
    }
}

```

Хорошая новость заключается в том, что при вызове на `DbSet<T>` метод `PrintInventory()` заполняет список отслеживаемыми сущностями, т.е. после вызова метода `SaveChanges()` любые изменения или удаления будут переданы в базу данных. Плохая новость (как можно заметить по тексту команды SQL) связана с тем, что метод `SqlQuery()` не воспринимает изменения в отображении, которые были сделаны ранее. Потребуется не только использовать имена таблицы базы данных и ее полей, но любые изменения в именах полей (вроде изменения на `PetName`) должны сопровождаться сопоставлением имени поля базы данных с именем свойства модели.

Запрашивание данных с помощью SQL из `DbContext.Database`

Свойство `Database` класса `DbContext` также имеет метод `SqlQuery()`, который может применяться для заполнения сущностей `DbSet<T>` и объектов, не являющихся частью модели (таких как модель представления). Предположим, что есть еще один класс (по имени `ShortCar`), определенный следующим образом:

```

public class ShortCar
{
    public int CarId { get; set; }
    public string Make { get; set; }
    public override string ToString() => $"{this.Make} with ID {this.CarId}.";
}

```

Вот как можно создать и заполнить список объектов `ShortCar`:

```

foreach (ShortCar c in context.Database.SqlQuery<ShortCar>(
    "Select CarId,Make from Inventory"))
{
    WriteLine(c);
}

```


Важно понимать, что при вызове метода `SqlQuery()` на объекте `Database` возвращаемые классы *никогда* не отслеживаются, даже если такие объекты определены как `DbSet<T>` в контекстном классе.

Запрашивание с помощью LINQ

Инфраструктура EF становится намного более мощной, когда объединяется с запросами LINQ. Рассмотрим следующее обновление метода `PrintInventory()`, в котором для получения записей из базы данных используется LINQ:

```
private static void PrintAllInventory()
{
    foreach (Car c in context.Cars.Where(c => c.Make == "BMW"))
    {
        WriteLine(c);
    }
}
```

Оператор LINQ транслируется в запрос SQL, который похож на показанный ниже:

```
Select * from Inventory where Make = 'BMW';
```

С учетом того, что вы уже работали со многими выражениями LINQ в главе 13, на текущий момент еще нескольких примеров будет вполне достаточно:

```
private static void FunWithLinqQueries()
{
    using (var context = new AutoLotEntities())
    {
        // Получить проекцию новых данных.
        var colorsMakes =
            from item in context.Cars select new { item.Color, item.Make };
        foreach (var item in colorsMakes)
        {
            WriteLine(item);
        }

        // Получить только элементы, в которых Color == "Black".
        var blackCars =
            from item in context.Cars where item.Color == "Black" select item;
        foreach (var item in blackCars)
        {
            WriteLine(item);
        }
    }
}
```

Поиск с помощью Find()

`Find()` — это специфический метод LINQ, поскольку он сначала ищет в `DbChangeTracker` запрошенную сущность и в случае ее нахождения возвращает сущность вызывающему методу. Если сущность не найдена, тогда инфраструктура EF делает обращение к базе данных, чтобы создать затребованную сущность.

Метод `Find()` ищет только по первичному ключу (простому или сложному), так что при его применении это следует иметь в виду. Пример использования метода `Find()` выглядит так:

```
WriteLine(context.Cars.Find(5));
```

Определение времени выполнения запросов инфраструктурой EF

Каждый предшествующий метод возвращает данные из базы данных. Важно понимать, когда именно запросы выполняются. Одно из достоинств инфраструктуры EF заключается в том, что запросы выборки не выполняются до тех пор, пока не начнется проход по результирующей коллекции. Запрос также выполнится в случае вызова на нем метода `ToList()` или `ToArray()`. Это делает возможным объединение вызовов в цепочку и построение запроса одновременно с точным контролем момента, когда совершается первое обращение к базе данных. Тем не менее, благодаря ленивой загрузке появляется контроль над тем, когда запросы делаются на навигационных свойствах, которые не были загружены первоначальным вызовом.

Рассмотрим следующую модифицированную версию метода `FunWithLinqQueries()`. В первых двух строках кода настраивается запрос. Обращение к базе данных не произойдет до тех пор, пока не начнется проход по коллекции с помощью `foreach`.

```
private static void ChainingLinqQueries()
{
    using (var context = new AutoLotEntities())
    {
        // Не выполняется.
        DbSet<Car> allData = context.Cars;

        // Не выполняется.
        var colorsMakes = from item in allData select new { item.Color, item.Make };

        // Теперь выполняется.
        foreach (var item in colorsMakes)
        {
            WriteLine(item);
        }
    }
}
```

Роль навигационных свойств

Как следует из названия, *навигационные свойства* позволяют находить связанные данные в других сущностях без необходимости в написании сложных запросов `JOIN`. В базе данных `SQL Server` навигационные свойства представляются посредством отношений внешних ключей. Чтобы учесть такие отношения в EF, каждый класс в модели содержит виртуальные свойства, которые соединяют вместе соответствующие классы. Например, в классе `Inventory` свойство `Orders` определено как `virtual ICollection<Order>`:

```
public virtual ICollection<Order> Orders { get; set; }
```

Тем самым инфраструктура EF информируется о том, что каждая запись базы данных `Inventory` (переименованная в класс `Car` в примерах кода) может иметь ноль или множество записей `Order`.

С другой стороны, модель `Order` имеет отношение "один к одному" с моделью `Inventory (Car)`. Модель `Order` перемещается обратно в модель `Inventory` через еще одно виртуальное свойство типа `Inventory`:

```
public virtual Car Car { get; set; }
```

В базе данных `SQL Server` внешние ключи являются свойствами, связывающими таблицы друг с другом. В приведенном примере таблица `Orders` имеет внешний ключ по имени `CarId`. В модели `Orders` он представлен следующим свойством:

```
public int CarId { get; set; }
```

На заметку! Если бы внешний ключ CarId имел тип int, допускающий значения null, тогда образовалось бы отношение “ноль к одному”.

По соглашению, если инфраструктура EF находит свойство по имени <Класс>Id, то оно будет применяться в качестве внешнего ключа для навигационного свойства типа <Класс>. Как и с другими именами классов и свойств, это можно изменить. Например, если нужно назначить свойству имя Foo, тогда класс понадобится обновить, как показано ниже:

```
public partial class Order
{
    [Column("CarId")]
    public int Foo { get; set; }
    [ForeignKey(nameof(Foo))]
    // Для краткости оставшийся код не показан.
}
```

Ленивая, энергичная и явная загрузка

Загрузка данных из базы данных в сущностные классы может осуществляться тремя разными способами: ленивым, энергичным и явным. Ленивая и энергичная загрузка основана на настройках контекста, а явная загрузка управляется разработчиком.

Ленивая загрузка

Ленивая загрузка означает, что инфраструктура EF загружает затребованный непосредственный объект (или объекты), но заменяет навигационные свойства посредниками. Назначение навигационным свойствам посредников делает возможным модификатор virtual. Когда в коде запрашивается любое свойство на навигационном свойстве, инфраструктура EF создает новое обращение к базе данных, выполняет его и заполняет детали объекта. Например, при наличии следующего кода инфраструктура EF будет запускать один запрос, чтобы получить все объекты Car, и затем для каждого объекта Car выполнять другой запрос, чтобы получить все объекты Order:

```
using (var context = new AutoLotEntities())
{
    foreach (Car c in context.Cars)
    {
        foreach (Order o in c.Orders)
        {
            WriteLine(o.OrderId);
        }
    }
}
```

Хотя загрузка только необходимых данных дает преимущество, в предыдущем примере видно, что она может стать кошмаром в смысле производительности, если не проявить внимание к тому, как ленивая загрузка утилизируется. Отключить ленивую загрузку можно, установив свойство LazyLoadingEnabled конфигурации объекта DbContext в false:

```
context.Configuration.LazyLoadingEnabled = false;
```

Если известно, что для каждого объекта автомобиля нужны объекты заказов, тогда необходимо подумать об использовании энергичной загрузки, которая рассматривается следующей.

Энергичная загрузка

Когда для объекта желательно загрузить связанные записи, написание множества запросов к реляционной базе данных будет неэффективным. Чтобы получить связанные данные, благоразумный разработчик баз данных написал бы один запрос, утилизующий соединения SQL. Именно это делает энергичная загрузка для разработчиков на C#, применяющих EF. Например, если требуются все объекты Car и все связанные с ними объекты Order, тогда предыдущий код можно изменить следующим образом:

```
using (var context = new AutoLotEntities())
{
    foreach (Car c in context.Cars.Include(c=>c.Orders))
    {
        foreach (Order o in c.Orders)
        {
            WriteLine(o.OrderId);
        }
    }
}
```

Метод Include() из LINQ информирует инфраструктуру EF о необходимости создания оператора SQL, который соединяет таблицы вместе, как если бы мы написали такой код SQL самостоятельно. Результирующий запрос, выполняемый в отношении базы данных, выглядит так:

```
SELECT
    [Project1].[CarId] AS [CarId],
    [Project1].[Make] AS [Make],
    [Project1].[Color] AS [Color],
    [Project1].[PetName] AS [PetName],
    [Project1].[C1] AS [C1],
    [Project1].[OrderId] AS [OrderId],
    [Project1].[CustId] AS [CustId],
    [Project1].[CarId1] AS [CarId1]
FROM ( SELECT
    [Extent1].[CarId] AS [CarId],
    [Extent1].[Make] AS [Make],
    [Extent1].[Color] AS [Color],
    [Extent1].[PetName] AS [PetName],
    [Extent2].[OrderId] AS [OrderId],
    [Extent2].[CustId] AS [CustId],
    [Extent2].[CarId] AS [CarId1],
    CASE WHEN ([Extent2].[OrderId] IS NULL) THEN CAST(NULL AS int)
          ELSE 1 END AS [C1]
    FROM [dbo].[Inventory] AS [Extent1]
    LEFT OUTER JOIN [dbo].[Orders] AS [Extent2]
        ON [Extent1].[CarId] = [Extent2].[CarId]
    ) AS [Project1]
ORDER BY [Project1].[CarId] ASC, [Project1].[C1] ASC
```

Точный синтаксис запроса не играет никакой роли; он показан в целях демонстрации того, что все объекты Car и связанные с ними объекты Order извлекаются за одно обращение к базе данных.

Явная загрузка

Последний способ загрузить записи — воспользоваться явной загрузкой, которая позволяет явным образом загрузить коллекцию или класс в конец навигационно-

го свойства. Для получения связанного объекта (или объектов) применяются методы `Collection()` (для коллекций) или `Property()` (для одиночных объектов) контекста, чтобы выбрать подлежащие загрузке данные, и метод `Load()`. В следующем коде оба метода демонстрируются в действии:

```
context.Configuration.LazyLoadingEnabled = false;
foreach (Car c in context.Cars)
{
    context.Entry(c).Collection(x => x.Orders).Load();
    foreach (Order o in c.Orders)
    {
        WriteLine(o.OrderId);
    }
}
foreach (Order o in context.Orders)
{
    context.Entry(o).Reference(x => x.Car).Load();
}
```

На выбор модели доступа к данным влияют потребности проекта. Если оставить ленивую загрузку включенной (стандартная настройка), то придется соблюдать осторожность, чтобы приложение не стало слишком “словоохотливым”. В случае отключения ленивой загрузки понадобится гарантировать загрузку связанных данных перед их потреблением.

Удаление данных

Удаление записей из базы данных может быть реализовано с использованием `DbSet<T>` (концептуально аналогично добавлению записей), но также с применением `EntityState`. Далее мы посмотрим на оба подхода в действии.

Удаление одной записи

Один из способов удаления одиночной записи из базы данных предусматривает определение местоположения нужного элемента в `DbSet<T>` и вызов метода `DbSet<T>.Remove()` с передачей ему этого экземпляра. Вызов метода `Remove()` приводит к удалению элемента из коллекции и установке `EntityState` в `EntityState.Deleted`. Несмотря на его удаление из `DbSet<T>`, из контекста (или базы данных) элемент не удаляется, пока не будет вызван метод `SaveChanges()`.

С таким процессом связана одна ловушка: элемент, подлежащий удалению, должен уже быть отслеженным (другими словами, загруженным в `DbSet<T>`). Распространенный паттерн используется в инфраструктуре MVC (исследуется позже в книге), где первичный ключ удаляемого элемента передается действию `Delete`. Элемент извлекается с применением `Find()`, затем удаляется из `DbSet<T>` посредством `Remove()` и, наконец, изменения сохраняются в базе данных с помощью `SaveChanges()`, как показано в примере ниже:

```
private static void RemoveRecord(int carId)
{
    // Найти запись об автомобиле, подлежащую удалению, по первичному ключу.
    using (var context = new AutoLotEntities())
    {
        // Проверить наличие записи.
        Car carToDelete = context.Cars.Find(carId);
        if (carToDelete != null)
        {
            context.Cars.Remove(carToDelete);
        }
    }
}
```

```
// Этот код предназначен чисто для демонстрации того,
// что состояние сущности изменилось на Deleted.
if (context.Entry(carToDelete).State != EntityState.Deleted)
{
    throw new Exception("Unable to delete the record");
}
context.SaveChanges();
}
}
```

На заметку! Вызов метода `Find()` перед удалением записи требует дополнительного обращения к базе данных. Сначала запись извлекается, а затем удаляется. Как вскоре будет объяснено, удаление данных также можно обеспечить, изменяя свойство `EntityState`.

Удаление множества записей

Можно также удалять сразу несколько записей, используя метод `RemoveRange()` на `DbSet<T>`. Как и в случае метода `Remove()`, подлежащие удалению элементы должны быть отслеженными.

Метод `RemoveRange()` принимает в качестве параметра `IEnumerable<T>`, что видно в следующем примере:

```
private static void RemoveMultipleRecords(IEnumerable<Car> carsToRemove)
{
    using (var context = new AutoLotEntities())
    {
        // Каждая запись должна быть загружена в DbChangeTracker.
        context.Cars.RemoveRange(carsToRemove);
        context.SaveChanges();
    }
}
```

Однако помните, что до вызова `SaveChanges()` в базе данных ничего не произойдет.

Удаление записи с использованием *EntityState*

Финальный способ удаления записи предполагает применение `EntityState`. Для этого создается экземпляр элемента, подлежащего удалению, первичному ключу нового экземпляра присваивается ключ удаляемого элемента, и свойство `EntityState` устанавливается в `EntityState.Deleted`. Такие действия приведут к добавлению элемента в `DbChangeTracker`, так что когда будет вызван метод `SaveChanges()`, запись удалится. Обратите внимание, что запись не требует предварительного запрашивания из базы данных. Все сказанное демонстрируется в показанном ниже коде.

```
private static void RemoveRecordUsingEntityState(int carId)
{
    using (var context = new AutoLotEntities())
    {
        Car carToDelete = new Car() { CarId = carId };
        context.Entry(carToDelete).State = EntityState.Deleted;
        try
        {
            context.SaveChanges();
        }
    }
}
```

```

        catch (DbUpdateConcurrencyException ex)
        {
            WriteLine(ex);
        }
    }
}

```

Здесь достигается выигрыш в производительности (т.к. не производится дополнительное обращение к базе данных), но утрачивается возможность проверки, существует ли объект в базе данных (если это имеет значение в конкретном сценарии). В случае если запись с указанным значением CarId в базе данных отсутствует, то инфраструктура EF сгенерирует исключение типа DbUpdateConcurrencyException, определенного в пространстве имен System.Data.Entity.Infrastructure. Исключение DbUpdateConcurrencyException подробно рассматривается далее в главе.

На заметку! Если элемент с таким же первичным ключом уже отслежен, тогда метод RemoveRecordUsingEntityState() потерпит неудачу, потому что отслеживание с помощью DbChangeTracker не допускает наличие сущностей с одинаковыми первичными ключами.

Обновление записи

Обновление записи следует практически такому же паттерну. Понадобится определить местоположение объекта, подлежащего обновлению, установить новые значения для свойств возвращенной сущности и сохранить изменения:

```

private static void UpdateRecord(int carId)
{
    // Найти запись об автомобиле, подлежащую обновлению, по первичному ключу.
    using (var context = new AutoLotEntities())
    {
        // Получить запись об автомобиле, обновить ее и сохранить!
        Car carToUpdate = context.Cars.Find(carId);
        if (carToUpdate != null)
        {
            WriteLine(context.Entry(carToUpdate).State);
            carToUpdate.Color = "Blue";
            WriteLine(context.Entry(carToUpdate).State);
            context.SaveChanges();
        }
    }
}

```

Обработка изменений в базе данных

В настоящем разделе мы создали решение EF, основанное на существующей базе данных. Такое решение хорошо подходит в ситуации, когда в организации есть свои администраторы баз данных, и вам предоставляется база данных, которую вы не контролируете. Если база данных со временем изменяется, то все, что вам потребуется предпринять — запустить мастер заново и повторно создать класс AutoLotEntities; классы модели будут перестроены автоматически. Удостоверьтесь, что любые добавления к классам моделей делались с использованием частичных классов, иначе после повторного запуска мастера вся выполненная ранее работа будет утеряна.

В начальном примере вами был пройден долгий путь к пониманию особенностей работы с Entity Framework.

Исходный код. Проект AutoLotConsoleApp доступен в подкаталоге Chapter_22.

Создание уровня доступа к данным AutoLotDAL

В предыдущем разделе для создания сущностей и контекста из существующей базы данных применялся мастер. Инфраструктура EF может также самостоятельно создать базу данных на основе классов модели и производного от DbContext класса. Вдобавок к созданию начальной базы данных инфраструктура EF делает возможным использование миграций с целью обновления базы данных для соответствия любым изменениям в модели. Еще лучше то, что мастер можно применять для создания начальных моделей и контекста, после чего переключиться на подход, ориентированный на C#, и использовать миграции, сохраняя базу данных в синхронизированном виде.

На заметку! Данная версия сборки AutoLotDAL.dll будет применяться в оставшихся главах книги.

Первым делом создадим новый проект библиотеки классов по имени AutoLotDAL. Удалим стандартный класс, который был создан, и добавим две папки, EF и Models. Теперь добавим в проект инфраструктуру Entity Framework, используя NuGet. Щелкнем правой кнопкой мыши на имени проекта и выберем в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet). В предыдущем примере явно добавлять EF не требовалось, потому что об этом позаботился мастер. После открытия окна NuGet Package Manager (Диспетчер пакетов NuGet) введем EntityFramework в поле поиска, выберем пакет EntityFramework и щелкнем на кнопке Install (Установить), как показано на рис. 22.7.

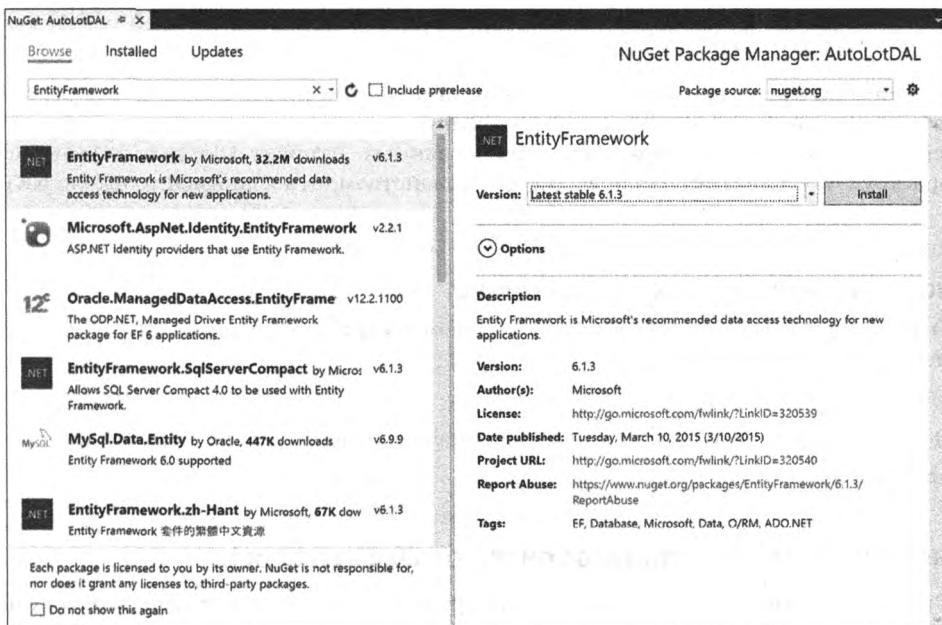


Рис. 22.7. Установка Entity Framework в проекте

После принятия изменений и условий лицензионного соглашения инфраструктура Entity Framework (версии 6.1.3 на время написания главы) установится в проект.

Добавление классов моделей

Начнем с добавления моделей (*Car.cs*, *CarPartial.cs*, *CreditRisk.cs*, *Customer.cs* и *Order.cs*) из проекта *AutoLotConsoleApp* в папку *Models* нового приложения. Скорректируем все пространства имен (изменив их на *AutoLotDAL.Models*) и избавимся от изменений, внесенных в последнем разделе.

- Изменим имена класса *Car*, конструктора и файла обратно на *Inventory*.
- Изменим имя файла *CarPartial* на *InventoryPartial* и имя класса на *Inventory*.
- В классе *Order* изменим имя типа *Car* на *Inventory*.
- Вернем свойству *Inventory.CarNickName* имя *PetName*.
- Обновим метод *ToString()* в частичном классе *Inventory* для применения *PetName*.
- В классе *Order* изменим имя *Foo* обратно на *CarId*.

Обновление класса модели *Inventory*

Откроем файл *Inventory.cs* и переместим код инициализации *HashSet<Order>* в свойство *Orders*:

```
public virtual ICollection<Order> Orders { get; set; } = new HashSet<Order>();
```

Работа код совершенно не меняется; просто используются преимущества новых средств C# для приведения кода в порядок.

Обновление класса *InventoryPartial*

Импортируем следующее пространство имен в файл *InventoryPartial.cs*:

```
using System.ComponentModel.DataAnnotations.Schema;
```

Добавим вычисляемое поле, которое объединяет значения *Make* и *Color* объекта автомобиля. Оно не будет сохраняться в базе данных, равно как не будет заполняться при материализации объекта из объекта чтения данных. Атрибут *[NotMapped]* сообщает инфраструктуре EF о том, что поле является свойством, относящимся только к .NET.

```
[NotMapped]
public string MakeColor => $"{Make} + ({Color})";
```

Обновление класса модели *Customer*

Откроем файл класса *Customer.cs* и переместим код создания *HashSet<Order>* в свойство следующего вида:

```
public virtual ICollection<Order> Orders { get; set; } = new HashSet<Order>();
```

Добавим поле только для .NET для возвращения полного имени заказчика:

```
[NotMapped]
public string FullName => FirstName + " " + LastName;
```

Обновление класса, производного от *DbContext*

Скопируем класс *AutoLotEntities* из предыдущего проекта в папку EF текущего проекта. Обновим пространство имен и изменим *DbSet<Car>* на *DbSet<Inventory>*. Также изменим *Car* на *Inventory* в методе *OnModelCreating()*.

Обновление файла *App.config*

Откроем файл *App.config* и посмотрим, какие изменения были сделаны NuGet в результате установки пакета *EntityFramework*. Большинство из них должны выглядеть знакомо. В файле *App.config* не хватает строки подключения, так что добавим ее:

```
<connectionStrings>
  <add name="AutoLotConnection"
        connectionString="data source=(LocalDb)\MSSQLLocalDB;
initial catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;
App=EntityFramework"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

На заметку! Вас может удивить вся проделанная работа, поскольку мы всего лишь воссоздали то, что было создано в результате реверсивного проектирования базы данных в предыдущем разделе. То, что мы сделали, значительно сократило объем клавиатурного ввода и сэкономило немало печатных страниц. Была воспроизведена ситуация, когда база данных еще не существует, так что все создается сначала в коде C# и затем (как вскоре будет показано) мигрирует в базу данных. Просто так получилось, что нам подвернулся хороший код, который сберег время и объем без того длинной главы.

Инициализация базы данных

Мощным средством инфраструктуры EF является возможность обеспечить соответствие базы данных и модели, а также инициализировать базу данных начальными данными. Это особенно удобно во время разработки и тестирования, т.к. позволяет восстанавливать базу данных в известном состоянии перед каждым запуском кода. Для включения средства инициализации необходимо создать класс, производный от *DropCreateDatabaseIfModelChanges<TContext>* или *DropCreateDatabaseAlways<TContext>*.

Начнем с создания в папке EF нового класса по имени *MyDataInitializer*, сделав его открытым и унаследованным от *DropCreateDatabaseAlways<AutoLotEntities>*. Добавим следующий оператор *using*:

```
using AutoLotDAL.Models;
```

Далее переопределим метод *Seed()*:

```
public class MyDataInitializer : DropCreateDatabaseAlways<AutoLotEntities>
{
    protected override void Seed(AutoLotEntities context)
    {
        base.Seed(context);
    }
}
```

Класс *DropCreateDatabaseAlways* является обобщенным классом, который типизирован для класса, производного от *DbContext*, в данном случае *AutoLotEntities*. Он будет удалять и воссоздавать базу данных каждый раз, когда выполняется инициализатор. Есть также класс *DropCreateDatabaseIfModelChanges<TContext>*, который удаляет и воссоздает базу данных, когда в модели появляются изменения.

Выполнение обновления или вставки

Обновление или вставка поддерживается в EF методом *AddOrUpdate()* типа *DbSet<T>*. Метод принимает лямбда-выражение с определением уникальности каждой

записи и список объектов, предназначенных для обновления или вставки. Если запись существует (основываясь на ключе уникальности) в базе данных, тогда она будет обновлена. Если запись не существует, то она будет вставлена. Ниже приведен пример обновления или вставки для класса `Inventory`, в котором перед вставкой записей принимается проверка свойств `Make` и `Color` каждого объекта автомобиля:

```
context.Cars.AddOrUpdate(x=>new {x.Make,x.Color},car);
```

Начальное заполнение базы данных

Для заполнения базы данных применяется метод `Seed()`, доступный в обоих классах инициализации. За счет использования метода `AddOrUpdate()` можно обеспечить восстановление базы данных в то же самое состояние без дублирования данных.

Ниже показан пример начального заполнения базы данных теми же записями, которые использовались в предыдущей главе:

```
protected override void Seed(AutoLotEntities context)
{
    var customers = new List<Customer>
    {
        new Customer {FirstName = "Dave", LastName = "Brenner"},
        new Customer {FirstName = "Matt", LastName = "Walton"},
        new Customer {FirstName = "Steve", LastName = "Hagen"},
        new Customer {FirstName = "Pat", LastName = "Walton"},
        new Customer {FirstName = "Bad", LastName = "Customer"},
    };
    customers.ForEach(x => context.Customers.AddOrUpdate(
        c=>new {c.FirstName, c.LastName},x));
    var cars = new List<Inventory>
    {
        new Inventory {Make = "VW", Color = "Black", PetName = "Zippy"},
        new Inventory {Make = "Ford", Color = "Rust", PetName = "Rusty"},
        new Inventory {Make = "Saab", Color = "Black", PetName = "Mel"},
        new Inventory {Make = "Yugo", Color = "Yellow", PetName = "Clunker"},
        new Inventory {Make = "BMW", Color = "Black", PetName = "Bimmer"},
        new Inventory {Make = "BMW", Color = "Green", PetName = "Hank"},
        new Inventory {Make = "BMW", Color = "Pink", PetName = "Pinky"},
        new Inventory {Make = "Pinto", Color = "Black", PetName = "Pete"},
        new Inventory {Make = "Yugo", Color = "Brown", PetName = "Brownie"},
    };
    context.Inventory.AddOrUpdate(x=>new {x.Make,x.Color},cars.ToArray());
    var orders = new List<Order>
    {
        new Order {Inventory= cars[0], Customer = customers[0]},
        new Order {Inventory= cars[1], Customer = customers[1]},
        new Order {Inventory= cars[2], Customer = customers[2]},
        new Order {Inventory= cars[3], Customer = customers[3]},
    };
    orders.ForEach(x => context.Orders.AddOrUpdate(c=>new {c.CarId,c.CustId},x));
    context.CreditRisks.AddOrUpdate(x=>new {x.FirstName,x.LastName},
    new CreditRisk
    {
        CustID = customers[4].CustID,
        FirstName = customers[4].FirstName,
        LastName = customers[4].LastName,
    });
}
```

Последний шаг заключается в установке инициализатора с помощью такого кода (который будет добавлен в следующем разделе):

```
Database.SetInitializer(new MyDataInitializer());
```

Тестирование сборки AutoLotDAL

Код для тестирования похож на код, который применялся при тестировании предыдущей версии сборки AutoLotDAL.dll. Позже в главе мы обновим его, чтобы использовать специальные хранилища, а пока займемся тестированием кода инициализации данных и начального заполнения. Первым делом добавим в решение новый проект консольного приложения по имени AutoLotTestDrive и установим его в качестве запускаемого. Посредством NuGet добавим в проект инфраструктуру EF и обновим элемент connectionStrings в файле App.config следующим образом:

```
<connectionStrings>
  <add name="AutoLotConnection"
        connectionString="data source=(localdb)\mssqllocaldb;
initial catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;
App=EntityFramework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Добавим ссылку на проект AutoLotDAL. Откроем файл Program.cs и добавим следующие операторы using:

```
using AutoLotDAL.EF;
using AutoLotDAL.Models;
```

Теперь поместим в метод Main() такой код:

```
static void Main(string[] args)
{
    Database.SetInitializer(new MyDataInitializer());
    Console.WriteLine("***** Fun with ADO.NET EF Code First *****\n");
    using (var context = new AutoLotEntities())
    {
        foreach (Inventory c in context.Inventory)
        {
            Console.WriteLine(c);
        }
    }
    Console.ReadLine();
}
```

Метод SetInitializer() удаляет и воссоздает базу данных, после чего запускает метод Seed() для заполнения базы данных. Наряду с тем, что результат в значительной степени совпадает с тем, что было достигнуто в предыдущем разделе, настоящая мощь EF проявляется в миграциях, которые рассматриваются далее.

Миграции Entity Framework

После того как приложение развернуто в производственной среде нельзя продолжать удалять базу данных всякий раз, когда пользователи запускают приложение. Если модель изменяется, то необходимо поддерживать базу данных в синхронизированном с ней состоянии. Именно здесь в игру вступают миграции EF. Перед созданием первой миграции мы внесем некоторые изменения, чтобы проиллюстрировать проблему. Откроем файл Program.cs и закомментируем следующую строку:

```
Database.SetInitializer(new MyDataInitializer());
```

На заметку! Как обсуждалось ранее, инициализатор удаляет и повторно создает базу данных либо при каждом запуске приложения, либо в случае изменения модели. Если не закомментировать строку с вызовом `SetInitializer()`, тогда проработать материал следующего раздела не удастся.

Создание начальной миграции

Каждый раз, когда контекст создается и применяется для выполнения операций в базе данных, он проверяет, существует ли таблица `__MigrationHistory`. При ее наличии контекст проверяет самую последнюю запись в таблице и сравнивает хеш текущей модели EF с самым последним хешем, сохраненным в таблице. В случае если указанные хеши отличаются, тогда инфраструктура EF сгенерирует исключение.

На заметку! При создании модели из существующей базы данных таблица `__MigrationHistory` не создается. Почему это имеет значение? Когда создается экземпляр класса `DbContext` и перед первым обращением к базе данных из специального кода, инфраструктура EF проверяет хронологию миграций. Поскольку таблица `__MigrationHistory` не существует, генерируется последовательность исключений, которые поглощаются инфраструктурой. Как вам хорошо известно, исключения могут быть затратными операциями и потому порождать проблему в плане производительности. Даже если вы не планируете использовать миграции, то все равно должны включить их, как объясняется в следующем разделе.

Включение миграций

Начнем с удаления базы данных `AutoLot` с применением окна `Object Explorer` (Проводник объектов) инструмента `SSMS`. Включим миграции в проекте, открыв окно `Package Manager Console` (Консоль диспетчера пакетов), которое является инструментом командной строки, управляющим пакетами `NuGet`, путем выбора пункта меню `View⇒Other Windows⇒Package Manager Console` (Вид⇒Другие окна⇒Консоль диспетчера пакетов). Удостоверимся, что в поле `Default Project` (Стандартный проект) указано имя `AutoLotDAL` и введем команду `enable-migrations` (рис. 22.8).

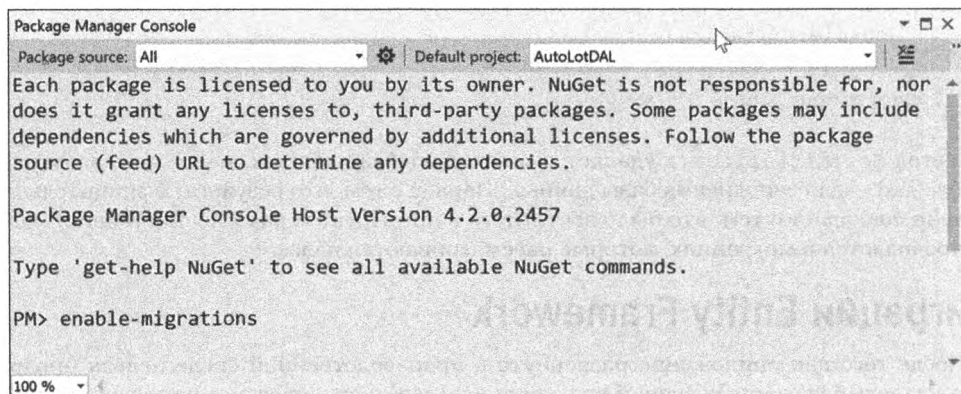


Рис. 22.8. Включение миграций для проекта `AutoLotDAL`

Появится сообщение “Checking if the context targets an existing database...” (“Проверка, нацелен ли контекст на существующую базу данных...”). Создается папка `Migrations` с одним файлом класса по имени `Configuration.cs`.

Давайте исследуем класс `Configuration`. Код в конструкторе указывает инфраструктуре EF о необходимости отключения автоматических миграций (что будет делаться большую часть времени, поскольку нам нужен контроль над тем, как работают миграции). Метод `Seed()` позволяет добавлять данные в базу данных подобно методу `Seed()` в инициализаторе базы данных.

```
internal sealed class Configuration :
    DbMigrationsConfiguration<AutoLotDAL.EF.AutoLotEntities>
{
    public Configuration()
    {
        AutomaticMigrationsEnabled = false;
    }

    protected override void Seed(AutoLotDAL.EF.AutoLotEntities context)
    {
    }
}
```

Формирование начальной миграции

В окне Package Manager Console введем следующую команду:

```
add-migration Initial
```

Команда приводит к созданию в папке `Migrations` дополнительного файла, имя которого выглядит примерно так: `201707262033409_Initial.cs`. Имя файла начинается с даты и времени центрального процессора, после чего находится название миграции. Такой формат именования позволяет инфраструктуре EF запускать миграции в правильном хронологическом порядке (если миграций более одной).

Откроем файл `201707262033409_Initial.cs` и исследуем два метода, один с именем `Up()` и еще один с именем `Down()`. Метод `Up()` предназначен для применения изменений в базе данных, а метод `Down()` — для отката изменений. Когда применяется какая-то миграция, все более ранние миграции, которые еще не применялись, будут применены за счет выполнения методов `Up()`, и любые более поздние миграции автоматически подвергаются откату с использованием методов `Down()`.

Обновление базы данных

Чтобы обновить базу данных, введем в окне Package Manager Console команду:

```
update-database
```

В результате создается база данных и таблицы, а также таблица `__MigrationHistory`, куда добавляется строка для только что примененной миграции:

MigrationId	ContextKey	Model	ProductVersion
201707262033409_Initial	AutoLotDAL.Migrations.Configuration	0x1F8B08000000...	6.1.3-40302

Теперь база данных и модель синхронизированы, поэтому можно безопасно заняться обновлением модели.

Обновление модели

В модель необходимо внести несколько изменений. После того, как процесс их внесения завершен, мы создадим еще одну миграцию, чтобы обеспечить синхронизацию базы данных.

Добавление класса *EntityBase*

Имена имеющихся моделей не согласованы с их первичными ключами, и мы хотим изменить положение дел. Мы собираемся назначить всем полям первичных ключей имя *Id* и поместить поля *Id* в базовый класс. Начнем с создания новой папки по имени *Base* внутри папки *Models*. Добавим новый файл класса под названием *EntityBase.cs*. Приведем код к следующему виду (обратите внимание на оператор *using*):

```
using System.ComponentModel.DataAnnotations;

namespace AutoLotDAL.Models.Base
{
    public class EntityBase
    {
        [Key]
        public int Id { get; set; }
    }
}
```

Модифицируем все классы моделей, сделав их унаследованными от *EntityBase*, удалим поля первичных ключей и обновим имена внешних ключей в классе *Order*. Также обновим навигационные свойства класса *Order*, добавив к ним аннотацию данных *ForeignKey*. Вот как выглядит измененный класс *Order*:

```
public partial class Order : EntityBase
{
    public int CustomerId { get; set; }
    public int CarId { get; set; }
    [ForeignKey(nameof(CustomerId))]
    public virtual Customer Customer { get; set; }
    [ForeignKey(nameof(CarId))]
    public virtual Inventory Car { get; set; }
}
```

Понадобится также исправить ошибки построения в *MyDataInitializer*.

Добавление свойства *TimeStamp*

Еще одно изменение связано с добавлением проверки параллелизма к базе данных. Для этого ко всем таблицам будет добавлено свойство *TimeStamp*, использующее аннотацию данных *TimeStamp*. Модифицируем класс *EntityBase*, как показано ниже:

```
public class EntityBase
{
    [Key]
    public int Id { get; set; }
    [TimeStamp]
    public byte[] TimeStamp { get; set; }
}
```

Аннотация данных *TimeStamp* отображает поле на тип данных *RowVersion*, поддерживаемый *SQL Server*, который в языке *C#* представлен как *byte[]*. Теперь поле будет принимать участие в проверке параллелизма, рассматриваемой далее в главе.

Обновление класса *CreditRisk*

Наконец, мы создадим уникальный индекс на свойствах *FirstName* и *LastName* с применением аннотаций данных. Поскольку это составной ключ, также понадобится указать имя для индекса и порядок для каждого столбца в индексе. В текущем примере именем индекса является *IDX_CreditRisk_Name*, порядком следования столбцов для ин-

декса — LastName и затем FirstName, а сам индекс создается как уникальный. Добавим следующий оператор using:

```
using System.ComponentModel.DataAnnotations.Schema;
```

Обновим класс, приведя его к такому виду:

```
public partial class CreditRisk : EntityBase
{
    [StringLength(50)]
    [Index("IDX_CreditRisk_Name", IsUnique = true, Order=2)]
    public string FirstName { get; set; }
    [StringLength(50)]
    [Index("IDX_CreditRisk_Name", IsUnique = true, Order = 1)]
    public string LastName { get; set; }
}
```

Создание финальной миграции

Модель обновлена, так что самое время создать финальную миграцию для рассматриваемого примера. Введем в окне Package Manager Console следующую команду:

```
add-migration Final
```

Команда создает в папке Migrations дополнительный файл под названием 201707262025040_Final.cs. Если запустить миграцию прямо сейчас, то возникнут ошибки. В конце концов, процесс не идеален! Необходимо внести небольшое изменение в код метода Up(). Переместим последние четыре строки вверх в позицию, выделенную полужирным.

```
public override void Up()
{
    DropForeignKey("dbo.Orders", "CarId", "dbo.Inventory");
    DropForeignKey("dbo.Orders", "CustId", "dbo.Customers");
    RenameColumn(table: "dbo.Orders", name: "CustId", newName: "CustomerId");
    RenameIndex(table: "dbo.Orders", name: "IX_CustId", newName: "IX_CustomerId");
    DropPrimaryKey("dbo.Inventory");
    DropPrimaryKey("dbo.Orders");
    DropPrimaryKey("dbo.Customers");
    DropPrimaryKey("dbo.CreditRisks");
    DropColumn("dbo.Inventory", "CarId");
    DropColumn("dbo.Orders", "OrderId");
    DropColumn("dbo.Customers", "CustID");
    DropColumn("dbo.CreditRisks", "CustID");
    AddColumn("dbo.Inventory", "Id", c => c.Int(nullable: false, identity: true));
    AddColumn("dbo.Orders", "Id", c => c.Int(nullable: false, identity: true));
    AddColumn("dbo.Customers", "Id", c => c.Int(nullable: false, identity: true));
    AddColumn("dbo.CreditRisks", "Id", c => c.Int(nullable: false, identity: true));
    AddPrimaryKey("dbo.Inventory", "Id");
    AddPrimaryKey("dbo.Orders", "Id");
    AddPrimaryKey("dbo.Customers", "Id");
    AddPrimaryKey("dbo.CreditRisks", "Id");
    CreateIndex("dbo.CreditRisks", new[] { "LastName", "FirstName" },
        unique: true, name: "IDX_CreditRisk_Name");
    AddForeignKey("dbo.Orders", "CarId", "dbo.Inventory", "Id");
    AddForeignKey("dbo.Orders", "CustomerId", "dbo.Customers", "Id",
        cascadeDelete: true);
}
```



```
// DropColumn("dbo.Inventory", "CarId");
// DropColumn("dbo.Orders", "OrderId");
// DropColumn("dbo.Customers", "CustID");
// DropColumn("dbo.CreditRisks", "CustID");
}
```

Потребуется также обновить код метода `Down()`:

```
public override void Down()
{
    DropForeignKey("dbo.Orders", "CustomerId", "dbo.Customers");
    DropForeignKey("dbo.Orders", "CarId", "dbo.Inventory");
    DropPrimaryKey("dbo.CreditRisks");
    DropPrimaryKey("dbo.Customers");
    DropPrimaryKey("dbo.Orders");
    DropPrimaryKey("dbo.Inventory");
    DropColumn("dbo.CreditRisks", "Id");
    DropColumn("dbo.Customers", "Id");
    DropColumn("dbo.Orders", "Id");
    DropColumn("dbo.Inventory", "Id");
    AddColumn("dbo.CreditRisks", "CustID",
        c => c.Int(nullable: false, identity: true));
    AddColumn("dbo.Customers", "CustID",
        c => c.Int(nullable: false, identity: true));
    AddColumn("dbo.Orders", "OrderId", c => c.Int(nullable: false, identity: true));
    AddColumn("dbo.Inventory", "CarId", c => c.Int(nullable: false, identity: true));
    // DropForeignKey("dbo.Orders", "CustomerId", "dbo.Customers");
    // DropForeignKey("dbo.Orders", "CarId", "dbo.Inventory");
    DropIndex("dbo.CreditRisks", "IDX_CreditRisk_Name");
    // DropPrimaryKey("dbo.CreditRisks");
    // DropPrimaryKey("dbo.Customers");
    // DropPrimaryKey("dbo.Orders");
    // DropPrimaryKey("dbo.Inventory");
    // DropColumn("dbo.CreditRisks", "Id");
    // DropColumn("dbo.Customers", "Id");
    // DropColumn("dbo.Orders", "Id");
    // DropColumn("dbo.Inventory", "Id");
    AddPrimaryKey("dbo.CreditRisks", "CustID");
    AddPrimaryKey("dbo.Customers", "CustID");
    AddPrimaryKey("dbo.Orders", "OrderId");
    AddPrimaryKey("dbo.Inventory", "CarId");
    RenameIndex(table: "dbo.Orders", name: "IX_CustomerId", newName: "IX_CustId");
    RenameColumn(table: "dbo.Orders", name: "CustomerId", newName: "CustId");
    AddForeignKey("dbo.Orders", "CustId", "dbo.Customers", "CustID",
        cascadeDelete: true);
    AddForeignKey("dbo.Orders", "CarId", "dbo.Inventory", "CarId");
}
```

Последней задачей является обновление базы данных, используя окно Package Manager Console. После ввода команды `update-database` мы получим сообщение о том, что миграция была применена.

Начальное заполнение базы данных

Скопируем код метода `Seed()` из класса `MyDataInitializer` в метод `Seed()` класса `Configure`. С помощью окна Package Manager Console снова обновим базу данных, после чего выполнится метод `Seed()`, заполняя базу начальными данными.

Добавление хранилищ для многократного использования кода

Распространенный паттерн проектирования для доступа к данным называется "Хранилище" (Repository). Согласно описанию Мартина Фаулера сущность этого паттерна заключается в том, чтобы служить посредником между предметной областью и уровнями отображения данных.

На заметку! Рассматриваемый далее материал не предназначен (да и не претендует) быть точной интерпретацией паттерна проектирования Мартина Фаулера. Если вас интересует оригинальный паттерн, побудивший нас написать свой код, тогда можете ознакомиться с дополнительными сведениями на веб-сайте Мартина Фаулера по адресу <https://www.martinfowler.com/eaCatalog/repository.html>.

Добавление интерфейса IRepo

Наша версия паттерна начинается с интерфейса, который открывает доступ к подавляющему большинству стандартных методов, которые будут применяться в библиотеке доступа к данным. Для начала добавим в проект AutoLotDAL новую папку по имени Repos. Затем добавим в папку Repos новый интерфейс под названием IRepo. Модифицируем оператор using следующим образом:

```
using System.Collections.Generic;
```

Ниже приведен полный интерфейс:

```
public interface IRepo<T>
{
    int Add(T entity);
    int AddRange(IList<T> entities);
    int Save(T entity);
    int Delete(int id, byte[] timeStamp);
    int Delete(T entity);
    T GetOne(int? id);
    List<T> GetAll();

    List<T> ExecuteQuery(string sql);
    List<T> ExecuteQuery(string sql, object[] sqlParametersObjects);
}
```

Добавление класса BaseRepo

Теперь добавим в папку Repos еще один класс по имени BaseRepo. Он будет реализовывать интерфейс IRepo и предоставлять основную функциональность для хранилищ, специфических к типам (рассматриваются следующими). Обновим операторы using:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using AutoLotDAL.EF;
using AutoLotDAL.Models.Base;
```

Сделаем класс открытым и реализующим интерфейсы IRepo и IDisposable. Внутри конструктора создадим новый экземпляр AutoLotEntities и присвоим его переменной

уровня класса. Метод `Set<T>()` указывает на свойство контекста, которое имеет тот же тип, что и `T`, такое как свойство `Cars`, и доступное через `Set<Inventory>`. Кроме того, добавим защищенное свойство для открытия доступа к контексту производным классам. Код представлен ниже:

```
public class BaseRepo<T> : IDisposable, IRepo<T> where T:EntityBase, new()
{
    private readonly DbSet<T> _table;
    private readonly AutoLotEntities _db;
    public BaseRepo()
    {
        _db = new AutoLotEntities();
        _table = _db.Set<T>();
    }
    protected AutoLotEntities Context => _db;
    public void Dispose()
    {
        _db?.Dispose();
    }
}
```

Реализация вспомогательного метода *SaveChanges()*

Добавим метод, предназначенный для помещения в оболочку метода `SaveChanges()` контекста. Обычно с вызовом методов подобного рода связан значительный объем кода, отвечающего за обработку ошибок, поэтому лучше написать такой код только раз и поместить его в базовый класс. Обработчики исключений для метода `SaveChanges()` класса `DbContext` реализованы как заглушки. В производственном приложении потребуется надлежащим образом обрабатывать любые исключения.

```
internal int SaveChanges()
{
    try
    {
        return _db.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
        // Генерируется, когда возникает ошибка, связанная с параллелизмом.
        // Пока что просто повторно сгенерировать исключение.
        throw;
    }
    catch (DbUpdateException ex)
    {
        // Генерируется, когда обновление базы данных терпит неудачу.
        // Проверить внутреннее исключение (исключения), чтобы получить
        // дополнительные сведения и выяснить, на какие объекты это повлияло.
        // Пока что просто повторно сгенерировать исключение.
        throw;
    }
    catch (CommitFailedException ex)
    {
        // Обработать здесь отказы транзакции.
        // Пока что просто повторно сгенерировать исключение.
        throw;
    }
}
```

```

catch (Exception ex)
{
    // Произошло какое-то другое исключение, которое должно быть обработано.
    throw;
}
}

```

На заметку! Создание нового экземпляра `DbContext` может оказаться затратным процессом с точки зрения производительности. В коде примера новый экземпляр `AutoLotEntities` создается с каждым экземпляром хранилища. Если такое действие не выполняется настолько хорошо, как хотелось бы (или необходимо), тогда следует обдумать возможность использования только одного контекстного класса и его разделение между хранилищами. Не существует единственного правильного способа кодирования этого, потому что каждая ситуация отличается и должна быть подстроена под конкретное приложение.

Извлечение записей

Метод `GetOne()` представляет собой оболочку для метода `Find()` класса `DbSet<T>`. Аналогично метод `GetAll()` является оболочкой для метода `ToList()`. Ниже показан код:

```

public T GetOne(int? id) => _table.Find(id);
public virtual List<T> GetAll() => _table.ToList();

```

Извлечение записей с помощью SQL

Последние четыре метода интерфейса, подлежащие реализации, работают со строками кода SQL. Они передают строку и параметры `DbSet<T>`:

```

public List<T> ExecuteQuery(string sql) => _table.SqlQuery(sql).ToList();
public List<T> ExecuteQuery(string sql, object[] sqlParametersObjects)
    => _table.SqlQuery(sql, sqlParametersObjects).ToList();

```

На заметку! Вы должны соблюдать предельную осторожность при запуске необработанных операторов SQL в отношении хранилища данных, особенно если строка принимает ввод от пользователя. Поступая так, вы делаете свое приложение уязвимым к атакам внедрением SQL. Безопасность в настоящей книге не рассматривается, но мы хотим подчеркнуть опасность выполнения необработанных операторов SQL.

Добавление записей

Методы `Add()` представляют собой оболочки для связанных методов `Add()` контекста. Преимущество заключается в инкапсуляции метода `SaveChanges()` и связанной обработки ошибок.

```

public int Add(T entity)
{
    _table.Add(entity);
    return SaveChanges();
}
public int AddRange(IList<T> entities)
{
    _table.AddRange(entities);
    return SaveChanges();
}

```

На заметку! В коде примера изменения сохраняются в базу данных каждый раз, когда методы выполняются. Код определенно можно модифицировать, чтобы обеспечить сохранение изменений как пакета.

Обновление записей

В методе `Save()` сначала свойство `EntityState` сущности устанавливается в `EntityState.Modified` и затем вызывается `SaveChanges()`. Установка состояния гарантирует, что контекст передаст изменения на сервер. Вот необходимый код:

```
public int Save(T entity)
{
    _db.Entry(entity).State = EntityState.Modified;
    return SaveChanges();
}
```

Удаление записей

Для метода `Delete()` мы добавим похожий код. Если вызывающий код передает объект, то обобщенные методы в `BaseRepo` устанавливают состояние в `EntityState.Deleted` и вызывают `SaveChanges()`. Если вызывающий код передает значение ключа и отметку времени, тогда создается новый объект, свойство `EntityState` изменяется на `Deleted` и вызывается метод `SaveChanges()`. Код выглядит следующим образом:

```
public int Delete(int id, byte[] timeStamp)
{
    _db.Entry(new T() { Id = id, Timestamp = timeStamp }).State = EntityState.Deleted;
    return SaveChanges();
}
public int Delete(T entity)
{
    _db.Entry(entity).State = EntityState.Deleted;
    return SaveChanges();
}
```

Хранилища, специфичные для сущностей

Данное проектное решение делает возможными хранилища, специфичные для сущностей, которые предоставляют дополнительную функциональность, такую как сортированный поиск или энергичная выборка. В текущий момент они не нужны, но будут созданы для будущего применения. Хранилища, специфичные для сущностей, следуют тому же самому паттерну: наследование от `BaseRepo<T>` и добавление специальных методов доступа к данным. Например, вот класс `InventoryRepo` со специальным методом `GetAll()`:

```
public class InventoryRepo : BaseRepo<Inventory>
{
    public override List<Inventory> GetAll()
        => Context.Inventory.OrderBy(x=>x.PetName).ToList();
}
```

Продолжение тестирования AutoLotDAL

Теперь у нас есть законченный уровень доступа к данным, а потому самое время вернуться к проекту `AutoLotTestDrive` и продолжить писать код для него.

Вывод инвентаризационных записей

Добавим в метод `Main()` класса `Program` следующий код:

```
Console.WriteLine("***** Using a Repository *****\n");
using (var repo = new InventoryRepo())
{
    foreach (Inventory c in repo.GetAll())
    {
        Console.WriteLine(c);
    }
}
```

Будет выведен список автомобилей, отсортированный по `PetName`.

Добавление инвентаризационных записей

Добавление новых записей демонстрирует простоту обращения к EF, используя хранилище. Определим новый метод `AddNewRecord()`, в котором создается экземпляр `InventoryRepo` и на нем вызывается `Add()`:

```
private static void AddNewRecord(Inventory car)
{
    // Добавить запись в таблицу Inventory базы данных AutoLot.
    using (var repo = new InventoryRepo())
    {
        repo.Add(car);
    }
}
```

Редактирование записей

Сохранять изменения в записях в той же степени просто. Метод `UpdateRecord()` извлекает объект `Inventory`, вносит в него некоторые изменения и вызывает метод `Save()` на экземпляре `InventoryRepo`:

```
private static void UpdateRecord(int carId)
{
    using (var repo = new InventoryRepo())
    {
        // Извлечь запись об автомобиле, изменить ее и сохранить.
        var carToUpdate = repo.GetOne(carId);
        if (carToUpdate == null) return;
        carToUpdate.Color = "Blue";
        repo.Save(carToUpdate);
    }
}
```

Удаление записей

Удалять записи можно с помощью сущности или свойств ключа сущности, в данном случае `Id` и `Timestamp`. Параллелизм будет вскоре рассмотрен, а пока просто имейте в виду, что свойство `Timestamp` требуется для уникальной идентификации записи.

```
private static void RemoveRecordByCar(Inventory carToDelete)
{
    using (var repo = new InventoryRepo())
    {
        repo.Delete(carToDelete);
    }
}
```

```
private static void RemoveRecordById(int carId, byte[] timeStamp)
{
    using (var repo = new InventoryRepo())
    {
        repo.Delete(carId, timeStamp);
    }
}
```

Параллелизм

Проблемы параллелизма в многопользовательских приложениях являются распространенным явлением. Если не программировать с учетом параллелизма, то когда два пользователя обновляют одну и ту же запись, в выигрыше оказывается последний из них. Это может быть вполне подходящим для приложения, но если нет, то EF и SQL Server предоставляют удобный механизм для проверки наличия конфликтов, связанных с параллелизмом.

Аннотация данных `Timestamp` заставляет SQL Server создать столбец `RowVersion`. Значение в таком столбце поддерживается SQL Server и обновляется при каждом добавлении или обновлении записи. Аннотация данных `Timestamp` также изменяет то, как EF строит и запускает запросы, которые обновляют или удаляют данные из базы. Скажем, запрос удаления теперь содержит в конструкции `where` поле `Timestamp`, а не ищет один лишь первичный ключ. Например, в случае удаления записи `Inventory` сгенерированный оператор SQL принимает следующий вид:

```
Execute NonQuery "DELETE [dbo].[Inventory] WHERE (([CarId] = @0)
AND ([Timestamp] = @1))"
```

Если два пользователя извлекают одну и ту же запись, обновляют ее и затем пытаются сохранить, то первый пользователь успешно обновит запись, потому что значение поля `Timestamp` в его объекте совпадает со значением столбца `RowVersion` записи (предполагая отсутствие других действий в отношении записи). Когда второй пользователь пытается сохранить запись, то ничего не произойдет, т.к. конструкция `where` не сможет найти запись. В итоге генерируется исключение `DbUpdateConcurrencyException`, поскольку количество записей, модифицированных (или удаленных) в `DbChangeTracker` (в данном примере одна), не совпадает с количеством записей, на которые действительно было оказано воздействие (ноль).

Класс `DbUpdateConcurrencyException` открывает доступ к коллекции `Entries`, хранящей все сущности, которые не удалось успешно обработать. Каждый элемент коллекции позволяет получить доступ к исходным свойствам, текущим свойствам и (через обращение к базе данных) к текущим свойствам из базы данных. Сказанное демонстрируется в приведенном далее коде. Наличие двух экземпляров `InventoryRepo` обеспечивает генерацию исключения, связанного с параллелизмом.

```
private static void TestConcurrency()
{
    var repo1 = new InventoryRepo();
    // Использовать второе хранилище, чтобы гарантировать
    // применение отличающегося контекста.
    var repo2 = new InventoryRepo();
    var car1 = repo1.GetOne(1);
    var car2 = repo2.GetOne(1);
    car1.PetName = "NewName";
    repo1.Save(car1);
    car2.PetName = "OtherName";
```

```

try
{
    repo2.Save(car2);
}
catch (DbUpdateConcurrencyException ex)
{
    var entry = ex.Entries.Single();
    var currentValues = entry.CurrentValues;
    var originalValues = entry.OriginalValues;
    var dbValues = entry.GetDatabaseValues();
    Console.WriteLine(" ***** Concurrency *****");
    Console.WriteLine("Type\tPetName");
    Console.WriteLine($"Current:\t{currentValues[nameof(Inventory.PetName)]}");
    Console.WriteLine($"Orig:\t{originalValues[nameof(Inventory.PetName)]}");
    Console.WriteLine($"db:\t{dbValues[nameof(Inventory.PetName)]}");
}
}
**** Concurrency ****
Type: PetName
Current: OtherName
Orig: Zippy
Db: NewName

```

То, что нужно делать с этой информацией, зависит от требований приложения. В EF и SQL Server доступны инструменты, необходимые для выявления конфликтов, относящихся к параллелизму.

Перехват

Последней темой, касающейся EF, является перехват. В рассмотренных ранее примерах вы видели, что “за кулисами” происходит немало “магических” действий для перемещения данных из хранилища в объектную модель и наоборот. Перехват — это выполнение кода на различных фазах такого процесса.

Интерфейс IDbCommandInterceptor

Все начинается с интерфейса `IDbCommandInterceptor`:

```

public interface IDbCommandInterceptor : IDbInterceptor
{
    void NonQueryExecuted(DbCommand command,
        DbCommandInterceptionContext<int> interceptionContext);
    void NonQueryExecuting(DbCommand command,
        DbCommandInterceptionContext<int> interceptionContext);
    void ReaderExecuted(DbCommand command,
        DbCommandInterceptionContext<DbDataReader> interceptionContext);
    void ReaderExecuting(DbCommand command,
        DbCommandInterceptionContext<DbDataReader> interceptionContext);
    void ScalarExecuted(DbCommand command,
        DbCommandInterceptionContext<object> interceptionContext);
    void ScalarExecuting(DbCommand command,
        DbCommandInterceptionContext<object> interceptionContext);
}

```

Как видите, интерфейс `IDbCommandInterceptor` содержит методы, которые вызываются инфраструктурой EF непосредственно до и после возникновения определен-

ных событий. Например, метод `ReaderExecuting()` вызывается прямо *перед* запуском средства чтения, а метод `ReaderExecuted()` — сразу *после* того, как средство чтения выполнилось. Мы рассмотрим пример вывода на консоль сообщений в каждом из упомянутых методов. В производственной системе логика будет соответствовать существующим требованиям.

Добавление перехвата в AutoLotDAL

Добавим в проект AutoLotDAL новую папку по имени `Interception` и поместим в нее новый класс `ConsoleWriterInterceptor`. Сделаем этот класс открытым, добавим оператор `using` для пространства имен `System.Data.Entity.Infrastructure.Interception` и унаследуем класс от интерфейса `IDbCommandInterceptor`. После реализации необходимых членов код должен выглядеть следующим образом:

```
public class ConsoleWriterInterceptor : IDbCommandInterceptor
{
    public void NonQueryExecuting(DbCommand command,
        DbCommandInterceptionContext<int> interceptionContext)
    {
    }

    public void NonQueryExecuted(DbCommand command,
        DbCommandInterceptionContext<int> interceptionContext)
    {
    }

    public void ReaderExecuting(DbCommand command,
        DbCommandInterceptionContext<DbDataReader> interceptionContext)
    {
    }

    public void ReaderExecuted(DbCommand command,
        DbCommandInterceptionContext<DbDataReader> interceptionContext)
    {
    }

    public void ScalarExecuting(DbCommand command,
        DbCommandInterceptionContext<object> interceptionContext)
    {
    }

    public void ScalarExecuted(DbCommand command,
        DbCommandInterceptionContext<object> interceptionContext)
    {
    }
}
```

Для простоты мы собираемся только выводить на консоль информацию о том, является ли вызов асинхронным, и текст команды.

Добавим оператор `using static System.Console;` и закрытый метод по имени `WriteInfo()`, который принимает параметры типа `bool` и `string`.

Ниже показан его код:

```
private void WriteInfo(bool isAsync, string commandText)
{
    WriteLine($"IsAsync: {isAsync}, Command Text: {commandText}");
}
```

В каждый метод интерфейса поместим вызов `WriteInfo()` следующего вида:

```
WriteInfo(interceptionContext.IsAsync, command.CommandText);
```

Регистрация перехватчика

Перехватчики могут регистрироваться в коде или в конфигурационном файле приложения. Регистрация в коде изолирует их от изменений конфигурационного файла и по этой причине гарантирует, что они всегда будут зарегистрированы. Если нужна более высокая гибкость, тогда конфигурационный файл может оказаться лучшим вариантом. В рассматриваемом примере мы зарегистрируем перехватчик в коде.

Откроем файл `AutoLotEntities.cs` и добавим в его начало такой оператор `using`:

```
using System.Data.Entity.Infrastructure.Interception;
```

Поместим в конструктор следующую строку кода:

```
DbInterception.Add(new ConsoleWriterInterceptor());
```

Выполнив один из тестовых методов, приведенных ранее в главе, можно заметить дополнительный вывод в окне консоли. Хотя данный пример прост, он демонстрирует возможности перехватчика.

На заметку! Класс `DbCommandInterceptionContext<T>` содержит намного больше членов, чем те, что представлены здесь. За дополнительными сведениями обращайтесь в документацию .NET Framework 4.7 SDK.

Добавление перехватчика DatabaseLogger

Инфраструктура EF теперь поставляется со встроенным регистрирующим перехватчиком, который позволяет выполнять простую регистрацию в журнале. Чтобы добавить такую возможность, откроем файл `AutoLotEntities.cs` и закомментируем код нашего консольного регистратора. Добавим статический член только для чтения типа `DatabaseLogger` (из пространства имен `System.Data.Entity.Infrastructure.Interception`). Его конструктор принимает два параметра; первый из них — имя журнального файла, а второй необязательный параметр указывает, должна ли информация записываться в журнальный файл (стандартным значением является `false`). Вызовем в конструкторе метод `StartLogging()` объекта перехватчика и добавим этот объект в список перехватчиков. Ниже показан модифицированный код:

```
static readonly DatabaseLogger DatabaseLogger =
    new DatabaseLogger("sqllog.txt", true);
public AutoLotEntities() : base("name=AutoLotConnection")
{
    // DbInterception.Add(new ConsoleWriterInterceptor());
    DatabaseLogger.StartLogging();
    DbInterception.Add(DatabaseLogger);
}
```

Последнее изменение касается использования реализации `DbContext` паттерна `IDisposable` для прекращения регистрации в журнале и удаления объекта перехватчика:

```
protected override void Dispose(bool disposing)
{
    DbInterception.Remove(DatabaseLogger);
    DatabaseLogger.StopLogging();
    base.Dispose(disposing);
}
```

События ObjectMaterialized и SavingChanges

Создание специального перехватчика может предоставить значительный объем функциональности, но также и требовать много работы. Два из наиболее распространенных сценариев поддерживаются двумя событиями класса `ObjectContext` — `ObjectMaterialized` и `SavingChanges`. Событие `ObjectMaterialized` инициируется, когда объект реконструируется из хранилища данных, прямо перед возвращением экземпляра вызывающему коду, а событие `SavingChanges` происходит, когда данные объекта распространяются в хранилище, сразу после вызова метода `SaveChanges()` контекста.

Доступ к объектному контексту

Класс `DbContext` не открывает объект контекста напрямую, но реализует интерфейс `IObjectContextAdapter`, который предоставляет доступ к `ObjectContext`. Чтобы обратиться к `ObjectContext`, необходимо привести `AutoLotEntities` к `IObjectContextAdapter`. Модифицируем операторы `using` следующим образом:

```
using System;
using System.Data.Entity;
using System.Data.Entity.Core.Objects;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.Infrastructure.Interception;
using AutoLotDAL.Interception;
using AutoLotDAL.Models;
```

Далее обновим конструктор и добавим два обработчика событий:

```
public AutoLotEntities() : base("name=AutoLotConnection")
{
    // Код перехватчика.
    var context = (this as IObjectContextAdapter).ObjectContext;
    context.ObjectMaterialized += OnObjectMaterialized;
    context.SavingChanges += OnSavingChanges;
}
private void OnSavingChanges(object sender, EventArgs EventArgs)
{
}
private void OnObjectMaterialized(object sender,
    System.Data.Entity.Core.Objects.ObjectMaterializedEventArgs e)
{
}
```

Событие ObjectMaterialized

Аргументы события `ObjectMaterialized` обеспечивают доступ к реконструируемой сущности. Данное событие инициируется немедленно после заполнения свойств класса модели инфраструктурой EF и перед тем, как контекст передаст его вызывающему коду. Хотя событие `ObjectMaterialized` в настоящей главе не применяется, оно неоценимо при работе с WPF, как вы увидите в главе 28.

Событие SavingChanges

Как уже упоминалось, событие `SavingChanges` возникает сразу после вызова метода `SaveChanges()` на `DbContext`, но перед обновлением базы данных. За счет обращения к объекту `ObjectContext`, переданному обработчику события, все сущности в транзакции доступны через свойство `ObjectStateEntry` класса `DbContext`. Некоторые основные свойства кратко описаны в табл. 22.6.

Таблица 22.6. Основные свойства класса EntityStateEntry

Член	Описание
CurrentValues	Текущие значения свойств сущности
OriginalValues	Исходные значения свойств сущности
Entity	Сущность, представленная объектом EntityStateEntry
State	Текущее состояние сущности (например, Modified, Added, Deleted)

Класс EntityStateEntry также предлагает набор методов, которые можно использовать в отношении сущности. Часть методов перечислена в табл. 22.7.

Таблица 22.7. Основные методы класса EntityStateEntry

Метод	Описание
AcceptChanges()	Принимает текущие значения как исходные
ApplyCurrentValues()	Устанавливает текущие значения в соответствии со значениями предоставленного объекта
ApplyOriginalValues()	Устанавливает исходные значения в соответствии со значениями предоставленного объекта
ChangeState()	Обновляет состояние сущности
GetModifiedProperties()	Возвращает имена всех измененных свойств
IsPropertyChanges()	Проверяет специфичное свойство на предмет изменений
RejectPropertyChanges()	Отклоняет любые изменения, внесенные в свойство

В результате появляется возможность писать код, который отклоняет любые изменения цвета автомобиля, если новый цвет является красным:

```
private void OnSavingChanges(object sender, EventArgs EventArgs)
{
    // Параметр sender имеет тип ObjectContext.
    // Можно получать текущие и исходные значения,
    // а также отменять/модифицировать операцию
    // сохранения любым желаемым образом.
    var context = sender as ObjectContext;
    if (context == null) return;
    foreach (EntityStateEntry item in
        context.ObjectStateManager.GetObjectStateEntries(
            EntityState.Modified | EntityState.Added))
    {
        // Делать здесь что-то важное.
        if ((item.Entity as Inventory) != null)
        {
            var entity = (Inventory) item.Entity;
            if (entity.Color == "Red")
            {
                item.RejectPropertyChanges(nameof(entity.Color));
            }
        }
    }
}
```

Отделение модели от уровня доступа к данным

В текущий момент уровень доступа к данным целиком находится в одном проекте, со сгруппированным вместе кодом EF и моделями. Хотя во многих приложениях поступать так вполне допустимо, иногда (как вы заметите при работе с WPF MMVM в главе 28 и ASP.NET в главах 29 и 30) лучше отделять модели от кода EF.

К счастью, делается это чрезвычайно просто. Добавим в решение новый проект библиотеки классов по имени `AutoLotDAL.Models` и удалим сгенерированный файл класса `Class1.cs`. Добавим ссылку на `System.ComponentModel.DataAnnotations` и затем с помощью диспетчера пакетов NuGet добавим к проекту пакет `EntityFramework`. Переместим все файлы и папки из папки `Models` проекта `AutoLotDAL` в новый проект и скорректируем пространства имен. Добавим в проект `AutoLotDAL` ссылку на `AutoLotDAL.Models`, а в проект `AutoLotTestDrive` — ссылку на `AutoLotDAL.Models`.

Исходный код. Обновленный проект `AutoLotDAL` доступен в подкаталоге `Chapter_22`.

Развертывание в SQL Server Express

Версия `LocalDb` системы `SQL Server` великолепно подходит для процессов разработки и локального тестирования, но в какой-то момент возникнет необходимость перемещения базы данных в другой экземпляр. В рассматриваемом примере мы будем развертывать базу данных в системе `SQL Server Express`, которая удобна для процесса разработки, но может также применяться более чем одним пользователем и поддерживает локальные службы (такие как `IIS` и `WCF`). Существуют два простых способа развертывания, которые будут описаны далее.

На заметку! Если вы еще не установили систему `SQL Server Express 2016`, то можете загрузить программу установки по следующему адресу: <https://www.microsoft.com/en-us/sql-server/sql-server-editions-express>

Развертывание в SQL Server Express с использованием миграций

После того как база данных готова к развертыванию в другом экземпляре `SQL Server`, первый механизм (при наличии доступа к экземпляру, где будет производиться развертывание) сводится просто к изменению строки подключения и выполнению команды `update-database`. Откроем файл `App.config` из проекта `AutoLotDAL` и обновим строку подключения для указания на `SQL Server Express`. Точная строка подключения будет зависеть от того, каким образом установлена система `SQL Server Express`, но должна выглядеть похожей на показанную ниже:

```
<connectionStrings>
  <add name="AutoLotConnection"
    connectionString="data source=.\SQLEXPRESS2016;initial catalog=AutoLot;
integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

После запуска команды `update-database` обновления применяются.

В команде `update-database` допускается также указывать другую строку подключения. Вместо изменения строки подключения в файле `App.config` можно было бы ввести в окне `Package Manager Console` следующую команду:

```
update-database -ProjectName AutolotDAL
-ConnectionString "data source=.\SQLEXPRESS2016;initial catalog=AutoLot;
```

```
integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"
-ConnectionString "System.Data.SqlClient"
```

На заметку! В команде `update-database` доступно много дополнительных параметров. Чтобы получить полный их список (и сведения о любых других командах EF), нужно использовать команду `get-help update-database`.

Создание сценария миграции

Безусловно, изменять строку подключения легко, но что если доступ к базе данных отсутствует и все должно делаться через администратора баз данных? Инфраструктура EF располагает решением проблемы в такой ситуации. Команда `update-database` способна также создавать сценарий SQL, делающий все необходимые изменения. Добавление параметра `-script` заставляет исследовать целевую базу данных и создать сценарий для любых миграций, которые еще не были применены. Введем в окне Package Manager Console показанную далее команду (обратите внимание на изменение имени каталога, чтобы файл был создан):

```
update-database -ProjectName AutolotDAL
-ConnectionString "data source=.\SQLEXPRESS2016;initial catalog=AutoLot2;
integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"
-ConnectionString "System.Data.SqlClient" -script
```

В результате выполнения команды создается сценарий, который можно отправить администратору баз данных для выполнения. Недостаток такого подхода в том, что при создании сценария подобного рода метод `Seed()` класса `Configuration` не выполняется и не включается в сценарий SQL. Заполнять базу начальными данными придется каким-то другим способом.

Резюме

В последних двух главах был представлен обзор манипуляций данными с использованием ADO.NET, в частности подключенный уровень и Entity Framework. Ценность инфраструктуры Entity Framework (и других систем ORM) заключается в том, что они позволяют быстрее выполнять проекты, но по-прежнему жизненно важно понимать внутреннюю работу ADO.NET. Конечно, здесь был предложен только краткий экскурс в темы, которые можно найти в рамках набора технологий ADO.NET. Для получения исчерпывающих сведений по темам, затронутым в книге (а также связанным темам), рекомендуем обратиться в документацию .NET Framework 4.7 SDK.

В настоящей главе формальное исследование программирования для баз данных с применением ADO.NET завершилось рассмотрением роли инфраструктуры Entity Framework. Инфраструктура EF позволяет программировать с помощью концептуальной модели, которая точно отображается на предметную область. Несмотря на то что форму сущностей можно изменять как угодно, исполняющая среда EF гарантирует, что измененные данные модели отображаются на корректные данные физической таблицы.

В ходе изучения вы узнали об аннотациях данных, которые представляют собой один из способов описания отображения между моделью предметной области и моделью базы данных. Вы увидели, каким образом инфраструктура EF обрабатывает транзакции, научились создавать, сохранять и удалять данные, и ознакомились с состоянием сущностей.

Было показано, как с помощью миграций базы данных синхронизировать изменения, внесенные в модель, с базой данных, проверять наличие ошибок, связанных с параллелизмом, а также добавлять регистрацию в журнале и перехват.

ГЛАВА 23

Введение в Windows Communication Foundation

Инфраструктура Windows Communication Foundation (WCF) представляет собой API-интерфейс, спроектированный специально для построения распределенных систем. В отличие от других распределенных API-интерфейсов, которые вы могли использовать в прошлом (например, DCOM, .NET Remoting, веб-службы XML, очереди сообщений), WCF предлагает единую, унифицированную и расширяемую объектную модель программирования, которую можно применять для взаимодействия с множеством ранее разрозненных распределенных технологий.

Глава начинается с объяснения потребности в инфраструктуре WCF и исследования задач, для решения которых она предназначена. После рассмотрения служб, предлагаемых WCF, внимание будет переключено на изучение основных сборок .NET, пространств имен и типов, которые представляют эту модель программирования. В главе будет построено несколько служб, хостов и клиентов WCF с использованием разнообразных инструментов разработки WCF.

На заметку! В настоящей главе будет создаваться код, который требует запуска Visual Studio с административными привилегиями (к тому же вы и сами должны иметь административные привилегии). Чтобы запустить Visual Studio с правами администратора, необходимо щелкнуть правой кнопкой мыши на значке Visual Studio и выбрать в контекстном меню пункт **Запуск от имени администратора**.

Выбор API-интерфейса распределенных вычислений

Операционная система Windows всегда предоставляла множество API-интерфейсов для построения распределенных систем. Хотя и верно то, что под *распределенной системой* большинство понимает систему, включающую минимум два сетевых компьютера, в широком смысле этот термин может относиться к двум исполняемым сборкам, которые нуждаются в обмене данными, даже если они запущены на одной физической машине. При таком определении выбор распределенного API-интерфейса для решения текущей задачи программирования обычно предусматривает ответ на следующий основополагающий вопрос.

Будет ли система использоваться исключительно внутренне или же доступ к функциональности приложения потребуется и внешним пользователям?

В случае построения распределенной системы для внутреннего применения есть гораздо больше возможностей гарантировать, что на каждом подключенном компьютере установлена та же самая операционная система и используется та же самая инфраструктура программирования (например, платформа .NET, COM или Java). Внутренние системы также позволяют задействовать существующую систему безопасности для целей аутентификации, авторизации и т.д. В ситуации подобного рода можно выбрать конкретный распределенный API-интерфейс, который с точки зрения производительности согласуется с имеющейся операционной системой и инфраструктурой программирования.

В противоположность этому при построении системы, которая должна быть доступна внешним потребителям, возникает целый ряд других проблем, подлежащих решению. Во-первых, вам вряд ли удастся диктовать внешним пользователям то, какую они могут применять операционную систему (системы) и инфраструктуру (инфраструктуры) программирования, а также как они должны конфигурировать свои настройки безопасности.

Во-вторых, если вы работаете в крупной компании или в университетской среде, где используются многочисленные операционные системы и технологии программирования, то даже внутреннее приложение неожиданно столкнется с теми же проблемами, что и приложение, ориентированное на внешний мир. В любом из указанных случаев вам необходимо ограничиться наиболее гибким распределенным API-интерфейсом, чтобы обеспечить максимальную доступность разрабатываемого приложения.

В зависимости от ответа на приведенный выше основополагающий вопрос распределенных вычислений далее потребуется выбрать конкретный API-интерфейс (либо их набор). В этой главе рассматривается WCF. Другим популярным вариантом является ASP.NET Web API (глава 27).

На заметку! Инфраструктура WCF (и охватываемые ею технологии) не имеет ничего общего с построением веб-сайтов, основанных на HTML. Наряду с тем, что веб-приложения могут считаться распределенными, поскольку обычно в обмен вовлечены две машины, инфраструктура WCF ориентирована на установление подключений между машинами с целью совместного использования функциональности удаленных компонентов, а не для отображения HTML-разметки в веб-браузере. Построение веб-сайтов с помощью платформы .NET будет обсуждаться в главе 28.

Роль WCF

Обширный комплект распределенных технологий затрудняет выбор правильного инструмента для работы. Ситуация еще больше усложняется из-за того факта, что функциональность некоторых технологий перекрывается (наиболее заметно в областях транзакций и безопасности).

Даже когда разработчик .NET выбрал технологию, которая кажется корректной для решения текущей задачи, построение, сопровождение и конфигурирование такого приложения будут в лучшем случае сложными. Каждый API-интерфейс имеет собственную программную модель, собственный уникальный набор инструментов конфигурирования и т.д. До появления WCF это означало, что подключать распределенные API-интерфейсы было нелегко без написания существенного объема кода специальной инфраструктуры. Например, если вы строите систему с использованием API-интерфейсов .NET Remoting и позже решаете, что веб-службы XML будут более подходящим решением, то придется полностью перепроектировать имеющуюся кодовую базу.

Инфраструктура WCF является инструментальным набором для распределенных вычислений, который интегрирует все ранее независимые технологии распределенной обработки в один рационализированный API-интерфейс, представленный главным образом пространством имен `System.ServiceModel`. С применением WCF можно открывать доступ к службам для вызывающих компонентов, используя широкое разнообразие приемов. Например, при построении внутреннего приложения, где все подключенные машины основаны на Windows, можно использовать разнообразные протоколы TCP для достижения максимально возможной производительности. Открыть доступ к той же службе можно также с помощью протоколов HTTP и SOAP, чтобы позволить внешним вызывающим компонентам задействовать ее функциональность независимо от языка программирования или операционной системы.

Так как инфраструктура WCF позволяет выбрать корректный протокол для выполнения работы (с применением общей программной модели), вы обнаружите, что подключать лежащий в основе связующий код распределенного приложения становится действительно легко. В большинстве случаев это можно делать без повторной компиляции или развертывания программного обеспечения клиента/службы, потому что утомительные детали часто переносятся в конфигурационные файлы приложения.

Обзор функциональных возможностей WCF

Возможность взаимодействия и интеграция разрозненных API-интерфейсов — только два важных аспекта WCF. Инфраструктура WCF также предлагает развитую программную архитектуру, которая дополняет поддерживаемые ею технологии удаленной разработки. Ниже приведен список главных средств WCF.

- Поддержка строго типизированных, а также нетипизированных сообщений. Такой подход позволяет приложениям .NET эффективно совместно использовать типы, в то время как программное обеспечение, созданное с применением других платформ (таких как Java), может потреблять потоки слабо типизированных данных XML.
- Поддержка нескольких привязок (например, низкоуровневый HTTP, TCP, MSMQ, WebSockets, именованные каналы и т.д.) дает возможность выбирать наиболее подходящий механизм для транспортировки данных сообщений.
- Поддержка последних спецификаций веб-служб (WS-*).
- Полностью интегрированная модель безопасности, охватывающая как собственные протоколы безопасности Windows/.NET, так и многочисленные нейтральные технологии защиты, построенные на основе стандартов веб-служб.
- Поддержка технологий сеансового управления состоянием, а также поддержка односторонних сообщений без состояния.

Как бы впечатляюще не выглядел представленный список, он лишь поверхностно касается функциональности, предоставляемой WCF. Технология WCF также предлагает средства трассировки и протоколирования, счетчики производительности, модель публикации и подписки на события, поддержку транзакций и многое другое.

Обзор архитектуры, ориентированной на службы

Еще одно преимущество инфраструктуры WCF связано с тем, что она базируется на принципах проектирования, которые установлены *архитектурой, ориентированной на службы* (service-oriented architecture — SOA). Конечно, SOA является модным словечком в отрасли, и подобно многим модным словечкам SOA может определяться многочисленными путями. Попросту говоря, SOA представляет собой способ проектирования рас-

пределенной системы, где несколько автономных *служб* работают совместно, передавая сообщения через границы (либо сетевых машин, либо двух процессов на одной машине) с использованием четко определенных *интерфейсов*.

В мире WCF такие четко определенные интерфейсы обычно создаются с применением интерфейсных типов CLR (см. главу 9). Однако в более общем смысле интерфейс службы просто описывает набор членов, которые могут быть вызваны внешними компонентами.

Команда разработчиков WCF следовала четырем принципам проектирования SOA. Несмотря на то что эти принципы соблюдаются автоматически, просто за счет построения приложения WCF, усвоение таких четырех важнейших правил проектирования SOA может содействовать лучшему пониманию WCF. В последующих разделах приведен краткий обзор каждого принципа.

Принцип 1: границы устанавливаются явно

Данный принцип подчеркивает тот факт, что функциональность службы WCF выражается с использованием четко определенных интерфейсов (например, описаний всех членов, их параметров и возвращаемых значений). Единственный способ, которым внешний компонент может взаимодействовать со службой WCF — через интерфейс, оставаясь в полном неведении относительно деталей ее внутренней реализации.

Принцип 2: службы являются автономными

Термин *автономные сущности* относится к тому факту, что заданная служба WCF является (насколько возможно) отдельным “островком”. Автономная служба должна быть независимой в отношении версии, развертывания и установки. Чтобы содействовать в поддержке этого принципа, можно возвратиться к ключевому аспекту программирования на основе интерфейсов. После того как интерфейс передан в производственную среду, он никогда не должен изменяться (или возникнет риск нарушения работы существующих клиентов). Когда требуется расширить функциональность службы WCF, необходимо просто написать новые интерфейсы, моделирующие желаемую функциональность.

Принцип 3: службы взаимодействуют через контракт, а не реализацию

Третий принцип представляет собой еще один побочный продукт программирования на основе интерфейсов. Детали реализации службы WCF (на каком языке она написана, как выполняет свою работу и т.п.) не касаются вызывающего внешнего компонента. Клиенты WCF взаимодействуют со службами исключительно через их открытые интерфейсы.

Принцип 4: совместимость служб основана на политике

Поскольку интерфейсы CLR предоставляют строго типизированные контракты всем клиентам WCF (и также могут применяться для генерации связанного документа WSDL (Web Services Description Language — язык описания веб-служб) на основе выбранной привязки), важно уяснить, что интерфейсы и WSDL сами по себе недостаточно выразительны, чтобы детализировать аспекты того, что способна делать служба. С учетом сказанного SOA позволяет определять *политики*, которые дополнительно проясняют семантику службы (например, ожидаемые требования безопасности, используемые для взаимодействия со службой). С помощью таких политик можно по существу отделять низкоуровневое синтаксическое описание службы (открытые интерфейсы) от семантических деталей ее работы и способа обращения к ней.

Итоговые сведения об инфраструктуре WCF

Инфраструктура WCF является рекомендуемым API-интерфейсом, когда нужно разрабатывать внутреннее приложение с применением протоколов TCP или перемещать данные между программами на одной машине с использованием именованных каналов. Инфраструктура WCF также может применяться для открытия доступа к данным внешнему миру в целом, используя протоколы на основе HTTP, хотя ASP.NET Web API (см. главу 30) — также популярная платформа для служб REST.

Речь вовсе не идет об отсутствии возможности применять в новых разработках первоначальные пространства имен, связанные с распределенными вычислениями (например, `System.Runtime.Remoting`, `System.Messaging`, `System.EnterpriseServices` и `System.Web.Services`). В некоторых ситуациях (скажем, когда необходимо строить объекты COM+) использовать их придется. В любом случае, если вы задействовали исходные API-интерфейсы в прошлых проектах, то изучение WCF не представит особого труда. Подобно предшествующим технологиям в WCF интенсивно применяются конфигурационные XML-файлы, атрибуты .NET и утилиты генерации прокси.

Вооружившись начальными знаниями, теперь можно сосредоточить внимание на процессе построения приложений WCF. Вы должны понимать, что полное раскрытие инфраструктуры WCF потребовало бы целой книги, т.к. описание каждой поддерживаемой службы (например, MSMQ, COM+, P2P и именованных каналов) могло бы занять отдельную главу. Здесь вы изучите общий процесс построения программ WCF, использующих протоколы на основе TCP и HTTP (т.е. веб-службы), что должно подготовить почву для дальнейшего углубления знаний в данной области.

Исследование основных сборок WCF

Как и можно было ожидать, программная архитектура WCF представлена набором сборок .NET, установленных в GAC. В табл. 23.1 описаны основные сборки WCF, которые придется применять почти в любом приложении WCF.

Таблица 23.1. Основные сборки WCF

Сборка	Описание
<code>System.Runtime.Serialization.dll</code>	В этой основной сборке определены пространства имен и типы, используемые для сериализации и десериализации объектов в инфраструктуре WCF
<code>System.ServiceModel.dll</code>	Эта основная сборка содержит типы, применяемые для построения приложений WCF любого рода

В двух сборках, перечисленных в табл. 23.1, определено много пространств имен и типов. Детальные сведения о них доступны в документации .NET Framework 4.7 SDK, а в табл. 23.2 описаны некоторые важные пространства имен.

Шаблоны проектов WCF в Visual Studio

Как будет более подробно объясняться позже в главе, приложение WCF обычно представлено тремя взаимосвязанными сборками; одной из них является библиотека *.dll, содержащая типы, с которыми могут взаимодействовать внешние вызывающие компоненты (другими словами, сама служба WCF). Когда вы хотите построить службу WCF, совершенно допустимо выбирать в качестве отправной точки стандартный шаблон проекта Class Library (Библиотека классов), как демонстрировалось в главе 14, и вручную добавлять ссылки на сборки WCF.

Таблица 23.2. Основные пространства имен WCF

Пространство имен	Описание
System.Runtime.Serialization	В этом пространстве имен определены многочисленные типы, которые используются для управления сериализацией и десериализацией данных в WCF
System.ServiceModel	В этом первичном пространстве имен WCF определены типы привязок и хостинга, а также базовые типы безопасности и транзакций
System.ServiceModel.Configuration	В этом пространстве имен определены многочисленные типы, обеспечивающие программный доступ к конфигурационным файлам WCF
System.ServiceModel.Description	В этом пространстве имен определены типы, которые предоставляют объектную модель для адресов, привязок и контрактов, заданных внутри конфигурационных файлов WCF
System.ServiceModel.MsmqIntegration	В этом пространстве имен определены типы для интеграции со службой MSMQ
System.ServiceModel.Security	В этом пространстве имен определены многочисленные типы для управления аспектами уровней безопасности WCF

В качестве альтернативы создать новую службу WCF можно, выбрав в Visual Studio шаблон проекта WCF Service Library (Библиотека служб WCF), как показано на рис. 23.1. Этот тип проекта автоматически устанавливает ссылки на обязательные сборки WCF, а также генерирует приличный объем начального кода, который довольно часто просто удаляется.

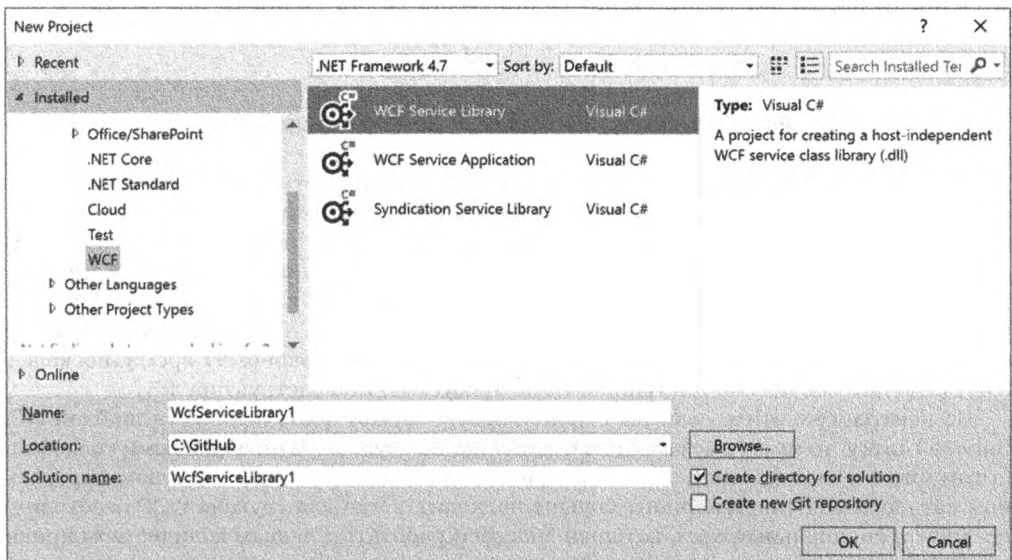


Рис. 23.1. Шаблон проекта WCF Service Library в Visual Studio

Одно из преимуществ выбора шаблона проекта WCF Service Library связано с тем, что он также снабжает вас файлом App.config, что поначалу может показаться странным, т.к. строится .NET-сборка *.dll, а не *.exe. Тем не менее, файл App.config полезен тем, что при отладке или запуске проекта IDE-среда Visual Studio автоматически запускает приложение WCF Test Client (Тестовый клиент WCF). Программа WcfTestClient.exe ожидает найти настройки в файле App.config и потому может применяться для тестирования службы. Более подробно данная программа рассматривается далее в главе.

На заметку! Файл App.config из проекта WCF Service Library также полезен тем, что показывает начальные настройки, используемые для конфигурирования хост-приложения WCF. В действительности большую часть его кода можно копировать и вставлять в конфигурационный файл для производственных служб.

Шаблон проекта WCF Service для веб-сайта

В Visual Studio доступен еще один шаблон проекта WCF Service (Служба WCF), находящийся в диалоговом окне New Web Site (Новый веб-сайт), которое открывается через пункт меню File⇒New⇒Web Site (Файл⇒Создать⇒Веб-сайт) и показано на рис. 23.2.

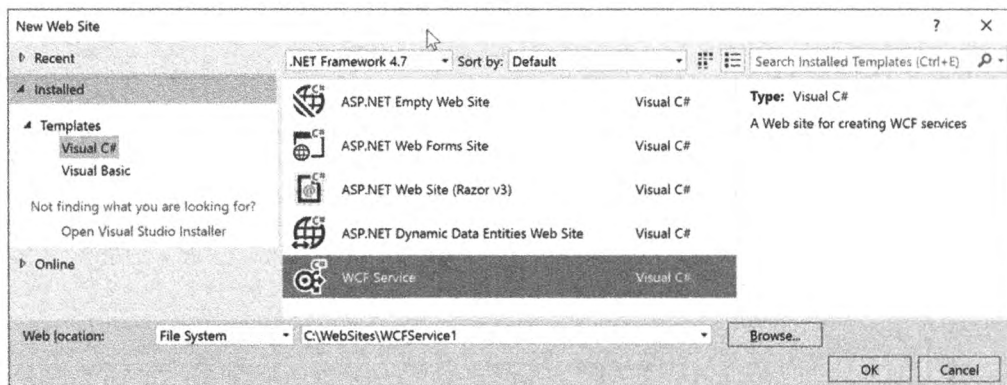


Рис. 23.2. Шаблон проекта WCF Service в Visual Studio

Шаблон проекта WCF Service удобен, когда заранее известно, что служба WCF будет применять протоколы на основе HTTP, а не, скажем, протокол TCP либо именованные каналы. Шаблон WCF Service может автоматически создать новый виртуальный каталог Internet Information Services (IIS) для хранения программных файлов WCF, сформировать подходящий файл Web.config для открытия доступа к службе по HTTP и сгенерировать необходимый файл *.svc (вы узнаете о файлах *.svc позже в главе). Таким образом, веб-ориентированный проект WCF Service просто экономит время, поскольку IDE-среда автоматически настраивает всю требуемую инфраструктуру IIS.

По контрасту с этим, если вы строите новую службу WCF, используя шаблон WCF Service Library, то имеете возможность размещения службы разнообразными способами (в том числе в специальном хосте, Windows-службе или вручную созданном виртуальном каталоге IIS). Такой вариант больше подходит, когда для службы WCF необходимо построить специальный хост, который способен работать с любым количеством привязок WCF.

Базовая структура приложения WCF

При построении распределенной системы WCF обычно создаются три взаимосвязанных сборки.

- *Сборка службы WCF.* Эта библиотека *.dll содержит классы и интерфейсы, представляющие общую функциональность, доступ к которой нужно открыть внешним вызывающим компонентам.
- *Хост службы WCF.* Этот программный модуль является сущностью, которая размещает в себе сборку службы WCF.
- *Клиент WCF.* Это приложение, которое получает доступ к функциональности службы через промежуточный прокси.

Как упоминалось ранее, сборка службы WCF представляет собой библиотеку классов .NET, которая содержит несколько контрактов WCF и их реализации. Ключевое отличие состоит в том, что интерфейсные контракты декорированы разнообразными атрибутами, которые управляют представлением типов данных, взаимодействием исполняющей среды WCF с открытыми типами и т.д.

Вторая сборка, хост службы WCF, может быть буквально любым исполняемым файлом .NET. Далее в главе вы увидите, что инфраструктура WCF настроена так, что доступ к службам можно легко открывать из приложения любого типа (например, приложения Windows Forms, Windows-службы, приложения WPF). При построении специального хоста применяется тип `ServiceHost` и возможно связанный с ним файл *.config. Файл *.config содержит детали, касающиеся механизма серверной стороны, который желательно использовать. Однако если в качестве хоста для службы WCF применяется IIS, тогда нет необходимости в программном построении специального хоста, т.к. IIS "за кулисами" будет использовать тип `ServiceHost`.

На заметку! Размещение службы WCF также возможно с применением службы активации Windows (Windows Activation Service — WAS); за подробными сведениями обращайтесь в документацию .NET Framework 4.7 SDK.

Финальная сборка представляет клиента, который выполняет обращения к службе WCF. Вполне ожидаемо таким клиентом может быть приложение .NET любого типа. Подобно хосту клиентское приложение обычно использует файл *.config клиентской стороны, в котором определен механизм на стороне клиента. Вы должны также знать, что если служба WCF построена с применением привязок, основанных на HTTP, то клиентское приложение может быть реализовано на другой платформе (скажем, Java).

На рис. 23.3 показаны высокоуровневые отношения между этими тремя взаимосвязанными сборками WCF. Для представления требуемого связующего механизма "за кулисами" используются многочисленные низкоуровневые детали (например, фабрики, каналы и прослушиватели). Такие низкоуровневые детали чаще всего скрыты от глаз; тем не менее, при необходимости они могут быть расширены или настроены. В большинстве случаев хорошо подходит стандартный механизм.

Также полезно отметить, что применять файл *.config серверной или клиентской стороны формально необязательно. При желании можно жестко закодировать хост (а также клиент), указав необходимый связующий механизм (т.е. конечные точки, привязку и адреса). Очевидная проблема такого подхода заключается в том, что если нужно изменить детали связующего механизма, тогда придется вносить изменения в код, перекомпилировать и заново развертывать множество сборок.

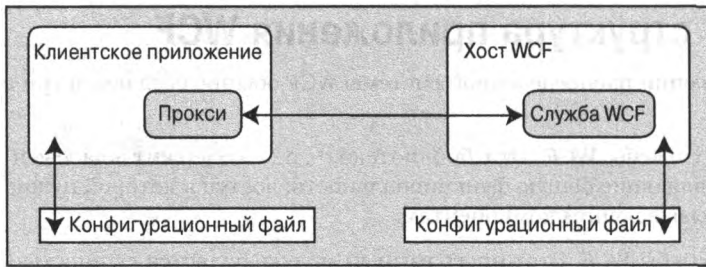


Рис. 23.3. Высокоуровневое представление типичного приложения WCF

Использование файла `*.config` делает кодовую базу намного более гибкой, поскольку изменение связующего механизма сводится к обновлению содержимого конфигурационного файла и перезапуску приложения. С другой стороны, программная конфигурация обеспечивает приложению более высокую динамическую гибкость — например, в зависимости от определенных условий связующий механизм можно конфигурировать по-разному.

Адрес, привязка и контракт в WCF

Хосты и клиенты взаимодействуют друг с другом путем согласования *адресов, привязок и контрактов*, которые являются основными строительными блоками приложения WCF и обозначаются посредством аббревиатуры ABC (address, binding, contract — адрес, привязка, контракт).

- **Адрес.** Описывает местоположение службы. В коде он представлен с помощью типа `System.Uri`, однако обычно значение адреса хранится в файлах `*.config`.
- **Привязка.** Инфраструктура WCF поставляется со многими разными привязками, которые указывают сетевые протоколы, механизмы кодирования и транспортный уровень.
- **Контракт.** Предоставляет описание каждого метода в службе WCF, к которому открыт доступ.

Вы должны понимать, что аббревиатура ABC вовсе не подразумевает, что разработчик обязан определить сначала адрес, за ним привязку и в конце контракт. Во многих случаях разработчик WCF начинает с определения контракта для службы, после чего устанавливает адрес и привязки (допустим любой порядок при условии, что учтены все аспекты). Прежде чем перейти к построению первого приложения WCF, давайте более детально рассмотрим ABC.

Понятие контрактов WCF

При построении службы WCF понятие *контракта* считается ключевым. Несмотря на то что это не является обязательным, подавляющее большинство приложений WCF начинается с определения набора интерфейсных типов .NET, применяемых для представления набора членов, которые будет поддерживать заданная служба WCF. В частности, интерфейсы, представляющие контракт WCF, называются *контрактами служб*. Классы (или структуры), которые реализуют их, носят название *типов служб*.

Контракты служб WCF декорируются разнообразными атрибутами, наиболее распространенные из которых определены в пространстве имен `System.ServiceModel`. Когда члены контракта службы (методы в интерфейсе) содержат только простые типы данных (такие как числовые, булевские и строковые), завершающую службу

WCF можно построить с использованием только атрибутов [ServiceContract] и [OperationContract].

Тем не менее, если члены открывают доступ к специальным типам, то вероятно будут применяться различные типы из пространства имен System.Runtime.Serialization (рис. 23.4), находящегося в сборке System.Runtime.Serialization.dll. Здесь вы обнаружите дополнительные атрибуты (например, [DataMember] и [DataContract]), которые предназначены для тонкой настройки процесса определения того, как составные типы будут сериализоваться и десериализоваться из XML при передаче в и из операций службы.

Строго говоря, вы не обязаны использовать интерфейсы CLR для определения контракта WCF. Многие из этих атрибутов могут применяться к открытым членам открытого класса (или структуры). Однако, учитывая многочисленные преимущества программирования на основе интерфейсов (например, полиморфизм и элегантная поддержка множества версий), использование интерфейсов CLR для описания контракта WCF имеет смысл рассматривать как рекомендуемый прием.

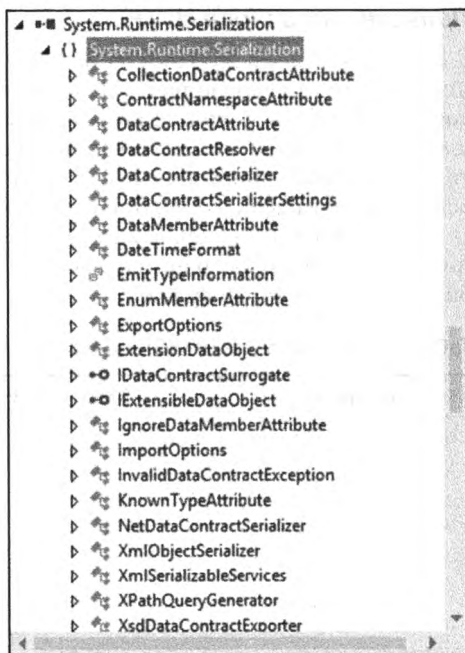


Рис. 23.4. В пространстве имен System.Runtime.Serialization определены атрибуты, используемые при построении контрактов данных WCF

Понятие привязок WCF

После определения и реализации контракта (или набора контрактов) в библиотеке службы следующий логический шаг предусматривает построение агента хостинга для самой службы WCF. Как упоминалось ранее, на выбор доступно множество возможных хостов, и все они должны указывать привязки, применяемые удаленными клиентами для получения доступа к функциональности типа службы.

Инфраструктура WCF поставляется со многими вариантами привязок, каждая из которых ориентирована на определенные нужды. Если ни одна из готовых привязок не удовлетворяет существующим потребностям, тогда можно создать собственную привязку, расширив тип CustomBinding (данная тема в главе не рассматривается). Привязка WCF может описывать следующие характеристики:

- транспортный уровень, используемый для перемещения данных (HTTP, MSMQ, именованные каналы, REST, WebSockets и TCP);
- каналы, применяемые транспортом (однаправленные, запрос-ответ и дуплексные);
- механизм кодирования, используемый для работы с самими данными, такой как XML или двоичный;
- любые поддерживаемые протоколы веб-служб (если разрешены привязкой), такие как WS-Security, WS-Transactions, WS-Reliability и т.д.

Давайте рассмотрим возможные варианты.

Привязки на основе HTTP

Классы `BasicHttpBinding`, `WSHttpBinding`, `WSDualHttpBinding` и `WSFederationHttpBinding` предназначены для открытия доступа к типам контрактов через протоколы HTTP/SOAP. Если для разрабатываемой службы требуется более широкий доступ (например, из множества операционных систем и множества программных архитектур), то на указанные привязки следует обратить внимание, т.к. все они кодируют данные на основе представления XML и применяют в сети протокол HTTP.

В табл. 23.3 показано, как можно представлять привязки WCF в коде (с использованием классов из пространства имен `System.ServiceModel`) или в виде атрибутов XML, определенных внутри файлов `*.config`.

Таблица 23.3. Привязки WCF на основе HTTP

Класс привязки	Элемент привязки	Описание
<code>BasicHttpBinding</code>	<code><basicHttpBinding></code>	Применяется для построения службы WCF, совместимой с профилем WS-Basic Profile (WS-I Basic Profile 1.1). Эта привязка использует HTTP в качестве транспорта и Text/XML в качестве стандартного метода кодирования сообщений
<code>WSHttpBinding</code>	<code><wsHttpBinding></code>	Подобен классу <code>BasicHttpBinding</code> , но предоставляет больше функциональных возможностей веб-службы. Эта привязка добавляет поддержку транзакций, надежной доставки сообщений и спецификации WS-Addressing
<code>WSDualHttpBinding</code>	<code><wsDualHttpBinding></code>	Подобен классу <code>WSHttpBinding</code> , но предназначен для применения с дуплексными контрактами (например, когда служба и клиент могут отправлять сообщения туда и обратно). Эта привязка поддерживает только безопасность SOAP и требует надежного обмена сообщениями
<code>WSFederationHttpBinding</code>	<code><wsFederationHttpBinding></code>	Безопасная привязка с возможностью взаимодействия, которая поддерживает протокол WS-Federation, позволяя объединенным в федерацию организациям эффективным образом выполнять аутентификацию и авторизацию пользователей

Как и можно было догадаться, класс `BasicHttpBinding` реализует простейший из всех протоколов, ориентированных на веб-службы. В частности, данная привязка гарантирует, что служба WCF соответствует спецификации под названием WS-I Basic Profile 1.1 (определенной WS-I). Главная причина использования такой привязки состоит в поддержке об-

ратной совместимости с приложениями, ранее построенными для взаимодействия с веб-службами ASP.NET (которые были частью библиотек .NET, начиная с версии 1.0).

Протокол `WSHttpBinding` не только включает поддержку подмножества спецификации WS-* (транзакции, безопасность и надежные сеансы), но также предоставляет возможность обработки двоичного кодирования данных с применением механизма оптимизации передачи сообщений (`Message Transmission Optimization Mechanism` — MTOM).

Основное преимущество привязки `WSDualHttpBinding` в том, что она добавляет возможность *дуплексного обмена сообщениями* между отправителем и получателем, которая представляет собой причудливый способ заявления о способности двустороннего взаимодействия. При выборе `WSDualHttpBinding` можно задействовать модель публикации/подписки на события WCF.

Наконец, `WSFederationHttpBinding` — это протокол на основе веб-служб, об использовании которого стоит подумать, когда крайне важна безопасность в рамках группы организаций. Данная привязка поддерживает спецификации WS-Trust, WS-Security и WS-SecureConversation, которые представлены API-интерфейсами CardSpace в WCF.

Привязки на основе TCP

При построении распределенного приложения, которое функционирует на машинах с установленными библиотеками .NET 4.7 (другими словами, все машины работают под управлением Windows), можно получить выигрыш в производительности за счет обхода привязок веб-служб и применения привязки TCP, обеспечивающей кодирование данных в компактном двоичном формате вместо XML. Когда используются привязки, описанные в табл. 23.4, клиент и хост должны быть приложениями .NET.

Таблица 23.4. Привязки WCF на основе TCP

Класс привязки	Элемент привязки	Описание
<code>NetNamedPipeBinding</code>	<code><netNamedPipeBinding></code>	Безопасная, надежная и оптимизированная привязка для коммуникаций между приложениями .NET на одной машине
<code>NetPeerTcpBinding</code>	<code><netPeerTcpBinding></code>	Безопасная привязка для сетевых приложений P2P
<code>NetTcpBinding</code>	<code><netTcpBinding></code>	Безопасная и оптимизированная привязка, подходящая для межмашинных коммуникаций между приложениями .NET

Для перемещения двоичных данных между клиентом и службой WCF класс `NetTcpBinding` применяет протокол TCP. Как упоминалось ранее, в результате обеспечивается более высокая производительность по сравнению с протоколами веб-служб, но все ограничивается внутренними решениями Windows. Положительной стороной является поддержка классом `NetTcpBinding` транзакций, надежных сеансов и безопасных коммуникаций.

Подобно `NetTcpBinding` класс `NetNamedPipeBinding` поддерживает транзакции, надежные сеансы и безопасные коммуникации; тем не менее, он не обладает возможностью делать межмашинные вызовы. Если вы ищете самый быстрый способ передачи данных между приложениями WCF на одной машине (например, для реализации взаимодействия между приложениями в домене), то привязка `NetNamedPipeBinding` не будет иметь себе равных. Более подробные сведения о классе `NetPeerTcpBinding` можно найти в разделе документации .NET Framework 4.7 SDK, посвященном сетям P2P.

Привязки на основе MSMQ

Наконец, если цель заключается в интеграции с сервером MSMQ, то непосредственный интерес представляют привязки `NetMsmqBinding` и `MsmqIntegrationBinding`. Детали использования привязок MSMQ в настоящей главе не рассматриваются, но в табл. 23.5 описано основное назначение каждой из них.

Таблица 23.5. Привязки WCF на основе MSMQ

Класс привязки	Элемент привязки	Описание
<code>MsmqIntegrationBinding</code>	<code><msmqIntegrationBinding></code>	Эту привязку можно применять для того, чтобы позволить приложениям WCF отправлять и получать сообщения от существующих приложений MSMQ, которые используют COM, собственный C++ или типы, определенные в пространстве имен <code>System.Messaging</code>
<code>NetMsmqBinding</code>	<code><netMsmqBinding></code>	Эту привязку на основе очередей можно применять для межмашинных коммуникаций между приложениями .NET. В рамках привязок, основанных на MSMQ, такой подход является предпочтительным

Понятие адресов WCF

После установления контрактов и привязок остается указать *адрес* для службы WCF. Это важно, поскольку удаленные вызывающие компоненты не смогут взаимодействовать с удаленными типами, если не удастся найти их местоположение. Подобно большинству аспектов WCF адрес может быть жестко закодирован в сборке (с использованием типа `System.Uri`) или вынесен в файл `*.config`.

В любом случае точный формат адреса WCF будет отличаться в зависимости от выбранной привязки (на основе HTTP, именованные каналы, на основе TCP или на основе MSMQ). На самом высоком уровне адреса WCF могут задавать перечисленные ниже единицы информации.

- **Scheme.** Транспортный протокол (например, HTTP).
- **MachineName.** Полностью заданное доменное имя машины.
- **Port.** Номер порта, который во многих ситуациях является необязательным параметром; скажем, привязка HTTP по умолчанию работает с портом 80.
- **Path.** Путь к службе WCF.

Указанная информация может быть представлена с помощью следующего обобщенного шаблона (значение `Port` необязательно, т.к. в некоторых привязках порт не применяется):

```
Scheme://<MachineName>[:Port]/Path
```

Когда используется привязка на основе HTTP (`basicHttpBinding`, `wsHttpBinding`, `wsDualHttpBinding` или `wsFederationHttpBinding`), адрес разбивается примерно так (вспомните, что если номер порта не указан, то протоколы на основе HTTP по умолчанию выбирают порт 80):

```
http://localhost:8080/MyWCFService
```

Если применяется привязка, основанная на TCP (вроде `NetTcpBinding` или `NetPeerTcpBinding`), то URI принимает следующий формат:

```
net.tcp://localhost:8080/MyWCFSvcService
```

Привязки на основе MSMQ (`NetMsmqBinding` и `MsmqIntegrationBinding`) уникальны в своем формате URI, потому что MSMQ может использовать открытые или закрытые очереди (доступные только на локальной машине), а номера портов не имеют смысла в URI, связанных с MSMQ. Взгляните на приведенный ниже URI, который описывает закрытую очередь по имени `MyPrivateQ`:

```
net.msmq://localhost/private$/MyPrivateQ
```

И последнее, но не менее важное замечание: формат адреса, применяемый привязкой для именованных каналов (`NetNamedPipeBinding`), выглядит так, как показано ниже (вспомните, что именованные каналы делают возможными межпроцессные коммуникации приложений на одной физической машине):

```
net.pipe://localhost/MyWCFSvcService
```

Хотя одиночная служба WCF может открывать доступ только к одному адресу (основанному на единственной привязке), есть возможность сконфигурировать коллекцию уникальных адресов (с разными привязками). Это делается в файле `*.config` за счет определения множества элементов `<endpoint>`. Для одной и той же службы можно указывать любое количество ABC. Такой подход полезен, когда необходимо позволить вызывающим компонентам выбирать протокол, который они желают использовать для взаимодействия со службой.

Построение службы WCF

Теперь, когда вы получили представление о строительных блоках приложения WCF, наступило время создать первый пример приложения, чтобы посмотреть, как ABC учитываются в коде и конфигурации. В первом примере шаблоны проектов WCF из Visual Studio не применяются, так что можно будет сконцентрироваться на специфических шагах по созданию службы WCF.

Первым делом создадим новый проект библиотеки классов C# по имени `MagicEightBallServiceLib` и назначим решению имя `MagicEightBallServiceHTTP`. Переименуем начальный файл `Class1.cs` в `MagicEightBallService.cs` и добавим ссылку на сборку `System.ServiceModel.dll`. Добавим в начальный файл кода оператор `using` для пространства имен `System.ServiceModel`. Код в файле C# должен выглядеть следующим образом (обратите внимание, что в данный момент мы имеем открытый класс):

```
// Основное пространство имен WCF.
using System.ServiceModel;

namespace MagicEightBallServiceLib
{
    public class MagicEightBallService
    {
    }
}
```

В классе реализован единственный контракт службы WCF, представленный строго типизированным интерфейсом CLR по имени `IEightBall`. Как вам наверняка известно, магический шар Magic 8-Ball — это игрушка, позволяющая получить ответ на задаваемый вопрос из набора фиксированных ответов. В интерфейсе будет определен единс-

твенный метод, который позволит клиенту задать вопрос магическому шару, чтобы получить случайный ответ.

Интерфейсы служб WCF оснащены атрибутом [ServiceContract], в то время как каждый член интерфейса декорирован атрибутом [OperationContract] (вскоре вы получите больше сведений о двух упомянутых атрибутах). Создадим новый файл по имени IEightBall.cs и поместим в него следующий код:

```
[ServiceContract]
public interface IEightBall
{
    // Задайте вопрос, получите ответ!
    [OperationContract]
    string ObtainAnswerToQuestion(string userQuestion);
}
```

На заметку! Допускается определять интерфейс контракта службы, который содержит методы, не оснащенные атрибутом [OperationContract]; однако к таким членам не будет открываться доступ через исполняющую среду WCF.

Как известно из главы 8, интерфейс — довольно бесполезная вещь, пока он не реализован классом или структурой, которая наполняет его функциональностью. Подобно реальному магическому шару реализация типа службы (MagicEightBallService) будет возвращать случайно выбранный заготовленный ответ из массива строк. Стандартный конструктор будет отображать информационное сообщение, которое (в конечном итоге) выводится в окне консоли хоста (в целях диагностики).

```
public class MagicEightBallService : IEightBall
{
    // Просто для отображения на хосте.
    public MagicEightBallService()
    {
        Console.WriteLine("The 8-Ball awaits your question...");
    }

    public string ObtainAnswerToQuestion(string userQuestion)
    {
        string[] answers = { "Future Uncertain", "Yes", "No",
            "Hazy", "Ask again later", "Definitely" };

        // Возвратить случайный ответ.
        Random r = new Random();
        return answers[r.Next(answers.Length)];
    }
}
```

Итак, библиотека службы WCF завершена. Тем не менее, перед конструированием хоста для этой службы необходимо ознакомимся с дополнительными деталями, касающимися атрибутов [ServiceContract] и [OperationContract].

Атрибут [ServiceContract]

Чтобы интерфейс CLR принимал участие в службах, предоставляемых WCF, он должен быть декорирован атрибутом [ServiceContract]. Подобно многим другим атрибутам .NET тип ServiceContractAttribute поддерживает много свойств, с помощью которых производится дальнейшее уточнение его целевого назначения. Два свойства, Name и Namespace, могут быть установлены для управления именем типа службы и названием пространства имен XML, где определен тип службы. Если используется при-

языка HTTP, то данные значения применяются для определения элементов <portType> связанного документа WSDL.

Здесь мы не заботимся о присваивании значения свойству Name, т.к. стандартное имя типа службы основано непосредственно на имени класса C#. Однако стандартным названием для лежащего в основе пространства имен XML будет просто `http://tempuri.org` (оно должно быть изменено для всех создаваемых служб WCF).

При построении службы WCF, которая будет отправлять и получать данные специальных типов (чего мы в настоящий момент не делаем), важно установить осмысленное значение для лежащего в основе пространства имен XML, поскольку это обеспечивает уникальность специальных типов. Как вам может быть известно из опыта построения веб-служб XML, пространства имен XML предоставляют способ помещения специальных типов в уникальный контейнер, гарантируя отсутствие конфликтов с типами из другой организации.

По указанной причине определение интерфейса можно обновить, задав в качестве пространства имен URI место происхождения службы, что очень похоже на процесс определения пространства имен XML в проекте веб-службы .NET:

```
[ServiceContract(Namespace = "http://MyCompany.com")]
public interface IEightBall
{
    ...
}
```

Помимо Namespace и Name атрибут [ServiceContract] может быть сконфигурирован посредством дополнительных свойств, которые кратко описаны в табл. 23.6. Имейте в виду, что в зависимости от выбранной привязки некоторые из перечисленных настроек будут игнорироваться.

Таблица 23.6. Разнообразные именованные свойства атрибута [ServiceContract]

Свойство	Описание
CallbackContract	Устанавливает, требует ли этот контракт службы функциональность обратного вызова для двустороннего обмена сообщениями (например, дуплексные привязки)
ConfigurationName	Определяет местоположение элемента службы в конфигурационном файле приложения. По умолчанию представляет собой имя класса, реализующего службу
ProtectionLevel	Позволяет указать степень, до которой привязка контракта требует шифрования, цифровых подписей или того и другого для конечных точек, открываемых контрактом
SessionMode	Устанавливает, разрешены ли сеансы, не разрешены или являются обязательными для этого контракта службы

Атрибут [OperationContract]

Методы, которые планируется использовать внутри инфраструктуры WCF, должны быть декорированы атрибутом [OperationContract], который также может быть сконфигурирован с помощью различных именованных свойств. Свойства, описанные в табл. 23.7, можно применять для объявления о том, что конкретный метод предназначен для однонаправленной работы, поддерживает асинхронные вызовы, требует шифрования данных сообщений и т.д. (в зависимости от выбранной привязки многие из этих значений могут быть проигнорированы).

Таблица 23.7. Различные именованные свойства атрибута [OperationContract]

Свойство	Описание
AsyncPattern	Указывает, реализована ли операция асинхронно с использованием пары методов Begin/End службы. Это позволяет службе передавать обработку другому потоку серверной стороны, но не имеет ничего общего с асинхронным вызовом метода клиентом!
IsInitiating	Указывает, может ли эта операция быть начальной операцией в сеансе
IsOneWay	Указывает, состоит ли операция из только одного входного сообщения (и не имеет ассоциированного вывода)
IsTerminating	Указывает, должна ли исполняющая среда WCF пытаться завершить текущий сеанс после выполнения операции

В начальном примере дополнительное конфигурирование метода `ObtainAnswerToQuestion()` не требуется, т.е. атрибут `[OperationContract]` можно оставить в его текущем виде.

Типы служб как контракты операций

Вспомните, что при построении типов служб WCF использовать интерфейсы не обязательно. На самом деле атрибуты `[ServiceContract]` и `[OperationContract]` можно применять прямо к самому типу службы:

```
// Только в целях иллюстрации;
// в текущем примере не используется.
[ServiceContract(Namespace = "http://MyCompany.com")]
public class ServiceTypeAsContract
{
    [OperationContract]
    void SomeMethod() { }

    [OperationContract]
    void AnotherMethod() { }
}
```

Вы можете принять такой подход; тем не менее, явное определение интерфейсного типа для представления контракта службы обеспечивает массу преимуществ. Самый очевидный выигрыш заключается в том, что отдельно взятый интерфейс можно применять к нескольким типам служб (написанных с использованием разных языков и архитектур) и достичь высокой степени полиморфизма. Еще одно преимущество связано с тем, что интерфейс контракта службы может выступать в качестве основы для новых контрактов (с применением наследования интерфейсов) без необходимости заботиться о реализации.

В любом случае разработка первой библиотеки службы WCF завершена. Скомпилируем проект, чтобы удостовериться в отсутствии опечаток.

Хостинг службы WCF

Теперь все готово для определения хоста. Хотя служба производственного уровня должна размещаться в Windows-службе или в виртуальном каталоге IIS, наш первый хост будет просто консольным приложением по имени `MagicEightBallServiceHost`.

Добавим в решение новый проект консольного приложения по имени `MagicEightBallServiceHost` и установим его в качестве запускаемого.

Добавим ссылку на сборку `System.ServiceModel.dll` и проект `MagicEightBallServiceLib`, после чего обновим начальный файл кода, импортировав пространства имен `System.ServiceModel` и `MagicEightBallServiceLib`:

```
using System;
...
using System.ServiceModel;
using MagicEightBallServiceLib;
namespace MagicEightBallServiceHost
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Console Based WCF Host *****");
            Console.ReadLine();
        }
    }
}
```

Первый шаг, который должен быть предпринят при построении хоста для служебного типа службы WCF, касается решения относительно того, определяется ли необходимая логика хостинга полностью в коде или же несколько низкоуровневых деталей будут вынесены в конфигурационный файл приложения. Как упоминалось ранее, преимущество файлов `*.config` заключается в том, что хост может изменять связующий механизм без перекомпиляции и повторного развертывания исполняемого файла. Однако всегда помните о том, что это совершенно не обязательно, поскольку логику хостинга можно жестко закодировать с использованием типов из сборки `System.ServiceModel.dll`.

В создаваемом консольном хосте будет применяться конфигурационный файл приложения, поэтому добавим в текущий проект новый такой файл (если его еще нет), выбрав пункт меню `Project⇒Add New Item` (Проект⇒Добавить новый элемент) и указав элемент `Application Configuration File` (Конфигурационный файл приложения).

Установка ABC внутри файла `App.config`

При построении хоста для типа службы WCF необходимо следовать предсказуемому набору шагов, часть из которых опирается на конфигурацию, а часть — на код.

- Определить конечную точку для службы WCF в конфигурационном файле хоста.
- Использовать в коде тип `ServiceHost` для открытия доступа к типам служб, видимых из этой конечной точки.
- Обеспечить, чтобы хост остался функционирующим для обслуживания входящих клиентских запросов. Очевидно, что данный шаг не является обязательным, если для хостинга применяется `Windows-служба` или `IIS`.

В мире WCF термин *конечная точка* представляет адрес, привязку и контракт, которые вместе объединены в аккуратный пакет. В XML конечная точка выражается с использованием элемента `<endpoint>` и его атрибутов `address`, `binding` и `contract`. Модифицируем файл `*.config`, указав в нем единственную конечную точку (достижимую через порт 8080), доступ к которой открывает данный хост:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
```



```

    <service name = "MagicEightBallServiceLib.MagicEightBallService">
      <endpoint address = "http://localhost:8080/MagicEightBallService"
        binding = "basicHttpBinding"
        contract = "MagicEightBallServiceLib.IEightBall"/>
    </service>
  </services>
</system.serviceModel>
<startup>
  <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7" />
</startup>
</configuration>

```

Обратите внимание, что элемент `<system.serviceModel>` является корнем всех настроек WCF хоста. Каждая служба, открываемая хостом, представлена элементом `<service>`, который находится внутри базового элемента `<services>`. Здесь в единственном элементе `<service>` для указания дружественного имени типа службы применяется (необязательный) атрибут `name`.

С помощью вложенного элемента `<endpoint>` устанавливается адрес, модель привязки (`basicHttpBinding` в рассматриваемом примере) и полностью заданное имя интерфейсного типа, определяющего контракт службы WCF (`IEightBall`). Поскольку используется привязка на основе HTTP, применяется схема `http://` с указанием произвольного номера порта.

Кодирование с использованием типа `ServiceHost`

Благодаря текущему конфигурационному файлу действительная программная логика, требуемая для завершения хоста, будет простой. Когда исполняемая программа запускается, создается экземпляр типа `ServiceHost`, которому сообщается служба WCF, отвечающая за хостинг. Во время выполнения этот объект автоматически читает данные из элемента `<system.serviceModel>` в файле `*.config` хоста для определения корректного адреса, привязки и контракта. Затем объект создает необходимый связующий механизм:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Console Based WCF Host *****");
    using (ServiceHost serviceHost =
        new ServiceHost(typeof(MagicEightBallService)))
    {
        // Открыть хост и начать прослушивание входящих сообщений.
        serviceHost.Open();

        // Оставить службу функционирующей до тех пор,
        // пока не будет нажата клавиша <Enter>.
        Console.WriteLine("The service is ready.");
        Console.WriteLine("Press the Enter key to terminate service.");
        Console.ReadLine();
    }
}

```

После запуска приложения обнаружится, что хост расположился в памяти и готов к приему входящих запросов от удаленных клиентов.

На заметку! Вспомните, что для выполнения многих типов проектов WCF среда Visual Studio должна быть запущена с административными привилегиями!

Указание базовых адресов

В настоящий момент объект `ServiceHost` создается с применением конструктора, который требует только информацию о типе службы. Тем не менее, в качестве аргумента конструктору можно также передавать массив элементов типа `System.Uri`, чтобы представить коллекцию адресов, по которым доступна данная служба. В текущее время адрес находится с использованием файла `*.config`. Однако предположим, что область `using` модифицирована следующим образом:

```
using (ServiceHost serviceHost = new
    ServiceHost(typeof(MagicEightBallService),
        new Uri[]{new Uri("http://localhost:8080/MagicEightBallService")}))
{
    ...
}
```

Теперь конечную точку можно определить так:

```
<endpoint address = ""
    binding = "basicHttpBinding"
    contract = "MagicEightBallServiceLib.IEightBall"/>
```

Разумеется, слишком большой объем жесткого кодирования внутри кодовой базы хоста снижает гибкость, поэтому в текущем примере предполагается, что хост службы создается просто путем предоставления информации о типе, как делалось раньше:

```
using (ServiceHost serviceHost = new ServiceHost(typeof(MagicEightBallService)))
{
    ...
}
```

Слегка обескураживающий аспект написания файлов `*.config` для хостов связан с тем, что в зависимости от объема жесткого кодирования существует несколько способов создания дескрипторов XML (как было в случае с необязательным массивом `Uri`). В показанной ниже модификации демонстрируется еще один способ написания файла `*.config`:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService">
        <!-- Адрес получен из <baseAddresses> -->
        <endpoint address = ""
            binding = "basicHttpBinding"
            contract = "MagicEightBallServiceLib.IEightBall"/>
        <!-- Перечислить все базовые адреса в выделенном разделе -->
      </service>
    </services>
  </system.serviceModel>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7" />
  </startup>
</configuration>
```

В данном случае атрибут `address` элемента `<endpoint>` по-прежнему пуст; невзирая на то, что при создании `ServiceHost` массив `Uri` в коде не указывается, приложение функционирует, как и ранее, т.к. нужное значение извлекается из элемента `baseAddresses`. Преимущество хранения базового адреса в разделе `<baseAddresses>` элемента `<host>` связано с тем, что другим частям файла `*.config` также необходимо знать адрес конечной точки службы. Таким образом, вместо копирования и вставки значений адресов внутрь файла `*.config` можно изолировать единственное значение, как было показано в предшествующем фрагменте.

На заметку! В примере, приведенном позже в главе, будет представлен графический инструмент конфигурирования, который позволяет создавать конфигурационные файлы менее утомительным способом.

В любом случае перед построением клиентского приложения, предназначенного для взаимодействия со службой, придется проделать чуть больше работы. В частности, необходимо более глубоко изучить роль класса `ServiceHost` и элемента `<service.serviceModel>`, а также роль служб обмена метаданными (`metadata exchange` — MEX).

Подробный анализ типа `ServiceHost`

Класс `ServiceHost` применяется для конфигурирования и открытия доступа к службе WCF из размещающей исполняемой сборки. Тем не менее, имейте в виду, что вы будете использовать данный тип напрямую только при построении специальной сборки `*.exe` для размещения ваших служб. Если для открытия доступа к службе применяется IIS, то объект `ServiceHost` создается автоматически.

Как уже было показано, тип `ServiceHost` требует полного описания службы, которое получается динамически через конфигурационные настройки файла `*.config` хоста. Хотя это происходит автоматически при создании объекта, состояние объекта `ServiceHost` можно сконфигурировать вручную с помощью его членов. Помимо методов `Open()` и `Close()` (которые взаимодействуют со службой в синхронной манере) класс `ServiceHost` имеет и другие члены, перечисленные в табл. 23.8.

Таблица 23.8. Избранные члены класса `ServiceHost`

Член	Описание
<code>Authorization</code>	Это свойство получает уровень авторизации для размещаемой службы
<code>AddDefaultEndpoints()</code>	Этот метод используется для программного конфигурирования хоста службы WCF, чтобы он воспринимал любое количество готовых конечных точек, предоставленных инфраструктурой
<code>AddServiceEndpoint()</code>	Этот метод позволяет программно регистрировать конечную точку для хоста
<code>BaseAddresses</code>	Это свойство получает список зарегистрированных базовых адресов для текущей службы
<code>BeginOpen()</code> <code>BeginClose()</code>	Эти методы позволяют асинхронно открывать и закрывать объект <code>ServiceHost</code> с применением стандартного асинхронного синтаксиса делегатов .NET
<code>CloseTimeout</code>	Это свойство позволяет устанавливать и получать время, отведенное службе на закрытие
<code>Credentials</code>	Это свойство получает удостоверения безопасности, используемые текущей службой

Член	Описание
EndOpen() EndClose()	Эти методы являются асинхронными аналогами методов BeginOpen() и BeginClose()
OpenTimeout	Это свойство позволяет устанавливать и получать время, отведенное службе на запуск
State	Это свойство получает значение, которое указывает текущее состояние коммуникационного объекта, представленное перечислением CommunicationState (например, Opened (открыт), Closed (закрыт), Created (создан))

Чтобы продемонстрировать некоторые дополнительные аспекты ServiceHost в действии, модифицируем класс Program, добавив новый статический метод, который выводит на консоль ABC каждой конечной точки, применяемой хостом:

```
static void DisplayHostInfo(ServiceHost host)
{
    Console.WriteLine();
    Console.WriteLine("***** Host Info *****");

    foreach (System.ServiceModel.Description.ServiceEndpoint se
        in host.Description.Endpoints)
    {
        Console.WriteLine("Address: {0}", se.Address);           // Адрес
        Console.WriteLine("Binding: {0}", se.Binding.Name);     // Привязка
        Console.WriteLine("Contract: {0}", se.Contract.Name);   // Контракт
        Console.WriteLine();
    }
    Console.WriteLine («*****»);
}
```

Предположим, что новый метод вызывается внутри Main() после открытия хоста:

```
using (ServiceHost serviceHost = new ServiceHost(typeof(MagicEightBallService)))
{
    // Открыть хост и начать прослушивание входящих сообщений.
    serviceHost.Open();
    DisplayHostInfo(serviceHost);
    ...
}
```

В результате выводится следующая статистика:

```
***** Console Based WCF Host *****

***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall
*****
The service is ready.
Press the Enter key to terminate service.
```

На заметку! При запуске хоста (или клиента) в примерах настоящей главы удостоверьтесь в том, что действительно “запускаете” программу из Visual Studio (<Ctrl+F5>), а не отлаживаете ее (<F5>), чтобы обеспечить выполнение процессов хоста и клиента независимым образом.

Подробный анализ элемента <system.serviceModel>

Подобно любому элементу XML внутри <system.serviceModel> допускается определять набор подэлементов, каждый из которых может быть снабжен разнообразными атрибутами. Несмотря на то что за детальными сведениями о наборе возможных атрибутов следует обращаться в документацию .NET Framework 4.7 SDK, ниже приведен каркас со списком некоторых (но не всех) полезных подэлементов:

```
<system.serviceModel>
  <behaviors>
  </behaviors>
  <client>
  </client>
  <commonBehaviors>
  </commonBehaviors>
  <diagnostics>
  </diagnostics>
  <comContracts>
  </comContracts>
  <services>
  </services>
  <bindings>
  </bindings>
</system.serviceModel>
```

Далее в главе вы встретите более экзотические конфигурационные файлы; избранные подэлементы кратко описаны в табл. 23.9.

Таблица 23.9. Избранные подэлементы <service.serviceModel>

Подэлемент	Описание
behaviors	Инфраструктура WCF поддерживает различные линии поведения конечных точек и служб. По существу <i>линия поведения</i> позволяет дополнительно уточнять функциональность хоста, службы или клиента
bindings	Этот элемент позволяет тонко настраивать каждую привязку WCF (например, <code>basicHttpBinding</code> и <code>netMsmqBinding</code>), а также указывать любые специальные привязки, используемые хостом
client	Этот элемент содержит список конечных точек, которые клиент применяет для подключения к службе. Очевидно, что он не особенно полезен в файле <code>*.config</code> хоста
comContracts	Этот элемент определяет контракты COM, обеспечивающие возможность взаимодействия WCF и COM
commonBehaviors	Этот элемент может устанавливаться только внутри файла <code>machine.config</code> . Он используется для определения всех линий поведения, применяемых каждой службой WCF на заданной машине
diagnostics	Этот элемент содержит настройки для средств диагностики WCF. Пользователь может включать/отключать трассировку, счетчики производительности и поставщика WMI, а также добавлять специальные фильтры сообщений
services	Этот элемент содержит коллекцию служб WCF, к которым хост открывает доступ

Включение обмена метаданными

Вспомните, что клиентские приложения WCF взаимодействуют со службой WCF через промежуточный тип прокси. Наряду с тем, что код прокси можно было бы писать целиком вручную, такой процесс будет утомительным и подверженным ошибкам. В идеале должен использоваться какой-то инструмент для генерации необходимого рутинного кода (включая файл *.config клиентской стороны). К счастью, в .NET Framework 4.7 SDK доступен инструмент командной строки (svcutil.exe), предназначенный именно для данной цели. Вдобавок Visual Studio предлагает похожую функциональность, доступную в результате выбора пункта меню Project⇒Add Service Reference (Проект⇒Добавить ссылку на службу).

Однако для того, чтобы упомянутые инструменты генерировали нужный код прокси и файл *.config, они должны быть в состоянии выяснять формат интерфейсов службы WCF и любых определенных контрактов данных (т.е. имена методов и типы параметров).

Обмен метаданными (MEX) — это *линия поведения службы* WCF, которая может применяться для тонкой настройки способа обработки службы исполняющей средой WCF. Выражаясь просто, каждый элемент <behavior> позволяет определять набор действий, на которые заданная служба может подписываться. Инфраструктура WCF предоставляет многочисленные линии поведения в готовом виде, к тому же можно строить собственные линии поведения.

Линия поведения MEX (которая по умолчанию отключена) будет перехватывать любые запросы метаданных, отправленные посредством HTTP-запроса GET. Чтобы позволить инструменту svcutil.exe или Visual Studio автоматизировать создание требуемого прокси клиентской стороны и файла *.config, понадобится включить MEX.

Включение MEX предусматривает корректировку файла *.config хоста с помощью подходящих настроек (или написание соответствующего кода C#). Во-первых, необходимо добавить новый элемент <endpoint> конкретно для MEX. Во-вторых, нужно определить линию поведения WCF для разрешения доступа HTTP-запросам GET. В-третьих, линию поведения MEX потребуется ассоциировать по имени со службой, используя атрибут behaviorConfiguration в открывающем элементе <service>. Наконец, в-четвертых, понадобится добавить элемент <host> для определения базового класса службы (MEX будет искать здесь местоположения описываемых типов).

На заметку! Финальный шаг можно опустить, если для представления базового адреса конструктору класса ServiceHost передается в качестве параметра объект System.Uri.

Взгляните на следующий измененный файл *.config хоста, который создает специальный элемент <behavior> (по имени EightBallServiceMEXBehavior), ассоциированный со службой через атрибут behaviorConfiguration внутри определения <service>:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService"
        behaviorConfiguration="EightBallServiceMEXBehavior">
        <endpoint address = ""
          binding = "basicHttpBinding"
          contract = "MagicEightBallServiceLib.IEightBall"/>
      <!-- Включить конечную точку MEX -->
```

```

    <endpoint address = "mex"
              binding = "mexHttpBinding"
              contract = "IMetadataExchange" />
<!-- Это необходимо добавить, чтобы MEX был известен адрес нашей службы -->
    <host>
      <baseAddresses>
        <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
      </baseAddresses>
    </host>
  </service>
</services>
<!-- Определение линии поведения для MEX -->
<behaviors>
  <serviceBehaviors>
    <behavior name = "EightBallServiceMEXBehavior" >
      <serviceMetadata httpGetEnabled = "true" />
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

Теперь приложение хоста службы можно перезапустить и просмотреть описание метаданных в веб-браузере. Для этого при функционирующем хосте необходимо ввести в строке адреса такой URL:

```
http://localhost:8080/MagicEightBallService
```

На домашней странице службы WCF (рис. 23.5) предоставляются базовые сведения о том, как программно взаимодействовать с данной службой.

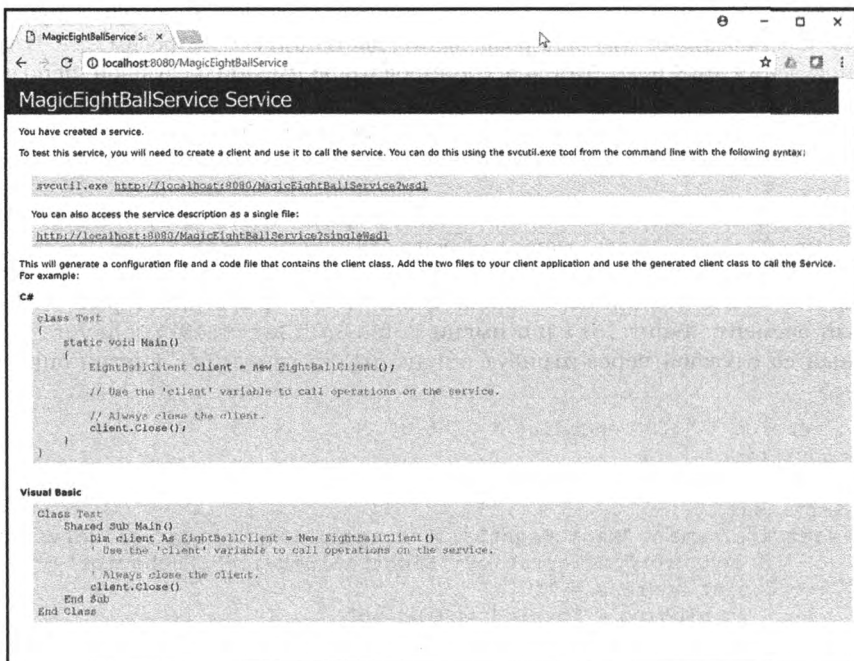


Рис. 23.5. Просмотр метаданных с применением MEX

Щелчок на гиперссылке в верхней части страницы позволяет просмотреть контракт WSDL. Вспомните, что язык описания веб-служб (Web Service Description Language — WSDL) — это грамматика, описывающая структуру веб-служб в заданной конечной точке.

Хост теперь открывает доступ к двум разным конечным точкам (одна для службы и одна для MEX), так что консольный вывод хоста будет выглядеть следующим образом:

```
***** Console Based WCF Host *****
***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall

Address: http://localhost:8080/MagicEightBallService/mex
Binding: MetadataExchangeHttpBinding
Contract: IMetadataExchange
*****
The service is ready.
```

Построение клиентского приложения WCF

При наличии готового хоста последней задачей будет построение фрагмента программного обеспечения для взаимодействия с таким типом службы WCF. Хотя можно избрать длинный путь и построить всю необходимую инфраструктуру вручную (осуществимая, но трудоемкая задача), в .NET Framework 4.7 SDK предлагается несколько подходов для быстрой генерации прокси клиентской стороны.

Генерация кода прокси с использованием `svcutil.exe`

Первый способ построения прокси клиентской стороны предусматривает применение инструмента командной строки `svcutil.exe`. С его помощью можно генерировать новый файл на языке C#, представляющий сам код прокси, а также конфигурационный файл клиентской стороны. Для этого в первом параметре указывается конечная точка службы. Параметр `/out:` используется для определения имени файла `*.cs`, содержащего код прокси, а параметр `/config:` позволяет указать имя генерируемого файла `*.config` клиентской стороны.

Предполагая, что служба в текущий момент запущена, следующий набор параметров, переданный `svcutil.exe`, приведет к генерации двух новых файлов в рабочем каталоге (вся команда с параметрами должна вводиться в одной строке):

```
svcutil http://localhost:8080/MagicEightBallService
/out:myProxy.cs /config:app.config
```

Открыв файл `myProxy.cs`, в нем можно обнаружить представление клиентской стороны интерфейса `IEightBall`, а также новый класс по имени `EightBallClient`, который и является классом прокси. Данный класс будет производным от обобщенного класса `System.ServiceModel.ClientBase<T>`, где `T` — зарегистрированный интерфейс службы.

В дополнение к нескольким специальным конструкторам каждый метод класса прокси (который основан на исходных методах интерфейса) будет реализован для применения унаследованного свойства `Channel` с целью вызова корректного метода службы. Вот частичный код типа прокси:

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel",
    "4.0.0.0")]
```



```
public partial class EightBallClient :
    System.ServiceModel.ClientBase<IEightBall>, IEightBall
{
    ...
    public string ObtainAnswerToQuestion(string userQuestion)
    {
        return base.Channel.ObtainAnswerToQuestion(userQuestion);
    }
}
```

При создании экземпляра типа прокси в клиентском приложении базовый класс установит подключение к конечной точке с использованием настроек, указанных в конфигурационном файле приложения клиентской стороны. Во многом подобно конфигурационному файлу серверной стороны сгенерированный файл App.config клиентской стороны содержит элемент <endpoint> и детали о привязке basicHttpBinding, которая применяется для взаимодействия со службой.

Вы также найдете следующий элемент <client>, который устанавливает АВС с точки зрения клиента:

```
<client>
  <endpoint
    address = "http://localhost:8080/MagicEightBallService"
    binding = "basicHttpBinding"
    bindingConfiguration = "BasicHttpBinding_IEightBall"
    contract = "IEightBall" name = "BasicHttpBinding_IEightBall" />
</client>
```

Сейчас можно было бы включить указанные два файла в проект клиента (вместе со ссылкой на сборку System.ServiceModel.dll) и затем использовать тип прокси для коммуникаций с удаленной службой WCF. Тем не менее, мы примем другой подход и посмотрим, каким образом Visual Studio может помочь в дальнейшей автоматизации создания файлов прокси клиентской стороны.

Генерация кода прокси в Visual Studio

Как и в любом хорошем инструменте командной строки, в утилите svcutil.exe предусмотрено огромное количество параметров, которые можно применять для управления генерацией прокси. Если расширенные параметры не нужны, то те же два файла можно сгенерировать в IDE-среде Visual Studio. Для клиента добавим в решение новый проект консольного приложения по имени MagicEightBallServiceClient и запустим (не в режиме отладки) приложение. Пока служба функционирует, выберем в меню Project (Проект) пункт Add Service Reference (Добавить ссылку на службу).

После выбора указанного пункта меню выберем MagicEightBallService в раскрываемом списке и щелкнем на кнопке Go (Перейти), чтобы просмотреть описание службы (рис. 23.6).

Помимо создания и вставки файлов прокси в текущий проект будут автоматически добавлены ссылки на сборки WCF. В соответствии с соглашением об именовании класс прокси определен внутри пространства имен ServiceReferencel, которое вложено в пространство имен клиента (во избежание возможных конфликтов имен). Ниже приведен полный код клиента:

```
// Местоположение прокси.
using MagicEightBallServiceClient.ServiceReferencel;
namespace MagicEightBallServiceClient
{
```

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Ask the Magic 8 Ball *****\n");
        using (EightBallClient ball = new EightBallClient())
        {
            Console.Write("Your question: ");
            string question = Console.ReadLine();
            string answer =
                ball.ObtainAnswerToQuestion(question);
            Console.WriteLine("8-Ball says: {0}", answer);
        }
        Console.ReadLine();
    }
}

```

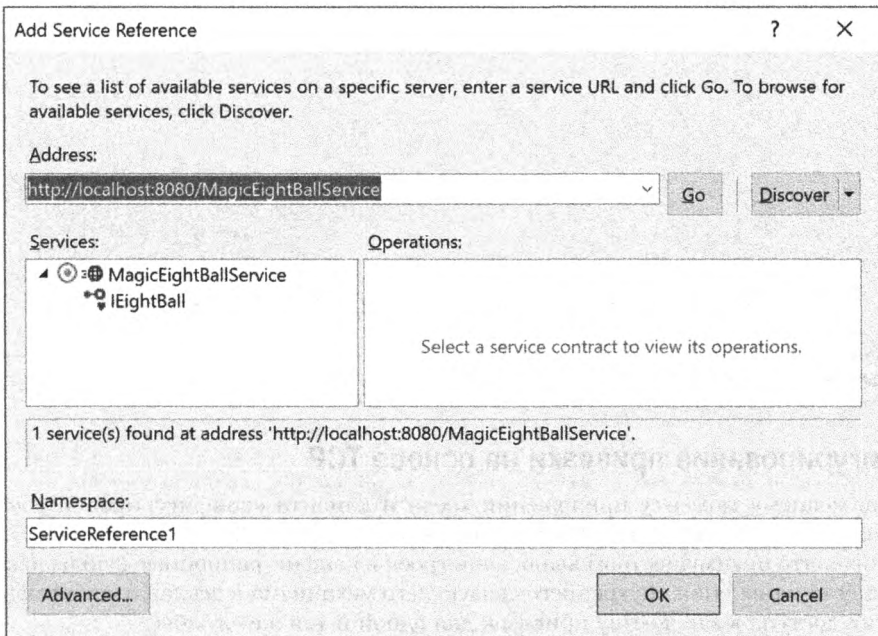


Рис. 23.6. Генерация файлов прокси в Visual Studio

Щелчком правой кнопкой мыши на решении и выберем в контекстном меню пункт **Set as StartUp Project** (Установить как запускаемый проект). Выберем переключатель **Multiple startup projects** (Множество запускаемых проектов), переместим **MagicEightBallServiceHost** в самый верх списка и в столбце **Action** (Действие) установим для проектов **MagicEightBallServiceHost** и **MagicEightBallServiceClient** действие **Start** (Запуск), как показано на рис. 23.7.

Теперь после щелчка на кнопке **Start** (Пуск) и предоставления службе времени на запуск можно задать вопрос в окне клиента и получить случайный ответ в окне консоли. Вот возможный вывод:

***** Ask the Magic 8 Ball *****

Your question: Will I ever finish Fallout 4?

8-Ball says: No

Press any key to continue . . .

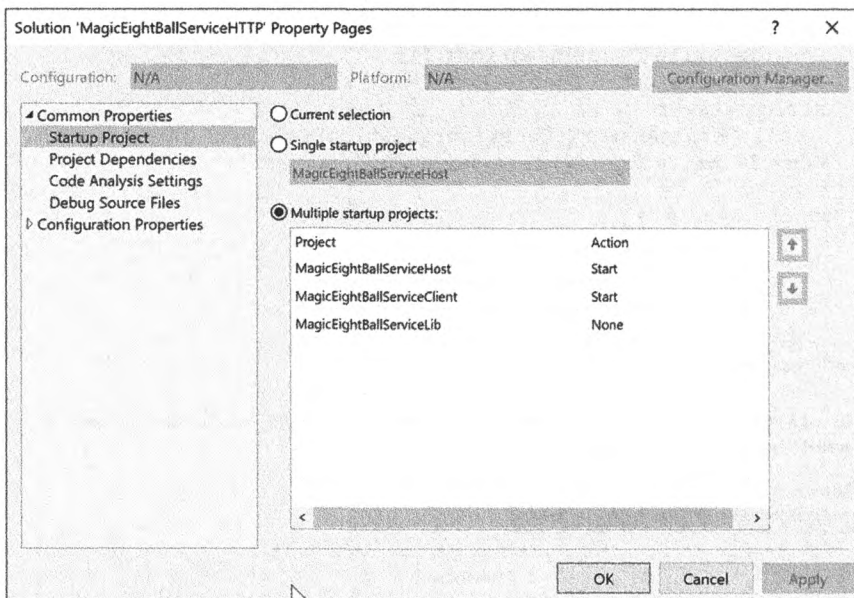


Рис. 23.7. Модификация прокси и файла *.config клиентской стороны

Исходный код. Решение MagicEightBallServiceHTTP доступно в подкаталоге Chapter_23.

Конфигурирование привязки на основе TCP

К настоящему моменту приложения хоста и клиента сконфигурированы для использования простейшей из привязок, основанных на HTTP — `basicHttpBinding`. Вспомните, что преимуществом выноса настроек в конфигурационные файлы является возможность изменения внутреннего связующего механизма в декларативной манере и открытия доступа к множеству привязок для одной и той же службы.

В целях иллюстрации проведем небольшой эксперимент. Создадим новую папку на диске C: (или там, где был сохранен код) по имени `EightBallTCP` и внутри нее две папки `Host` и `Client`.

Затем в проводнике Windows перейдем в папку `\bin\Debug` проекта хоста (рассмотренного ранее в главе) и скопируем файлы `MagicEightBallServiceHost.exe`, `MagicEightBallServiceHost.exe.config` и `MagicEightBallServiceLib.dll` в папку `C:\EightBallTCP\Host`. Откроем файл `*.config` в простом текстовом редакторе и модифицируем его содержимое следующим образом:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService">
```

```

<endpoint address = ""
    binding = "netTcpBinding"
    contract = "MagicEightBallServiceLib.IEightBall"/>
<host>
  <baseAddresses>
    <add baseAddress = "net.tcp://localhost:8090/MagicEightBallService"/>
  </baseAddresses>
</host>
</service>
</services>
</system.serviceModel>
</configuration>

```

По существу из файла *.config удалены все настройки MEX (т.к. уже создан прокси) и установлено применение типа привязки NetTcpBinding через уникальный порт. Запустим приложение, дважды щелкнув на файле *.exe. Если все было сделано правильно, тогда должен появиться показанный ниже вывод:

```

***** Console Based WCF Host *****
***** Host Info *****
Address: net.tcp://localhost:8090/MagicEightBallService
Binding: NetTcpBinding
Contract: IEightBall
*****
The service is ready.
Press the Enter key to terminate service.

```

Чтобы завершить тест, скопируем файлы MagicEightBallServiceClient.exe и MagicEightBallServiceClient.exe.config из папки \bin\Debug клиентского приложения (обсуждалось ранее в главе) в папку C:\EightBallTCP\Client. Модифицируем конфигурационный файл следующим образом:

```

<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <client>
      <endpoint address = "net.tcp://localhost:8090/MagicEightBallService"
        binding = "netTcpBinding"
        contract = "ServiceReferencel.IEightBall"
        name = "netTcpBinding_IEightBall" />
    </client>
  </system.serviceModel>
</configuration>

```

Данный конфигурационный файл клиентской стороны значительно проще файла, создаваемого генератором прокси Visual Studio. Обратите внимание, что существующий элемент <bindings> удален. Первоначально файл *.config содержал элемент <bindings> с подэлементом <basicHttpBinding>, который определял множество деталей, касающихся настроек привязки клиента (например, таймауты).

На самом деле для рассматриваемого примера такие детали никогда не понадобятся, поскольку мы автоматически получаем стандартные значения лежащего в основе объекта BasicHttpBinding. Конечно, при необходимости можно было бы обновить существующий элемент <bindings>, определив детали подэлемента <netTcpBinding>, но это совершенно не обязательно, если устраивают стандартные значения объекта NetTcpBinding.

Теперь должно быть все готово к запуску клиентского приложения. При условии, что хост все еще функционирует в фоновом режиме, появится возможность перемещения данных между сборками по протоколу TCP.

Исходный код. Конфигурационные файлы проекта MagicEightBallTCP доступны в подкаталоге Chapter_23.

Упрощение конфигурационных настроек

При проработке первого примера в настоящей главе вы могли заметить, что логика конфигурации хостинга довольно многословна. Например, файл *.config хоста (для исходной базовой привязки HTTP) должен определять один элемент <endpoint> для службы, еще один элемент <endpoint> для MEX, элемент <baseAddresses> (формально необязательный) для сокращения избыточных URI и затем раздел <behaviors> для указания характеристик обмена метаданными во время выполнения.

По правде говоря, изучение правил написания файлов *.config при построении служб WCF может оказаться трудной задачей. Чтобы еще более усложнить положение дел, порядочное число служб WCF склонно требовать те же самые базовые настройки в конфигурационном файле хоста. Например, если создается новая служба WCF и новый хост, и нужно открыть доступ к этой службе, используя элемент <basicHttpBinding> с поддержкой MEX, то необходимое содержимое файла *.config будет выглядеть практически идентично созданному ранее.

К счастью, начиная с выпуска .NET 4.0, инфраструктура Windows Communication Foundation включает ряд упрощений, в числе которых стандартные настройки (и другие сокращения), намного облегчающие процесс построения конфигурации хоста.

Использование стандартных конечных точек

До появления поддержки стандартных конечных точек, когда вызывался метод Open() на объекте ServiceHost, а в конфигурационном файле не было определено ни одного элемента <endpoint>, исполняющая среда генерировала исключение. Аналогичный результат получался при вызове метода AddServiceEndpoint() в коде для указания конечной точки. Однако, начиная с версии .NET 4.5, каждая служба WCF автоматически получает *стандартные конечные точки*, которые фиксируют общепринятые детали конфигурации для каждого поддерживаемого протокола.

После открытия файла machine.config для .NET 4.5 в нем обнаружится новый элемент по имени <protocolMapping>, документирующий привязки WCF, которые будут применяться по умолчанию, если ни одной привязки не указано:

```
<system.serviceModel>
...
<protocolMapping>
  <add scheme = "http" binding="basicHttpBinding"/>
  <add scheme = "net.tcp" binding="netTcpBinding"/>
  <add scheme = "net.pipe" binding="netNamedPipeBinding"/>
  <add scheme = "net.msmsg" binding="netMsmqBinding"/>
</protocolMapping>
...
</system.serviceModel>
```

Для использования таких стандартных привязок потребуется только указать базовые адреса в конфигурационном файле хоста. Чтобы увидеть это в действии, откроем в Visual Studio проект MagicEightBallServiceHost, основанный на HTTP. Модифицируем

файл *.config хоста, полностью удалив элемент <endpoint> для службы WCF и все данные, связанные с MEX. Теперь конфигурационный файл должен выглядеть примерно так:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Поскольку в <baseAddress> указан допустимый базовый адрес HTTP, хост автоматически будет применять привязку basicHttpBinding. Запустив хост снова, можно увидеть тот же самый вывод данных ABC:

```
***** Console Based WCF Host *****
***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall
*****
The service is ready.
Press the Enter key to terminate service.
```

Здесь пока еще не включены службы MEX, но это вскоре будет сделано с использованием другого упрощения, которое называется *конфигурациями стандартного поведения*. Тем не менее, давайте сначала посмотрим, как открывать доступ к одиночной службе WCF с множеством привязок.

Открытие доступа к одиночной службе WCF, использующей множество привязок

Со времен своего первого выпуска инфраструктура WCF позволяет одному хосту открывать доступ к службе WCF с несколькими конечными точками. Например, открыть доступ к службе MagicEightBallService, применяющей привязки HTTP, TCP и именованных каналов, можно за счет добавления новых конечных точек в конфигурационный файл. После перезапуска хоста весь необходимый связующий механизм создается автоматически.

Указанная характеристика дает огромное преимущество по многим причинам. До появления WCF открывать доступ к одной службе с множеством привязок было трудно, т.к. каждый тип привязки (например, HTTP и TCP) имел собственную модель программирования.

Однако возможность разрешения вызывающему коду выбирать наиболее подходящую привязку чрезвычайно удобна. Внутренние вызывающие компоненты могут отдавать предпочтение привязкам TCP, внешние клиенты (находящиеся за брандмауэром компании) — использовать для доступа HTTP, в то время как клиенты на той же самой машине — применять именованный канал.

Чтобы сделать подобное в версиях, предшествующих .NET 4.5, внутри конфигурационного файла хоста требовалось вручную определять множество элементов <endpoint>. Также для каждого протокола необходимо было определять множество элементов <baseAddress>. Тем не менее, теперь можно просто подготовить следующий конфигурационный файл:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
            <add baseAddress =
              "net.tcp://localhost:8099/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Скомпилировав проект (для обновления развернутого файла *.config) и перезапустив хост, можно будет увидеть следующие данные конечной точки:

```
***** Console Based WCF Host *****

***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall

Address: net.tcp://localhost:8099/MagicEightBallService
Binding: NetTcpBinding
Contract: IEightBall

*****
The service is ready.
Press the Enter key to terminate service.
```

Теперь, когда служба WCF достижима из двух конечных точек, возникает вопрос: каким образом клиент может производить выбор между ними? При генерации прокси клиентской стороны инструмент, запускаемый выбором пункта меню Add Service Reference, назначит каждой доступной конечной точке строковое имя в файле *.config клиентской стороны. В коде можно передавать корректное строковое имя конструктору прокси и иметь уверенность в том, что будет выбрана правильная привязка. Однако прежде чем поступать так, потребуется переустановить MEX для модифицированного конфигурационного файла хоста и научиться настраивать параметры стандартной привязки.

Изменение параметров привязки WCF

В случае указания ABC службы в коде C# (что будет осуществляться далее в главе) способ изменения стандартных параметров привязки WCF вполне очевиден: нужно просто модифицировать значения свойств объекта. Например, если необходимо использовать привязку BasicHttpBinding, но изменить параметры таймаута, то можно написать такой код:

```

void ConfigureBindingInCode()
{
    BasicHttpBinding binding = new BasicHttpBinding();
    binding.OpenTimeout = TimeSpan.FromSeconds(30);
    ...
}

```

Параметры привязки всегда допускается конфигурировать в декларативной манере. Например, версия платформы .NET 3.5 позволяет строить конфигурационный файл хоста, в котором изменяется свойство `OpenTimeout` класса `BasicHttpBinding`, как показано ниже:

```

<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name = "myCustomHttpBinding"
          openTimeout = "00:00:30" />
      </basicHttpBinding>
    </bindings>
    <services>
      <service name = "WcfMathService.MyCalc">
        <endpoint address = "http://localhost:8080/MyCalc"
          binding = "basicHttpBinding"
          bindingConfiguration = "myCustomHttpBinding"
          contract = "WcfMathService.IBasicMath" />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

Здесь мы имеем конфигурационный файл для службы по имени `WcfMathService.MyCalc`, которая поддерживает единственный интерфейс `IBasicMath`. Обратите внимание, что раздел `<bindings>` позволяет определить именованный элемент `<binding>`, который изменяет параметры для заданной привязки. Внутри элемента `<endpoint>` службы можно подключать специфические параметры с применением атрибута `bindingConfiguration`.

Такой вид конфигурации хостинга по-прежнему работает ожидаемым образом; тем не менее, если задействуется стандартная конечная точка, тогда подключить `<binding>` к `<endpoint>` не удастся! К счастью, параметрами стандартной конечной точки можно управлять, просто опуская атрибут `name` элемента `<binding>`. Например, в следующем фрагменте кода изменяются некоторые свойства стандартных объектов `BasicHttpBinding` и `NetTcpBinding`, используемые на заднем плане:

```

<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
            <add baseAddress =
              "net.tcp://localhost:8099/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
    <bindings>

```



```

<basicHttpBinding>
  <binding openTimeout = "00:00:30" />
</basicHttpBinding>
<netTcpBinding>
  <binding closeTimeout = "00:00:15" />
</netTcpBinding>
</bindings>
</system.serviceModel>
</configuration>

```

Использование конфигурации стандартного поведения MEX

Инструмент генерации прокси должен сначала обнаружить композицию службы во время выполнения. В WCF такое обнаружение во время выполнения разрешается путем включения MEX. В большинстве конфигурационных файлов хоста MEX понадобится включить (по крайней мере, на протяжении разработки); к счастью, способ конфигурирования MEX изменяется редко, поэтому в .NET 4.5 и последующих версиях предлагается несколько удобных сокращений.

Наиболее полезным сокращением является готовая поддержка MEX. Вам не придется добавлять конечную точку MEX, определять именованную линию поведения служб MEX и затем подключать именованную привязку к службе (как делалось в HTTP-версии MagicEightBallServiceHost); взамен можно просто добавить следующий код:

```

<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
            <add baseAddress =
              "net.tcp://localhost:8099/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
    <bindings>
      <basicHttpBinding>
        <binding openTimeout = "00:00:30" />
      </basicHttpBinding>
      <netTcpBinding>
        <binding closeTimeout = "00:00:15" />
      </netTcpBinding>
    </bindings>
    <behaviors>
      <serviceBehaviors>
        <behavior>
<!-- Для получения стандартного MEX не именуйте элемент <serviceMetadata> -->
          <serviceMetadata httpGetEnabled = "true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```

Трюк заключается в том, что элемент <serviceMetadata> больше не имеет атрибута name (кроме того, элемент <service> больше не нуждается в атрибуте behavior

Configuration). После указанных корректировок во время выполнения становится доступной поддержка MEX. Чтобы проверить это, можно запустить хост (после компиляции с целью обновления конфигурационного файла) и ввести в браузере такой URL:

`http://localhost:8080/MagicEightBallService`

Затем можно щелкнуть на ссылке `wsdl` в верхней части веб-страницы и посмотреть описание WSDL службы (см. рис. 23.5). Обратите внимание, что в выводе внутри окна консоли хоста отсутствуют данные о конечной точке MEX, т.к. конечная точка для `IMetadataExchange` в конфигурационном файле явно не определялась. Однако службы MEX включены, и можно приступать к построению клиентских прокси.

Обновление клиентского прокси и выбор привязки

Предполагая, что обновленный хост скомпилирован и функционирует в фоновом режиме, теперь необходимо открыть клиентское приложение и обновить текущую ссылку на службу. Начнем с разворачивания узла `Service References` (Ссылки на службы) в окне `Solution Explorer`. Далее щелкнем правой кнопкой мыши на элементе `ServiceReference` и выберем в контекстном меню пункт `Update Service Reference` (Обновить ссылку на службу), как показано на рис. 23.8.

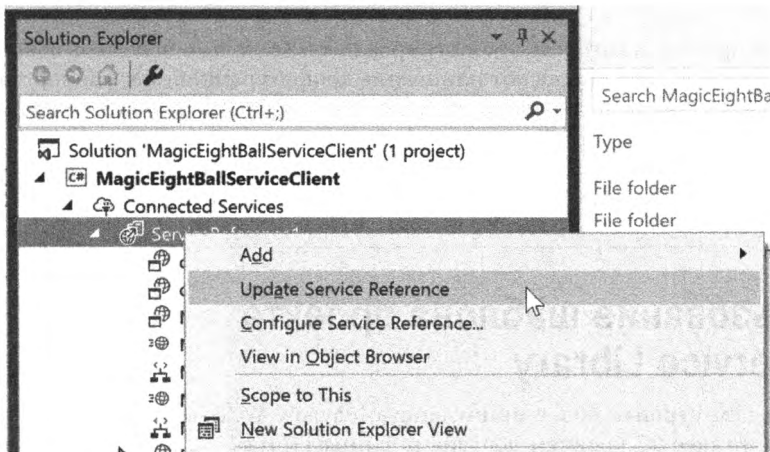


Рис. 23.8. Обновление прокси и файла *.config клиентской стороны

В результате внутри файла *.config клиентской стороны появятся две привязки на выбор: одна для HTTP и одна для TCP. Каждой привязке назначено подходящее имя. Ниже приведен частичный листинг обновленного конфигурационного файла:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name = "BasicHttpBinding_IEightBall" ... />
      </basicHttpBinding>
      <netTcpBinding>
        <binding name = "NetTcpBinding_IEightBall" ... />
      </netTcpBinding>
    </bindings>
    ...
  </system.serviceModel>
</configuration>
```

Клиент может применять указанные имена при создании прокси-объекта для выбора желаемой привязки. Таким образом, если клиент предпочитает использовать TCP, то код C# клиентской стороны можно модифицировать следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Ask the Magic 8 Ball *****\n");
    using (EightBallClient ball = new EightBallClient("NetTcpBinding_IEightBall"))
    {
        ...
    }
    Console.ReadLine();
}
```

Если же клиент взамен будет применять привязку HTTP, тогда можно написать такой код:

```
using (EightBallClient ball = new
    EightBallClient("BasicHttpBinding_IEightBall"))
{
    ...
}
```

Текущий пример, в котором демонстрировалось несколько полезных сокращений, завершен. Такие средства упрощают написание конфигурационных файлов хостов. Далее будет показано, как использовать шаблон проекта WCF Service Library (Библиотека служб WCF).

Исходный код. Проект MagicEightBallServiceHTTPDefaultBindings доступен в подкаталоге Chapter_23.

Использование шаблона проекта WCF Service Library

Прежде чем строить более причудливую службу WCF, которая взаимодействует с созданным в главе 22 уровнем доступа к данным AutoLot, в следующем примере будут проиллюстрированы важные аспекты, в том числе преимущества шаблона проекта WCF Service Library, приложение WCF Test Client, редактор конфигурации WCF, размещение служб WCF внутри Windows-службы и асинхронные клиентские вызовы. Чтобы сосредоточить все внимание на новых концепциях, создаваемая служба WCF будет намеренно сохранена простой.

Построение простой математической службы

Для начала создадим новый проект WCF Service Library по имени MathServiceLibrary. Затем изменим имя начального файла IService1.cs на IBasicMath.cs. Далее удалим весь код из пространства имен MathServiceLibrary и поместим в него следующий код:

```
[ServiceContract (Namespace="http://MyCompany.com")]
public interface IBasicMath
{
    [OperationContract]
    int Add(int x, int y);
}
```

Переименуем файл `Service1.cs` в `MathService.cs`, удалим весь код из пространства имен `MathServiceLibrary` и реализуем контракт службы, как показано ниже:

```
public class MathService : IBasicMath
{
    public int Add(int x, int y)
    {
        // Для эмуляции длительного запроса.
        System.Threading.Thread.Sleep(5000);
        return x + y;
    }
}
```

Обратите внимание, что данный файл `*.config` уже включает поддержку MEX; по умолчанию конечная точка службы применяет протокол `basicHttpBinding`.

Тестирование службы WCF с помощью `WcfTestClient.exe`

Одно из преимуществ использования проекта WCF Service Library связано с тем, что при отладке или запуске библиотеки он будет читать настройки из файла `*.config` и применять их для загрузки приложения WCF Test Client (`WcfTestClient.exe`). Упомянутое приложение с графическим пользовательским интерфейсом позволяет протестировать каждый член интерфейса службы по мере ее построения, т.е. вручную строить хост/клиент просто для целей тестирования, как делалось ранее, не придется.

На рис. 23.9 показана тестовая среда для службы `MathService`. Обратите внимание, что двойной щелчок на методе интерфейса позволяет указать входные параметры и вызвать метод.

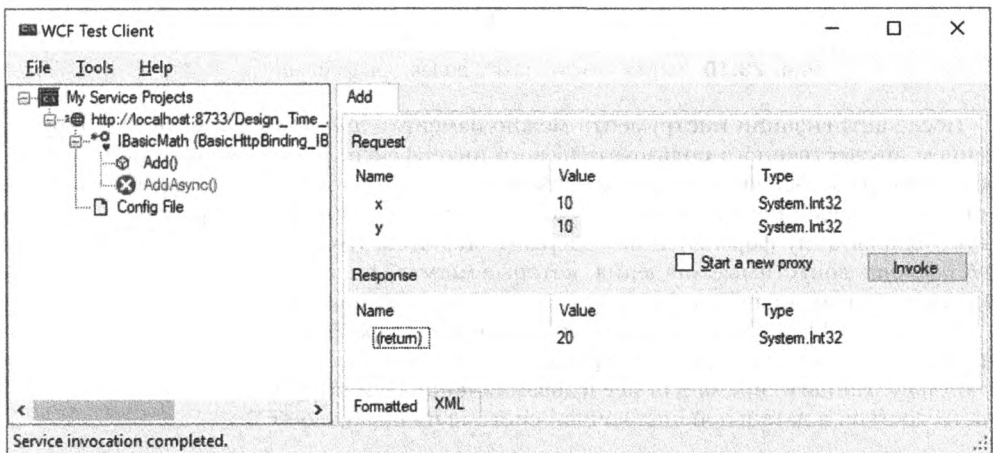


Рис. 23.9. Тестирование службы WCF с использованием `WcfTestClient.exe`

Эта утилита запускается сразу же после создания проекта WCF Service Library; тем не менее, имейте в виду, что данный инструмент можно применять для тестирования любой службы WCF, запустив его в командной строке и указав конечную точку MEX. Например, если запустить приложение `MagicEightBallServiceHost.exe`, то в окне командной строки разработчика можно ввести следующую команду:

```
wcftestclient http://localhost:8080/MagicEightBallService
```

Затем в похожей манере можно вызвать метод `ObtainAnswerToQuestion()`.

Изменение конфигурационных файлов с использованием SvcConfigEditor.exe

Еще одно преимущество применения проекта WCF Service Library связано с тем, что по щелчку правой кнопкой мыши на файле `App.config` в окне Solution Explorer можно активизировать графический редактор конфигурирования службы (Service Configuration Editor), `SvcConfigEditor.exe` (рис. 23.10). Тот же самый прием может использоваться в клиентском приложении, которое ссылается на службу WCF.

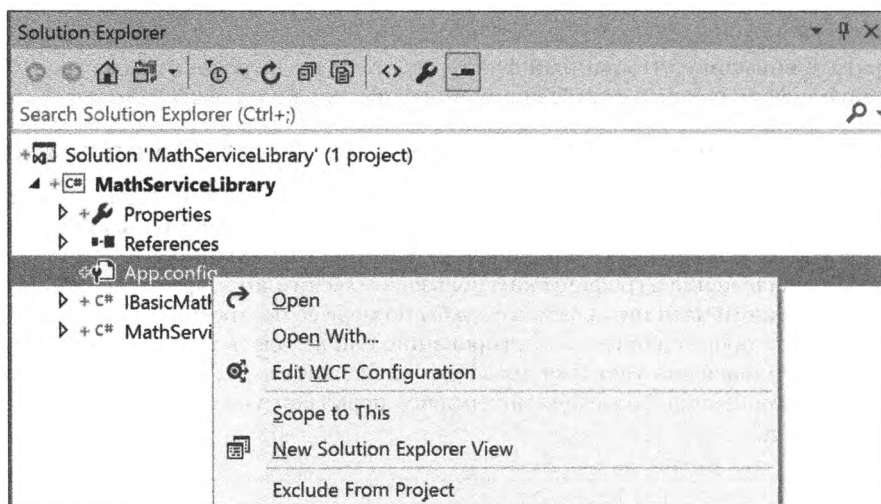


Рис. 23.10. Запуск графического редактора файлов *.config

После активизации инструмента можно изменять данные в формате XML с применением дружелюбного пользовательского интерфейса. Использование инструментов подобного рода для сопровождения файлов *.config дает много преимуществ. Первое (и самое главное): появляется гарантия того, что сгенерированная разметка соответствует ожидаемому формату и не содержит опечаток. Второе: это замечательный способ увидеть допустимые значения, которые могут быть присвоены каждому атрибуту. И третье: больше не понадобится вручную вводить данные XML.

На рис. 23.11 показан внешний вид окна редактора Service Configuration Editor. По правде говоря, описание всех интересных средств `SvcConfigEditor.exe` заняло бы целую главу. Найдите время для исследования данного инструмента; помните о возможности доступа к детальной справочной системе по нажатию <F1>.

На заметку! Утилита `SvcConfigEditor.exe` позволяет редактировать (или создавать) конфигурационные файлы, даже если не был выбран начальный проект WCF Service Library. Необходимо просто запустить инструмент в окне командной строки разработчика и с помощью пункта меню `File⇒Open` (Файл⇒Открыть) загрузить существующий файл *.config для редактирования.

Конфигурировать службу `MathService` больше не понадобится, так что можно перейти к задаче построения специального хоста.

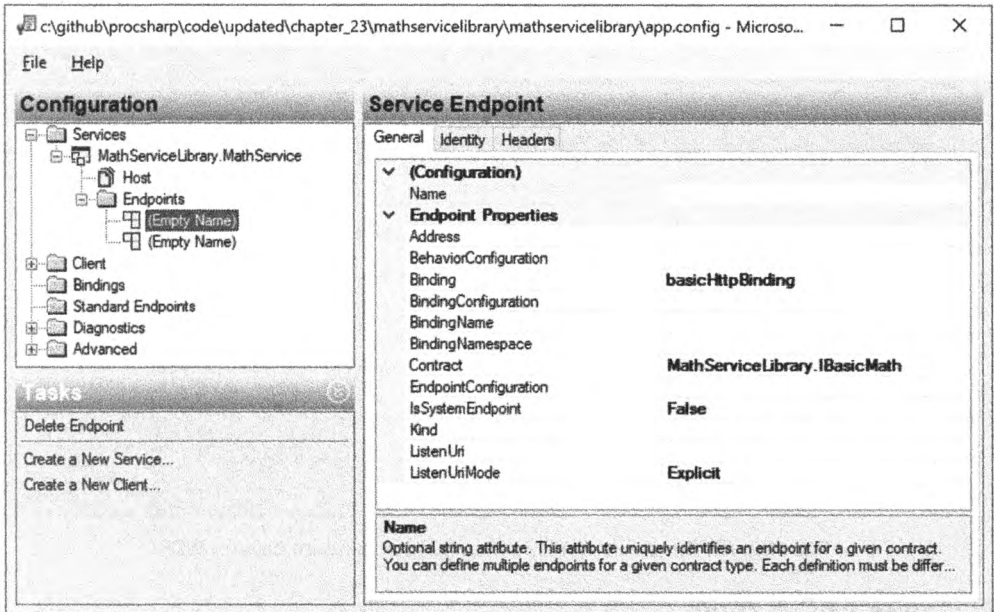


Рис. 23.11. Работа с редактором Service Configuration Editor

Хостинг службы WCF внутри Windows-службы

Хостинг службы WCF в консольном приложении (или внутри настольного приложения с графическим пользовательским интерфейсом) — не идеальный вариант для сервера производственного уровня, поскольку хост всегда должен оставаться функционирующим в фоновом режиме и готовым к обслуживанию клиентов. Даже если свернуть приложение-хост в панели задач Windows, то все равно очень легко случайно прекратить его работу и тем самым разорвать подключения клиентских приложений.

На заметку! Хотя и верно то, что настольное приложение Windows *не обязано* отображать главное окно, типичная программа *.exe требует взаимодействия с пользователем для ее загрузки и запуска. Однако службу Windows (описанную далее) можно сконфигурировать на запуск даже при отсутствии пользователей, вошедших в систему рабочей станции.

Если вы строите внутреннее приложение WCF, то еще одной альтернативой для хостинга библиотеки службы WCF будет ее размещение внутри Windows-службы. Преимущество такого решения связано с тем, что Windows-служба может быть сконфигурирована для автоматического запуска при загрузке системы на целевой машине. Другое преимущество заключается в том, что Windows-служба выполняется невидимо в фоновом режиме (в отличие от консольного приложения) и не требует участия пользователя (к тому же на машине хостинга не требуется наличие установленного сервера IIS).

Давайте посмотрим, как построить такой хост. Начнем с добавления в решение нового проекта Windows Service (Служба Windows) по имени MathWindowsServiceHost (рис. 23.12). В окне Solution Explorer переименуем начальный файл Service1.cs на MathService.cs.

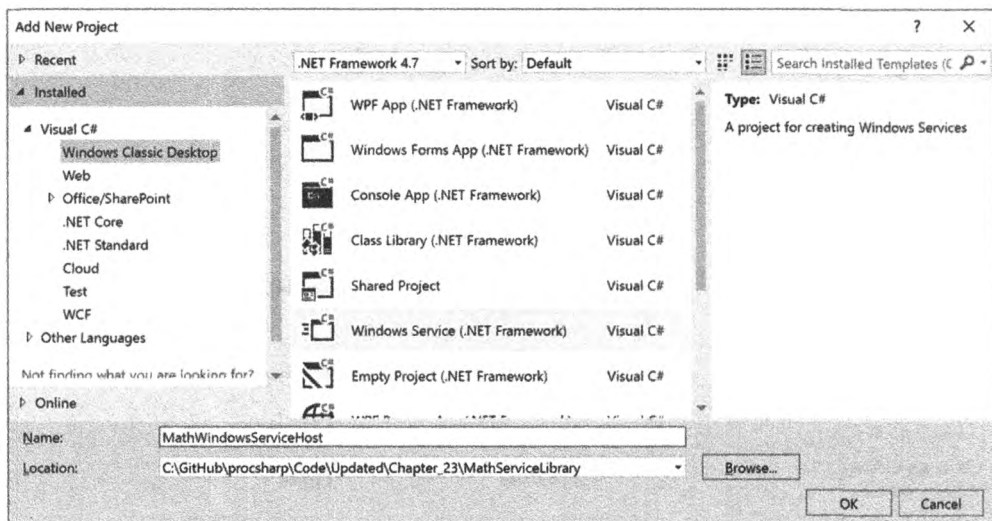


Рис. 23.12. Создание Windows-службы для хостинга службы WCF

Указание ABC в коде

Теперь предположим, что установлены ссылки на сборки `MathServiceLibrary.dll` и `System.ServiceModel.dll`. Все, что осталось сделать — применить тип `ServiceHost` внутри методов `OnStart()` и `OnStop()` типа Windows-службы. Откроем файл кода для класса хоста службы (щелкнув правой кнопкой мыши на поверхности визуального конструктора и выбрав в контекстном меню пункт `View Code` (Просмотреть код)) и поместим в него следующий код:

```
// Должны быть импортированы следующие пространства имен:
using MathServiceLibrary;
using System.ServiceModel;
using System.ServiceProcesses;

namespace MathWindowsServiceHost
{
    public partial class MathWinService: ServiceBase
    {
        // Переменная-член типа ServiceHost.
        private ServiceHost myHost;

        public MathWinService()
        {
            InitializeComponent();
        }

        protected override void OnStart(string[] args)
        {
            // Проверить для подстраховки.
            myHost?.Close();
            // Создать хост.
            myHost = new ServiceHost (typeof(MathService));
            // Указать ABC в коде.
            Uri address = new Uri("http://localhost:8080/MathServiceLibrary");
            WSHttpBinding binding = new WSHttpBinding();
            Type contract = typeof(IBasicMath);
```

```

        // Добавить эту конечную точку.
        myHost.AddServiceEndpoint(contract, binding, address);
        // Открыть хост.
        myHost.Open();
    }
    protected override void OnStop()
    {
        // Остановить хост.
        myHost?.Close();
    }
}

```

Хотя при построении хоста для службы WCF на основе Windows-службы ничто не препятствует использованию конфигурационного файла, здесь (для разнообразия) вместо файла *.config конечная точка устанавливается в коде с применением классов Uri, WSHttpBinding и Type. После создания всех аспектов ABC хост программно информируется о них с помощью метода AddServiceEndpoint().

Если исполняющей среде нужно указать о том, что необходим доступ ко всем привязкам стандартных конечных точек, которые хранятся в конфигурационном файле machine.config платформы .NET 4.5, то программную логику можно упростить, указывая базовые адреса при вызове конструктора ServiceHost. В таком случае не придется задавать ABC в коде вручную или вызывать метод AddServiceEndPoint(); взамен необходимо просто вызвать AddDefaultEndpoints(). Взгляните на следующее изменение кода:

```

protected override void OnStart(string[] args)
{
    myHost?.Close();
    // Создать хост и указать URL для привязки HTTP.
    myHost = new ServiceHost(typeof(MathService),
        new Uri("http://localhost:8080/MathServiceLibrary"));
    // Выбрать стандартные конечные точки.
    myHost.AddDefaultEndpoints();
    // Открыть хост.
    myHost.Open();
}

```

Включение MEX

Несмотря на то что включить MEX можно программно, мы сделаем это в конфигурационном файле. Модифицируем файл App.config в проекте Windows-службы, поместив в него следующие стандартные настройки MEX:

```

<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MathServiceLibrary.MathService">
        </service>
      </services>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled = "true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```



```
</serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>
```

Создание программы установки для Windows-службы

Чтобы зарегистрировать Windows-службу в операционной системе, к проекту понадобится добавить программу установки, которая будет содержать код, необходимый для регистрации службы. Щелкнем правой кнопкой мыши на поверхности визуального конструктора Windows-службы и выберем в контекстном меню пункт Add Installer (Добавить программу установки), как показано на рис. 23.13.

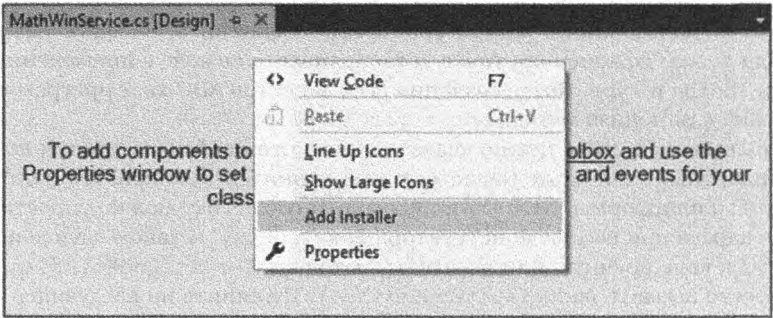


Рис. 23.13. Добавление программы установки для Windows-службы

В результате на поверхность визуального конструктора добавятся два новых компонента. Первый компонент (именуемый по умолчанию serviceProcessInstaller1) представляет элемент, который может установить новую Windows-службу на целевой машине. Выберем его в визуальном конструкторе и в окне Properties (Свойства) установим свойство Account в LocalSystem (рис. 23.14).

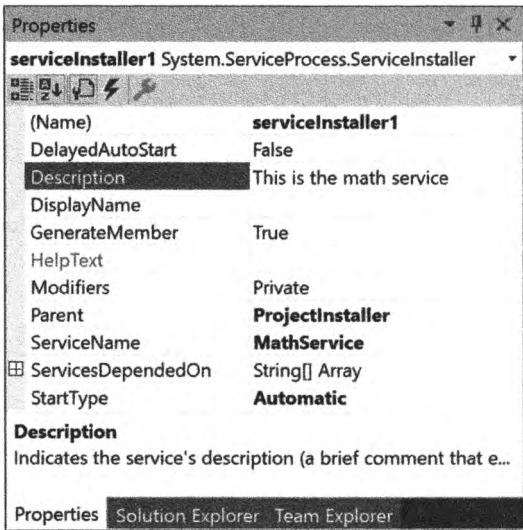


Рис. 23.14. Служба Windows должна запускаться от имени учетной записи локальной системы

Второй компонент (под названием `serviceInstaller`) представляет тип, который будет устанавливать конкретную Windows-службу. В окне **Properties** изменим значение свойства `ServiceName` на `MathService`, установим свойство `StartType` в `Automatic` и укажем дружественное описание зарегистрированной Windows-службы в свойстве `Description` (рис. 23.15).

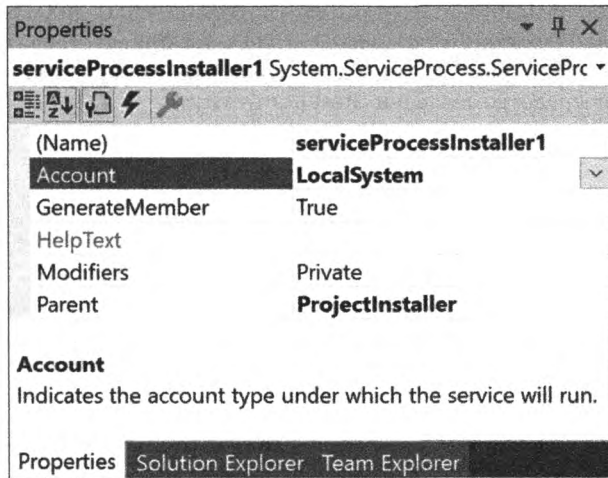


Рис. 23.15. Конфигурирование деталей, связанных с программой установки

Откроем файл `Program.cs` из проекта `MathWindowsServiceHost` и обновим метод `Main()` следующим образом:

```
static void Main()
{
    ServiceBase[] ServicesToRun;
    ServicesToRun = new ServiceBase[]
    {
        new MathWinService()
    };
    ServiceBase.Run(ServicesToRun);
}
```

Теперь приложение можно компилировать.

Установка Windows-службы

Служба Windows может быть установлена на машине-хосте с помощью традиционной программы установки (такой как установщик `*.msi`) либо инструмента командной строки `installutil.exe`.

На заметку! Для установки Windows-службы с использованием `installutil.exe` окно командной строки разработчика должно быть запущено с административными привилегиями. Чтобы сделать это, нужно щелкнуть правой кнопкой мыши на значке **Developer Command Prompt** (Командная строка разработчика) и выбрать в контекстном меню пункт **Запуск от имени администратора**.

В окне командной строки разработчика (запущенного от имени администратора) перейдем в папку `\bin\Debug` проекта `MathWindowsServiceHost` и введем следующую команду:

```
installutil MathWindowsServiceHost.exe
```

Предполагая, что установка прошла успешно, можно щелкнуть на значке **Services** (Службы) в папке **Administrative Tools** (Администрирование) панели управления Windows. В списке служб, упорядоченном по алфавиту, должно присутствовать дружественное имя службы `MathService` (рис. 23.16). Если служба еще не функционирует, то ее необходимо запустить посредством ссылки **Start** (Запустить).

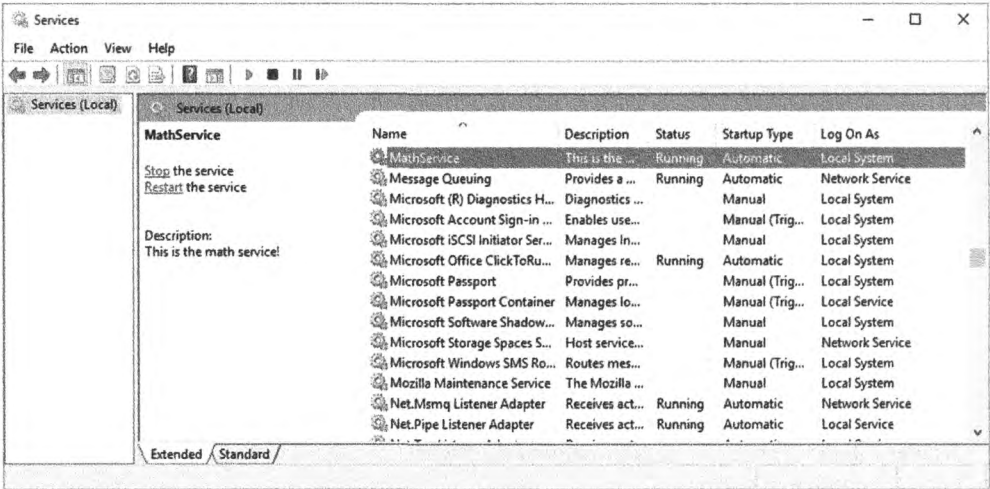


Рис. 23.16. Служба Windows, в которой размещается наша служба WCF

Теперь, когда служба установлена и работает, осталось построить клиентское приложение для ее потребления.

Исходный код. Проект `MathWindowsServiceHost` доступен в подкаталоге `Chapter_23`.

Асинхронный вызов службы из клиента

Добавим в решение новый проект консольного приложения по имени `MathClient` и пометим его как запускающийся. Установим ссылку на выполняющуюся службу WCF (в настоящий момент размещенную в Windows-службе, которая функционирует в фоновом режиме), выбрав пункт меню **Project**⇒**Add Service Reference** (Проект⇒Добавить ссылку на службу) в Visual Studio (в поле **Address** (Адрес) понадобится ввести URL вида `http://localhost:8080/MathServiceLibrary`). Пока не следует щелкать на кнопке **OK**! В нижнем левом углу диалогового окна **Add Service Reference** (Добавление ссылки на службу) находится кнопка **Advanced** (Дополнительно), как показано на рис. 23.17.

Щелкнем на кнопке **Advanced**, чтобы просмотреть дополнительные настройки конфигурации прокси в диалоговом окне **Service Reference Settings** (Настройки ссылки на службу), представленном на рис. 23.18. В этом диалоговом окне можно сгенерировать код, который позволит вызывать удаленные методы в асинхронной манере, если выбран переключатель **Generate asynchronous operations** (Генерировать асинхронные операции). Выберем упомянутый переключатель.

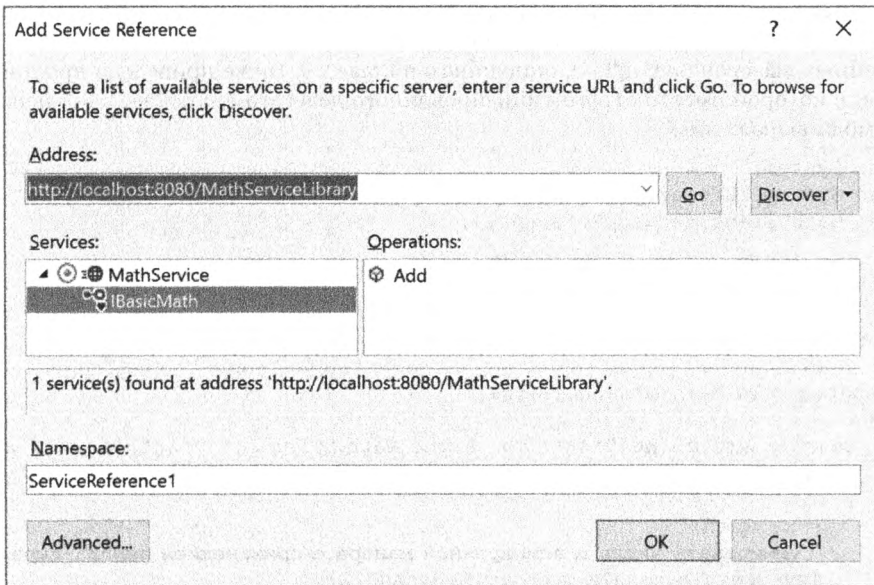


Рис. 23.17. Добавление ссылки на службу MathService и подготовка к конфигурированию расширенных настроек

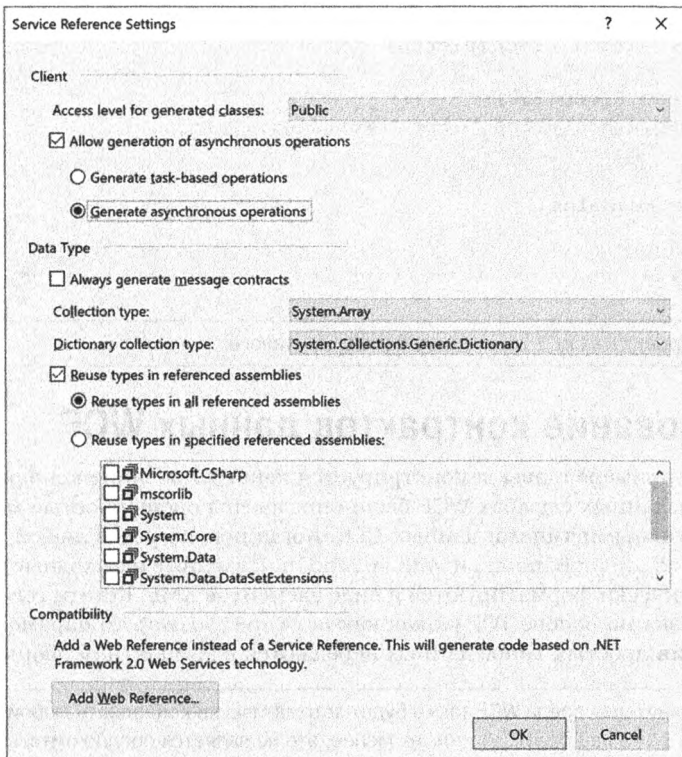


Рис. 23.18. Дополнительные настройки конфигурации прокси клиентской стороны

В данный момент код прокси содержит дополнительные методы, которые позволяют обращаться к каждому члену контракта службы с применением ожидаемого паттерна асинхронных вызовов `Begin/End`, описанного в главе 19. Ниже приведена простая реализация, в которой вместо строго типизированного делегата `AsyncCallback` используется лямбда-выражение:

```
using System;
using System.Threading;
using MathClient.ServiceReference1;
...
namespace MathClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** The Async Math Client *****\n");
            using (BasicMathClient proxy = new BasicMathClient())
            {
                proxy.Open();
                //Суммировать числа в асинхронной манере с применением лямбда-выражения
                IAsyncResult result = proxy.BeginAdd(2, 3,
                    ar =>
                    {
                        Console.WriteLine("2 + 3 = {0}", proxy.EndAdd(ar));
                    },
                    null);
                while (!result.IsCompleted)
                {
                    Thread.Sleep(200);
                    Console.WriteLine("Client working...");
                }
            }
            Console.ReadLine();
        }
    }
}
```

Исходный код. Проект `MathClient` доступен в подкаталоге `Chapter_23`.

Проектирование контрактов данных WCF

В последнем примере главы демонстрируется конструирование *контрактов данных* WCF. В ранее созданных службах WCF были определены очень простые методы, оперирующие примитивными типами данных CLR. Когда используется любой тип привязки HTTP (скажем, `basicHttpBinding` и `wsHttpBinding`), входные и выходные простые типы данных автоматически форматируются в виде элементов XML. Кстати говоря, если применяется привязка на основе TCP (такая как `netTcpBinding`), то параметры и возвращаемые значения простых типов данных передаются в компактном двоичном формате.

На заметку! Исполняющая среда WCF также будет автоматически кодировать любой тип, помеченный атрибутом `[Serializable]`; тем не менее, это не является предпочтительным способом определения контрактов WCF и предназначено только для обратной совместимости.

Однако при определении контрактов служб, которые используют специальные классы в качестве параметров или возвращаемых значений, такие данные рекомендуется моделировать с применением контрактов данных WCF. Выразаясь просто, контракт данных представляет собой тип, декорированный атрибутом [DataContract]. Подобным же образом каждое поле, которое планируется использовать как часть предполагаемого контракта, должно быть помечено атрибутом [DataMember].

На заметку! В ранних версиях платформы .NET применение атрибутов [DataContract] и [DataMember] было обязательным для обеспечения корректного представления специальных типов данных. Это требование в Microsoft было ослаблено; формально вы не обязаны использовать указанные атрибуты в специальных типах данных; тем не менее, в .NET такой прием считается рекомендуемым подходом.

Использование веб-ориентированного шаблона проекта WCF Service

Следующая служба WCF позволит внешним вызывающим компонентам взаимодействовать с уровнем доступа к данным AutoLot, построенным в главе 22. Более того, финальная служба WCF будет создана с применением веб-ориентированного шаблона проекта WCF Service и размещена в IIS.

Для начала запустим Visual Studio (с правами администратора) и выберем пункт меню File⇒New⇒Web Site (Файл⇒Создать⇒Веб-сайт). Укажем тип проекта WCF Service (Служба WCF) и удостоверимся в том, что в раскрывающемся списке Web Location (Веб-местоположение) выбран вариант HTTP (это приведет к установке службы в IIS). Откроем доступ к службе из следующего URI:

`http://localhost/AutoLotWCFService`

На рис. 23.19 показаны настройки для проекта.

Щелкнем правой кнопкой мыши на решении (в окне Solution Explorer), выберем в контекстном меню пункт Add⇒Existing Project (Добавить⇒Существующий проект), укажем проект AutoLotDAL из главы 22 и повторим действия для AutoLotDAL.Models.

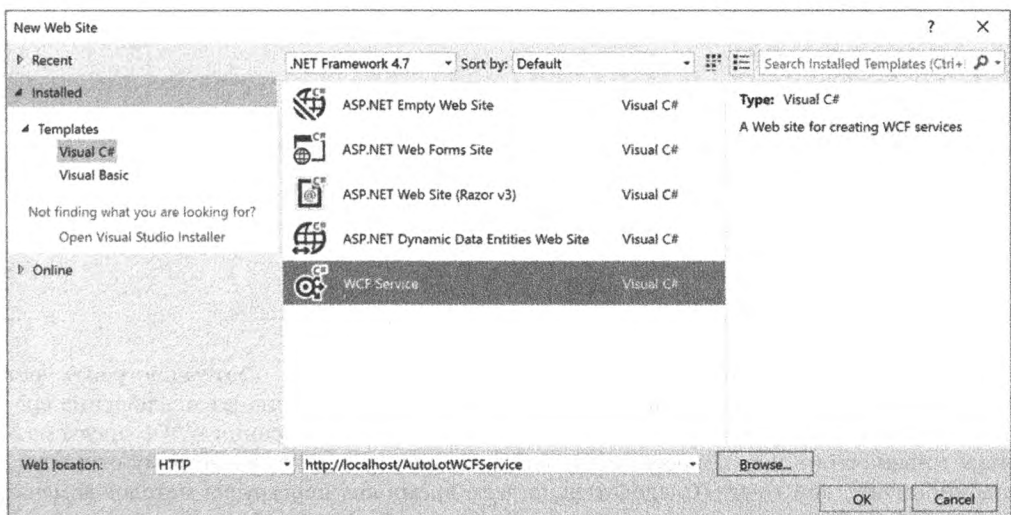


Рис. 23.19. Создание веб-ориентированной службы WCF

Можно было бы просто добавить ссылки на сборки `AutoLotDAL.dll` и `AutoLotDAL.Models.dll`, но добавление проекта делает возможной отладку уровня доступа к данным.

Внутри папки `App_Code` переименуем файл `IService.cs` в `IAutoLotService.cs` и определим в нем начальный контракт службы:

```
[ServiceContract]
public interface IAutoLotService
{
    [OperationContract]
    void InsertCar(string make, string color, string petname);

    [OperationContract]
    void InsertCar(InventoryRecord car);

    [OperationContract]
    List<InventoryRecord> GetInventory();
}
```

В интерфейсе `IAutoLotService` определены три метода, один из которых возвращает список объектов типа `InventoryRecord` (пока еще не созданного). Вы можете вспомнить, что метод `GetAll()` класса `InventoryRepo` возвращает список `List<T>` записей `Inventory`, из-за чего может удивить, почему метод `GetInventory()` службы возвращает список `List<T>` объектов `InventoryRecord`, т.е. другого класса.

Инфраструктура WCF построена так, чтобы соблюдать принципы SOA, в числе которых программирование на основе контрактов, а не реализаций. Таким образом, вместо возвращения внешнему вызывающему компоненту специфичного для .NET типа `Inventory` мы будем возвращать специальный контракт данных (`InventoryRecord`), корректно выраженный внутри документа WSDL в независимой манере.

Для создания специального контракта данных добавим в папку `App_Code` новый класс по имени `InventoryRecord.cs`. Определим контракт данных `InventoryRecord` следующим образом:

```
using System.Runtime.Serialization;

[DataContract]
public class InventoryRecord
{
    [DataMember]
    public int ID;

    [DataMember]
    public string Make;

    [DataMember]
    public string Color;

    [DataMember]
    public string PetName;
}
```

Если реализовать интерфейс `IAutoLotService` в таком виде, а затем построить хост и попытаться вызвать эти методы на стороне клиента, то возникнет исключение времени выполнения. Причина в том, что одно из требований описания WSDL предусматривает назначение *уникального имени* каждому методу, доступ к которому открыт из заданной конечной точки. Следовательно, в то время как перегрузка методов замечательно работает в контексте языка C#, текущие спецификации веб-служб не разрешают существование двух методов с одним и тем же именем `InsertCar()`.

К счастью, атрибут `[OperationContract]` поддерживает свойство `Name`, которое позволяет указать, каким образом метод C# будет представлен внутри описания WSDL. С учетом сказанного модифицируем вторую версию `InsertCar()`, как показано ниже:

```
public interface IAutoLotService
{
    ...
    [OperationContract(Name = "InsertCarWithDetails")]
    void InsertCar(InventoryRecord car);
}
```

Обновление пакетов NuGet и установка AutoMapper и EntityFramework

Прежде чем реализовывать контракт службы, понадобится добавить NuGet-пакет AutoMapper, который специализируется на преобразовании одного объекта в другой на основе имен и значений свойств.

На заметку! Пакет AutoMapper является исключительно мощным, но в данной главе мы рассмотрим его лишь поверхностно. Он будет задействован в нескольких главах далее в книге. Дополнительные сведения о том, что предлагает пакет AutoMapper, можно найти в документации по адресу <http://automapper.org/>.

Щелчком правой кнопкой мыши на проекте `AutoLotWCFService` и выберем в контекстном меню пункт `Manage NuGet Packages` (Управление пакетами NuGet). В открывшемся окне диспетчера пакетов NuGet может появиться сообщение о том, что в решении отсутствуют некоторые пакеты. В таком случае нужно щелкнуть кнопке `Restore` (Восстановить), расположенной после сообщения.

Щелкнем на кнопке `Browse` (Обзор) в верхней части окна и если пакет AutoMapper пока не отображается, то введем AutoMapper в поле поиска. Выберем AutoMapper и щелкнем на кнопке `Install` (Установить), как показано на рис. 23.20.

Наконец, установим с помощью диспетчера пакетов NuGet пакет EntityFramework версии 6.1.3.

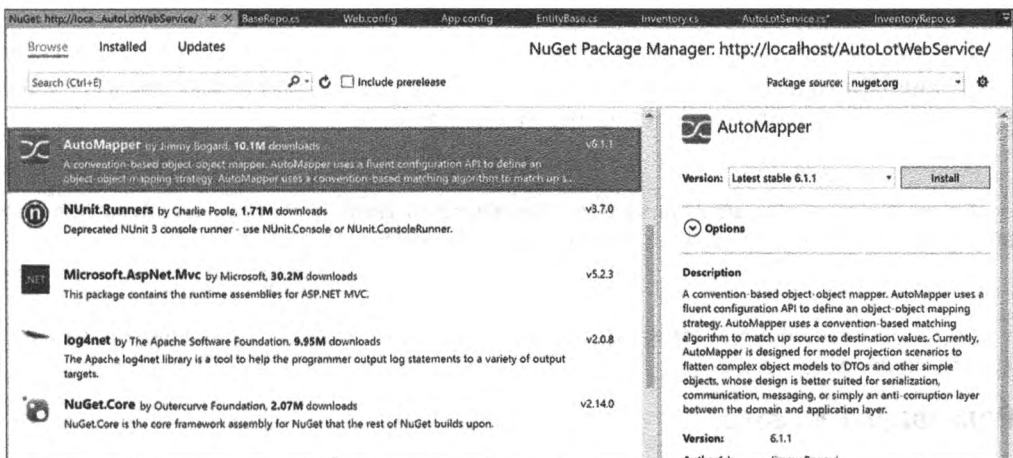


Рис. 23.20. Установка пакета AutoMapper

Реализация контракта службы

Переименуем файл `Service.cs` в `AutoLotService.cs`. Тип `AutoLotService` реализует интерфейс `IAutoLotService` следующим образом (в этот файл кода понадобится импортировать пространства имен `AutoLotDAL.Models`, `AutoLotDAL.Repos` и `AutoMapper`):

```
using AutoLotDAL.Models;
using AutoLotDAL.Repos;
using AutoMapper;

public class AutoLotService : IAutoLotService
{
    public AutoLotService()
    {
        Mapper.Initialize(cfg => cfg.CreateMap<Inventory, InventoryRecord>());
    }

    public void InsertCar(string make, string color, string petname)
    {
        var repo = new InventoryRepo();
        repo.Add(new Inventory { Color = color, Make = make, PetName = petname });
        repo.Dispose();
    }

    public void InsertCar(InventoryRecord car)
    {
        var repo = new InventoryRepo();
        repo.Add(new Inventory { Color = car.Color, Make = car.Make,
                                PetName = car.PetName });
        repo.Dispose();
    }

    public List<InventoryRecord> GetInventory()
    {
        var repo = new InventoryRepo();
        var records = repo.GetAll();
        var results = Mapper.Map<List<InventoryRecord>>(records);
        repo.Dispose();
        return results;
    }
}
```

В конструкторе посредством обобщенного метода `CreateMap()` мы сообщаем `AutoMapper` типы, которые будут преобразовываться:

```
Mapper.Initialize(cfg => cfg.CreateMap<Inventory, InventoryRecord>());
```

После преобразования списка `List<T>` объектов типа `Inventory` в список `List<T>` объектов типа `InventoryRecord` мы вызываем метод `Map()`:

```
var results = Mapper.Map<List<InventoryRecord>>(records);
```

Остаток кода совершенно прямолинеен; в нем просто вызываются методы `InventoryRepo`.

Роль файла `*.svc`

При создании веб-ориентированной службы WCF вы обнаружите, что проект содержит специфичный файл с расширением `*.svc`. Этот конкретный файл необходим лю-

бой службе WCF, размещаемой в IIS: в нем описано имя и местоположение реализации службы внутри точки установки. Поскольку имена начального файла и типов WCF были изменены, содержимое файла `Service.svc` потребуется модифицировать:

```
<%@ ServiceHost Language="C#" Debug="true"
    Service="AutoLotService" CodeBehind="~/App_Code/AutoLotService.cs" %>
```

Исследование файла `Web.config`

В файле `Web.config` службы WCF, созданной для HTTP, будет использоваться несколько упрощений WCF, обсужденных ранее в главе. Как подробно объясняется далее в книге при рассмотрении ASP.NET, предназначение файла `Web.config` аналогично файлу `*.config` исполняемой сборки; однако он также управляет рядом настроек, специфических для веб-среды. Например, обратите внимание, что службы MEX включены, и не нужно указывать специальный элемент `<endpoint>` вручную:

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <!-- Чтобы избежать раскрытия информации метаданных,
                установите следующее значение в false и удалите
                конечную точку метаданных перед развертыванием. -->
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
          <!-- Для получения деталей исключений при отказах в целях
                отладки установите следующее значение в true.
                Перед развертыванием установите его снова в false
                во избежание раскрытия информации об исключении. -->
          <serviceDebug includeExceptionDetailInFaults="false"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"
                              multipleSiteBindingsEnabled="true" />
  </system.serviceModel>
</configuration>
```

Добавление строки подключения в файл `Web.config`

Финальное изменение файла `Web.config` связано с добавлением строки подключения для базы данных AutoLot в SQL Server Express. В этой версии необходимо добавить элементы `User Id` и `Password`, а также удалить элемент `Integrated Security` либо изменить серию настроек в IIS и SQL Server. Вот как должна выглядеть строка подключения:

```
<connectionStrings>
  <add name="AutoLotConnection"
    connectionString="data source=.\SQLEXPRESS2016;
    initial catalog=AutoLot;MultipleActiveResultSets=True;
    App=EntityFramework;User Id=myUser;Password=myPassword"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

На заметку! Чтобы пример заработал, на странице Security (Безопасность) окна свойств сервера SQL Server должен быть выбран переключатель SQL Server and Windows Authentication mode (Режим аутентификации SQL Server и Windows). Если вы применяете загружаемые файлы примеров, тогда понадобится привести настройки User Id и Password в соответствие с имеющейся конфигурацией. Имейте в виду, что LocalDb и IIS вместе работают не особенно хорошо. Хотя и можно заставить их работать вместе, лучше использовать SQL Server Express или последующие редакции с WPF.

Тестирование службы

Теперь для тестирования службы можно построить клиент любого вида, включая передачу конечной точки файла *.svc приложению WcfTestClient.exe:

```
WcfTestClient http://localhost/AutoLotWCFSvc/Service.svc
```

Если необходимо создать специальное клиентское приложение, тогда можно применить диалоговое окно Add Service Reference, как делалось в примерах проектов MagicEightBallServiceClient и MathClient ранее в главе.

Исходный код. Проект AutoLotWCFSvc доступен в подкаталоге Chapter_23.

На этом исследование API-интерфейса WCF завершено. Разумеется, данная тема слишком обширна, чтобы ее удалось полностью раскрыть в одной ознакомительной главе, но благодаря изложенному здесь материалу вы можете продолжить изучение WCF самостоятельно. Дополнительные сведения об инфраструктуре WCF приведены в документации .NET Framework 4.7 SDK.

Резюме

В настоящей главе вы ознакомились с инфраструктурой Windows Communication Foundation (WCF), которая представляет основной API-интерфейс распределенного программирования на платформе .NET. Как здесь объяснялось, главной мотивацией создания WCF было предоставление унифицированной объектной модели, которая открывает доступ к нескольким собранным воедино (ранее несвязанным) API-интерфейсам распределенных вычислений. Служба WCF представляется путем указания адресов, привязок и контрактов (обозначаемых аббревиатурой ABC).

Вы также узнали, что типичное приложение WCF предусматривает использование трех взаимосвязанных сборок. В первой сборке определены контракты и типы службы, которые представляют ее функциональность. Эта сборка затем размещается в специальной исполняемой программе, в виртуальном каталоге IIS либо в Windows-службе. Наконец, клиентская сборка применяет сгенерированный файл кода, в котором определен тип прокси (и настройки внутри конфигурационного файла приложения) для взаимодействия с удаленным типом.

В главе также было показано, как использовать некоторые инструменты программирования для WCF, в том числе редактор SvcConfigEditor.exe (позволяющий модифицировать содержимое файлов *.config), приложение WcfTestClient.exe (для быстрого тестирования служб WCF) и разнообразные шаблоны проектов Visual Studio. Кроме того, вы узнали о ряде упрощений конфигурации, включая стандартные конечные точки и линии поведения.

ЧАСТЬ VII

Windows Presentation Foundation

В этой части

Глава 24. Введение в Windows Presentation Foundation и XAML

Глава 25. Элементы управления, компоновки, события и привязка данных в WPF

Глава 26. Службы визуализации графики WPF

Глава 27. Ресурсы, анимация, стили и шаблоны WPF

Глава 28. Уведомления, проверка достоверности, команды и MVVM

ГЛАВА 24

Введение в Windows Presentation Foundation и XAML

Когда была выпущена версия 1.0 платформы .NET, программисты, нуждающиеся в построении графических настольных приложений, использовали два API-интерфейса под названиями Windows Forms и GDI+, упакованные преимущественно в сборках `System.Windows.Forms.dll` и `System.Drawing.dll`. Наряду с тем, что Windows Forms и GDI+ — все еще жизнеспособные API-интерфейсы для построения традиционных настольных графических пользовательских интерфейсов, начиная с версии .NET 3.0, поставляется альтернативный API-интерфейс с таким же предназначением, который называется Windows Presentation Foundation (WPF).

В начале этой вводной главы, посвященной WPF, вы ознакомитесь с мотивацией, лежащей в основе новой инфраструктуры для построения графических пользовательских интерфейсов, что поможет увидеть отличия между моделями программирования Windows Forms/GDI+ и WPF. Затем анализируется роль ряда важных классов, включая `Application`, `Window`, `ContentControl`, `Control`, `UIElement` и `FrameworkElement`.

В настоящей главе будет представлена грамматика на основе XML, которая называется *расширяемым языком разметки приложений* (Extensible Application Markup Language — XAML). Вы изучите синтаксис и семантику XAML (в том числе синтаксис присоединяемых свойств, роль преобразователей типов и расширений разметки).

Глава завершается исследованием визуальных конструкторов WPF, встроенных в Visual Studio, за счет построения вашего первого приложения WPF. Вы научитесь перехватывать действия клавиатуры и мыши, определять данные уровня приложения и выполнять другие распространенные задачи WPF.

Мотивация, лежащая в основе WPF

На протяжении многих лет в Microsoft создавали инструменты для построения графических пользовательских интерфейсов (для низкоуровневой разработки на C/C++/Windows API, VB6, MFC и т.д.) настольных приложений. Каждый инструмент предлагает кодовую базу для представления основных аспектов приложения с графическим пользовательским интерфейсом, включая главные окна, диалоговые окна, элементы управления, системы меню и другие базовые аспекты. После начального выпуска платформы .NET инфраструктура Windows Forms быстро стала предпочтительным подходом к разработке пользовательских интерфейсов благодаря своей простой, но очень мощной объектной модели.

Хотя с помощью Windows Forms было успешно создано множество полноценных настольных приложений, дело в том, что данная программная модель несколько *асимметрична*. Попросту говоря, сборки `System.Windows.Forms.dll` и `System.Drawing.dll` не предоставляют прямую поддержку для многих дополнительных технологий, требуемых при построении полнофункционального настольного приложения. Чтобы проиллюстрировать сказанное, рассмотрим узкоспециализированную разработку графических пользовательских интерфейсов до выпуска WPF (табл. 24.1).

Таблица 24.1. Решения, предшествующие WPF, для обеспечения желаемой функциональности

Желаемая функциональность	Технология
Построение окон с элементами управления	Windows Forms
Поддержка двумерной графики	GDI+ (<code>System.Drawing.dll</code>)
Поддержка трехмерной графики	API-интерфейсы DirectX
Поддержка потокового видео	API-интерфейсы Windows Media Player
Поддержка документов нефиксированного формата	Программное манипулирование файлами PDF

Как видите, разработчик, использующий Windows Forms, вынужден заимствовать типы из нескольких несвязанных API-интерфейсов и объектных моделей. Несмотря на то что применение всех разрозненных API-интерфейсов синтаксически похоже (в конце концов, это просто код C#), каждая технология требует радикально иного мышления. Например, навыки, необходимые для создания трехмерной анимации с использованием DirectX, совершенно отличаются от тех, что нужны для привязки данных к экранной сетке. Конечно, программисту Windows Forms чрезвычайно трудно в равной степени хорошо овладеть природой каждого API-интерфейса.

Унификация несходных API-интерфейсов

Инфраструктура WPF специально создавалась для объединения ранее несвязанных задач программирования в одну унифицированную объектную модель. Таким образом, при разработке трехмерной анимации больше не возникает необходимость в ручном кодировании с применением API-интерфейса DirectX (хотя это можно делать), поскольку нужная функциональность уже встроена в WPF. Чтобы продемонстрировать, насколько все стало яснее, в табл. 24.2 представлена модель разработки настольных приложений, введенная в .NET 3.0.

Таблица 24.2. Решения .NET 3.0 для обеспечения желаемой функциональности

Желаемая функциональность	Технология
Построение форм с элементами управления	WPF
Поддержка двумерной графики	WPF
Поддержка трехмерной графики	WPF
Поддержка потокового видео	WPF
Поддержка документов нефиксированного формата	WPF

Очевидное преимущество здесь в том, что программисты .NET теперь имеют единственный *симметричный* API-интерфейс для всех распространенных потребностей, возникающих во время разработки графических пользовательских интерфейсов настольных приложений. Освоившись с функциональностью основных сборок WPF и грамматикой XAML, вы будете приятно удивлены, насколько быстро с их помощью можно создавать сложные пользовательские интерфейсы.

Обеспечение разделения обязанностей через XAML

Возможно, одно из наиболее значительных преимуществ заключается в том, что инфраструктура WPF предлагает способ аккуратного отделения внешнего вида и поведения приложения с графическим пользовательским интерфейсом от программной логики, которая им управляет. Используя язык XAML, пользовательский интерфейс приложения можно определять через *разметку XML*. Такая разметка (в идеале генерируемая с помощью инструментов вроде Microsoft Visual Studio или Microsoft Expression Blend) затем может быть подключена к связанному файлу кода для обеспечения внутренней части функциональности программы.

На заметку! Язык XAML не ограничивается приложениями WPF. Любое приложение может применять XAML для описания дерева объектов .NET, даже если они не имеют никакого отношения к видимому пользовательскому интерфейсу.

По мере погружения в WPF вас может удивить, насколько высокую гибкость обеспечивает эта “настольная разметка”. Язык XAML позволяет определять в разметке не только простые элементы пользовательского интерфейса (кнопки, таблицы, окна со списками и т.д.), но также интерактивную двумерную и трехмерную графику, анимацию, логику привязки данных и функциональность мультимедиа (наподобие воспроизведения видео).

Кроме того, XAML облегчает настройку визуализации элемента управления. Например, определение круглой кнопки, на которой выполняется анимация логотипа компании, требует всего нескольких строк разметки. Как показано в главе 27, элементы управления WPF могут быть модифицированы посредством стилей и шаблонов, которые позволяют изменять весь внешний вид приложения с минимальными усилиями. В отличие от разработки с помощью Windows Forms единственной веской причиной для построения специального элемента управления WPF с нуля является необходимость в изменении *поведения* элемента управления (например, добавление специальных методов, свойств или событий либо создание подкласса существующего элемента управления с целью переопределения виртуальных членов). Если нужно просто изменить *внешний вид* элемента управления (как в случае с круглой анимированной кнопкой), то это можно делать полностью через разметку.

Обеспечение оптимизированной модели визуализации

Наборы инструментов для построения графических пользовательских интерфейсов, такие как Windows Forms, MFC или VB6, выполняют все запросы графической визуализации (включая визуализацию элементов управления вроде кнопок и окон со списком) с применением низкоуровневого API-интерфейса на основе C (GDI), который в течение многих лет был частью Windows. Интерфейс GDI обеспечивает адекватную производительность для типовых бизнес-приложений или простых графических программ; однако если приложению с пользовательским интерфейсом нужна была высокопроизводительная графика, то приходилось обращаться к услугам DirectX.

Программная модель WPF полностью отличается тем, что при визуализации графических данных GDI не используется. Все операции визуализации (двумерная и трехмерная графика, анимация, визуализация элементов управления и т.д.) теперь работают с API-интерфейсом DirectX. Очевидная выгода такого подхода в том, что приложения WPF будут автоматически получать преимущества аппаратной и программной оптимизации. Вдобавок приложения WPF могут задействовать очень развитые графические службы (эффекты размытия, сглаживания, прозрачности и т.п.) без сложностей, присущих программированию напрямую с применением API-интерфейса DirectX.

На заметку! Хотя WPF переносит все запросы визуализации на уровень DirectX, нельзя утверждать, что приложение WPF будет работать настолько же быстро, как приложение, построенное с использованием неуправляемого языка C++ и DirectX. Несмотря на значительные усовершенствования, внесенные в инфраструктуру WPF в версии .NET 4.7, если вы намереваетесь строить настольное приложение, которое требует максимально возможной скорости выполнения (вроде трехмерной игры), то неуправляемый C++ и DirectX по-прежнему являются наилучшим выбором.

Упрощение программирования сложных пользовательских интерфейсов

Чтобы подвести итоги сказанному до сих пор: Windows Presentation Foundation (WPF) — это API-интерфейс, предназначенный для построения настольных приложений, который интегрирует разнообразные настольные API-интерфейсы в единую объектную модель и обеспечивает четкое разделение обязанностей через XAML. В дополнение к указанным важнейшим моментам приложения WPF также выигрывают от простого способа интеграции со службами, что исторически было довольно сложным. Ниже кратко перечислены основные функциональные возможности WPF.

- Множество диспетчеров компоновки (гораздо больше, чем в Windows Forms) для обеспечения исключительно гибкого контроля над размещением и изменением позиций содержимого.
- Применение расширенного механизма привязки данных для связывания содержимого с элементами пользовательского интерфейса разнообразными способами.
- Встроенный механизм стилей, который позволяет определять “темы” для приложения WPF.
- Использование векторной графики, поддерживающей автоматическое изменение размеров содержимого с целью соответствия размерам и разрешающей способности экрана, который отображает пользовательский интерфейс приложения.
- Поддержка двумерной и трехмерной графики, анимации, а также воспроизведения видео и аудио.
- Развитый типографский API-интерфейс, который поддерживает документы XML Paper Specification (XPS), фиксированные документы (WYSIWYG), документы нефиксированного формата и аннотации в документах (например, API-интерфейс Sticky Notes).
- Поддержка взаимодействия с унаследованными моделями графических пользовательских интерфейсов (такими как Windows Forms, ActiveX и HWND-дескрипторы Win32). Например, в приложение WPF можно встраивать специальные элементы управления Windows Forms и наоборот.

Теперь, получив определенное представление о том, что инфраструктура WPF приносит в платформу, давайте рассмотрим разнообразные типы приложений, которые могут быть созданы с применением данного API-интерфейса. Многие из перечисленных выше возможностей будут подробно исследованы в последующих главах.

Исследование сборок WPF

В конечном итоге инфраструктура WPF — не многим более чем коллекция типов, встроенных в сборки .NET. В табл. 24.3 описаны основные сборки, используемые при разработке приложений WPF, на каждую из которых должна быть добавлена ссылка, когда создается новый проект. Как и следовало ожидать, проекты WPF в Visual Studio ссылаются на эти обязательные сборки автоматически.

Таблица 24.3. Основные сборки WPF

Сборка	Описание
PresentationCore.dll	В этой сборке определены многочисленные пространства имен, которые образуют фундамент уровня графического пользовательского интерфейса в WPF. Например, она включает поддержку API-интерфейса WPF Ink, примитивы анимации и множество типов графической визуализации
PresentationFramework.dll	В этой сборке содержится большинство элементов управления WPF, классы Application и Window, поддержка интерактивной двумерной графики и многочисленные типы, применяемые для привязки данных
System.Xaml.dll	Эта сборка предоставляет пространства имен, которые позволяют программно взаимодействовать с документами XAML во время выполнения. В общем и целом она полезна только при разработке инструментов поддержки WPF или когда нужен абсолютный контроль над разметкой XAML во время выполнения
WindowsBase.dll	В этой сборке определены типы, которые формируют инфраструктуру API-интерфейса WPF, включая типы потоков WPF, типы, связанные с безопасностью, разнообразные преобразователи типов и поддержку свойств зависимости и маршрутизируемых событий (описанных в главе 25)

В совокупности указанные в табл. 24.3 четыре сборки определяют несколько новых пространств имен и сотни новых классов, интерфейсов, структур, перечислений и делегатов .NET. В то время как подробные сведения следует искать в документации .NET Framework 4.7 SDK, в табл. 24.4 описана роль некоторых (но далеко не всех) важных пространств имен.

Таблица 24.4. Основные пространства имен WPF

Пространство имен	Описание
System.Windows	Корневое пространство имен WPF. Здесь находятся основные классы (такие как Application и Window), которые требуются в любом проекте настольного приложения WPF
System.Windows.Controls	Содержит все ожидаемые графические элементы (виджеты) WPF, включая типы для построения систем меню, всплывающие подсказки и многочисленные диспетчеры компоновки

Пространство имен	Описание
<code>System.Windows.Data</code>	Содержит типы для работы с механизмом привязки данных WPF, а также для поддержки шаблонов привязки данных
<code>System.Windows.Documents</code>	Содержит типы для работы с API-интерфейсом документов, который позволяет интегрировать в приложения WPF функциональность в стиле PDF через протокол XML Paper Specification (XPS)
<code>System.Windows.Ink</code>	Предоставляет поддержку Ink API — интерфейса, который позволяет получать ввод от пера или мыши, реагировать на входные жесты и т.д. Этот API-интерфейс полезен при программировании для Tablet PC, но может использоваться также в любых приложениях WPF
<code>System.Windows.Markup</code>	Здесь определено множество типов, обеспечивающих программный анализ и обработку разметки XAML (и эквивалентного двоичного формата BAML)
<code>System.Windows.Media</code>	Корневое пространство имен для нескольких связанных с мультимедиа пространств имен, внутри которых определены типы для работы с анимацией, визуализацией трехмерной графики, визуализацией текста и прочими мультимедийными примитивами
<code>System.Windows.Navigation</code>	Предоставляет типы для обеспечения логики навигации, применяемой браузерными приложениями XAML (XAML browser application — XBAP), а также настольными приложениями, которые требуют страничной модели навигации
<code>System.Windows.Shapes</code>	Здесь определены классы, которые позволяют визуализировать двумерную графику, автоматически реагирующую на ввод с помощью мыши

В начале путешествия по программной модели WPF мы исследуем два члена пространства имен `System.Windows`, которые являются общими при традиционной разработке любого настольного приложения: `Application` и `Window`.

На заметку! Если вы создавали пользовательские интерфейсы для настольных приложений с использованием API-интерфейса `Windows Forms`, то имейте в виду, что сборки `System.Windows.Forms.*` и `System.Drawing.*` никак не связаны с WPF. Они относятся к первоначальному инструментальному набору .NET для построения графических пользовательских интерфейсов, т.е. `Windows Forms/GDI+`.

Роль класса `Application`

Класс `System.Windows.Application` представляет глобальный экземпляр выполняющегося приложения WPF. В нем имеется метод `Run()` (для запуска приложения), комплект событий, которые можно обрабатывать для взаимодействия с приложением на протяжении его времени жизни (наподобие `Startup` и `Exit`), и набор членов, специфичных для браузерных приложений XAML (таких как события, инициируемые во время навигации пользователя по страницам). В табл. 24.5 описаны основные свойства класса `Application`.

Таблица 24.5. Основные свойства класса *Application*

Свойство	Описание
Current	Это статическое свойство позволяет получать доступ к работающему объекту <i>Application</i> из любого места кода. Может быть полезно, когда обычному или диалоговому окну необходим доступ к создавшему его объекту <i>Application</i> , как правило, для взаимодействия с переменными или функциональностью уровня приложения
MainWindow	Это свойство позволяет программно получать или устанавливать главное окно приложения
Properties	Это свойство позволяет устанавливать и получать данные, доступные через все аспекты приложения WPF (окна, диалоговые окна и т.д.)
StartupUri	Это свойство получает или устанавливает URI, который указывает окно или страницу для автоматического открытия при запуске приложения
Windows	Это свойство возвращает объект тип <i>WindowCollection</i> , предоставляющий доступ ко всем окнам, которые созданы в потоке, создавшем объект <i>Application</i> . Может быть удобным, когда нужно пройти по всем открытым окнам приложения и изменить их состояние (скажем, свернуть все окна)

Построение класса *Application*

В любом приложении WPF понадобится определить класс, расширяющий *Application*. Внутри такого класса будет определена точка входа программы (метод *Main()*), которая создает экземпляр данного подкласса и обычно обрабатывает события *Startup* и *Exit* (при необходимости). Вот пример:

```
// Определить глобальный объект приложения для этой программы WPF.
class MyApp : Application
{
    [STAThread]
    static void Main(string[] args)
    {
        // Создать объект приложения.
        MyApp app = new MyApp();

        // Зарегистрировать события Startup/Exit.
        app.Startup += (s, e) => { /* Запуск приложения */ };
        app.Exit += (s, e) => { /* Завершение приложения */ };
    }
}
```

В обработчике события *Startup* чаще всего будут обрабатываться входные аргументы командной строки и запускаться главное окно программы. Как и следовало ожидать, обработчик события *Exit* представляет собой место, куда можно поместить любую необходимую логику завершения программы (например, сохранение пользовательских предпочтений).

На заметку! Метод *Main()* приложения WPF должен быть снабжен атрибутом *[STAThread]*, который гарантирует, что любые унаследованные объекты COM, используемые приложением, являются безопасными в отношении потоков. Если не аннотировать метод *Main()* подобным образом, тогда во время выполнения возникнет исключение.

Перечисление элементов коллекции *Windows*

Еще одним интересным свойством класса `Application` является `Windows`, обеспечивающее доступ к коллекции, которая представляет все окна, загруженные в память для текущего приложения WPF. Вспомните, что создаваемые новые объекты `Window` автоматически добавляются в коллекцию `Application.Windows`. Ниже приведен пример метода, который сворачивает все окна приложения (возможно в ответ на нажатие определенного сочетания клавиш или выбор пункта меню конечным пользователем):

```
static void MinimizeAllWindows()
{
    foreach (Window wnd in Application.Current.Windows)
    {
        wnd.WindowState = WindowState.Minimized;
    }
}
```

Вскоре будет построено несколько приложений WPF, а пока давайте выясним основную функциональность типа `Window` и изучим несколько важных базовых классов WPF.

Роль класса `Window`

Класс `System.Windows.Window` (из сборки `PresentationFramework.dll`) представляет одиночное окно, которым владеет производный от `Application` класс, включая все отображаемые главным окном диалоговые окна. Тип `Window` вполне ожидаемо имеет несколько родительских классов, каждый из которых привносит дополнительную функциональность. На рис. 24.1 показана цепочка наследования (и реализуемые интерфейсы) для класса `System.Windows.Window`, как она выглядит в браузере объектов `Visual Studio`.

По мере чтения этой и последующих глав вы начнете понимать функциональность, предлагаемую многими базовыми классами WPF. Далее представлен краткий обзор функциональности каждого базового класса (полные сведения ищите в документации .NET Framework 4.7 SDK).

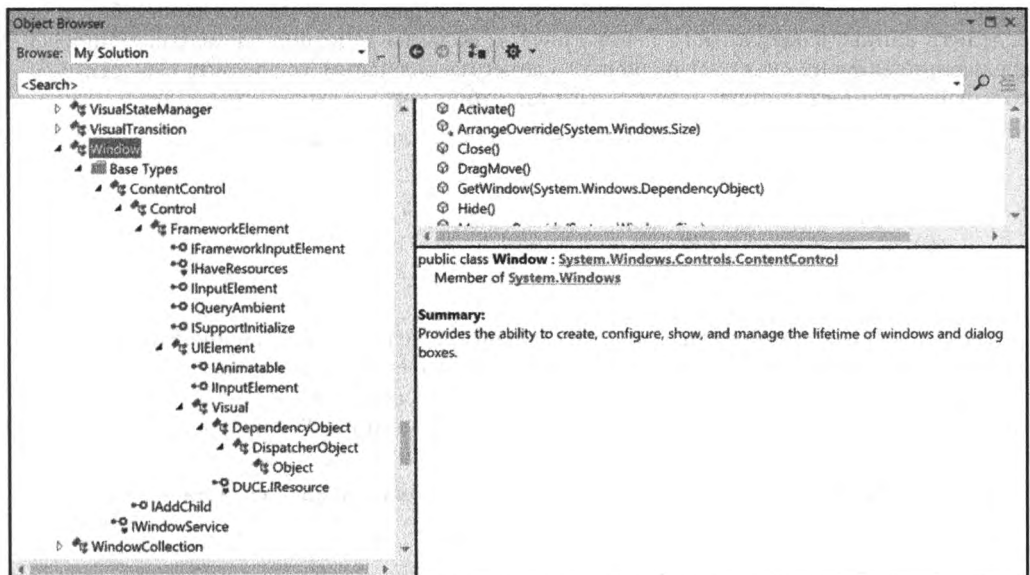


Рис. 24.1. Иерархия наследования класса `Window`

Роль класса `System.Windows.Controls.ContentControl`

Непосредственным родительским классом `Window` является класс `ContentControl`, который вполне можно считать наиболее впечатляющим из всех классов WPF. Базовый класс `ContentControl` снабжает производные типы способностью размещать в себе одиночный фрагмент *содержимого*, который, выражаясь упрощенно, относится к визуальным данным, помещенным внутрь области элемента управления через свойство `Content`. Модель содержимого WPF позволяет очень легко настраивать базовый вид и поведение элемента управления `ContentControl`.

Например, когда речь идет о типичном “кнопочном” элементе управления, то обычно предполагается, что его содержимым будет простой строковый литерал (OK, Cancel, Abort и т.д.). Если для описания элемента управления WPF применяется XAML, а значение, которое необходимо присвоить свойству `Content`, может быть выражено в виде простой строки, тогда вот как установить свойство `Content` внутри открывающего определения элемента:

```
<!-- Установка значения Content в открывающем элементе -->
<Button Height="80" Width="100" Content="OK"/>
```

На заметку! Свойство `Content` можно также устанавливать в коде C#, что позволяет изменять внутренности элемента управления во время выполнения.

Однако содержимое может быть практически любым. Например, пусть нужна “кнопка”, которая содержит в себе что-то более интересное, нежели простую строку — возможно специальную графику или текстовый фрагмент. В других инфраструктурах для построения пользовательских интерфейсов, таких как Windows Forms, потребовалось бы создать специальный элемент управления, что могло повлечь за собой написание значительного объема кода и сопровождение полностью нового класса. Благодаря модели содержимого WPF необходимость в этом отпадает.

Когда свойству `Content` должно быть присвоено значение, которое невозможно выразить в виде простого массива символов, его нельзя присвоить с использованием атрибута в открывающем определении элемента управления. Взамен понадобится определить данные содержимого неявно внутри области действия элемента. Например, следующий элемент `<Button>` включает в качестве содержимого элемент `<StackPanel>`, который сам имеет уникальные данные (а именно — `<Ellipse>` и `<Label>`):

```
<!-- Неявная установка для свойства Content сложных данных -->
<Button Height="80" Width="100">
  <StackPanel>
    <Ellipse Fill="Red" Width="25" Height="25"/>
    <Label Content="OK!"/>
  </StackPanel>
</Button>
```

Для установки сложного содержимого можно также применять синтаксис “свойство-элемент” языка XAML. Взгляните на показанное далее функционально эквивалентное определение `<Button>`, которое явно устанавливает свойство `Content` с помощью синтаксиса “свойство-элемент” (дополнительная информация о XAML будет дана позже в главе, так что пока не обращайтесь внимания на детали):

```
<!-- Установка свойства Content с использованием синтаксиса "свойство-элемент" -->
<Button Height="80" Width="100">
  <Button.Content>
    <StackPanel>
      <Ellipse Fill="Red" Width="25" Height="25"/>
```

```

<Label Content = "OK!" />
</StackPanel>
</Button.Content>
</Button>

```

Имейте в виду, что не каждый элемент WPF является производным от класса `ContentControl`, поэтому не все элементы поддерживают такую уникальную модель содержимого (хотя большинство поддерживает). Кроме того, некоторые элементы управления WPF вносят несколько усовершенствований в только что рассмотренную базовую модель содержимого. В главе 25 роль содержимого WPF раскрывается более подробно.

Роль класса `System.Windows.Controls.Control`

В отличие от `ContentControl` все элементы управления WPF разделяют в качестве общего родительского класса базовый класс `Control`. Он предоставляет многочисленные члены, которые необходимы для обеспечения основной функциональности пользовательского интерфейса. Например, в классе `Control` определены свойства для установки размеров элемента управления, прозрачности, порядка обхода по нажатию клавиши `<Tab>`, отображаемого курсора, цвета фона и т.д. Более того, данный родительский класс предлагает поддержку *шаблонных служб*. Как объясняется в главе 27, элементы управления WPF могут полностью изменять способ визуализации своего внешнего вида, используя шаблоны и стили. В табл. 24.6 кратко описаны основные члены типа `Control`, сгруппированные по связанной функциональности.

Таблица 24.6. Основные члены класса `Control`

Член	Описание
<code>Background</code> , <code>Foreground</code> , <code>BorderBrush</code> , <code>BorderThickness</code> , <code>Padding</code> , <code>HorizontalAlignment</code> , <code>VerticalContentAlignment</code>	Эти свойства позволяют устанавливать базовые настройки, касающиеся того, как элемент управления будет визуализироваться и позиционироваться
<code>FontFamily</code> , <code>FontSize</code> , <code>FontStretch</code> , <code>FontWeight</code>	Эти свойства управляют разнообразными настройками шрифтов
<code>IsTabStop</code> , <code>TabIndex</code>	Эти свойства применяются для установки порядка обхода элементов управления в окне по нажатию клавиши <code><Tab></code>
<code>MouseDoubleClick</code> , <code>PreviewMouseDoubleClick</code>	Эти события обрабатывают действие двойного щелчка на виджете
<code>Template</code>	Это свойство позволяет получать и устанавливать шаблон элемента управления, который может быть использован для изменения вывода визуализации виджета

Роль класса `System.Windows.FrameworkElement`

Базовый класс `FrameworkElement` предоставляет несколько членов, которые применяются повсюду в инфраструктуре WPF, в том числе для поддержки раскадровки (в целях анимации) и привязки данных, а также возможности именования членов (через свойство `Name`), получения любых ресурсов, определенных производным типом, и установки общих измерений производного типа. Основные члены класса `FrameworkElement` кратко описаны в табл. 24.7.

Таблица 24.7. Основные члены класса FrameworkElement

Член	Описание
ActualHeight, ActualWidth, MaxHeight, MaxWidth, MinHeight, MinWidth, Height, Width	Эти свойства управляют размерами производного типа
ContextMenu	Это свойство получает или устанавливает всплывающее меню, ассоциированное с производным типом
Cursor	Это свойство получает или устанавливает курсор мыши, ассоциированный с производным типом
HorizontalAlignment, VerticalAlignment	Это свойство управляет позиционированием типа внутри контейнера (такого как панель или окно со списком)
Name	Это свойство позволяет назначать имя типу, чтобы обращаться к его функциональности в файле кода
Resources	Это свойство предоставляет доступ к любым ресурсам, которые определены типом (система ресурсов WPF объясняется в главе 27)
ToolTip	Это свойство получает или устанавливает всплывающую подсказку, ассоциированную с производным типом

Роль класса System.Windows.UIElement

Из всех типов в цепочке наследования класса Window наибольший объем функциональности обеспечивает базовый класс UIElement. Его основная задача — предоставить производному типу многочисленные события, чтобы он мог получать фокус и обрабатывать входные запросы. Например, в классе UIElement предусмотрено множество событий для обслуживания операций перетаскивания, перемещений курсора мыши, клавиатурного ввода, а также ввода с помощью пера (для Pocket PC и Tablet PC).

Модель событий WPF будет подробно описана в главе 25; тем не менее, многие основные события будут выглядеть вполне знакомыми (MouseMove, MouseDown, MouseEnter, MouseLeave, KeyUp и т.д.). В дополнение к десяткам событий родительский класс UIElement предлагает свойства, предназначенные для управления фокусом, состоянием доступности, видимостью и логикой проверки попаданий (табл. 24.8).

Таблица 24.8. Основные члены класса UIElement

Член	Описание
Focusable, IsFocused	Эти свойства позволяют устанавливать фокус на заданный производный тип
IsEnabled	Это свойство позволяет управлять доступностью заданного производного типа
IsMouseDirectlyOver, IsMouseOver	Эти свойства предоставляют простой способ выполнения логики проверки попадания
IsVisible, Visibility	Эти свойства позволяют работать с настройкой видимости производного типа
RenderTransform	Это свойство позволяет устанавливать трансформацию, которая будет использоваться при визуализации производного типа

Роль класса `System.Windows.Media.Visual`

Класс `Visual` предлагает основную поддержку визуализации в WPF, которая включает проверку попадания для графических данных, координатную трансформацию и вычисление ограничивающих прямоугольников. В действительности при рисовании данных на экране класс `Visual` взаимодействует с подсистемой `DirectX`. Как будет показано в главе 26, инфраструктура WPF поддерживает три возможных способа визуализации графических данных, каждый из которых отличается в плане функциональности и производительности. Применение типа `Visual` (и его потомков вроде `DrawingVisual`) является наиболее легковесным путем визуализации графических данных, но также подразумевает написание вручную большого объема кода для учета всех требуемых служб. Более подробно об этом пойдет речь в главе 26.

Роль класса `System.Windows.DependencyObject`

Инфраструктура WPF поддерживает отдельную разновидность свойств .NET под названием *свойства зависимости*. Выражаясь упрощенно, данный стиль свойств предоставляет дополнительный код, чтобы позволить свойству реагировать на определенные технологии WPF, такие как стили, привязка данных, анимация и т.д. Чтобы тип поддерживал подобную схему свойств, он должен быть производным от базового класса `DependencyObject`. Несмотря на то что свойства зависимости являются ключевым аспектом разработки WPF, большую часть времени их детали скрыты от глаз. В главе 25 мы рассмотрим свойства зависимости более подробно.

Роль класса

`System.Windows.Threading.DispatcherObject`

Последним базовым классом для типа `Window` (помимо `System.Object`, который здесь не требует дополнительных пояснений) является `DispatcherObject`. В нем определено одно интересное свойство `Dispatcher`, которое возвращает ассоциированный объект `System.Windows.Threading.Dispatcher`. Класс `Dispatcher` — это точка входа в очередь событий приложения WPF, и он предоставляет базовые конструкции для организации параллелизма и многопоточности.

Синтаксис XAML для WPF

Приложения WPF производственного уровня обычно будут использовать отдельные инструменты для генерации необходимой разметки XAML. Как бы ни были удобны такие инструменты, важно понимать общую структуру языка XAML. Для содействия процессу изучения доступен популярный (и бесплатный) инструмент, который позволяет легко экспериментировать с XAML.

Введение в `Kaxaml`

Когда вы только приступаете к изучению грамматики XAML, может оказаться удобным в применении бесплатный инструмент под названием *Kaxaml*. Этот популярный редактор/анализатор XAML доступен в каталоге `kaxaml` загружаемого кода примеров.

На заметку! Во многих предшествующих изданиях книги мы направляли читателей на веб-сайт www.kaxaml.com, но, к сожалению, он прекратил свою работу. Мы предоставили копию пакета `.msi` в рамках загружаемых материалов для данной книги и также создали ответвление в своем хранилище GitHub ([www.github.com/skimedic/kaxaml](https://github.com/skimedic/kaxaml)), чтобы гарантировать его существование. Мы благодарны создателям за великолепный инструмент *Kaxaml*, который помог многочисленным разработчикам изучить XAML.

Редактор `Kaхамl` полезен тем, что не имеет никакого понятия об исходном коде C#, обработчиках ошибок или логике реализации. Он предлагает намного более прямолинейный способ тестирования фрагментов XAML, чем использование полноценного шаблона проекта WPF в Visual Studio. К тому же `Kaхамl` обладает набором интегрированных инструментов, в том числе средством выбора цвета, диспетчером фрагментов XAML и даже средством "очистки XAML", которое форматирует разметку XAML на основе заданных настроек. Открыв `Kaхамl` в первый раз, вы найдете в нем простую разметку для элемента управления `<Page>`:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>

  </Grid>
</Page>
```

Подобно объекту `Window` объект `Page` содержит разнообразные диспетчеры компоновки и элементы управления. Тем не менее, в отличие от `Window` объекты `Page` не могут запускаться как отдельные сущности. Взамен они должны помещаться внутрь подходящего хоста, такого как `NavigationWindow` или `Frame`. Хорошая новость в том, что в элементах `<Page>` и `<Window>` можно вводить идентичную разметку.

На заметку! Если в окне разметки `Kaхамl` заменить элементы `<Page>` и `</Page>` элементами `<Window>` и `</Window>`, тогда можно нажать клавишу `<F5>` и отобразить на экране новое окно.

В качестве начального теста введем следующую разметку в панели XAML, находящейся в нижней части окна `Kaхамl`:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Кнопка со специальным содержанием -->
    <Button Height="100" Width="100">
      <Ellipse Fill="Green" Height="50" Width="50"/>
    </Button>
  </Grid>
</Page>
```

В верхней части окна `Kaхамl` появится визуализированная страница (рис. 24.2).

Во время работы с `Kaхамl` помните, что данный инструмент не позволяет писать разметку, которая влечет за собой любую компиляцию кода (но разрешено использовать `x:Name`). Сюда входит определение атрибута `x:Class` (для указания файла кода), ввод имен обработчиков событий в разметке или применение любых ключевых слов XAML, которые также предусматривают компиляцию кода (вроде `FieldModifier` или `ClassModifier`). Попытка поступить так приводит к ошибке разметки.

Пространства имен XML и "ключевые слова" XAML

Корневой элемент XAML-документа WPF (такой как `<Window>`, `<Page>`, `<UserControl>` или `<Application>`) почти всегда будет ссылаться на два заранее определенные пространства имен XML:

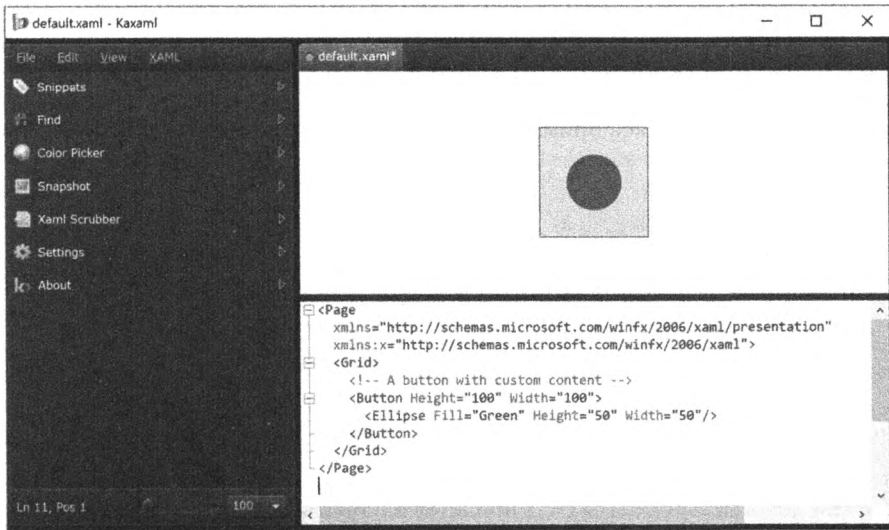


Рис. 24.2. Редактор KaXaml является удобным (и бесплатным) инструментом, применяемым при изучении грамматики XAML

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- A Button with custom content -->
    <Button Height="100" Width="100">
      <Ellipse Fill="Green" Height="50" Width="50"/>
    </Button>
  </Grid>
</Page>
```

Первое пространство имен XML, `http://schemas.microsoft.com/winfx/2006/xaml/presentation`, отображает множество связанных с WPF пространств имен .NET для использования текущим файлом *.xaml (`System.Windows`, `System.Windows.Controls`, `System.Windows.Data`, `System.Windows.Ink`, `System.Windows.Media`, `System.Windows.Navigation` и т.д.).

Это отображение "один ко многим" в действительности жестко закодировано внутри сборки WPF (`WindowsBase.dll`, `PresentationCore.dll` и `PresentationFramework.dll`) с применением атрибута [`XmlnsDefinition`] уровня сборки. Например, если открыть браузер объектов Visual Studio и выбрать сборку `PresentationCore.dll`, то можно увидеть списки, подобные показанному ниже, в котором импортируется пространство имен `System.Windows`:

```
[assembly: XmlnsDefinition(
  "http://schemas.microsoft.com/winfx/2006/xaml/presentation",
  "System.Windows")]
```

Второе пространство имен XML, `http://schemas.microsoft.com/winfx/2006/xaml`, используется для добавления специфичных для XAML "ключевых слов" (термин выбран за неимением лучшего), а также пространства имен `System.Windows.Markup`:

```
[assembly: XmlnsDefinition("http://schemas.microsoft.com/winfx/2006/xaml",
  "System.Windows.Markup")]
```

Одно из правил любого корректно сформированного документа XML (не забывайте, что грамматика XAML основана на XML) состоит в том, что открывающий корневой элемент назначает одно пространство имен XML в качестве *первичного пространства*

имен, которое обычно представляет собой пространство имен, содержащее самые часто применяемые элементы. Если корневой элемент требует включения дополнительных вторичных пространств имен (как видно здесь), то они должны быть определены с использованием уникального префикса (чтобы устранить возможные конфликты имен). По соглашению для префикса применяется просто `x`, однако он может быть любым уникальным маркером, таким как `XamlSpecificStuff`:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:XamlSpecificStuff="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Кнопка со специальным содержимым -->
    <Button XamlSpecificStuff:Name="button1" Height="100" Width="100">
      <Ellipse Fill="Green" Height="50" Width="50"/>
    </Button>
  </Grid>
</Page>
```

Очевидный недостаток определения длинных префиксов для пространств имен XML связан с тем, что `XamlSpecificStuff` придется набирать всякий раз, когда в файле XAML нужно сослаться на один из элементов, определенных в этом пространстве имен XML. Из-за того, что префикс `XamlSpecificStuff` намного длиннее, давайте ограничимся `x`.

Помимо ключевых слов `x:Name`, `x:Class` и `x:Code` пространство имен `http://schemas.microsoft.com/winfx/2006/xaml` также предоставляет доступ к дополнительным ключевым словам XAML, наиболее распространенные из которых кратко описаны в табл. 24.9.

Таблица 24.9. Ключевые слова XAML

Ключевое слово XAML	Описание
<code>x:Array</code>	Представляет в XAML тип массива <code>.NET</code>
<code>x:ClassModifier</code>	Позволяет определять видимость класса C# (<code>internal</code> или <code>public</code>), обозначенного ключевым словом <code>Class</code>
<code>x:FieldModifier</code>	Позволяет определять видимость члена типа (<code>internal</code> , <code>public</code> , <code>private</code> или <code>protected</code>) для любого именованного подэлемента корня (например, <code><Button></code> внутри элемента <code><Window></code>). Именованный элемент определяется с использованием ключевого слова <code>Name</code> в XAML
<code>x:Key</code>	Позволяет устанавливать значение ключа для элемента XAML, которое будет помещено в элемент словаря
<code>x:Name</code>	Позволяет указывать сгенерированное имя C# заданного элемента XAML
<code>x:Null</code>	Представляет ссылку <code>null</code>
<code>x:Static</code>	Позволяет ссылаться на статический член типа
<code>x:Type</code>	Эквивалент XAML операции <code>typeof</code> языка C# (она будет выдавать объект <code>System.Type</code> на основе предоставленного имени)
<code>x:TypeArguments</code>	Позволяет устанавливать элемент как обобщенный тип с определенным параметром типа (например, <code>List<int></code> или <code>List<bool></code>)

В дополнение к двум указанным объявлениям пространств имен XML можно (а иногда и нужно) определить дополнительные префиксы дескрипторов в открывающем элементе документа XAML. Обычно так поступают, когда необходимо описать в XAML класс `.NET`, определенный во внешней сборке.

Например, предположим, что было построено несколько специальных элементов управления WPF, которые упакованы в библиотеку по имени `MyControls.dll`. Если теперь требуется создать новый объект `Window`, в котором применяются созданные элементы, то можно установить специальное пространство имен XML, отображаемое на библиотеку `MyControls.dll`, с использованием маркеров `clr-namespace` и `assembly`. Ниже приведен пример разметки, создающей префикс дескриптора по имени `myCtrls`, который может применяться для доступа к элементам управления в этой библиотеке:

```
<Window x:Class="WpfApplication1.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:myCtrls="clr-namespace:MyControls;assembly=MyControls"
  Title="MainWindow" Height="350" Width="525">
  <Grid>
    <myCtrls:MyCustomControl />
  </Grid>
</Window>
```

Маркеру `clr-namespace` назначается название пространства имен .NET в сборке, в то время как маркер `assembly` устанавливается в дружественное имя внешней сборки `*.dll`. Такой синтаксис можно использовать для любой внешней библиотеки .NET, которой желательно манипулировать внутри разметки. В настоящее время в этом нет необходимости, но в последующих главах понадобится определять специальные объявления пространств имен XML для описания типов в разметке.

На заметку! Если нужно определить в разметке класс, который является частью текущей сборки, но находится в другом пространстве имен .NET, то префикс дескриптора `xmlns` определяется без атрибута `assembly`:

```
xmlns:myCtrls="clr-namespace:SomeNamespaceInMyApp"
```

Управление видимостью классов и переменных-членов

Многие ключевые слова вы увидите в действии в последующих главах там, где они потребуются; тем не менее, в качестве простого примера взгляните на следующее XAML-определение `<Window>`, в котором применяются ключевые слова `ClassModifier` и `FieldModifier`, а также `x:Name` и `x:Class` (вспомните, что редактор `KaXaml` не позволяет использовать ключевые слова XAML, вовлекающие компиляцию, такие как `x:Code`, `x:FieldModifier` или `x:ClassModifier`):

```
<!-- Этот класс теперь будет объявлен как internal в файле *.g.cs -->
<Window x:Class="MyWPFApp.MainWindow" x:ClassModifier="internal"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <!-- Эта кнопка будет объявлена как public в файле *.g.cs -->
  <Button x:Name="myButton" x:FieldModifier="public" Content="OK"/>
</Window>
```

По умолчанию все определения типов C#/XAML являются открытыми (`public`), а члены — внутренними (`internal`). Однако на основе показанного определения XAML результирующий автоматический сгенерированный файл содержит внутренний тип класса с открытой переменной-членом `Button`:

```
internal partial class MainWindow : System.Windows.Window,
  System.Windows.Markup.IComponentConnector
{
```

```
public System.Windows.Controls.Button myButton;
...
}
```

Элементы XAML, атрибуты XAML и преобразователи типов

После установки корневого элемента и необходимых пространств имен XML следующая задача заключается в наполнении корня *дочерним элементом*. В реальном приложении WPF дочерним элементом будет диспетчер компоновки (такой как Grid или StackPanel), который в свою очередь содержит любое количество дополнительных элементов, описывающих пользовательский интерфейс. Такие диспетчеры компоновки рассматриваются в главе 25, а пока предположим, что элемент <Window> будет содержать единственный элемент Button.

Как было показано ранее в главе, *элементы XAML* отображаются на типы классов или структур внутри заданного пространства имен .NET, тогда как *атрибуты* в открывающем дескрипторе элемента отображаются на свойства или события конкретного типа. В целях иллюстрации введем в редакторе KaXaml следующее определение <Button>:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Сконфигурировать внешний вид элемента Button -->
    <Button Height="50" Width="100" Content="OK!"
      FontSize="20" Background="Green" Foreground="Yellow"/>
  </Grid>
</Page>
```

Обратите внимание, что значения, присвоенные свойствам, представлены с помощью простого текста. Это может выглядеть как полное несоответствие типам данных, поскольку после создания такого элемента Button в коде C# данным свойствам будут присваиваться *не* строковые объекты, а значения специфических типов данных. Например, ниже показано, как та же самая кнопка описана в коде:

```
public void MakeAButton()
{
    Button myBtn = new Button();
    myBtn.Height = 50;
    myBtn.Width = 100;
    myBtn.FontSize = 20;
    myBtn.Content = "OK!";
    myBtn.Background = new SolidColorBrush(Colors.Green);
    myBtn.Foreground = new SolidColorBrush(Colors.Yellow);
}
```

Оказывается, что инфраструктура WPF поставляется с несколькими классами *преобразователей типов*, которые будут применяться для трансформации простых текстовых значений в корректные типы данных. Такой процесс происходит прозрачно (и автоматически).

Тем не менее, нередко возникает потребность в присваивании атрибуту XAML намного более сложного значения, которое невозможно выразить посредством простой строки. Например, пусть необходимо построить специальную кисть для установки свойства Background элемента Button. Создать кисть подобного рода в коде довольно просто:

```

public void MakeAButton()
{
    ...
    // Необычная кисть для фона.
    LinearGradientBrush fancyBrush =
        new LinearGradientBrush(Colors.DarkGreen, Colors.LightGreen, 45);
    myBtn.Background = fancyBrush;
    myBtn.Foreground = new SolidColorBrush(Colors.Yellow);
}

```

Но можно ли представить эту сложную кисть в виде строки? Нет, нельзя! К счастью, в XAML предусмотрен специальный синтаксис, который можно использовать всякий раз, когда нужно присвоить сложный объект в качестве значения свойства; он называется *синтаксисом "свойство-элемент"*.

Понятие синтаксиса "свойство-элемент" в XAML

Синтаксис "свойство-элемент" позволяет присваивать свойству сложные объекты. Ниже показано описание XAML элемента `Button`, в котором для установки свойства `Background` применяется объект `LinearGradientBrush`:

```

<Button Height="50" Width="100" Content="OK!"
        FontSize="20" Foreground="Yellow">
  <Button.Background>
    <LinearGradientBrush>
      <GradientStop Color="DarkGreen" Offset="0"/>
      <GradientStop Color="LightGreen" Offset="1"/>
    </LinearGradientBrush>
  </Button.Background>
</Button>

```

Обратите внимание, что внутри дескрипторов `<Button>` и `</Button>` определена вложенная область по имени `<Button.Background>`, а в ней — специальный элемент `<LinearGradientBrush>`. (Пока не беспокойтесь о коде кисти; вы освоите графику WPF в главе 26.)

Вообще говоря, любое свойство может быть установлено с использованием синтаксиса "свойство-элемент", который всегда сводится к следующему шаблону:

```

<ОпределяющийКласс>
  <ОпределяющийКласс.СвойствоОпределяющегоКласса>
    <!-- Значение для свойства определяющего класса -->
  </ОпределяющийКласс.СвойствоОпределяющегоКласса>
</ОпределяющийКласс>

```

Хотя любое свойство может быть установлено с применением такого синтаксиса, указание значения в виде простой строки, когда подобное возможно, будет экономить время ввода. Например, вот гораздо более многословный способ установки свойства `Width` элемента `Button`:

```

<Button Height="50" Content="OK!"
        FontSize="20" Foreground="Yellow">
    ...
  <Button.Width>
    100
  </Button.Width>
</Button>

```

Понятие присоединяемых свойств XAML

В дополнение к синтаксису “свойство-элемент” в XAML поддерживается специальный синтаксис, используемый для установки значения *присоединяемого свойства*. По существу присоединяемое свойство позволяет дочернему элементу устанавливать значение свойства, которое на самом деле определено в родительском элементе. Общий шаблон, которому нужно следовать, выглядит так:

```
<РодительскийЭлемент>
  <ДочернийЭлемент РодительскийЭлемент.СвойствоРодительскогоЭлемента =
    "Значение">
</РодительскийЭлемент>
```

Самое распространенное применение синтаксиса присоединяемых свойств связано с позиционированием элементов пользовательского интерфейса внутри одного из классов диспетчеров компоновки (Grid, DockPanel и т.д.). Диспетчеры компоновки более подробно рассматриваются в главе 25, а пока введем в редакторе Kaхaml следующую разметку:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Canvas Height="200" Width="200" Background="LightBlue">
    <Ellipse Canvas.Top="40" Canvas.Left="40" Height="20"
      Width="20" Fill="DarkBlue"/>
  </Canvas>
</Page>
```

Здесь определен диспетчер компоновки Canvas, который содержит элемент Ellipse. Обратите внимание, что с помощью синтаксиса присоединяемых свойств элемент Ellipse способен информировать свой родительский элемент (Canvas) о том, где располагать позицию его левого верхнего угла.

В отношении присоединяемых свойств следует иметь в виду несколько моментов. Прежде всего, это не универсальный синтаксис, который может применяться к любому свойству любого родительского элемента. Скажем, приведенная далее разметка XAML содержит ошибку:

```
<!-- Попытка установки свойства Background в Canvas через присоединяемое
свойство. Ошибка! -->
<Canvas Height="200" Width="200">
  <Ellipse Canvas.Background="LightBlue"
    Canvas.Top="40" Canvas.Left="90"
    Height="20" Width="20" Fill="DarkBlue"/>
</Canvas>
```

В действительности присоединяемые свойства являются специализированной формой специфичной для WPF концепции, которая называется *свойством зависимости*. Если только свойство не было реализовано в весьма специальной манере, то его значение не может быть установлено с использованием синтаксиса присоединяемых свойств. Свойства зависимости подробно исследуются в главе 25.

На заметку! Инструменты Kaхaml и Visual Studio имеют средство IntelliSense, которое отображает допустимые присоединяемые свойства, доступные для установки заданным элементом.

Понятие расширений разметки XAML

Как уже объяснялось, значения свойств чаще всего представляются в виде простой строки или через синтаксис "свойство-элемент". Однако существует еще один способ указать значение атрибута XAML — применение *расширений разметки*. Расширения разметки позволяют анализатору XAML получать значение для свойства из выделенного внешнего класса. Это может обеспечить большие преимущества, поскольку для получения значений некоторых свойств требуется выполнение множества операторов кода.

Расширения разметки предлагают способ аккуратного расширения грамматики XAML новой функциональностью. Расширение разметки внутренне представлено как класс, производный от `MarkupExtension`. Следует отметить, что необходимость в построении специального расширения разметки возникает крайне редко. Тем не менее, некоторые ключевые слова XAML (вроде `x:Array`, `x:Null`, `x:Static` и `x:Type`) являются замаскированными расширениями разметки!

Расширение разметки помещается между фигурными скобками:

```
<Элемент УстанавливаемоеСвойство = "{РасширениеРазметки}"/>
```

Чтобы увидеть расширение разметки в действии, введем в редакторе `Ka.Xaml` следующий код:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:CorLib="clr-namespace:System;assembly=mscorlib">

  <StackPanel>
    <!-- Расширение разметки Static позволяет получать значение
         статического члена класса -->
    <Label Content="{x:Static CorLib:Environment.OSVersion}"/>
    <Label Content="{x:Static CorLib:Environment.ProcessorCount}"/>

    <!-- Расширение разметки Type - это версия XAML
         операции typeof языка C# -->
    <Label Content="{x:Type Button}"/>
    <Label Content="{x:Type CorLib:Boolean}"/>

    <!-- Наполнение элемента ListBox массивом строк -->
    <ListBox Width="200" Height="50">
      <ListBox.ItemsSource>
        <x:Array Type="CorLib:String">
          <CorLib:String>Sun Kil Moon</CorLib:String>
          <CorLib:String>Red House Painters</CorLib:String>
          <CorLib:String>Besnard Lakes</CorLib:String>
        </x:Array>
      </ListBox.ItemsSource>
    </ListBox>
  </StackPanel>
</Page>
```

Прежде всего, обратите внимание, что определение `<Page>` содержит новое объявление пространства имен XML, которое позволяет получать доступ к пространству имен `System` сборки `mscorlib.dll`. После установления этого пространства имен XML первым делом с помощью расширения разметки `x:Static` извлекаются значения свойств `OSVersion` и `ProcessorCount` класса `System.Environment`.

Расширение разметки `x:Type` обеспечивает доступ к описанию метаданных указанного элемента. Здесь содержимому элементов `Label` просто присваиваются полностью заданные имена типов `Button` и `System.Boolean` из WPF.

Наиболее интересная часть показанной выше разметки связана с элементом `ListBox`. Его свойство `ItemsSource` устанавливается в массив строк, полностью объявленный в разметке. Взгляните, каким образом расширение разметки `x:Array` позволяет указывать набор подэлементов внутри своей области действия:

```
<x:Array Type="CorLib:String">
  <CorLib:String>Sun Kil Moon</CorLib:String>
  <CorLib:String>Red House Painters</CorLib:String>
  <CorLib:String>Besnard Lakes</CorLib:String>
</x:Array>
```

На заметку! Предыдущий пример XAML служит только для иллюстрации расширения разметки в действии. Как будет показано в главе 25, существуют гораздо более простые способы наполнения элементов управления `ListBox`.

На рис. 24.3 представлена разметка этого элемента `<Page>` в редакторе `Kaxaml`.

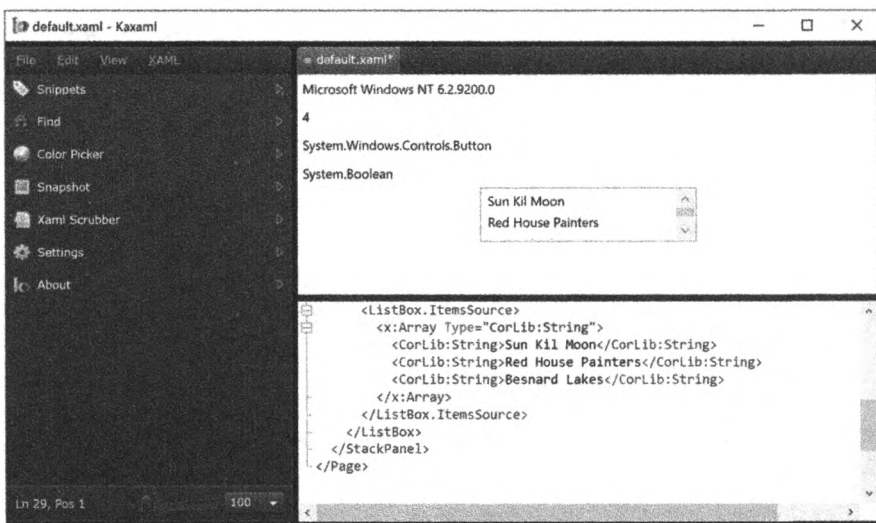


Рис. 24.3. Расширения разметки позволяют устанавливать значения через функциональность выделенного класса

Вы уже видели многочисленные примеры, которые демонстрировали основные аспекты синтаксиса XAML. Вы наверняка согласитесь, что XAML интересен своей возможностью описывать деревья объектов .NET в декларативной манере. Хотя это исключительно полезно при конфигурировании графических пользовательских интерфейсов, не забывайте о том, что с помощью XAML можно описывать любой тип из любой сборки при условии, что он является неабстрактным и содержит стандартный конструктор.

Построение приложений WPF с использованием Visual Studio

Давайте выясним, как Visual Studio может упростить их создание программ WPF.

На заметку! Далее будут представлены основные особенности применения Visual Studio для построения приложений WPF. В последующих главах при необходимости будут иллюстрироваться дополнительные аспекты этой IDE-среды.

Шаблоны проектов WPF

В диалоговом окне New Project (Новый проект) среды Visual Studio определен набор рабочих пространств проектов WPF, которые расположены в узле Windows Classic Desktop (Классический рабочий стол Windows) внутри корневого узла Visual C#. Здесь можно выбирать из нескольких типов проектов, включая WPF App (Приложение WPF), Windows Forms App (Приложение Windows Forms) и многие другие. Для начала создадим новый проект приложения WPF по имени WpfTesterApp (рис. 24.4).

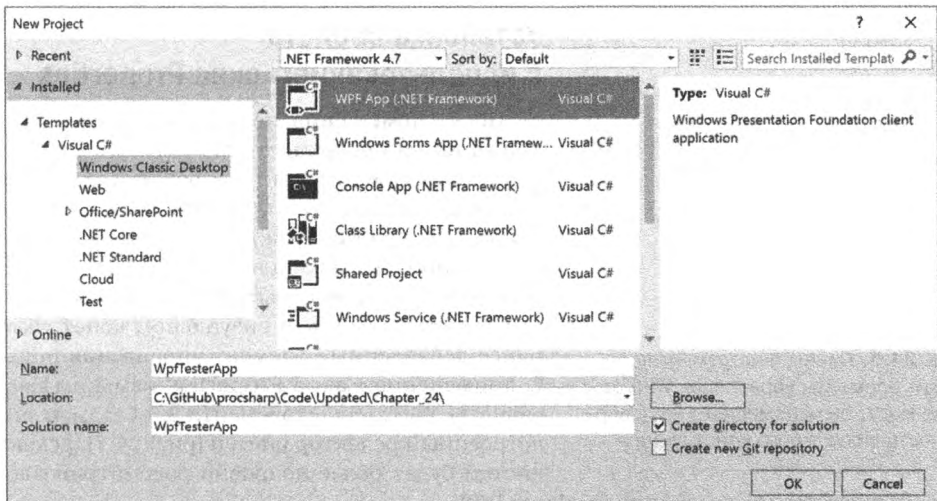


Рис. 24.4. Шаблоны проектов WPF в Visual Studio находятся в узле Windows Classic Desktop

Кроме установки ссылок на все сборки WPF (PresentationCore.dll, PresentationFramework.dll, System.Xaml.dll и WindowsBase.dll) вы получите начальные классы, производные от Window и Application, каждый из которых представлен с применением XAML и файла кода C#.

Панель инструментов и визуальный конструктор/редактор XAML

В Visual Studio имеется панель инструментов (открываемая через меню View (Вид)), которая содержит многочисленные элементы управления WPF. В верхней части панели расположены наиболее часто используемые элементы управления, а в нижней части — все элементы управления (рис. 24.5).

С применением стандартной операции перетаскивания посредством мыши любой из элементов управления можно поместить на поверхность визуального конструктора элемента Window или перетащить его на область редактора разметки XAML внизу окна визуального конструктора. После этого начальная разметка XAML сгенерируется автоматически. Давайте перетащим с помощью мыши элементы управления Button и Calendar на поверхность визуального конструктора. Обратите внимание на возможность изменения позиции и размера элементов управления (а также просмотрите результирующую разметку XAML, генерируемую на основе изменений).

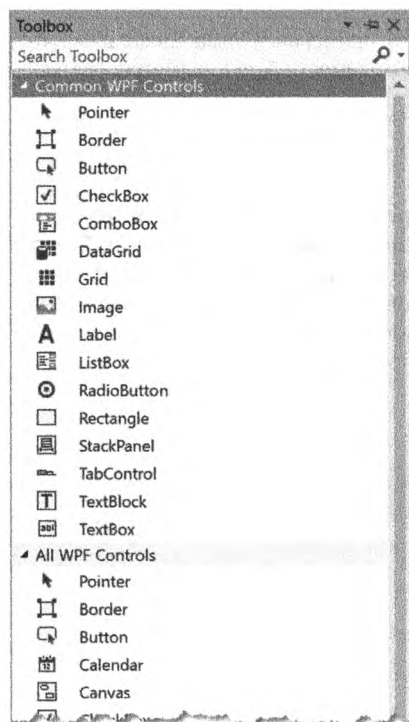


Рис. 24.5. Панель инструментов содержит элементы управления WPF, которые могут быть помещены на поверхность визуального конструктора

В дополнение к построению пользовательского интерфейса, используя мышь и панель инструментов, разметку можно также вводить вручную с применением интегрированного редактора XAML. Как показано на рис. 24.6, вы получаете поддержку средства IntelliSense, которое помогает упростить написание разметки. Для примера можно добавить свойство `Background` в открывающий элемент `<Window>`.

Уделите некоторое время на добавление значений свойств напрямую в редакторе XAML. Обязательно освоите данный аспект визуального конструктора WPF.

Установка свойств с использованием окна Properties

После помещения нескольких элементов управления на поверхность визуального конструктора (или определения их в редакторе вручную) можно открыть окно Properties (Свойства) для установки значений свойств выделенного элемента управления, а также для создания связанных с ним обработчиков событий. В качестве простого примера выберем в визуальном конструкторе ранее добавленный элемент управления `Button`. С применением окна Properties изменим цвет в свойстве `Background` элемента `Button`, используя встроенный редактор кистей (рис. 24.7); редактор кистей будет более подробно рассматриваться в

главе 26 во время исследования графики WPF.

На заметку! В верхней части окна Properties имеется текстовая область, предназначенная для поиска. Чтобы быстро найти свойство, которое требуется установить, понадобится ввести его имя.

После завершения работы с редактором кистей имеет смысл взглянуть на сгенерированную разметку, которая может выглядеть так:

```
<Button x:Name="button" Content="Button"
        HorizontalAlignment="Left" Margin="10,10,0,0"
        VerticalAlignment="Top" Width="75">
  <Button.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="Black" Offset="0"/>
      <GradientStop Color="#FFE90E0E" Offset="1"/>
      <GradientStop Color="#FF1F4CE3"/>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

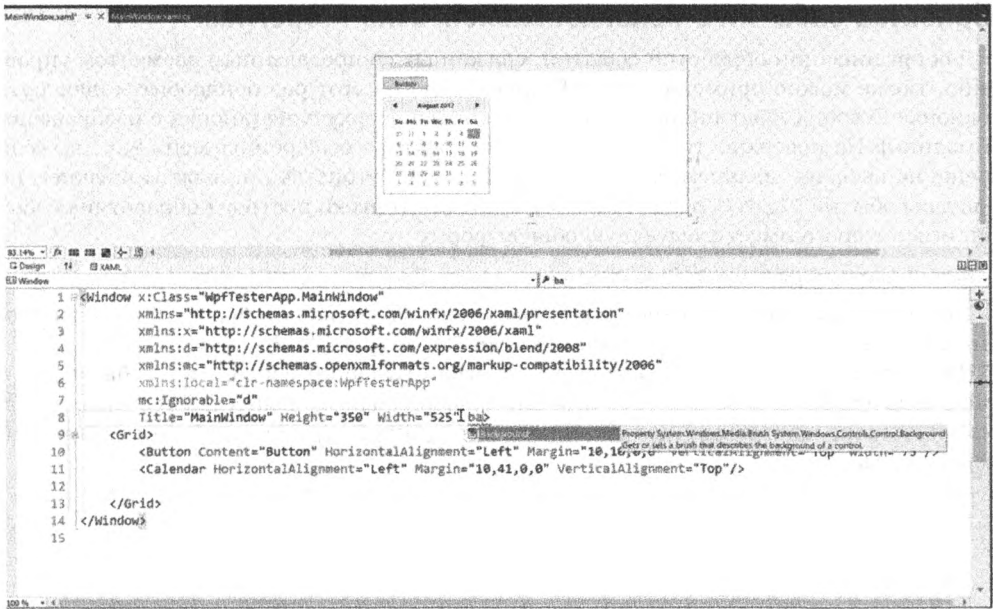


Рис. 24.6. Визуальный конструктор элемента Window

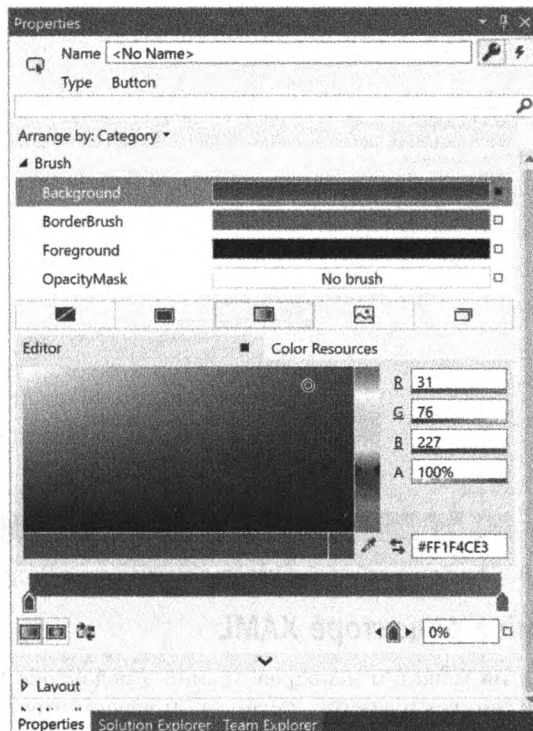


Рис. 24.7. Окно Properties может применяться для конфигурирования пользовательского интерфейса элемента управления WPF

Обработка событий с использованием окна Properties

Для организации обработки событий, связанных с определенным элементом управления, также можно применять окно Properties, но на этот раз понадобится щелкнуть на кнопке Events (События), расположенной справа вверху окна (кнопка с изображением молнии). На поверхности визуального конструктора выберем элемент Button, если он еще не выбран, щелкнем на кнопке Events в окне Properties и дважды щелкнем на поле для события Click. Среда Visual Studio автоматически построит обработчик событий, имя которого имеет следующую общую форму:

ИмяЭлементаУправления_ИмяСобытия

Так как кнопка не была переименована, в окне Properties отображается сгенерированный обработчик событий по имени `button_Click` (рис. 24.8).

Кроме того, Visual Studio сгенерирует соответствующий обработчик события C# в файле кода для окна. В него можно поместить любой код, который должен выполняться, когда на кнопке произведен щелчок. В качестве простого примера добавим следующий оператор кода:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("You clicked the button!");
    // Вы щелкнули на кнопке!
}
```

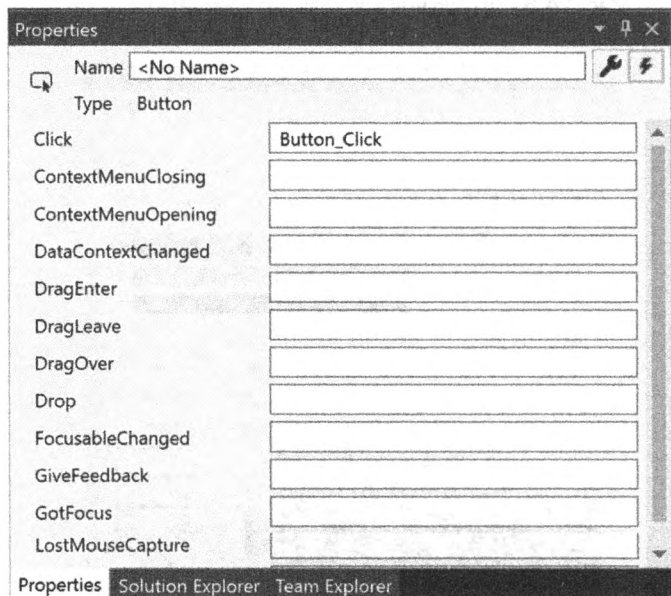


Рис. 24.8. Обработка событий с использованием окна Properties

Обработка событий в редакторе XAML

Обрабатывать события можно и непосредственно в редакторе XAML. Для примера поместим курсор мыши внутрь элемента `<Window>` и введем имя события `MouseMove`, а за ним знак равенства. Среда Visual Studio отобразит все совместимые обработчики из файла кода (если они существуют), а также пункт `Create method` (Создать метод), как показано на рис. 24.9.



Рис. 24.9. Обработка событий с применением редактора XAML

Позволим IDE-среде создать обработчик события `MouseMove`, введем следующий код и запустим приложение, чтобы увидеть конечный результат:

```

private void MainWindow_MouseMove (object sender, MouseEventArgs e)
{
    this.Title = e.GetPosition(this).ToString();
}

```

На заметку! В главе 28 описаны паттерны MVVM и “Команда” (Command), которые являются гораздо лучшим способом обработки событий щелчков в приложениях уровня предприятия. Но если вас интересует только простое приложение, тогда обработка событий щелчков с помощью прямолинейного обработчика будет вполне приемлемой.

Окно Document Outline

Во время работы с любым основанным на XAML проектом вы определенно будете использовать значительный объем разметки для представления пользовательского интерфейса. Когда вы начнете сталкиваться с более сложной разметкой XAML, может оказаться удобной визуализация разметки для быстрого выбора элементов с целью редактирования в визуальном конструкторе Visual Studio.

В настоящее время наша разметка довольно проста, т.к. было определено лишь несколько элементов управления внутри начального элемента `<Grid>`. Тем не менее, необходимо найти окно Document Outline (Схема документа), которое по умолчанию располагается в левой части окна IDE-среды (если обнаружить его не удастся, то данное окно можно открыть через пункт меню `View → Other Windows (Вид → Другие окна)`). При активном окне визуального конструктора XAML (не окне с файлом кода C#) в IDE-среде можно заметить, что в окне Document Outline отображаются вложенные элементы (рис. 24.10).

Этот инструмент также предоставляет способ временного сокрытия заданного элемента (или набора элементов) на повер-

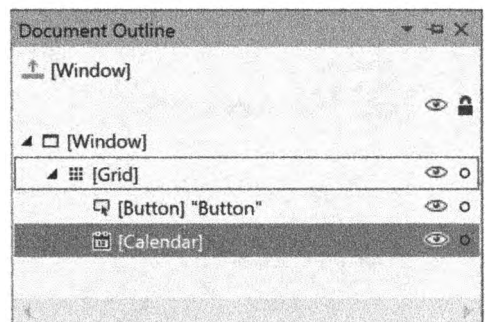


Рис. 24.10. Визуализация разметки XAML в окне Document Outline

хности визуального конструктора, а также блокировки элементов с целью предотвращения их дальнейшего редактирования. В главе 25 вы увидите, что окно Document Outline предлагает много других возможностей для группирования выбранных элементов внутри новых диспетчеров компоновки (помимо прочих средств).

Включение и отключение отладчика XAML

После запуска приложения на экране появляется окно MainWindow. Кроме того, можно также видеть интерактивный отладчик (рис. 24.11).

При желании отключить его понадобится найти настройки, касающиеся отладки XAML, на вкладке Tools⇒Options⇒Debugging⇒General (Сервис⇒Параметры⇒Отладка⇒Общие). Снятие отметки с флажков предотвращает перекрывание окон приложения окном отладчика. Настройки, связанные с отладкой, представлены на рис. 24.12.

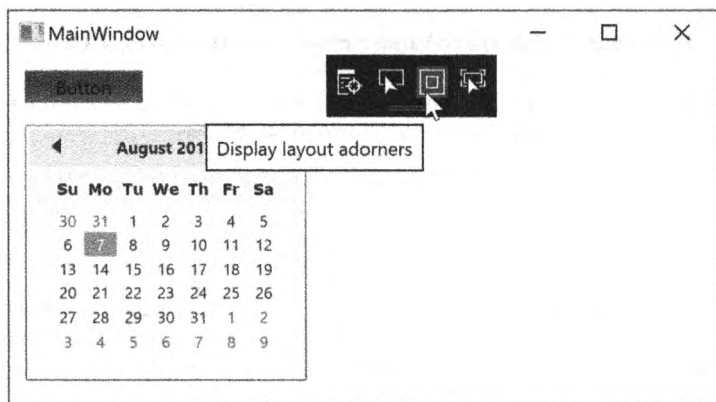


Рис. 24.11. Отладка пользовательского интерфейса XAML

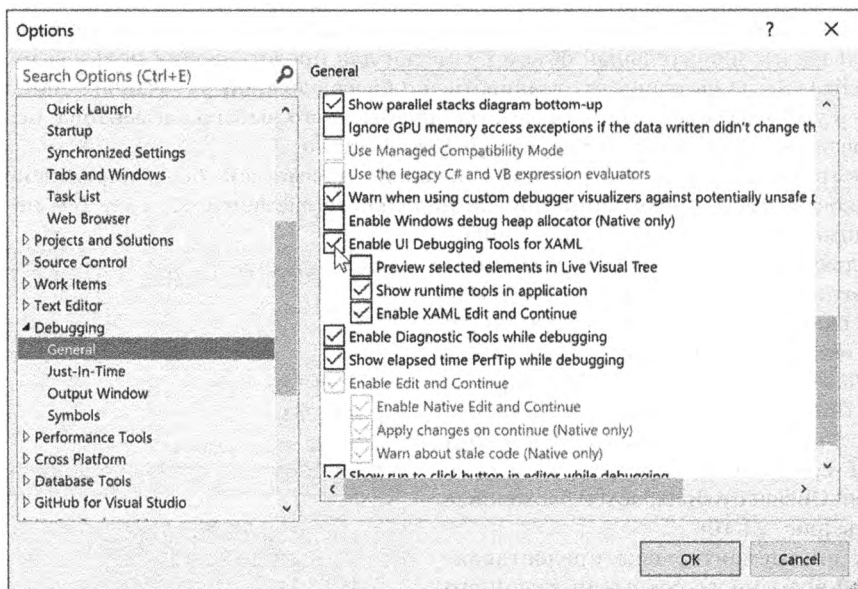


Рис. 24.12. Настройки, связанные с отладкой пользовательского интерфейса XAML

Исследование файла App.xaml

Как проект узнает, какое окно отображать? Еще большая интрига в том, что в результате исследования файлов кода, относящихся к приложению, метод `Main()` обнаружить не удастся. Вы уже знаете, что приложения обязаны иметь точку входа, так как же инфраструктуре .NET становится известно, каким образом запускать приложение? К счастью, оба связующих элемента автоматически поддерживаются через шаблоны Visual Studio и инфраструктуру WPF.

Чтобы разгадать загадку, какое окно открывать, в файле App.xaml посредством разметки определен класс приложения. В дополнение к определениям пространств имен он определяет свойства приложения, такие как `StartupUri`, ресурсы уровня приложения (рассматриваемые в главе 27) и специфические обработчики для событий приложения вроде `Startup` и `Exit`. В `StartupUri` указано окно, подлежащее загрузке при запуске. Откроем файл App.xaml и проанализируем разметку в нем:

```
<Application x:Class="WpfTesterApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfTesterApp"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

С применением визуального конструктора XAML и средства завершения кода Visual Studio добавим обработчики для событий `Startup` и `Exit`. Обновленная разметка XAML должна выглядеть примерно так (изменение выделено полужирным):

```
<Application x:Class="WpfTesterApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfTesterApp"
    StartupUri="MainWindow.xaml"
    Startup="App_OnStartup" Exit="App_OnExit">
    <Application.Resources>
    </Application.Resources>
</Application>
```

Содержимое файла App.xaml.cs должно быть похожим на приведенное ниже:

```
public partial class App : Application
{
    private void App_OnStartup(object sender, StartupEventArgs e)
    {
    }
    private void App_OnExit(object sender, ExitEventArgs e)
    {
    }
}
```

Обратите внимание, что класс помечен как частичный (`partial`). На самом деле все оконные классы в отделенном коде для файлов XAML помечаются как частичные. В том-то и кроется решение вопроса, где находится метод `Main()`. Но сначала необходимо выяснить, что происходит при обработке файлов XAML утилитой `msbuild.exe`.

Отображение разметки XAML окна на код C#

Когда утилита `msbuild.exe` обрабатывает файл `*.csproj`, она создает для каждого файла XAML в проекте три файла: `*.g.cs` (где *g* означает *autogenerated* (автоматически сгенерированный)), `*.g.i.cs` (где *i* означает *IntelliSense*) и `*.baml` (для BAML (Binary Application Markup Language — двоичный язык разметки приложений)). Такие файлы сохраняются в каталоге `\obj\Debug` (и могут просматриваться в окне *Solution Explorer* за счет щелчка на кнопке *Show All Files* (Показать все файлы)). Чтобы их увидеть, может потребоваться щелкнуть на кнопке *Refresh* (Обновить) в окне *Solution Explorer*, т.к. они не являются частью фактического проекта, а представляют собой артефакты построения.

Чтобы сделать процесс более осмысленным, элементам управления полезно назначить имена. Назначим имена элементам управления `Button` и `Calendar`, как показано ниже:

```
<Button Name="ClickMe" Content="Button"
        HorizontalAlignment="Left" Margin="10,10,0,0"
        VerticalAlignment="Top" Width="75" Click="Button_Click">
    // Для краткости разметка не показана.
</Button>
<Calendar Name="MyCalendar" HorizontalAlignment="Left"
        Margin="10,41,0,0" VerticalAlignment="Top"/>
```

Теперь перестроим решение (или проект) и обновим файлы в окне *Solution Explorer*. Если открыть файл `MainWindow.g.cs` в текстовом редакторе, то внутри обнаружится класс по имени `MainWindow`, который расширяет базовый класс `Window`. Имя данного класса является прямым результатом действия атрибута `x:Class` в начальном дескрипторе `<Window>`.

В классе `MainWindow` определена закрытая переменная-член типа `bool` (с именем `_contentLoaded`), которая не была напрямую представлена в разметке XAML. Указанный член данных используется для того, чтобы определить (и гарантировать) присваивание содержимого окна только один раз. Класс также содержит переменную-член типа `System.Windows.Controls.Button` по имени `ClickMe`. Имя элемента управления основано на значении атрибута `x>Name` в открывающем объявлении `<Button>`. В классе не будет присутствовать переменная для элемента управления `Calendar`. Причина в том, что утилита `msbuild.exe` создает переменную для каждого именованного элемента управления в разметке XAML, который имеет связанный код в отделенном коде. Когда такого кода нет, потребность в переменной отпадает. Чтобы еще больше запутать ситуацию, если бы элементу управления `Button` не назначалось имя, то и для него не было бы предусмотрено переменной. Это часть магии WPF, которая связана с реализацией интерфейса `IComponentConnector`.

Сгенерированный компилятором класс также явно реализует интерфейс `IComponentConnector` из WPF, определенный в пространстве имен `System.Windows.Markup`. В интерфейсе `IComponentConnector` имеется единственный метод `Connect()`, который реализован для подготовки каждого элемента управления, определенного в разметке, и обеспечения логики событий, как указано в исходном файле `MainWindow.xaml`. Можно заметить обработчик, настроенный для события щелчка на кнопке `ClickMe`. Перед завершением метода переменная-член `_contentLoaded` устанавливается в `true`. Вот как выглядит данный метод:

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId,
        object target)
{
    switch (connectionId)
    {
```

```

case 1:
    this.ClickMe = ((System.Windows.Controls.Button) (target));
    #line 11 "..\..\MainWindow.xaml"
    this.ClickMe.Click +=
        new System.Windows.RoutedEventHandler(this.Button_Click);
    #line default
    #line hidden
    return;
}
this._contentLoaded = true;
}

```

Чтобы продемонстрировать влияние неименованных элементов управления на код, добавим к календарю обработчик события `SelectedDatesChanged`. Перестроим приложение, обновим файлы и заново загрузим файл `MainWindow.g.cs`. Теперь в методе `Connect()` присутствует следующий блок кода:

```

#line 20 "..\..\MainWindow.xaml"
this.MyCalendar.SelectedDatesChanged += new
    System.EventHandler<System.Windows.Controls.SelectionChangedEventArgs>(
        this.MyCalendar_OnSelectedDatesChanged);

```

Он сообщает инфраструктуре о том, что элементу управления в строке 20 файла XAML назначен обработчик события `SelectedDatesChanged`, как показано в предыдущем коде.

Последнее, но не менее важное: класс `MainWindow` определяет и реализует метод по имени `InitializeComponent()`. Вы могли бы ожидать, что данный метод содержит код, который настраивает внешний вид и поведение каждого элемента управления, устанавливая его разнообразные свойства (`Height`, `Width`, `Content` и т.д.). Однако это совсем не так! Как тогда элементы управления получают корректный пользовательский интерфейс? Логика в методе `InitializeComponent()` выясняет местоположение встроенного в сборку ресурса, который именован идентично исходному файлу `*.xaml`:

```

public void InitializeComponent() {
    if (_contentLoaded) {
        return;
    }
    _contentLoaded = true;
    System.Uri resourceLocator =
        new System.Uri("/WpfTesterApp;component/mainwindow.xaml",
            System.UriKind.Relative);
    #line 1 "..\..\MainWindow.xaml"
    System.Windows.Application.LoadComponent(this, resourceLocator);
    #line default
    #line hidden
}

```

Здесь возникает вопрос: что собой представляет этот встроенный ресурс?

Роль BAML

Как и можно было предположить, формат BAML является компактным двоичным представлением исходных данных XAML. Файл `*.baml` встраивается в виде ресурса (через сгенерированный файл `*.g.resources`) в скомпилированную сборку. Ресурс BAML содержит все данные, необходимые для настройки внешнего вида и поведения виджетов пользовательского интерфейса (свойства вроде `Height` и `Width`).

Здесь важно понимать, что приложение WPF содержит внутри себя двоичное представление (BAML) разметки. Во время выполнения ресурс BAML извлекается из контейнера ресурсов и применяется для настройки внешнего вида и поведения всех окон и элементов управления.

Вдобавок запомните, что имена таких двоичных ресурсов идентичны именам написанных автономных файлов *.xaml. Тем не менее, отсюда вовсе не следует необходимость распространения файлов *.xaml вместе со скомпилированной программой WPF. Если только не строится приложение WPF, которое должно динамически загружать и анализировать файлы *.xaml во время выполнения, то поставлять исходную разметку никогда не придется.

Разгадывание загадки Main ()

Теперь, когда известно, как работает процесс msbuild.exe, откроем файл App.g.cs. В нем обнаружится автоматически сгенерированный метод Main(), который инициализирует и запускает наш объект приложения.

```
public static void Main() {
    WpfTesterApp.App app = new WpfTesterApp.App();
    app.InitializeComponent();
    app.Run();
}
```

Метод InitializeComponent() конфигурирует свойства приложения, включая StartupUri и обработчики событий Startup и Exit.

```
public void InitializeComponent() {
    #line 5 "..\..\App.xaml"
    this.Startup += new System.Windows.StartupEventHandler(this.App_OnStartup);
    #line default
    #line hidden
    #line 5 "..\..\App.xaml"
    this.Exit += new System.Windows.ExitEventHandler(this.App_OnExit);
    #line default
    #line hidden
    #line 5 "..\..\App.xaml"
    this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);
    #line default
    #line hidden
}
```

Взаимодействие с данными уровня приложения

Вспомните, что в классе Application имеется свойство по имени Properties, которое позволяет определить коллекцию пар "имя/значение" через индексатор типа. Поскольку этот индексатор предназначен для оперирования на типе System.Object, в коллекцию можно сохранять элементы любого вида (в том числе экземпляры специальных классов) с целью последующего извлечения по дружественному имени. С использованием такого подхода легко разделять данные между всеми окнами в приложении WPF.

В целях иллюстрации мы обновим текущий обработчик события Startup, чтобы он проверял входящие аргументы командной строки на присутствие значения /GODMODE (распространенный мошеннический код во многих играх). Если оно найдено, тогда значение bool по имени GodMode внутри коллекции свойств устанавливается в true (в противном случае оно устанавливается в false).

Звучит достаточно просто, но как передать обработчику события Startup входные аргументы командной строки (обычно получаемые методом `Main()`)? Один из подходов предусматривает вызов статического метода `Environment.GetCommandLineArgs()`. Однако те же самые аргументы автоматически добавляются во входной параметр `StartupEventArgs` и доступны через свойство `Args`. Таким образом, ниже приведена первая модификация текущей кодовой базы:

```
private void App_OnStartup(object sender, StartupEventArgs e)
{
    Application.Current.Properties["GodMode"] = false;
    // Проверить входные аргументы командной строки
    // на предмет наличия флага /GODMODE.
    foreach (string arg in e.Args)
    {
        if (arg.Equals("/godmode", StringComparison.OrdinalIgnoreCase))
        {
            Application.Current.Properties["GodMode"] = true;
            break;
        }
    }
}
```

Данные уровня приложения доступны из любого места внутри приложения WPF. Для обращения к ним потребуется лишь получить точку доступа к глобальному объекту приложения (через `Application.Current`) и просмотреть коллекцию. Например, обработчик события `Click` для кнопки можно было бы изменить следующим образом:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Указал ли пользователь /godmode?
    if ((bool)Application.Current.Properties["GodMode"])
    {
        MessageBox.Show("Cheater!");
        // Мошенник!
    }
}
```

Если теперь ввести аргумент командной строки `/godmode` на вкладке **Debug** (Отладка) в окне свойств проекта и запустить программу, то отобразится окно сообщения и программа завершится. Можно также запустить программу из командной строки с помощью показанной ниже команды (предварительно открыв окно командной строки и перейдя в каталог `bin/debug`):

```
WpfAppAllCode.exe /godmode
```

Отобразится окно сообщения, а программа завершится.

На заметку! Вспомните, что аргументы командной строки можно указывать внутри Visual Studio. Нужно просто дважды щелкнуть на значке **Properties** (Свойства) в окне **Solution Explorer**, в открывшемся диалоговом окне перейти на вкладку **Debug** (Отладка) и ввести `/godmode` в поле **Command line arguments** (Аргументы командной строки).

Обработка закрытия объекта Window

Конечные пользователи могут завершать работу окна с помощью многочисленных встроенных приемов уровня системы (например, щелкнув на кнопке закрытия X внутри рамки окна) или вызвав метод `Close()` в ответ на некоторое действие с интерактивным элементом (скажем, выбор пункта меню `File⇒Exit` (Файл⇒Выход)). Инфраструктура WPF предлагает два события, которые можно перехватывать для выяснения, действительно ли пользователь намерен закрыть окно и удалить его из памяти. Первое такое событие — `Closing`, которое работает в сочетании с делегатом `CancelEventHandler`.

Указанный делегат ожидает целевые методы, принимающие тип `System.ComponentModel.CancelEventArgs` во втором параметре. Класс `CancelEventArgs` предоставляет свойство `Cancel`, установка которого в `true` предотвращает фактическое закрытие окна (что удобно, когда пользователю должен быть задан вопрос о том, на самом ли деле он желает закрыть окно или сначала нужно сохранить сделанную работу).

Если пользователь действительно хочет закрыть окно, тогда свойство `CancelEventArgs.Cancel` можно установить в `false` (стандартное значение). В итоге сгенерируется событие `Closed` (которое работает с делегатом `System.EventHandler`), представляющее собой точку, где окно полностью и безвозвратно готово к закрытию.

Модифицируем класс `MainWindow` для обработки упомянутых двух событий, добавив в текущий код конструктора такие операторы:

```
public MainWindow()
{
    InitializeComponent();
    this.Closed+=MainWindow_OnClosed;
    this.Closing += MainWindow_Closing;
}
```

Теперь реализуем соответствующие обработчики событий:

```
private void MainWindow_Closing(object sender,
    System.ComponentModel.CancelEventArgs e)
{
    // Выяснить, на самом ли деле пользователь хочет закрыть окно.
    string msg = "Do you want to close without saving?";
    MessageBoxResult result = MessageBox.Show(msg,
        "My App", MessageBoxButton.YesNo, MessageBoxImage.Warning);
    if (result == MessageBoxResult.No)
    {
        // Если пользователю не желает закрывать окно, то отменить закрытие.
        e.Cancel = true;
    }
}

private void MainWindow_Closed(object sender, EventArgs e)
{
    MessageBox.Show("See ya!");
}
```

Запустим программу и попробуем закрыть окно, щелкнув либо на значке X в правом верхнем углу окна, либо на кнопке. Должно появиться диалоговое окно с запросом подтверждения. Щелчок на кнопке `Yes` (Да) приведет к отображению окна с прощальным сообщением, а щелчок на кнопке `No` (Нет) оставит окно в памяти.

Перехват событий мыши

Инфраструктура WPF предоставляет несколько событий, которые можно перехватывать, чтобы взаимодействовать с мышью. В частности, базовый класс `UIElement` определяет такие связанные с мышью события, как `MouseMove`, `MouseUp`, `MouseDown`, `MouseEnter`, `MouseLeave` и т.д.

В качестве примера обработаем событие `MouseMove`. Это событие работает в сочетании с делегатом `System.Windows.Input.MouseEventHandler`, который ожидает, что его целевой метод будет принимать во втором параметре объект типа `System.Windows.Input.MouseEventArgs`. С применением класса `MouseEventArgs` можно извлекать позицию (*x, y*) курсора мыши и другие важные детали. Взгляните на следующее частичное определение:

```
public class MouseEventArgs : InputEventArgs
{
    ...
    public Point GetPosition(IInputElement relativeTo);
    public MouseButtonState LeftButton { get; }
    public MouseButtonState MiddleButton { get; }
    public MouseDevice MouseDevice { get; }
    public MouseButtonState RightButton { get; }
    public StylusDevice StylusDevice { get; }
    public MouseButtonState XButton1 { get; }
    public MouseButtonState XButton2 { get; }
}
```

На заметку! Свойства `XButton1` и `XButton2` позволяют взаимодействовать с “расширенными кнопками мыши” (вроде кнопок “вперед” и “назад”, которые имеются в некоторых устройствах). Они часто используются для взаимодействия с хронологией навигации браузера, чтобы переключаться между посещенными страницами.

Метод `GetPosition()` позволяет получать значение (*x, y*) относительно элемента пользовательского интерфейса в окне. Если интересует позиция относительно активного окна, то нужно просто передать `this`. Обеспечим обработку события `MouseMove` в конструкторе класса `MainWindow`:

```
public MainWindow(string windowTitle, int height, int width)
{
    ...
    this.MouseMove += MainWindow_MouseMove;
}
```

Ниже приведен обработчик события `MouseMove`, который отобразит местоположение курсора мыши в области заголовка окна (обратите внимание, что возвращенный тип `Point` транслируется в строковое значение посредством вызова `ToString()`):

```
private void MainWindow_MouseMove(object sender,
    System.Windows.Input.MouseEventArgs e)
{
    // Отобразить в заголовке окна текущую позицию (x, y) курсора мыши.
    this.Title = e.GetPosition(this).ToString();
}
```

Перехват событий клавиатуры

Обработка клавиатурного ввода для окна, на котором находится фокус, также очень проста. В классе `UIElement` определено несколько событий, которые можно перехватывать для отслеживания нажатий клавиш клавиатуры на активном элементе (например, `KeyUp` и `KeyDown`). События `KeyUp` и `KeyDown` работают с делегатом `System.Windows.Input.KeyEventHandler`, который ожидает во втором параметре тип `KeyEventArgs`, определяющий набор важных открытых свойств:

```
public class KeyEventArgs : KeyboardEventArgs
{
    ...
    public bool IsDown { get; }
    public bool IsRepeat { get; }
    public bool IsToggled { get; }
    public bool IsUp { get; }
    public Key Key { get; }
    public KeyStates KeyStates { get; }
    public Key SystemKey { get; }
}
```

Чтобы проиллюстрировать организацию обработки события `KeyDown` в конструкторе `MainWindow` (как делалось для предыдущих событий), мы реализуем следующий обработчик события, который изменяет содержимое кнопки на информацию о текущей нажатой клавише:

```
private void MainWindow_KeyDown(object sender,
    System.Windows.Input.KeyEventArgs e)
{
    // Отобразить на кнопке информацию о нажатой клавише.
    ClickMe.Content = e.Key.ToString();
}
```

К настоящему моменту WPF может показаться всего лишь очередной инфраструктурой для построения графических пользовательских интерфейсов, которая предлагает (в большей или меньшей степени) те же самые службы, что и Windows Forms, MFC или VB6. Если бы это было именно так, тогда возникает вопрос о смысле наличия еще одного инструментального набора, ориентированного на создание пользовательских интерфейсов. Чтобы реально оценить уникальность WPF, потребуется освоить основанную на XML грамматику — XAML.

Исходный код. Проект `WpfTesterApp` доступен в подкаталоге `Chapter_24`.

Изучение документации WPF

В заключение главы следует отметить, что тематике WPF в документации .NET 4.7 Framework SDK посвящен целый раздел. По мере исследования API-интерфейса WPF и чтения остальных глав, раскрывающих инфраструктуру WPF, вы обнаружите настоятельную потребность в обращении к справочной системе. Там вы найдете множество примеров разметки XAML и подробные обучающие руководства по широкому спектру тем, начиная с программирования трехмерной графики и заканчивая сложными операциями привязки данных.

Документация WPF доступна через меню `Docs` ⇒ `.NET` ⇒ `.NET Framework` ⇒ `Windows Presentation Foundation` (Документация ⇒ `.NET` ⇒ `.NET Framework` ⇒ `Windows Presentation Foundation`); она находится по адресу <https://docs.microsoft.com/ru-ru/dotnet/framework/wpf/index>.

Резюме

Инфраструктура Windows Presentation Foundation (WPF) представляет собой набор инструментов для построения пользовательских интерфейсов, появившийся в версии .NET 3.0. Основная цель WPF заключается в интеграции и унификации множества ранее разрозненных настольных технологий (двумерная и трехмерная графика, разработка окон и элементов управления и т.п.) в единую унифицированную программную модель. Помимо этого в приложениях WPF обычно применяется язык XAML, который позволяет определять внешний вид и поведение элементов WPF через разметку.

Вспомните, что язык XAML позволяет описывать деревья объектов .NET с использованием декларативного синтаксиса. Во время исследования XAML в данной главе вы узнали о нескольких новых фрагментах синтаксиса, включая синтаксис “свойство-элемент” и присоединяемые свойства, а также о роли преобразователей типов и расширений разметки XAML.

Разметка XAML является ключевым аспектом любого приложения WPF производственного уровня. В финальном примере главы было построено приложение WPF, которое продемонстрировало многие концепции, обсужденные в главе. В последующих главах эти и многие другие концепции будут рассматриваться более подробно.

ГЛАВА 25

Элементы управления, компоновки, события и привязка данных в WPF

В главе 24 была представлена основа модели программирования WPF, включая классы `Window` и `Application`, грамматику XAML и использование файлов кода. Кроме того, в ней было дано введение в процесс построения приложений WPF с применением визуальных конструкторов IDE-среды Visual Studio. В настоящей главе мы углубимся в конструирование более сложных графических пользовательских интерфейсов с использованием нескольких новых элементов управления и диспетчеров компоновки, а также по ходу дела выясним дополнительные возможности визуальных конструкторов WPF, доступных в Visual Studio.

Здесь будут рассматриваться некоторые важные темы, связанные с элементами управления WPF, такие как программная модель привязки данных и применение команд управления. Вы узнаете, как работать с интерфейсами `Ink API` и `Documents API`, которые позволяют получать ввод от пера (или мыши) и создавать форматированные документы с использованием протокола `XML Paper Specification`.

Обзор основных элементов управления WPF

Если вы не являетесь новичком в области построения графических пользовательских интерфейсов, то общее назначение большинства элементов управления WPF не должно вызывать много вопросов. Независимо от того, какой набор инструментов для создания графических пользовательских интерфейсов вы применяли в прошлом (например, VB 6.0, MFC, Java AWT/Swing, Windows Forms, macOS, GTK+/GTK# и т.п.), основные элементы управления WPF, перечисленные в табл. 25.1, вероятно покажутся знакомыми.

Элементы управления Ink API

В дополнение к общепринятым элементам управления WPF, упомянутым в табл. 25.1, инфраструктура WPF определяет элементы управления для работы с интерфейсом `Ink API`. Данный аспект разработки WPF полезен при построении приложений для Tablet PC, т.к. он позволяет захватывать ввод от пера. Тем не менее, это вовсе не означает, что стандартное настольное приложение не может задействовать `Ink API`, поскольку те же самые элементы управления могут работать с вводом от мыши.

Таблица 25.1. Основные элементы управления WPF

Категория элементов управления WPF	Примеры членов	Описание
Основные элементы управления для пользовательского ввода	Button, RadioButton, ComboBox, CheckBox, Calendar, DatePicker, Expander, DataGrid, ListBox, ListView, ToggleButton, TreeView, ContextMenu, ScrollBar, Slider, TabControl, TextBlock, TextBox, RepeatButton, RichTextBox, Label	Инфраструктура WPF предлагает полное семейство элементов управления, которые можно задействовать при построении пользовательских интерфейсов
Боковые элементы окон и элементов управления	Menu, ToolBar, StatusBar, ToolTip, ProgressBar	Эти элементы пользовательского интерфейса служат для декорирования рамки объекта Window компонентами для ввода (наподобие Menu) и элементами информирования пользователя (скажем, StatusBar и ToolTip)
Элементы управления мультимедиа	Image, MediaElement, SoundPlayerAction	Эти элементы управления предоставляют поддержку воспроизведения аудио/видео и вывода изображений
Элементы управления компоновкой	Border, Canvas, DockPanel, Grid, GridView, GridSplitter, GroupBox, Panel, TabControl, StackPanel, Viewbox, WrapPanel	Инфраструктура WPF предлагает множество элементов управления, которые позволяют группировать и организовывать другие элементы в целях управления компоновкой

Пространство имен `System.Windows.Ink` из сборки `PresentationCore.dll` содержит разнообразные поддерживающие типы Ink API (например, `Stroke` и `StrokeCollection`). Однако большинство элементов управления Ink API (вроде `InkCanvas` и `InkPresenter`) упакованы вместе с общими элементами управления WPF в пространстве имен `System.Windows.Controls` внутри сборки `PresentationFramework.dll`. Мы будем работать с интерфейсом Ink API позже в главе.

Элементы управления документов WPF

Вдобавок инфраструктура WPF предоставляет элементы управления для расширенной обработки документов, позволяя строить приложения, которые включают функциональность в стиле Adobe PDF. С применением типов из пространства имен `System.Windows.Documents` (также находящегося в сборке `PresentationFramework.dll`) можно создавать готовые к печати документы, которые поддерживают масштабирование, поиск, пользовательские аннотации (“клеякие” заметки) и другие развитые средства работы с текстом.

Тем не менее, “за кулисами” элементы управления документов не используют API-интерфейсы Adobe PDF, а взамен работают с API-интерфейсом XML Paper Specification (XPS). Конечные пользователи никакой разницы не заметят, потому что документы PDF и XPS имеют практически идентичный вид и поведение. В действительности доступно

множество бесплатных утилит, которые позволяют выполнять преобразования между указанными двумя файловыми форматами на лету. Из-за ограничений по объему такие элементы управления в текущем издании не рассматриваются.

Общие диалоговые окна WPF

Инфраструктура WPF также предлагает несколько общих диалоговых окон, таких как `OpenFileDialog` и `SaveFileDialog`, которые определены в пространстве имен `Microsoft.Win32` внутри сборки `PresentationFramework.dll`. Работа с любым из указанных диалоговых окон сводится к созданию объекта и вызову метода `ShowDialog()`:

```
using Microsoft.Win32;
// Для краткости код не показан.
private void btnShowDlg_Click(object sender, RoutedEventArgs e)
{
    // Отобразить диалоговое окно сохранения файла.
    SaveFileDialog saveDlg = new SaveFileDialog();
    saveDlg.ShowDialog();
}
```

Как и можно было ожидать, в этих классах поддерживаются разнообразные члены, которые позволяют устанавливать фильтры файлов и пути к каталогам, а также получать доступ к выбранным пользователем файлам. Некоторые диалоговые окна применяются в последующих примерах; кроме того, будет показано, как строить специальные диалоговые окна для получения пользовательского ввода.

Подробные сведения находятся в документации

Невзирая на то что вам могло показаться, целью главы *не* является описание каждого члена абсолютно всех элементов управления WPF. Взамен будет дан обзор различных элементов управления с акцентом на лежащей в основе программной модели и ключевых службах, общих для большинства элементов управления WPF.

Полное представление о конкретной функциональности заданного элемента управления дает документация .NET Framework 4.7 SDK — в частности, раздел “Control Library” (“Библиотека элементов управления”) справочной системы, доступный по ссылке <https://docs.microsoft.com/ru-ru/dotnet/framework/wpf/controls/index>.

Здесь вы найдете исчерпывающие описания каждого элемента управления, разнообразные примеры кода (на XAML и C#), а также информацию о цепочке наследования, реализованных интерфейсах и примененных атрибутах для любого элемента управления. Обязательно уделите время на исследование элементов управления, рассматриваемых в настоящей главе.

Краткий обзор визуального конструктора WPF в Visual Studio

Большинство стандартных элементов управления WPF упаковано в пространство имен `System.Windows.Controls` внутри сборки `PresentationFramework.dll`. При построении приложения WPF в Visual Studio множество общих элементов управления находится в панели инструментов при условии, что активным окном является визуальный конструктор WPF.

Подобно другим инфраструктурам для построения пользовательских интерфейсов в Visual Studio такие элементы управления можно перетаскивать на поверхность визуального конструктора WPF и конфигурировать их в окне `Properties` (Свойства), как

было показано в главе 24. Хотя Visual Studio сгенерирует приличный объем разметки XAML автоматически, нет ничего необычного в том, чтобы затем редактировать разметку вручную. Давайте рассмотрим основы.

Работа с элементами управления WPF в Visual Studio

Вы можете вспомнить из главы 24, что после помещения элемента управления WPF на поверхность визуального конструктора Visual Studio в окне Properties (или прямо в разметке XAML) необходимо установить свойство `x:Name`, т.к. это позволяет обращаться к объекту в связанном файле кода C#. Кроме того, на вкладке Events (События) окна Properties можно генерировать обработчики событий для выбранного элемента управления. Таким образом, с помощью Visual Studio можно было бы сгенерировать следующую разметку для простого элемента управления Button:

```
<Button x:Name="btnMyButton" Content="Click Me!" Height="23" Width="140"
Click="btnMyButton_Click" />
```

Здесь свойство Content элемента Button устанавливается в простую строку "Click Me!". Однако благодаря модели содержимого элементов управления WPF можно создать элемент Button со следующим сложным содержимым:

```
<Button x:Name="btnMyButton" Height="121" Width="156" Click="btnMyButton_Click">
  <Button.Content>
    <StackPanel Height="95" Width="128" Orientation="Vertical">
      <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
      <Label Width="59" FontSize="20" Content="Click!" Height="36" />
    </StackPanel>
  </Button.Content>
</Button>
```

Вы можете также вспомнить, что непосредственным дочерним элементом производного от ContentControl класса является предполагаемое содержимое, а потому при указании сложного содержимого определять область Button.Content явно не требуется. Можно было бы написать такую разметку:

```
<Button x:Name="btnMyButton" Height="121" Width="156" Click="btnMyButton_Click">
  <StackPanel Height="95" Width="128" Orientation="Vertical">
    <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
    <Label Width="59" FontSize="20" Content="Click!" Height="36" />
  </StackPanel>
</Button>
```

В любом случае свойство Content кнопки устанавливается в элемент StackPanel со связанными элементами. Создавать сложное содержимое подобного рода можно также с применением визуального конструктора Visual Studio. После определения диспетчера компоновки для элемента управления содержимым его можно выбирать в визуальном конструкторе в качестве целевого компонента, на который будут перетаскиваться внутренние элементы управления. Каждый из них можно редактировать в окне Properties. Если окно Properties использовалось для обработки события Click элемента управления Button (как было показано в предшествующих объявлениях XAML), то IDE-среда сгенерирует пустой обработчик события, куда можно будет добавить специальный код, например:

```
private void btnMyButton_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("You clicked the button!");
    // Вы щелкнули на кнопке!
}
```

Работа с окном Document Outline

В главе 24 вы узнали, что окно Document Outline (Схема документа) в Visual Studio (открываемое через меню View⇒Other Windows (Вид⇒Другие окна)) удобно при проектировании элемента управления WPF со сложным содержимым. Для создаваемого элемента Window отображается логическое дерево XAML, а щелчок на любом узле в дереве приводит к его автоматическому выбору в визуальном конструкторе и в редакторе XAML для редактирования.

В текущей версии Visual Studio окно Document Outline имеет несколько дополнительных средств, которые вы можете счесть полезными. Справа от любого узла находится значок, напоминающий глазное яблоко. Щелчок на нем позволяет скрывать или отображать элемент в визуальном конструкторе, что оказывается удобным, когда необходимо сосредоточить внимание на отдельном сегменте, подлежащем редактированию (следует отметить, что элемент будет сокрыт только на поверхности визуального конструктора, но не во время выполнения).

Справа от значка с глазным яблоком есть еще один значок, который позволяет блокировать элемент в визуальном конструкторе. Как и можно было догадаться, это удобно, когда нужно воспрепятствовать случайному изменению разметки XAML для заданного элемента вами или коллегами по разработке. На самом деле блокировка элемента делает его допускающим только чтение на этапе проектирования (что вполне очевидно не мешает изменять состояние объекта во время выполнения).

Управление компоновкой содержимого с использованием панелей

Приложение WPF неизменно содержит определенное количество элементов пользовательского интерфейса (например, элементов ввода, графического содержимого, систем меню и строк состояния), которые должны быть хорошо организованы внутри разнообразных окон. После размещения элементов пользовательского интерфейса необходимо гарантировать их запланированное поведение, когда конечный пользователь изменяет размер окна или его части (как в случае окна с разделителем). Чтобы обеспечить сохранение элементами управления WPF своих позиций внутри окна, в котором они находятся, можно задействовать множество *типов панелей* (также известных как *диспетчеры компоновки*).

По умолчанию новый WPF-элемент Window, созданный с помощью Visual Studio, будет применять диспетчер компоновки типа Grid (вскоре мы опишем его более подробно). Тем не менее, пока что рассмотрим элемент Window без каких-либо объявленных диспетчеров компоновки:

```
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">
  </Window>
```

Когда элемент управления объявляется прямо внутри окна, в котором панели не используются, он позиционируется по центру контейнера. Рассмотрим показанное далее простое объявление окна, содержащего единственный элемент управления Button. Независимо от того, как изменяются размеры окна, этот виджет пользовательского интерфейса всегда будет находиться на равном удалении от всех четырех границ клиентской области. Размер элемента Button определяется установленными значениями его свойств Height и Width.

```

<!-- Эта кнопка всегда находится в центре окна -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">
  <Button x:Name="btnOK" Height="100"
    Width="80" Content="OK"/>
</Window>

```

Также вспомните, что попытка помещения внутрь области Window сразу нескольких элементов вызовет ошибки разметки и компиляции. Причина в том, что свойству Content окна (или по существу любого потомка ContentControl) может быть присвоен только один объект. Следовательно, приведенная далее разметка XAML приведет к ошибкам разметки и компиляции:

```

<!-- Ошибка! Свойство Content неявно устанавливается более одного раза! -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">
  <!-- Ошибка! Два непосредственных дочерних элемента в <Window>! -->
  <Label x:Name="lblInstructions" Width="328" Height="25"
    FontSize="15" Content="Enter Information"/>
  <Button x:Name="btnOK" Height="100" Width="80" Content="OK"/>
</Window>

```

Понятно, что от окна, допускающего наличие только одного элемента управления, мало толку. Когда окно должно содержать несколько элементов, их потребуется расположить внутри любого числа панелей. В панель будут помещены все элементы пользовательского интерфейса, которые представляют окно, после чего сама панель выступит в качестве единственного объекта, присваиваемого свойству Content окна.

Пространство имен System.Windows.Controls предлагает многочисленные панели, каждая из которых по-своему обслуживает внутренние элементы. С помощью панелей можно устанавливать поведение элементов управления при изменении размеров окна пользователем — будут они оставаться в те же местах, где были размещены на этапе проектирования, располагаться свободным потоком слева направо или сверху вниз и т.д.

Элементы управления типа панелей также разрешено помещать внутрь других панелей (например, элемент управления DockPanel может содержать StackPanel со своими элементами), чтобы обеспечить высокую гибкость и степень управления.

В табл. 25.2 кратко описаны некоторые распространенные элементы управления типа панелей WPF.

В последующих нескольких разделах вы узнаете, как применять распространенные типы панелей, копируя заранее определенную разметку XAML в приложение kaхaml.exe, которое было установлено в главе 24. Все необходимые файлы XAML находятся в подкаталоге PanelMarkup внутри Chapter_25. Во время работы с Kaхaml для эмуляции изменения размеров окна нужно изменить высоту или ширину элемента Page в разметке.

Позиционирование содержимого внутри панелей Canvas

При наличии опыта работы с Windows Forms панель Canvas вероятно покажется наиболее привычной, т.к. она делает возможным абсолютное позиционирование содержимого пользовательского интерфейса.

Таблица 25.2. Основные элементы управления типа панелей WPF

Элемент управления типа панели	Описание
Canvas	Предоставляет классический режим размещения содержимого. Элементы остаются в точности там, куда были помещены на этапе проектирования
DockPanel	Привязывает содержимое к указанной стороне панели (Top (верхняя), Bottom (нижняя), Left (левая) или Right (правая))
Grid	Располагает содержимое внутри серии ячеек, поддерживаемых внутри табличной сетки
StackPanel	Укладывает содержимое вертикально или горизонтально, как регламентируется свойством Orientation
WrapPanel	Позиционирует содержимое слева направо, перенося его на следующую строку по достижении границы панели. Дальнейшее упорядочение происходит последовательно сверху вниз или слева направо в зависимости от значения свойства Orientation

Если конечный пользователь изменяет размер окна, делая его меньше, чем размер компоновки, обслуживаемой панелью Canvas, то внутреннее содержимое будет невидимым до тех пор, пока контейнер не увеличится до размера, равного или превышающего размер области Canvas.

Чтобы добавить содержимое к Canvas, сначала понадобится определить требуемые элементы управления внутри области между открывающим и закрывающим дескрипторами Canvas. Затем для каждого элемента управления необходимо указать левый верхний угол с использованием свойств Canvas.Top и Canvas.Left; именно здесь должна начинаться визуализация. Нижний правый угол каждого элемента управления можно задать неявно, устанавливая свойства Canvas.Height и Canvas.Width, либо явно с применением свойств Canvas.Right и Canvas.Bottom.

Для демонстрации Canvas в действии откроем готовый файл SimpleCanvas.xaml в kaxaml.exe. Определение Canvas должно иметь следующий вид (в случае загрузки примеров в приложение WPF дескриптор Page нужно будет заменить дескриптором Window):

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">
  <Canvas Background="LightSteelBlue">
    <Button x:Name="btnOK" Canvas.Left="212" Canvas.Top="203"
      Width="80" Content="OK"/>
    <Label x:Name="lblInstructions" Canvas.Left="17" Canvas.Top="14"
      Width="328" Height="25" FontSize="15"
      Content="Enter Car Information"/>
    <Label x:Name="lblMake" Canvas.Left="17" Canvas.Top="60"
      Content="Make"/>
    <TextBox x:Name="txtMake" Canvas.Left="94" Canvas.Top="60"
      Width="193" Height="25"/>
    <Label x:Name="lblColor" Canvas.Left="17" Canvas.Top="109"
      Content="Color"/>
    <TextBox x:Name="txtColor" Canvas.Left="94" Canvas.Top="107"
      Width="193" Height="25"/>
  </Canvas>
</Page>
```

```

<Label x:Name="lblPetName" Canvas.Left="17" Canvas.Top="155"
      Content="Pet Name"/>
<TextBox x:Name="txtPetName" Canvas.Left="94" Canvas.Top="153"
        Width="193" Height="25"/>
</Canvas>
</Page>

```

В верхней половине экрана отобразится окно, показанное на рис. 25.1.

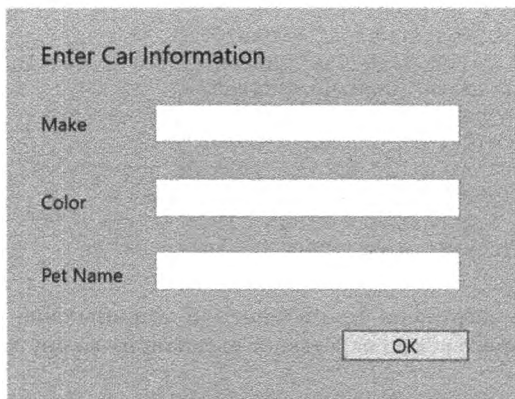


Рис. 25.1. Диспетчер компоновки Canvas делает возможным абсолютное позиционирование содержимого

Обратите внимание, что порядок объявления элементов содержимого внутри Canvas не влияет на расчет местоположения; на самом деле местоположение основано на размере элемента управления и значениях его свойств `Canvas.Top`, `Canvas.Bottom`, `Canvas.Left` и `Canvas.Right`.

На заметку! Если подэлементы внутри Canvas не определяют специфическое местоположение с использованием синтаксиса присоединяемых свойств (например, `Canvas.Left` и `Canvas.Top`), тогда они автоматически прикрепляются к верхнему левому углу Canvas.

Применение типа Canvas может показаться предпочтительным способом организации содержимого (т.к. он выглядит настолько знакомым), но данному подходу присущи некоторые ограничения. Во-первых, элементы внутри Canvas не изменяют свои размеры динамически при использовании стилей или шаблонов (скажем, их шрифты остаются незатронутыми). Во-вторых, панель Canvas не пытается сохранять элементы видимыми, когда конечный пользователь уменьшает размер окна.

Пожалуй, наилучшим применением типа Canvas является позиционирование *графического содержимого*. Например, при построении изображения с использованием XAML определенно понадобится сделать так, чтобы все линии, фигуры и текст оставались на своих местах, а не динамически перемещались в случае изменения пользователем размера окна. Мы еще вернемся к Canvas в главе 26 при обсуждении служб визуализации графики WPF.

Позиционирование содержимого внутри панелей `WrapPanel`

Панель `WrapPanel` позволяет определять содержимое, которое будет протекать сквозь панель, когда размер окна изменяется. При позиционировании элементов внутри `WrapPanel` их координаты верхнего левого и нижнего правого углов не указываются,

как обычно делается в Canvas. Однако для каждого подэлемента допускается определение значений свойств Height и Width (наряду с другими свойствами), чтобы управлять их общим размером в контейнере.

Поскольку содержимое внутри WrapPanel не пристыковывается к заданной стороне панели, порядок объявления элементов играет важную роль (содержимое визуализируется от первого элемента до последнего). В файле SimpleWrapPanel.xaml находится следующая разметка (заключенная внутри определения Page):

```
<WrapPanel Background="LightSteelBlue">
  <Label x:Name="lblInstruction" Width="328"
    Height="25" FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" Content="Make"/>
  <TextBox x:Name="txtMake" Width="193" Height="25"/>
  <Label x:Name="lblColor" Content="Color"/>
  <TextBox x:Name="txtColor" Width="193" Height="25"/>
  <Label x:Name="lblPetName" Content="Pet Name"/>
  <TextBox x:Name="txtPetName" Width="193" Height="25"/>
  <Button x:Name="btnOK" Width="80" Content="OK"/>
</WrapPanel>
```

Когда эта разметка загружена, при изменении ширины окна содержимое выглядит не особо привлекательно, т.к. оно перетекает слева направо внутри окна (рис. 25.2).

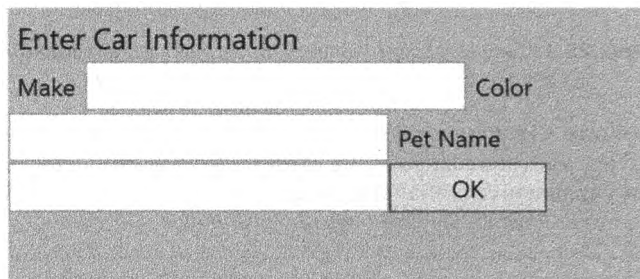


Рис. 25.2. Содержимое в панели WrapPanel ведет себя во многом подобно традиционной странице HTML

По умолчанию содержимое WrapPanel перетекает слева направо. Тем не менее, если изменить значение свойства Orientation на Vertical, то можно заставить содержимое перетекать сверху вниз:

```
<WrapPanel Background="LightSteelBlue" Orientation="Vertical">
```

Панель WrapPanel (как и ряд других типов панелей) может быть объявлена с указанием значений ItemWidth и ItemHeight, которые управляют стандартным размером каждого элемента. Если подэлемент предоставляет собственные значения Height и/или Width, то он будет позиционироваться относительно размера, установленного для него панелью. Взгляните на следующую разметку:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="100" Width="650">
  <WrapPanel Background="LightSteelBlue" Orientation="Horizontal"
    ItemWidth="200" ItemHeight="30">
    <Label x:Name="lblInstruction" FontSize="15"
      Content="Enter Car Information"/>
```

```

<Label x:Name="lblMake" Content="Make"/>
<TextBox x:Name="txtMake"/>
<Label x:Name="lblColor" Content="Color"/>
<TextBox x:Name="txtColor"/>
<Label x:Name="lblPetName" Content="Pet Name"/>
<TextBox x:Name="txtPetName"/>
<Button x:Name="btnOK" Width="80" Content="OK"/>
</WrapPanel>
</Page>

```

В результате визуализации получается окно, показанное на рис. 25.3 (обратите внимание на размер и позицию элемента управления Button, для которого было задано уникальное значение Width).

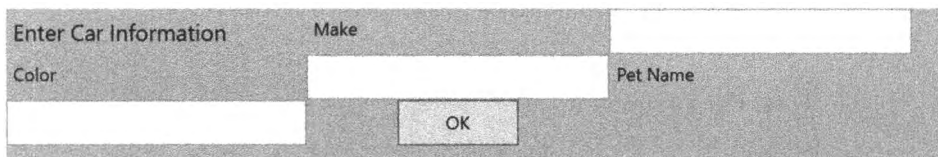


Рис. 25.3. Панель WrapPanel может устанавливать ширину и высоту отдельного элемента

После просмотра рис. 25.3 вы наверняка согласитесь с тем, что панель WrapPanel — обычно не лучший выбор для организации содержимого непосредственно в окне, поскольку ее элементы могут беспорядочно смешиваться, когда пользователь изменяет размер окна. В большинстве случаев WrapPanel будет подэлементом панели другого типа, позволяя небольшой области окна переносить свое содержимое при изменении размера (как, например, элемент управления ToolBar).

Позиционирование содержимого внутри панелей StackPanel

Подобно WrapPanel элемент управления StackPanel организует содержимое внутри одиночной строки, которая может быть ориентирована горизонтально или вертикально (по умолчанию) в зависимости от значения, присвоенного свойству Orientation. Однако отличие между ними заключается в том, что StackPanel не пытается переносить содержимое при изменении размера окна пользователем. Взамен элементы в StackPanel просто растягиваются (согласно выбранной ориентации), приспосабливаясь к размеру самой панели StackPanel. Например, в файле SimpleStackPanel.xaml содержится разметка, которая в результате дает вывод, показанный на рис. 25.4:

```

<StackPanel Background="LightSteelBlue">
  <Label x:Name="lblInstruction"
    FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" Content="Make"/>
  <TextBox Name="txtMake"/>
  <Label x:Name="lblColor" Content="Color"/>
  <TextBox x:Name="txtColor"/>
  <Label x:Name="lblPetName" Content="Pet Name"/>
  <TextBox x:Name="txtPetName"/>
  <Button x:Name="btnOK" Width="80" Content="OK"/>
</StackPanel>

```

Если присвоить свойству Orientation значение Horizontal, тогда визуализированный вывод станет таким, как на рис. 25.5:

```

<StackPanel Background="LightSteelBlue" Orientation="Horizontal">

```

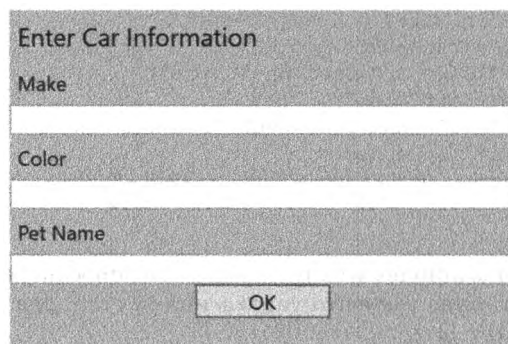


Рис. 25.4. Вертикальное укладывание содержимого

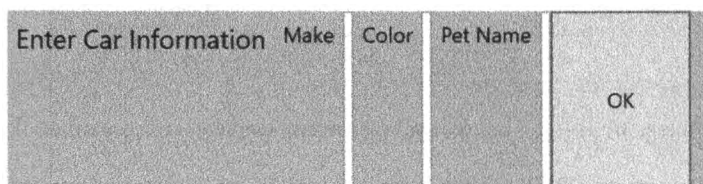


Рис. 25.5. Горизонтальное укладывание содержимого

Подобно `WrapPanel` панель `StackPanel` тоже редко применяется для организации содержимого прямо внутри окна. Панель `StackPanel` должна использоваться как вложенная панель в какой-то главной панели.

Позиционирование содержимого внутри панелей `Grid`

Из всех панелей, предоставляемых API-интерфейсами WPF, панель `Grid`, несомненно, является самой гибкой. Аналогично таблице HTML панель `Grid` может состоять из набора ячеек, каждая из которых имеет свое содержимое. При определении `Grid` выполняются перечисленные ниже шаги.

1. Определение и конфигурирование каждой колонки.
2. Определение и конфигурирование каждой строки.
3. Назначение содержимого каждой ячейке сетки с применением синтаксиса присоединяемых свойств.

На заметку! Если не определить какие-либо строки и колонки, то по умолчанию элемент `Grid` будет состоять из единственной ячейки, которая заполняет всю поверхность окна. Кроме того, если не установить ячейку (колонку и строку) для подэлемента внутри `Grid`, тогда он автоматически разместится в колонке 0 и строке 0.

Первые два шага (определение колонок и строк) выполняются с использованием элементов `Grid.ColumnDefinitions` и `Grid.RowDefinitions`, которые содержат коллекции элементов `ColumnDefinition` и `RowDefinition` соответственно. Каждая ячейка внутри сетки на самом деле является настоящим объектом .NET, так что можно желаемым образом настраивать внешний вид и поведение каждого элемента.

Ниже представлено простое определение `Grid` (из файла `SimpleGrid.xaml`), которое организует содержимое пользовательского интерфейса, как показано на рис. 25.6:

```

<Grid ShowGridLines="True" Background="LightSteelBlue">
  <!-- Определить строки и колонки -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>

  <!-- Добавить элементы в ячейки сетки -->
  <Label x:Name="lblInstruction" Grid.Column="0" Grid.Row="0"
    FontSize="15" Content="Enter Car Information"/>
  <Button x:Name="btnOK" Height="30" Grid.Column="0"
    Grid.Row="0" Content="OK"/>
  <Label x:Name="lblMake" Grid.Column="1" Grid.Row="0" Content="Make"/>
  <TextBox x:Name="txtMake" Grid.Column="1"
    Grid.Row="0" Width="193" Height="25"/>
  <Label x:Name="lblColor" Grid.Column="0" Grid.Row="1" Content="Color"/>
  <TextBox x:Name="txtColor" Width="193" Height="25"
    Grid.Column="0" Grid.Row="1" />

  <!--Добавить цвет к ячейке с именем, просто чтобы сделать картину интереснее-->
  <Rectangle Fill="LightGreen" Grid.Column="1" Grid.Row="1" />
  <Label x:Name="lblPetName" Grid.Column="1" Grid.Row="1" Content="Pet Name"/>
  <TextBox x:Name="txtPetName" Grid.Column="1" Grid.Row="1"
    Width="193" Height="25"/>
</Grid>

```

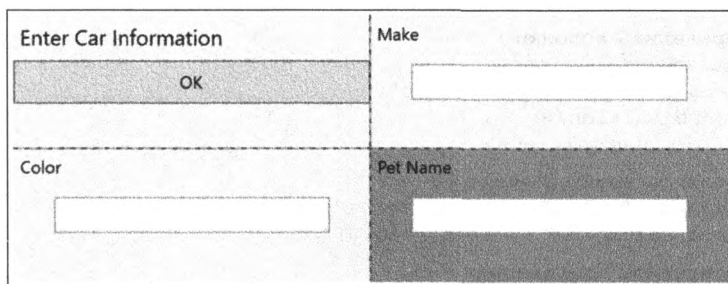


Рис. 25.6. Панель Grid в действии

Обратите внимание, что каждый элемент (включая элемент Rectangle светло-зеленого цвета) прикрепляется к ячейке сетки с применением присоединяемых свойств `Grid.Row` и `Grid.Column`. По умолчанию порядок ячеек начинается с левой верхней ячейки, которая указывается с использованием `Grid.Column="0"` и `Grid.Row="0"`. Учитывая, что сетка состоит всего из четырех ячеек, правая нижняя ячейка может быть идентифицирована как `Grid.Column="1"` и `Grid.Row="1"`.

Установка размеров столбцов и строк в панели Grid

Задавать размеры столбцов и строк в панели Grid можно одним из трех способов:

- установка абсолютных размеров (например, 100);
- установка автоматических размеров;
- установка относительных размеров (например, 3*).

Установка абсолютных размеров — именно то, что и можно было ожидать; для размера колонки (или строки) указывается специфическое число единиц, независимых от устройства. При установке автоматических размеров размер каждой колонки или строки определяется на основе элементов управления, содержащихся в колонке или строке. Установка относительных размеров практически эквивалентна заданию размеров в процентах внутри стиля CSS. Общая сумма чисел в колонках или строках с относительными размерами распределяется на общий объем доступного пространства.

В следующем примере первая строка получает 25% пространства, а вторая — 75% пространства:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="1*" />
  <ColumnDefinition Width="3*" />
</Grid.ColumnDefinitions>
```

Панели Grid с типами GridSplitter

Панели Grid также способны поддерживать *разделители*. Как вам возможно известно, разделители позволяют конечному пользователю изменять размеры колонок и строк сетки. При этом содержимое каждой ячейки с изменяемым размером реорганизуется на основе находящихся в нем элементов. Добавить разделители к Grid довольно просто: необходимо определить элемент управления GridSplitter и с применением синтаксиса присоединяемых свойств указать строку или колонку, на которую он воздействует.

Имейте в виду, что для того, чтобы разделитель был виден на экране, потребуется присвоить значение его свойству Width или Height (в зависимости от вертикального или горизонтального разделения). Ниже показана простая панель Grid с разделителем на первой колонке (Grid.Column="0") из файла GridWithSplitter.xaml:

```
<Grid Background="LightSteelBlue">
  <!-- Определить колонки -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>

  <!-- Добавить метку в ячейку 0 -->
  <Label x:Name="lblLeft" Background="GreenYellow"
    Grid.Column="0" Content="Left!"/>

  <!-- Определить разделитель -->
  <GridSplitter Grid.Column="0" Width="5"/>

  <!-- Добавить метку в ячейку 1 -->
  <Label x:Name="lblRight" Grid.Column="1" Content="Right!"/>
</Grid>
```

Прежде всего, обратите внимание, что колонка, которая будет поддерживать разделитель, имеет свойство Width, установленное в Auto. Вдобавок элемент GridSplitter использует синтаксис присоединяемых свойств для указания, с какой колонкой он работает. В выводе (рис. 25.7) можно заметить 5-пиксельный разделитель, который позволяет изменять размер каждого элемента Label. Из-за того, что для элементов Label не было задано свойство Height или Width, они заполняют всю ячейку.

Позиционирование содержимого внутри панелей DockPanel

Панель DockPanel обычно применяется в качестве контейнера, который содержит любое количество дополнительных панелей для группирования связанного содержимого.

го. Панели `DockPanel` используют синтаксис присоединяемых свойств (как было показано в типах `Canvas` и `Grid`) для управления местом, куда будет пристыковываться каждый элемент внутри `DockPanel`.

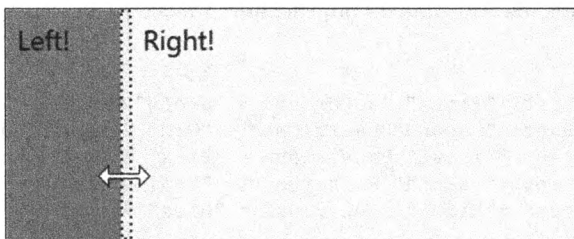


Рис. 25.7. Панель `Grid` с разделителем

В файле `SimpleDockPanel.xaml` определена следующая простая панель `DockPanel`, которая дает результат, показанный на рис. 25.8:

```
<DockPanel LastChildFill="True" Background="AliceBlue">
  <'-- Стыковать элементы к панели -->
  <Label DockPanel.Dock="Top" Name="lblInstruction" FontSize="15"
    Content="Enter Car Information"/>
  <Label DockPanel.Dock="Left" Name="lblMake" Content="Make"/>
  <Label DockPanel.Dock="Right" Name="lblColor" Content="Color"/>
  <Label DockPanel.Dock="Bottom" Name="lblPetName" Content="Pet Name"/>
  <Button Name="btnOK" Content="OK"/>
</DockPanel>
```

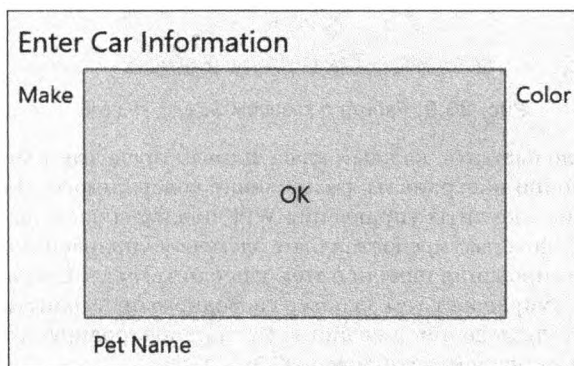


Рис. 25.8. Простая панель `DockPanel`

На заметку! Если добавить множество элементов к одной стороне `DockPanel`, то они выстроятся вдоль указанной грани в порядке их объявления.

Преимущество применения типов `DockPanel` заключается в том, что при изменении пользователем размера окна каждый элемент остается прикрепленным к указанной (посредством `DockPanel.Dock`) стороне панели. Также обратите внимание, что внутри открывающего дескриптора `DockPanel` в этом примере атрибут `LastChildFill` установлен в `true`. Поскольку элемент `Button` на самом деле является “последним дочерним” элементом в контейнере, он будет растянут, чтобы занять все оставшееся пространство.

Включение прокрутки в типах панелей

Полезно упомянуть, что в рамках инфраструктуры WPF поставляется класс `ScrollViewer`, который обеспечивает автоматическое поведение прокрутки данных внутри объектов панелей. Вот как он определяется в файле `SimpleScrollViewer.xaml`:

```
<ScrollViewer>
  <StackPanel>
    <Button Content ="First" Background = "Green" Height ="40"/>
    <Button Content ="Second" Background = "Red" Height ="40"/>
    <Button Content ="Third" Background = "Pink" Height ="40"/>
    <Button Content ="Fourth" Background = "Yellow" Height ="40"/>
    <Button Content ="Fifth" Background = "Blue" Height ="40"/>
  </StackPanel>
</ScrollViewer>
```

Результат визуализации приведенного определения XAML представлен на рис. 25.9 (обратите внимание на то, что справа в окне отображается линейка прокрутки, т.к. размера окна не хватает, чтобы показать все пять кнопок).

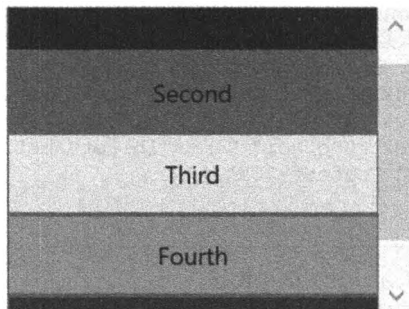


Рис. 25.9. Работа с классом `ScrollViewer`

Как и можно было ожидать, каждый класс панели предлагает многочисленные члены, позволяющие точно настраивать размещение содержимого. В качестве связанного замечания: многие элементы управления WPF поддерживают два удобных свойства (`Padding` и `Margin`), которые предоставляют элементу управления возможность самостоятельного информирования панели о том, как с ним следует обращаться. В частности, свойство `Padding` управляет тем, сколько свободного пространства должно окружать внутренний элемент управления, а свойство `Margin` контролирует объем дополнительного пространства вне элемента управления.

На этом краткий экскурс в основные типы панелей WPF и различные способы позиционирования их содержимого завершен. Далее мы покажем, как использовать визуальные конструкторы Visual Studio для создания компоновок.

Конфигурирование панелей с использованием визуальных конструкторов Visual Studio

Теперь, когда вы ознакомились с разметкой XAML, применяемой при определении ряда общих диспетчеров компоновки, полезно знать, что IDE-среда Visual Studio предлагает очень хорошую поддержку для конструирования компоновок. Ключевым компонентом является окно Document Outline, описанное ранее в главе. Чтобы проиллюстрировать некоторые основы, мы создадим новый проект приложения WPF по имени `VisualLayoutTester`.

В первоначальной разметке для Window по умолчанию используется диспетчер компоновки Grid:

```
<Window x:Class="VisualLayoutTester.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:VisualLayoutTesterApp"
  mc:Ignorable="d" Title="MainWindow" Height="350" Width="525">
  <Grid>
  </Grid>
</Window>
```

Если вы благополучно применяете систему компоновки Grid, то на рис. 25.10 заметите, что можно легко разделять и менять размеры ячеек сетки, используя визуальный конструктор. Сначала необходимо выбрать компонент Grid в окне Document Outline и затем щелкнуть на границе сетки, чтобы создать новые строки и колонки.

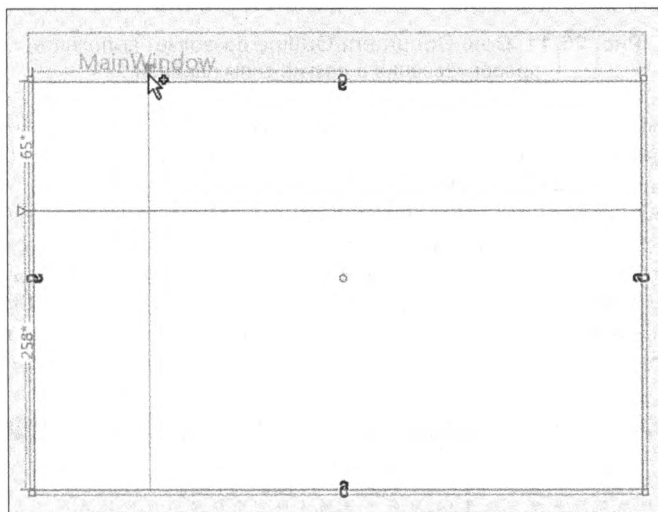


Рис. 25.10. Элемент управления Grid может быть разделен на ячейки с применением визуального конструктора IDE-среды

Теперь предположим, что определена сетка с каким-то числом ячеек. Далее можно перетаскивать элементы управления в интересующую ячейку сетки и IDE-среда будет автоматически устанавливать их свойства Grid.Row и Grid.Column. Вот как может выглядеть разметка, сгенерированная IDE-средой после перетаскивания элемента Button в предопределенную ячейку:

```
<Button x:Name="button" Content="Button" Grid.Column="1" HorizontalAlignment="Left"
  Margin="21,21.4,0,0" Grid.Row="1" VerticalAlignment="Top" Width="75"/>
```

Пусть, например, было решено вообще не использовать элемент Grid. Щелчок правой кнопкой мыши на любом узле разметки в окне Document Outline приводит к открытию контекстного меню, которое содержит пункт, позволяющий заменить текущий контейнер другим (рис. 25.11). Следует осознавать, что такое действие (с высокой вероятностью) радикально изменит позиции элементов управления, потому что они станут удовлетворять правилам нового типа панели.

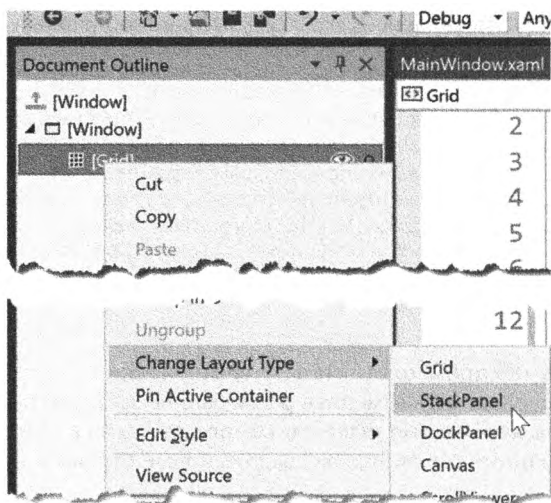


Рис. 25.11. Окно Document Outline позволяет выполнять преобразование в другие типы панелей

Еще один удобный трюк связан с возможностью выбора в визуальном конструкторе набора элементов управления и последующего их группирования внутри нового вложенного диспетчера компоновки. Предположим, что имеется панель Canvas, в которой определен набор произвольных объектов (при желании первоначальную панель Grid можно преобразовать в Canvas с применением приема, продемонстрированного на рис. 25.11). Выделим множество элементов на поверхности визуального конструктора, щелкая на каждом элементе левой кнопкой мыши при нажатой клавише <Ctrl>. Если затем щелкнуть правой кнопкой мыши на выделенном наборе элементов, то с помощью открывшегося контекстного меню их можно сгруппировать в новую вложенную панель (рис. 25.12).

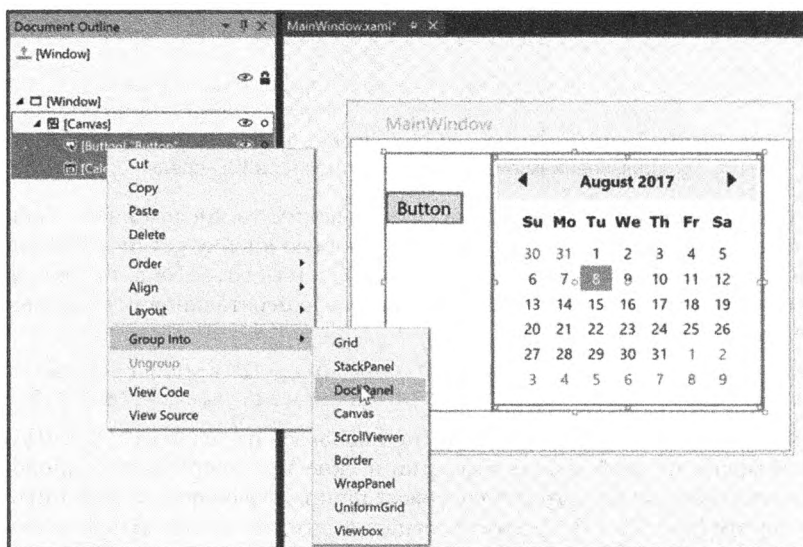


Рис. 25.12. Группирование элементов в новую вложенную панель

После этого снова нужно заглянуть в окно Document Outline, чтобы проконтролировать вложенную систему компоновки. Так как строятся полнофункциональные окна WPF, скорее всего, всегда нужно будет использовать систему вложенных компоновок, а не просто выбирать единственную панель для отображения всего пользовательского интерфейса (фактически в оставшихся примерах приложений WPF обычно так и будет делаться). В качестве финального замечания следует указать, что все узлы в окне Document Outline поддерживают перетаскивание. Например, если требуется переместить в родительскую панель элемент управления, который в текущий момент находится внутри Canvas, тогда можно поступить так, как иллюстрируется на рис. 25.13.

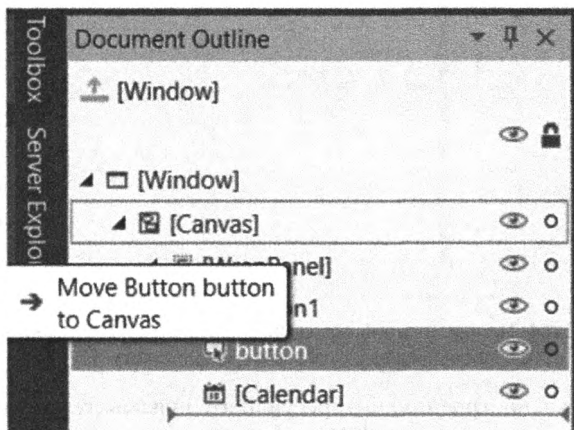


Рис. 25.13. Перемещение элементов с помощью окна Document Outline

В последующих главах, посвященных WPF, будут представлены дополнительные усовершенствованные приемы для работы с компоновкой там, где они возможны. Тем не менее, определенно полезно посвятить какое-то время самостоятельному экспериментированию и проверке разнообразных средств. В следующем примере данной главы будет демонстрироваться построение вложенного диспетчера компоновки для специального приложения обработки текста (с проверкой правописания).

Построение окна с использованием вложенных панелей

Как упоминалось ранее, в типичном окне WPF для получения желаемой системы компоновки применяется не единственный элемент управления типа панели, а одни панели вкладываются внутрь других. Начнем с создания нового проекта приложения WPF по имени MyWordPad.

Нашей целью является конструирование компоновки, в которой главное окно имеет расположенную в верхней части систему меню, под ней — панель инструментов и в нижней части окна — строку состояния. Строка состояния будет содержать область для текстовых подсказок, которые отображаются при выборе пользователем пункта меню (или кнопки в панели инструментов). Система меню и панель инструментов предоставят триггеры пользовательского интерфейса для закрытия приложения и отображения вариантов правописания в виджете Expander. На рис. 25.14 показана начальная компоновка; она также иллюстрирует возможности правописания в рамках инфраструктуры WPF.

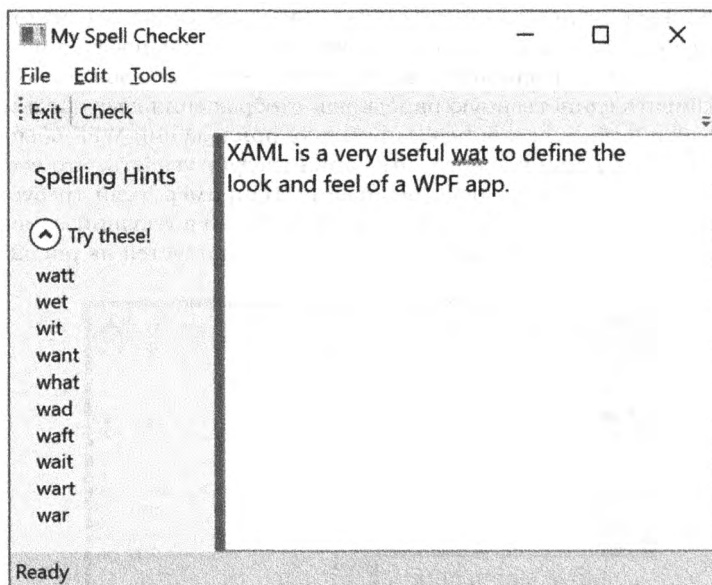


Рис. 25.14. Использование вложенных панелей для формирования пользовательского интерфейса окна

Чтобы приступить к построению интересующего пользовательского интерфейса, модифицируем начальное определение XAML типа Window для использования дочернего элемента DockPanel вместо стандартного элемента управления Grid:

```
<Window x:Class="MyWordPad.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:MyWordPad"
  mc:Ignorable="d"
  Title="My Spell Checker" Height="350" Width="525">
  <!-- Эта панель устанавливает содержимое окна -->
  <DockPanel>
  </DockPanel>
</Window>
```

Построение системы меню

Системы меню в WPF представлены классом Menu, который поддерживает коллекцию объектов MenuItem. При построении системы меню в XAML каждый объект MenuItem можно заставить обрабатывать разнообразные события, наиболее примечательным из которых является Click, возникающее при выборе подэлемента конечным пользователем. В рассматриваемом примере создаются два пункта меню верхнего уровня (File (Файл) и Tools (Сервис); позже будет построено меню Edit (Правка)), которые содержат в себе подэлементы Exit (Выход) и Spelling Hints (Подсказки по правописанию) соответственно.

В дополнение к обработке события Click для каждого подэлемента необходимо также обработать события MouseEnter и MouseExit, которые применяются для установки текста в строке состояния. Добавим в контекст элемента DockPanel следующую разметку:

```

<!-- Стыковать систему меню к верхней части -->
<Menu DockPanel.Dock="Top"
      HorizontalAlignment="Left" Background="White" BorderBrush="Black">
  <MenuItem Header="_File">
    <Separator/>
    <MenuItem Header="_Exit" MouseEnter="MouseEnterExitArea"
              MouseLeave="MouseLeaveArea" Click="FileExit_Click"/>
  </MenuItem>
  <MenuItem Header="_Tools">
    <MenuItem Header="_Spelling Hints"
              MouseEnter="MouseEnterToolsHintsArea"
              MouseLeave="MouseLeaveArea" Click="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>

```

Обратите внимание, что система меню стыкована с верхней частью DockPanel. Кроме того, элемент Separator применяется для добавления в систему меню тонкой горизонтальной линии прямо перед пунктом Exit. Значения Header для каждого MenuItem содержат символ подчеркивания (например, _Exit). Подобным образом указывается символ, который будет подчеркиваться, когда конечный пользователь нажмет клавишу <Alt> (для ввода клавиатурного сокращения). Символ подчеркивания используется вместо символа & в Windows Forms, т.к. язык XAML основан на XML, а символ & в XML имеет особый смысл.

После построения системы меню необходимо реализовать различные обработчики событий. Прежде всего, есть обработчик пункта меню File⇒Exit (Файл⇒Выход), FileExit_Click(), который просто закрывает окно, что в свою очередь приводит к завершению приложения, поскольку это окно самого высшего уровня. Обработчики событий MouseEnter и MouseExit для каждого подэлемента будут в итоге обновлять строку состояния; однако пока мы просто оставим их пустыми. Наконец, обработчик ToolsSpellingHints_Click() для пункта меню Tools⇒Spelling Hints также оставим пока пустым. Ниже показаны текущие обновления файла отдельного кода (в том числе добавленные операторы using для пространств имен Microsoft.Win32 и System.IO):

```

using Microsoft.Win32;
using System.IO;

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    protected void FileExit_Click(object sender, RoutedEventArgs args)
    {
        // Закрыть это окно.
        this.Close();
    }
    protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
    {
    }
    protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
    {
    }
    protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args)
    {
    }
}

```

```
protected void MouseLeaveArea(object sender, RoutedEventArgs args)
{
}
}
```

Визуальное построение меню

Наряду с тем, что всегда полезно знать, как вручную определять элементы в XAML, такая работа может быть слегка утомительной. В Visual Studio поддерживается возможность визуального конструирования систем меню, панелей инструментов, строк состояния и многих других элементов управления пользовательского интерфейса. Щелчок правой кнопкой мыши на элементе управления Menu приводит к открытию контекстного меню, содержащего Add MenuItem (Добавить MenuItem), который позволяет добавить новый пункт меню в элемент управления Menu. После добавления набора пунктов верхнего уровня можно заняться добавлением пунктов подменю, разделителей, разворачиванием и свертыванием самого меню и выполнением других связанных с меню операций посредством второго щелчка правой кнопкой мыши.

В оставшейся части примера MyWordPad вы увидите финальную сгенерированную разметку XAML; тем не менее, посвятите некоторое время экспериментированию с визуальными конструкциями.

Построение панели инструментов

Панели инструментов (представляемые в WPF классом ToolBar) обычно предлагают альтернативный способ активизации пунктов меню. Поместите следующую разметку непосредственно после закрывающего дескриптора определения Menu:

```
<!-- Поместить панель инструментов под область меню -->
<ToolBar DockPanel.Dock = "Top" >
  <Button Content = "Exit" MouseEnter = "MouseEnterExitArea"
    MouseLeave = "MouseLeaveArea" Click = "FileExit_Click"/>
  <Separator/>
  <Button Content = "Check" MouseEnter = "MouseEnterToolsHintsArea"
    MouseLeave = "MouseLeaveArea" Click = "ToolsSpellingHints_Click"
    Cursor = "Help" />
</ToolBar>
```

Наш элемент управления ToolBar образован из двух элементов управления Button, которые предназначены для обработки тех же самых событий теми же методами из файла кода. С помощью такого приема можно дублировать обработчики для обслуживания и пунктов меню, и кнопок панели инструментов. Хотя в данной панели применяются типичные нажимаемые кнопки, вы должны принимать во внимание, что тип ToolBar "является" ContentControl, а потому на его поверхность можно помещать любые типы (скажем, раскрывающиеся списки, изображения и графику). Еще один интересный аспект связан с тем, что кнопка Check (Проверить) поддерживает специальный курсор мыши через свойство Cursor.

На заметку! Элемент ToolBar может быть дополнительно помещен внутрь элемента ToolBar Tray, который управляет компоновкой, стыковкой и перетаскиванием для набора объектов ToolBar. За подробной информацией обращайтесь в документацию .NET Framework 4.7 SDK.

Построение строки состояния

Элемент управления строкой состояния (StatusBar) стыкуется с нижней частью DockPanel и содержит единственный элемент управления TextBlock, который ранее

в главе не использовался. Элемент `TextBlock` можно применять для хранения текста с форматированием вроде выделения полужирным и подчеркиванием, добавления разрывов строк и т.д. Поместим приведенную ниже разметку сразу после предыдущего определения элемента управления `ToolBar`:

```
<!-- Разместить строку состояния внизу -->
<StatusBar DockPanel.Dock="Bottom" Background="Beige" >
  <StatusBarItem>
    <TextBlock Name="statBarText" Text="Ready" />
  </StatusBarItem>
</StatusBar>
```

Завершение проектирования пользовательского интерфейса

Финальный аспект проектирования нашего пользовательского интерфейса связан с определением поддерживающего разделителя элемента `Grid`, в котором определены две колонки. Слева находится элемент управления `Expander`, помещенный внутрь `StackPanel`, который будет отображать список предполагаемых вариантов правописания, а справа — элемент `TextBox` с поддержкой многострочного текста, линеек прокрутки и включенной проверкой орфографии. Элемент `Grid` может быть целиком размещен в левой части родительской панели `DockPanel`. Чтобы завершить определение пользовательского интерфейса окна, добавим следующую разметку XAML, расположив ее непосредственно под разметкой, которая описывает `StatusBar`:

```
<Grid DockPanel.Dock="Left" Background="AliceBlue">
  <!-- Определить строки и колонки -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <GridSplitter Grid.Column="0" Width="5" Background="Gray" />
  <StackPanel Grid.Column="0" VerticalAlignment="Stretch" >
    <Label Name="lblSpellingInstructions" FontSize="14" Margin="10,10,0,0">
      Spelling Hints
    </Label>

    <Expander Name="expanderSpelling" Header="Try these!"
      Margin="10,10,10,10">
      <!-- Будет заполняться программно -->
      <Label Name="lblSpellingHints" FontSize="12" />
    </Expander>
  </StackPanel>

  <!-- Это будет областью для ввода -->
  <TextBox Grid.Column="1"
    SpellCheck.IsEnabled="True"
    AcceptsReturn="True"
    Name="txtData" FontSize="14"
    BorderBrush="Blue"
    VerticalScrollBarVisibility="Auto"
    HorizontalScrollBarVisibility="Auto">
  </TextBox>
</Grid>
```

Реализация обработчиков событий `MouseEnter/MouseLeave`

К настоящему моменту пользовательский интерфейс окна готов. Понадобится лишь предоставить реализации оставшихся обработчиков событий. Начнем с обновления файла кода C# так, чтобы каждый из обработчиков событий `MouseEnter` и `MouseLeave` устанавливал в текстовой панели строки состояния подходящее сообщение, которое окажет помощь конечному пользователю:

```
public partial class MainWindow : System.Windows.Window
{
    ...
    protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Exit the Application";
    }
    protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Show Spelling Suggestions";
    }
    protected void MouseLeaveArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Ready";
    }
}
```

Теперь приложение можно запустить. Текст в строке состояния должен изменяться в зависимости от того, над каким пунктом меню или кнопкой панели инструментов находится курсор.

Реализация логики проверки правописания

Инфраструктура WPF имеет встроенную поддержку проверки правописания, независимую от продуктов Microsoft Office. Это значит, что использовать уровень взаимодействия с COM для обращения к функции проверки правописания Microsoft Word не понадобится; та же самая функциональность добавляется с помощью всего нескольких строк кода.

Вспомните, что при определении элемента управления `TextBox` свойство `SpellCheck.IsEnabled` устанавливается в `true`. В результате неправильно написанные слова подчеркиваются красной волнистой линией, как происходит в Microsoft Office. Более того, лежащая в основе программная модель предоставляет доступ к механизму проверки правописания, который позволяет получить список предполагаемых вариантов для слов, написанных с ошибкой. Добавим в метод `ToolsSpellingHints_Click()` следующий код:

```
protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
{
    string spellingHints = string.Empty;

    // Попробовать получить ошибку правописания в текущем положении курсора ввода.
    SpellingError error = txtData.GetSpellingError(txtData.CaretIndex);
    if (error != null)
    {
        // Построить строку с предполагаемыми вариантами правописания.
        foreach (string s in error.Suggestions)
        {
            spellingHints += $"{s}\n";
        }
    }
}
```

```

// Отобразить предполагаемые варианты и раскрыть элемент Expander.
lblSpellingHints.Content = spellingHints;
expanderSpelling.IsExpanded = true;
}
}

```

Приведенный выше код довольно прост. С применением свойства `CaretIndex` извлекается объект `SpellingError` и вычисляется текущее положение курсора ввода в текстовом поле. Если в указанном месте присутствует ошибка (т.е. значение `error` не равно `null`), тогда осуществляется проход в цикле по списку предполагаемых вариантов с использованием свойства `Suggestions`. После того, как все предполагаемые варианты для неправильно написанного слова получены, они помещаются в элемент `Label` внутри элемента `Expander`.

Вот и все! С помощью нескольких строк процедурного кода (и приличной порции разметки XAML) заложены основы для функционирования текстового процессора. После изучения *управляющих команд* мы добавим дополнительные возможности.

Понятие команд WPF

Инфраструктура WPF предлагает поддержку того, что может считаться *независимыми от элементов управления событиями*, через *архитектуру команд*. Обычное событие .NET определяется внутри некоторого базового класса и может использоваться только этим классом или его потомками. Следовательно, нормальные события .NET тесно привязаны к классу, в котором они определены.

По контрасту команды WPF представляют собой похожие на события сущности, которые не зависят от специфического элемента управления и во многих случаях могут успешно применяться к многочисленным (и на вид несвязанным) типам элементов управления. Вот лишь несколько примеров: WPF поддерживает команды копирования, вырезания и вставки, которые могут использоваться в разнообразных элементах пользовательского интерфейса (вроде пунктов меню, кнопок панели инструментов и специальных кнопок), а также клавиатурные комбинации (скажем, `<Ctrl+C>` и `<Ctrl+V>`).

В то время как другие инструментальные наборы для построения пользовательских интерфейсов (вроде Windows Forms) предлагают для таких целей стандартные события, их применение обычно дает в результате избыточный и трудный в сопровождении код. Внутри модели WPF в качестве альтернативы можно использовать команды. Итогом обычно оказывается более компактная и гибкая кодовая база.

Внутренние объекты команд

Инфраструктура WPF поставляется с множеством встроенных команд, каждую из которых можно ассоциировать с соответствующей клавиатурной комбинацией (или другим входным жестом). С точки зрения программирования команда WPF — это любой объект, поддерживающий свойство (часто называемое `Command`), которое возвращает объект, реализующий показанный ниже интерфейс `ICommand`:

```

public interface ICommand
{
    // Возникает, когда происходят изменения, влияющие
    // на то, должна выполняться команда или нет.
    event EventHandler CanExecuteChanged;
    // Определяет метод, который выясняет, может ли
    // команда выполняться в ее текущем состоянии.
    bool CanExecute(object parameter);
    // Определяет метод для вызова при обращении к команде.
    void Execute(object parameter);
}

```


В WPF предлагаются разнообразные классы команд, которые открывают доступ к примерно сотне готовых объектов команд. В таких классах определены многочисленные свойства, представляющие специфические объекты команд, каждый из которых реализует интерфейс `ICommand` . В табл. 25.3 кратко описаны избранные стандартные объекты команд (более подробную информацию ищите в документации .NET Framework 4.7 SDK).

Таблица 25.3. Внутренние объекты команд WPF

Класс WPF	Объекты команд	Описание
ApplicationCommands	Close, Copy, Cut, Delete, Find, Open, Paste, Save, SaveAs, Redo, Undo	Разнообразные команды уровня приложения
ComponentCommands	MoveDown, MoveFocusBack, MoveLeft, MoveRight, ScrollToEnd, ScrollToHome	Разнообразные команды, которые являются общими для компонентов пользовательского интерфейса
MediaCommands	BoostBase, ChannelUp, ChannelDown, FastForward, NextTrack, Play, Rewind, Select, Stop	Разнообразные команды, связанные с мультимедиа
NavigationCommands	BrowseBack, BrowseForward, Favorites, LastPage, NextPage, Zoom	Разнообразные команды, связанные с навигационной моделью WPF
EditingCommands	AlignCenter, CorrectSpellingError, DecreaseFontSize, EnterLineBreak, EnterParagraphBreak, MoveDownByLine, MoveRightByWord	Разнообразные команды, связанные с интерфейсом Documents API в WPF

Подключение команд к свойству `Command`

Для подключения любого свойства команд WPF к элементу пользовательского интерфейса, который поддерживает свойство `Command` (такому как `Button` или `MenuItem`), потребуется проделать совсем небольшую работу. В качестве примера модифицируем текущую систему меню, добавив новый пункт верхнего уровня по имени `Edit` (Правка) с тремя подэлементами, которые позволяют копировать, вставлять и вырезать текстовые данные:

```
<Menu DockPanel.Dock="Top"
    HorizontalAlignment="Left"
    Background="White" BorderBrush="Black">
  <MenuItem Header="_File" Click="FileExit_Click" >
    <MenuItem Header="_Exit" MouseEnter="MouseEnterExitArea"
      MouseLeave="MouseLeaveArea" Click="FileExit_Click"/>
  </MenuItem>

  <!-- Новые пункты меню с командами -->
  <MenuItem Header="_Edit">
    <MenuItem Command="ApplicationCommands.Copy"/>
    <MenuItem Command="ApplicationCommands.Cut"/>
    <MenuItem Command="ApplicationCommands.Paste"/>
  </MenuItem>
```

```

<MenuItem Header="_Tools">
  <MenuItem Header ="_Spelling Hints"
    MouseEnter ="MouseEnterToolsHintsArea"
    MouseLeave ="MouseLeaveArea"
    Click ="ToolsSpellingHints_Click"/>
</MenuItem>
</Menu>

```

Обратите внимание, что свойству `Command` каждого подэлемента в меню `Edit` присвоено некоторое значение. В результате пункты меню автоматически получают корректные имена и горячие клавиши (например, `<Ctrl+C>` для операции вырезания) в пользовательском интерфейсе меню, и приложение теперь способно *копировать, вырезать и вставлять* текст без необходимости в написании процедурного кода.

Запустив приложение и выделив какую-то часть текста, новые пункты меню можно использовать сразу же. Вдобавок приложение также оснащено возможностью реагирования на стандартную операцию щелчка правой кнопкой мыши, предлагая пользователю те же самые пункты в контекстном меню

Подключение команд к произвольным действиям

Если объект команды нужно подключить к произвольному событию (специфичному для приложения), то придется прибегнуть к написанию процедурного кода. Задача непростая, но требует чуть больше логики, чем можно видеть в XAML. Например, пусть необходимо, чтобы все окно реагировало на нажатие клавиши `<F1>`, активизируя ассоциированную с ним справочную систему. Также предположим, что в файле кода для главного окна определен новый метод по имени `SetF1CommandBinding()`, который вызывается внутри конструктора после вызова `InitializeComponent()`:

```

public MainWindow()
{
    InitializeComponent();
    SetF1CommandBinding();
}

```

Метод `SetF1CommandBinding()` будет программно создавать новый объект `CommandBinding`, который можно применять всякий раз, когда требуется привязать объект команды к заданному обработчику событий в приложении. Сконфигурируем объект `CommandBinding` для работы с командой `ApplicationCommands.Help`, которая автоматически выдается по нажатию клавиши `<F1>`:

```

private void SetF1CommandBinding()
{
    CommandBinding helpBinding = new CommandBinding(ApplicationCommands.Help);
    helpBinding.CanExecute += CanHelpExecute;
    helpBinding.Executed += HelpExecuted;
    CommandBindings.Add(helpBinding);
}

```

Большинство объектов `CommandBinding` будет обрабатывать событие `CanExecute` (которое позволяет указать, иницируется ли команда для конкретной операции программы) и событие `Executed` (где можно определить код, который должен быть выполнен после того, как команда произошла). Добавим к нашему производному от `Window` типу следующие обработчики событий (форматы методов регламентируются ассоциированными делегатами):

```
private void CanHelpExecute(object sender, CanExecuteRoutedEventArgs e)
{
    // Если нужно предотвратить выполнение команды,
    // то можно установить CanExecute в false.
    e.CanExecute = true;
}

private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Look, it is not that difficult. Just type something!",
        "Help!");
}
```

В предыдущем фрагменте кода метод `CanHelpExecute()` реализован так, что справка по нажатию <F1> всегда разрешена; это делается путем возвращения `true`. Однако если в определенных ситуациях справочная система отображаться не должна, то необходимо предпринять соответствующую проверку и возвращать `false`. Наша “справочная система”, отображаемая внутри `HelpExecute()`, представляет собой всего лишь обычное окно сообщения. Теперь можно запустить приложение. После нажатия <F1> появляется наше окно сообщения.

Работа с командами Open и Save

Чтобы завершить текущий пример, мы добавим функциональность сохранения текстовых данных во внешнем файле и открытия файлов *.txt для редактирования. Можно пойти длинным путем, вручную добавив программную логику, которая включает и отключает пункты меню в зависимости от того, имеются ли данные внутри `TextBox`. Тем не менее, для сокращения усилий можно прибегнуть к услугам команд.

Начнем с обновления элемента `MenuItem`, который представляет меню `File` верхнего уровня, путем добавления двух новых подменю, использующих объекты `Save` и `Open` класса `ApplicationCommands`:

```
<MenuItem Header="_File">
    <MenuItem Command="ApplicationCommands.Open"/>
    <MenuItem Command="ApplicationCommands.Save"/>
    <Separator/>
    <MenuItem Header="_Exit"
        MouseEnter="MouseEnterExitArea"
        MouseLeave="MouseLeaveArea" Click="FileExit_Click"/>
</MenuItem>
```

Вспомните, что все объекты команд реализуют интерфейс `ICommand`, в котором определены два события (`CanExecute` и `Executed`). Теперь необходимо разрешить окну выполнять указанные команды, предварительно проверив возможность делать это в текущих обстоятельствах; таким образом, определим обработчик события для запуска специального кода.

Понадобится наполнить коллекцию `CommandBindings`, поддерживаемую окном. В разметке XAML потребуется применить синтаксис “свойство-элемент” для определения области `Window.CommandBindings`, в которую помещаются два определения `CommandBinding`. Модифицируем определение `Window`, как показано ниже:

```
<Window x:Class="MyWordPad.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MySpellChecker" Height="331" Width="508"
    WindowStartupLocation="CenterScreen" >
```

```

<!-- Это информирует элемент управления Window о том, какие
      обработчики вызывать при поступлении команд Open и Save -->
<Window.CommandBindings>
  <CommandBinding Command="ApplicationCommands.Open"
    Executed="OpenCmdExecuted"
    CanExecute="OpenCmdCanExecute"/>
  <CommandBinding Command="ApplicationCommands.Save"
    Executed="SaveCmdExecuted"
    CanExecute="SaveCmdCanExecute"/>
</Window.CommandBindings>
<!-- Эта панель устанавливает содержимое окна -->
<DockPanel>
  ...
</DockPanel>
</Window>

```

Щелчком правой кнопкой мыши на каждом из атрибутов Executed и CanExecute в редакторе XAML и выберем в контекстном меню пункт *Navigate to Event Handler* (Перейти к обработчику события). Как объяснялось в главе 24, в результате автоматически сгенерируется заготовка кода для обработчика события. Теперь в файле кода C# для окна должны присутствовать четыре пустых обработчика событий.

Реализация обработчиков события CanExecute будет сообщать окну, что можно инициировать соответствующие события Executed в любой момент, для чего свойство CanExecute входного объекта CanExecuteRoutedEventArgs устанавливается в true:

```

private void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}

private void SaveCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}

```

Обработчики соответствующего события Executed выполняют действительную работу по отображению диалоговых окон открытия и сохранения файла; они также управляют данными из TextBox в файл. Начнем с импортирования пространств имен System.IO и Microsoft.Win32 в файл кода. Окончательный код прямолинеен:

```

private void OpenCmdExecuted(object sender, ExecutedRoutedEventArgs e)
{
    // Создать диалоговое окно открытия файла
    // и показать в нем только текстовые файлы.
    var openDlg = new OpenFileDialog { Filter = "Text Files (*.txt)"};

    // Был ли совершен щелчок на кнопке OK?
    if (true == openDlg.ShowDialog())
    {
        // Загрузить содержимое выбранного файла.
        string dataFromFile = File.ReadAllText(openDlg.FileName);

        // Отобразить строку в TextBox.
        txtData.Text = dataFromFile;
    }
}

private void SaveCmdExecuted(object sender, ExecutedRoutedEventArgs e)
{
    var saveDlg = new SaveFileDialog { Filter = "Text Files (*.txt)"};

```

```
// Был ли совершен щелчок на кнопке ОК?
if (true == saveDlg.ShowDialog())
{
    // Сохранить данные из TextBox в указанном файле.
    File.WriteAllText(saveDlg.FileName, txtData.Text);
}
}
```

На заметку! Система команд WPF более подробно рассматривается в главе 28, где будут создаваться специальные команды на основе `ICommand` и `RelayCommands`.

Итак, пример и начальное знакомство с элементами управления WPF завершены. Вы узнали, как работать с базовыми командами, системами меню, строками состояния, панелями инструментов, вложенными панелями и несколькими основными элементами пользовательского интерфейса (вроде `TextBox` и `Expander`). Следующий пример будет иметь дело с более экзотическими элементами управления, а также с рядом важных служб WPF.

Исходный код. Проект `MyWordPad` доступен в подкаталоге `Chapter_25`.

Понятие маршрутизируемых событий

Вы могли заметить, что в предыдущем примере кода передавался параметр `RoutedEventArgs`, а не `EventArgs`. Модель маршрутизируемых событий является усовершенствованием стандартной модели событий CLR и спроектирована для того, чтобы обеспечить возможность обработки событий в манере, подходящей описанию XAML дерева объектов. Предположим, что имеется новый проект приложения WPF по имени `WPFRoutedEvents`. Модифицируем описание XAML начального окна, добавив следующий элемент управления `Button`, который определяет сложное содержимое:

```
<Button Name="btnClickMe" Height="75" Width = "250"
        Click ="btnClickMe_Clicked">
    <StackPanel Orientation ="Horizontal">
        <Label Height="50" FontSize ="20">Fancy Button!</Label>
        <Canvas Height ="50" Width ="100" >
            <Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
                Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
            <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15" Width ="36"
                Canvas.Top="17" Canvas.Left="32"/>
        </Canvas>
    </StackPanel>
</Button>
```

Обратите внимание, что в открывающем определении элемента `Button` было обработано событие `Click` за счет указания имени метода, который должен вызываться при возникновении события. Событие `Click` работает с делегатом `RoutedEventHandler`, который ожидает обработчик события, принимающий `object` в первом параметре и `System.Windows.RoutedEventArgs` во втором. Реализуем такой обработчик:

```
public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
{
    // Делать что-нибудь, когда на кнопке произведен щелчок.
    MessageBox.Show("Clicked the button");
}
```

После запуска приложения окно сообщения будет отображаться независимо от того, на какой части содержимого кнопки был выполнен щелчок (зеленый элемент `Ellipse`, желтый элемент `Ellipse`, элемент `Label` или поверхность элемента `Button`). В принципе это хорошо. Только представьте, насколько громоздким оказалась бы обработка событий WPF, если бы пришлось обрабатывать событие `Click` для каждого из упомянутых подэлементов. Дело не только в том, что создание отдельных обработчиков событий для каждого аспекта `Button` — трудоемкая задача, а еще и в том, что в результате получился бы сложный в сопровождении код.

К счастью, *маршрутизируемые события* WPF позаботятся об автоматическом вызове единственного обработчика события `Click` вне зависимости от того, на какой части кнопки был совершен щелчок. Выражаясь просто, модель маршрутизируемых событий автоматически распространяет событие вверх (или вниз) по дереву объектов в поисках подходящего обработчика.

Точнее говоря, маршрутизируемое событие может использовать три *стратегии маршрутизации*. Если событие перемещается от точки возникновения вверх к другим областям определений внутри дерева объектов, то его называют *пузырьковым событием*. И наоборот, если событие перемещается от самого внешнего элемента (например, `Window`) вниз к точке возникновения, то его называют *туннельным событием*. Наконец, если событие инициируется и обрабатывается только элементом, внутри которого оно возникло (что можно было бы описать как нормальное событие CLR), то его называют *прямым событием*.

Роль пузырьковых маршрутизируемых событий

В текущем примере, когда пользователь щелкает на внутреннем овале желтого цвета, событие `Click` поднимается на следующий уровень области определения (`Canvas`), затем на `StackPanel` и в итоге на уровень `Button`, где обрабатывается. Подобным же образом, если пользователь щелкает на `Label`, то событие всплывает на уровень `StackPanel` и, в конце концов, попадает в элемент `Button`.

Благодаря такому шаблону пузырьковых маршрутизируемых событий не придется беспокоиться о регистрации специфичных обработчиков события `Click` для всех членов составного элемента управления. Однако если необходимо выполнить специальную логику обработки щелчков для нескольких элементов внутри того же самого дерева объектов, то это вполне можно делать.

В целях иллюстрации предположим, что щелчок на элементе управления `outerEllipse` должен быть обработан в уникальной манере. Сначала обрабатываем событие `MouseDown` для этого подэлемента (графически визуализируемые типы вроде `Ellipse` не поддерживают событие `Click`, но могут отслеживать действия кнопки мыши через события `MouseDown`, `MouseUp` и т.д.):

```
<Button Name="btnClickMe" Height="75" Width = "250"
    Click ="btnClickMe_Clicked">
  <StackPanel Orientation ="Horizontal">
    <Label Height="50" FontSize ="20">Fancy Button!</Label>
    <Canvas Height ="50" Width ="100" >
      <Ellipse Name = "outerEllipse" Fill ="Green"
        Height ="25" MouseDown ="outerEllipse_MouseDown"
        Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
      <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15" Width ="36"
        Canvas.Top="17" Canvas.Left="32"/>
    </Canvas>
  </StackPanel>
</Button>
```

Затем реализуем подходящий обработчик событий, который в демонстрационных целях будет просто изменять свойство Title главного окна:

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Изменить заголовок окна.
    this.Title = "You clicked the outer ellipse!";
}
```

Далее можно выполнять разные действия в зависимости от того, на чем конкретно щелкнул конечный пользователь (на внешнем эллипсе или в любом другом месте внутри области кнопки).

На заметку! Пузырьковые маршрутизируемые события всегда перемещаются из точки возникновения до следующей определяющей области. Таким образом, в рассмотренном примере щелчок на элементе innerEllipse привел бы к попаданию события в контейнер Canvas, а не в элемент outerEllipse, потому что оба элемента являются типами Ellipse внутри области определения Canvas.

Продолжение или прекращение пузырькового распространения

В текущий момент, когда пользователь щелкает на объекте outerEllipse, запускается зарегистрированный обработчик события MouseDown для данного объекта Ellipse, после чего событие всплывет до события Click кнопки. Чтобы информировать WPF о необходимости остановки пузырькового распространения по дереву объектов, свойство Handled параметра MouseButtonEventArgs понадобится установить в true:

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Изменить заголовок окна.
    this.Title = "You clicked the outer ellipse!";

    // Остановить пузырьковое распространение.
    e.Handled = true;
}
```

В таком случае обнаружится, что заголовок окна изменился, но окно MessageBox, отображаемое обработчиком события Click элемента Button, не появляется. По существу пузырьковые маршрутизируемые события позволяют сложной группе содержимого действовать либо как единый логический элемент (например, Button), либо как отдельные элементы (скажем, Ellipse внутри Button).

Роль туннельных маршрутизируемых событий

Строго говоря, маршрутизируемые события по своей природе могут быть *пузырьковыми* (как было описано только что) или *туннельными*. Туннельные события (имена которых начинаются с префикса Preview — наподобие PreviewMouseDown) спускаются от самого верхнего элемента до внутренних областей определения дерева объектов. В общем и целом для каждого пузырькового события в библиотеках базовых классов WPF предусмотрено связанное туннельное событие, которое возникает *перед* его пузырьковым аналогом. Например, перед возникновением пузырькового события MouseDown сначала инициируется туннельное событие PreviewMouseDown.

Обработка туннельных событий выглядит очень похожей на обработку любых других событий: нужно просто указать имя обработчика события в разметке XAML (или при необходимости применить соответствующий синтаксис обработки событий C# в файле кода) и реализовать такой обработчик в коде. Для демонстрации взаимодей-

твия туннельных и пузырьковых событий начнем с организации обработки события PreviewMouseDown для объекта outerEllipse:

```
<Ellipse Name = "outerEllipse" Fill = "Green" Height = "25"
    MouseDown = "outerEllipse_MouseDown"
    PreviewMouseDown = "outerEllipse_PreviewMouseDown"
    Width = "50" Cursor = "Hand" Canvas.Left = "25" Canvas.Top = "12" />
```

Затем модифицируем текущее определение класса C#, обновив обработчики событий (для всех объектов) за счет добавления данных о событии в переменную-член _mouseActivity типа string с использованием входного объекта аргументов события. В результате появится возможность наблюдать за потоком событий, появляющихся в фоновом режиме.

```
public partial class MainWindow : Window
{
    string _mouseActivity = string.Empty;
    public MainWindow()
    {
        InitializeComponent();
    }

    public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
    {
        AddEventInfo(sender, e);
        MessageBox.Show(_mouseActivity, "Your Event Info");

        // Очистить строку для следующего цикла.
        _mouseActivity = "";
    }

    private void AddEventInfo(object sender, RoutedEventArgs e)
    {
        _mouseActivity += string.Format(
            "{0} sent a {1} event named {2}.\n", sender,
            e.RoutedEvent.RoutingStrategy,
            e.RoutedEvent.Name);
    }

    private void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
    {
        AddEventInfo(sender, e);
    }

    private void outerEllipse_PreviewMouseDown(object sender, MouseButtonEventArgs e)
    {
        AddEventInfo(sender, e);
    }
}
```

Обратите внимание, что ни в одном обработчике событий пузырьковое распространение не останавливается. После запуска приложения отобразится окно с уникальным сообщением, которое зависит от места на кнопке, где был произведен щелчок. На рис. 25.15 показан результат щелчка на внешнем объекте Ellipse.

Итак, почему события WPF обычно встречаются парами (одно туннельное и одно пузырьковое)? Ответ можно сформулировать так: благодаря предварительному просмотру событий появляется возможность выполнения любой специальной логики (проверки достоверности данных, отключения пузырькового распространения и т.п.) перед запуском пузырькового аналога событий.

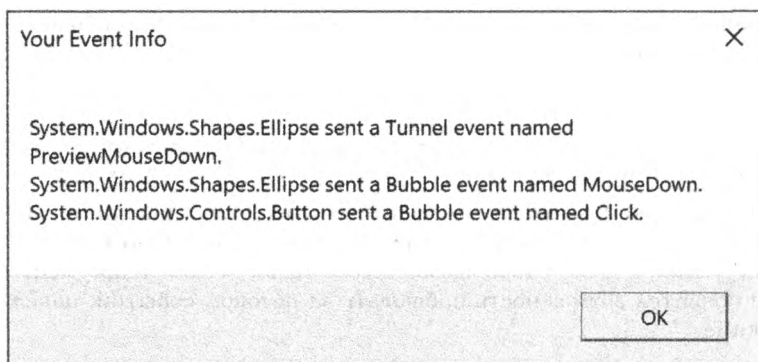


Рис. 25.15. Туннельное событие возникает первым, пузырьковое — вторым

В качестве примера предположим, что создается элемент `TextBox`, который должен содержать только числовые данные. В нем можно было бы обработать событие `PreviewKeyDown`: если выясняется, что пользователь ввел нечисловые данные, то пузырьковое событие легко отменить, установив свойство `Handled` в `true`.

Как несложно было предположить, при построении специального элемента управления, который поддерживает специальные события, событие допускается реализовать так, чтобы оно могло распространяться пузырьковым (или туннельным) образом по дереву разметки XAML. В настоящей главе мы не рассматриваем процесс создания специальных маршрутизируемых событий (хотя он не особо отличается от построения специального свойства зависимости). Если интересно, загляните в раздел “Routed Events Overview” (“Обзор маршрутизируемых событий”) документации .NET Framework 4.7 SDK, где предлагается несколько обучающих руководств, которые помогут в освоении этой темы.

Исходный код. Проект `WPF RoutedEvents` доступен в подкаталоге `Chapter_25`.

Более глубокий взгляд на API-интерфейсы и элементы управления WPF

В оставшемся материале главы будет построено новое приложение WPF с применением Visual Studio. Целью является создание пользовательского интерфейса, который состоит из виджета `TabControl`, содержащего набор вкладок. Каждая вкладка будет иллюстрировать несколько новых элементов управления WPF и интересные API-интерфейсы, которые могут быть задействованы в разрабатываемых проектах. Попутно вы также узнаете о дополнительных возможностях визуальных конструкторов WPF из Visual Studio.

Работа с элементом управления `TabControl`

Первым делом создадим новый проект приложения WPF по имени `WpfControlsAndAPIs`. Как упоминалось ранее, начальное окно будет содержать элемент управления `TabControl` с четырьмя вкладками, каждая из которых отображает набор связанных элементов управления и/или API-интерфейсов WPF. Установим свойство `Width` окна в 800, а свойство `Height` окна в 350.

Перетащим элемент управления `TabControl` из панели инструментов Visual Studio на поверхность визуального конструктора и обновим его разметку следующим образом:

```
<TabControl Name="MyTabControl" HorizontalAlignment="Stretch"
            VerticalAlignment="Stretch">
  <TabItem Header="TabItem">
    <Grid Background="#FFE5E5E5"/>
  </TabItem>
  <TabItem Header="TabItem">
    <Grid Background="#FFE5E5E5"/>
  </TabItem>
</TabControl>
```

Вы заметите, что два элемента типа вкладок предоставляются автоматически. Чтобы добавить дополнительные вкладки, нужно щелкнуть правой кнопкой мыши на узле `TabControl` в окне Document Outline и выбрать в контекстном меню пункт `Add TabItem` (Добавить `TabItem`). Можно также щелкнуть правой кнопкой мыши на элементе `TabControl` в визуальном конструкторе и выбрать тот же самый пункт меню или просто ввести разметку в редакторе XAML. Добавим одну дополнительную вкладку, используя любой из подходов.

Обновим разметку каждого элемента управления `TabItem` в редакторе XAML и изменим его свойство `Header`, указывая `Ink API`, `Data Binding` и `DataGrid`. Окно визуального конструктора должно выглядеть примерно так, как показано на рис. 25.16.

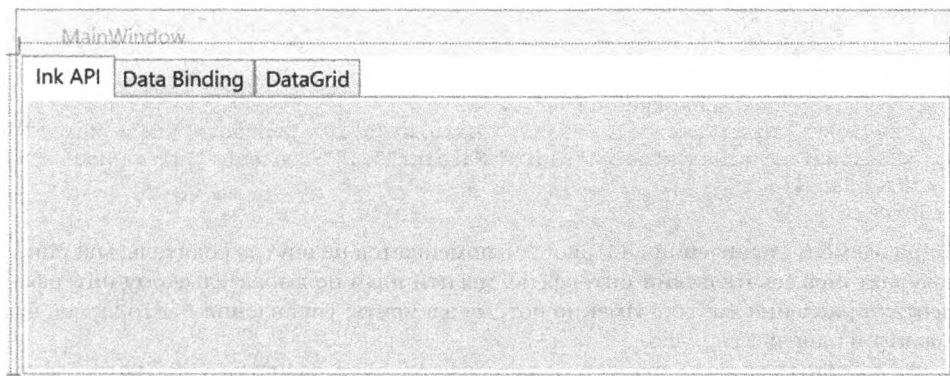


Рис. 25.16. Начальная компоновка системы вкладок

Имейте в виду, что выбранная для редактирования вкладка становится активной, и ее содержимое можно формировать, перетаскивая элементы управления из панели инструментов. Располагая определением основного элемента управления `TabControl`, можно проработать детали каждой вкладки, одновременно изучая дополнительные средства API-интерфейса WPF.

Построение вкладки `Ink API`

Первая вкладка предназначена для раскрытия общей роли интерфейса `Ink API`, который позволяет легко встраивать в программу функциональность рисования. Конечно, его применение не ограничивается приложениями для рисования; `Ink API` можно использовать для разнообразных целей, включая фиксацию рукописного ввода.

На заметку! В оставшейся части главы (и в следующих главах, посвященных WPF) вместо применения разнообразных окон будет главным образом напрямую редактироваться разметка XAML. Хотя процедура перетаскивания элементов управления работает нормально, чаще всего компоновка оказывается нежелательной (Visual Studio добавляет границы и заполнение на основе того, где размещен элемент), а потому приходится тратить значительное время на очистку разметки XAML.

Начнем с замены дескриптора Grid в элементе управления TabItem, помеченном как Ink API, дескриптором StackPanel и добавления закрывающего дескриптора. Разметка должна иметь такой вид:

```
<TabItem Header="Ink API">
  <StackPanel Background="#FFE5E5E5">
    </StackPanel>
  </TabItem>
```

Проектирование панели инструментов

Добавим (используя редактор XAML) новый элемент управления ToolBar по имени InkToolBar со свойством Height, установленным в 60:

```
<ToolBar Name="InkToolBar" Height="60">
</ToolBar>
```

Добавим три элемента управления RadioButton в панель WrapPanel внутри элемента управления Border:

```
<Border Margin="0,2,0,2.4" Width="280" VerticalAlignment="Center">
  <WrapPanel>
    <RadioButton x:Name="inkRadio" Margin="5,10" Content="Ink Mode!"
      IsChecked="True" />
    <RadioButton x:Name="eraseRadio" Margin="5,10" Content="Erase Mode!" />
    <RadioButton x:Name="selectRadio" Margin="5,10" Content="Select Mode!" />
  </WrapPanel>
</Border>
```

Когда элемент управления RadioButton помещается не внутрь родительской панели, он получает пользовательский интерфейс, идентичный пользовательскому интерфейсу элемента управления Button! Именно потому элементы управления RadioButton были упакованы в панель WrapPanel.

Далее добавим элемент Separator и элемент ComboBox, свойство Width которого установлено в 175, а свойство Margin — в 10, 0, 0, 0. Добавим три дескриптора ComboBoxItem с содержимым Red, Green и Blue, и сопроводим весь элемент управления ComboBox еще одним элементом Separator:

```
<Separator/>
<ComboBox x:Name="comboColors" Width="175" Margin="10,0,0,0">
  <ComboBoxItem Content="Red"/>
  <ComboBoxItem Content="Green"/>
  <ComboBoxItem Content="Blue"/>
</ComboBox>
<Separator/>
```

Элемент управления RadioButton

В данном примере необходимо, чтобы три добавленных элемента управления RadioButton были взаимно исключающими. В других инфраструктурах для построения графических пользовательских интерфейсов такие связанные элементы требуют

помещения в одну групповую рамку. Поступать подобным образом в WPF нет нужды. Взамен элементам управления просто назначается то же самое *групповое имя*, что очень удобно, поскольку связанные элементы не обязаны физически находиться внутри одной области, а могут располагаться где угодно в окне.

Класс `RadioButton` имеет свойство `IsChecked`, значения которого переключаются между `true` и `false`, когда конечный пользователь щелкает на элементе пользовательского интерфейса. К тому же элемент управления `RadioButton` предоставляет два события (`Checked` и `Unchecked`), которые можно применять для перехвата такого изменения состояния.

Добавление кнопок сохранения, загрузки и удаления

Финальным элементом управления внутри `ToolBar` будет `Grid`, содержащий три элемента управления `Button`. Добавим следующую разметку после последнего элемента управления `Separator`:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
  </Grid.ColumnDefinitions>
  <Button Grid.Column="0" x:Name="btnSave" Margin="10,10"
    Width="70" Content="Save Data"/>
  <Button Grid.Column="1" x:Name="btnLoad" Margin="10,10"
    Width="70" Content="Load Data"/>
  <Button Grid.Column="2" x:Name="btnClear" Margin="10,10"
    Width="70" Content="Clear"/>
</Grid>
```

Добавление элемента управления InkCanvas

Финальным элементом управления для `TabControl` является `InkCanvas`. Поместим показанную ниже разметку после закрывающего дескриптора `ToolBar`, но перед закрывающим дескриптором `StackPanel`:

```
<InkCanvas x:Name="MyInkCanvas" Background="#FFB6F4F1" />
```

Предварительный просмотр окна

Теперь все готово к тестированию программы, для чего понадобится нажать клавишу `<F5>`. Должны отобразиться три взаимно исключающих переключателя, раскрывающийся список с тремя элементами и три кнопки (рис. 25.17).

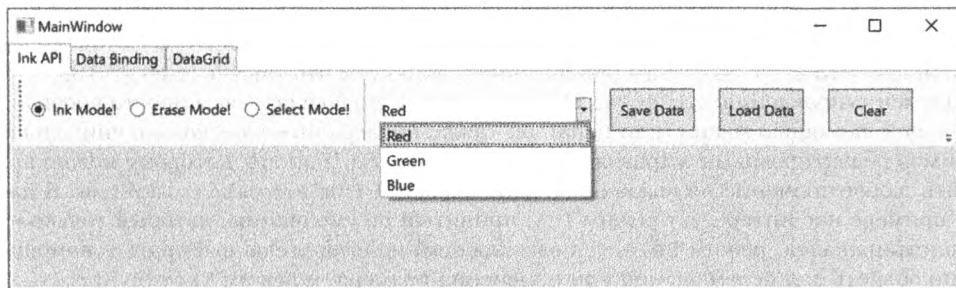


Рис. 25.17. Завершенная компоновка вкладки `Ink API`

Обработка событий для вкладки Ink API

Следующая задача для вкладки Ink API связана с организацией обработки события Click для каждого элемента управления RadioButton. Как делалось в других проектах WPF, нужно просто щелкнуть на значке с изображением молнии в окне Properties среды Visual Studio и ввести имена обработчиков событий. С помощью такого приема свяжем событие Click каждого элемента управления RadioButton с тем же самым обработчиком по имени RadioButtonClicked. После обработки всех трех событий Click обработаем событие SelectionChanged элемента управления ComboBox, используя обработчик по имени ColorChanged. В результате должен получиться следующий код C#:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        this.InitializeComponent();
        // Вставить здесь код, требуемый при создании объекта.
    }
    private void RadioButtonClicked(object sender, RoutedEventArgs e)
    {
        // TODO: Добавить сюда реализацию обработчика событий.
    }
    private void ColorChanged(object sender, SelectionChangedEventArgs e)
    {
        // TODO: Добавить сюда реализацию обработчика событий.
    }
}
```

Обработчики событий будут реализованы позже, так что оставим их пока пустыми.

Добавление элементов управления в панель инструментов

Элемент управления InkCanvas добавляется путем прямого редактирования разметки XAML. Имейте в виду, что панель инструментов Visual Studio по умолчанию не отображает все возможные компоненты WPF, но содержимое панели инструментов можно обновлять.

Щелкнем правой кнопкой мыши где-нибудь в области панели инструментов и выберем в контекстном меню пункт Choose Items (Выбрать элементы). Вскоре появится список возможных компонентов для добавления в панель инструментов. Нас интересует добавление элемента управления InkCanvas (рис. 25.18).

Элемент управления InkCanvas

Простое добавление InkCanvas делает возможным рисование в окне. Рисовать можно с помощью мыши или при наличии устройства, воспринимающего касания, пальца либо цифрового пера. Запустим приложение и нарисуем что-нибудь (рис. 25.19).

Элемент управления InkCanvas обеспечивает нечто большее, чем просто рисование штрихов с помощью мыши (или пера); он также поддерживает несколько уникальных режимов редактирования, управляемых свойством EditingMode, которому можно присвоить любое значение из связанного перечисления InkCanvasEditingMode. В данном примере нас интересует режим Ink, принятый по умолчанию, который только что демонстрировался, режим Select, позволяющий пользователю выбирать с помощью мыши область для перемещения или изменения размера, и режим EraseByStroke, который удаляет предыдущий штрих мыши.

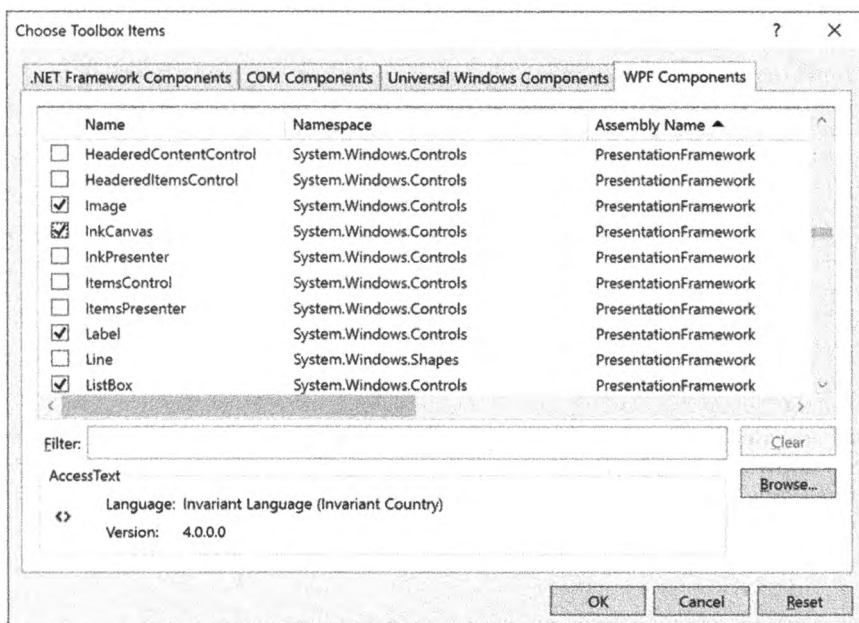


Рис. 25.18. Добавление новых компонентов в панель инструментов Visual Studio

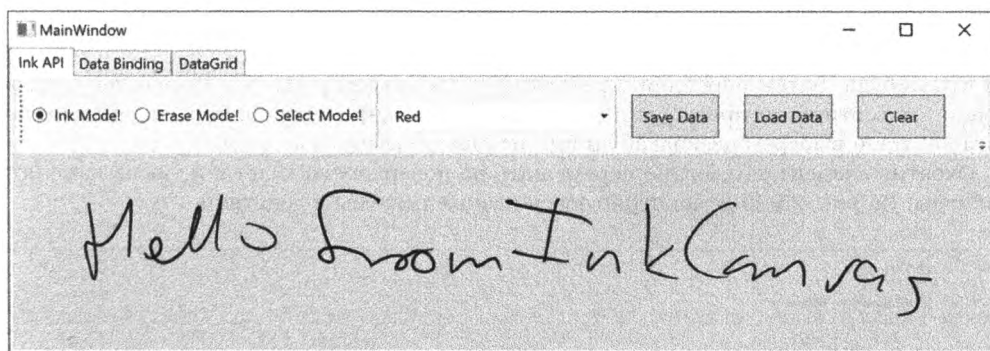


Рис. 25.19. Элемент управления InkCanvas в действии

На заметку! *Штрих* — это визуализация, которая происходит во время одиночной операции нажатия и отпускания кнопки мыши. Элемент управления InkCanvas сохраняет все штрихи в объекте `StrokeCollection`, который доступен с применением свойства `Strokes`.

Обновим обработчик `RadioButtonClicked()` следующей логикой, которая помещает InkCanvas в корректный режим в зависимости от выбранного переключателя RadioButton:

```
private void RadioButtonClicked(object sender, RoutedEventArgs e)
{
    // В зависимости от того, какая кнопка отправила событие,
    // поместить InkCanvas в нужный режим оперирования.
}
```

```

switch((sender as RadioButton)?.Content.ToString())
{
    // Эти строки должны совпадать со значениями свойства Content
    // каждого элемента RadioButton.
    case "Ink Mode!":
        this.MyInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
        break;
    case "Erase Mode!":
        this.MyInkCanvas.EditingMode = InkCanvasEditingMode.EraseByStroke;
        break;
    case "Select Mode!":
        this.MyInkCanvas.EditingMode = InkCanvasEditingMode.Select;
        break;
}
}

```

Вдобавок установим Ink как стандартный режим в конструкторе окна. Там же установим стандартный выбор для ComboBox (более подробно элемент управления ComboBox рассматривается в следующем разделе):

```

public MainWindow()
{
    this.InitializeComponent();

    // Установить режим Ink в качестве стандартного.
    this.MyInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
    this.inkRadio.IsChecked = true;
    this.comboColors.SelectedIndex = 0;
}

```

Теперь запустим программу еще раз, нажав <F5>. Войдем в режим Ink и нарисуем что-нибудь. Затем перейдем в режим Erase и сотрем ранее нарисованное (курсор мыши автоматически примет вид стирающей резинки). Наконец, переключимся в режим Select и выберем несколько линий, используя мышь в качестве лассо.

Охватив элемент, его можно перемещать по поверхности холста, а также изменять размеры. На рис. 25.20 демонстрируются разные режимы в действии.

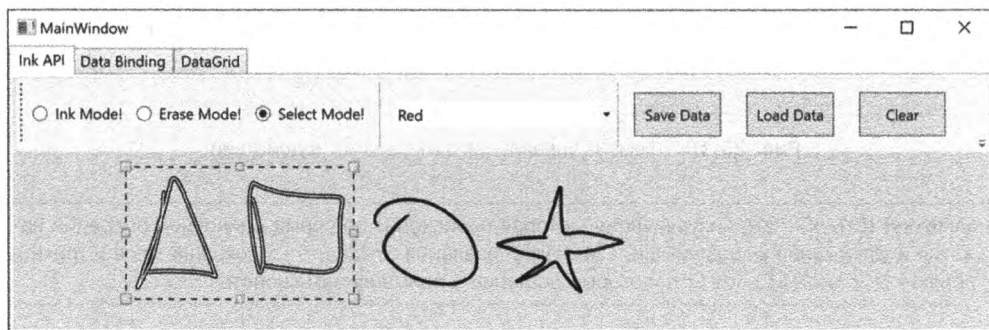


Рис. 25.20. Разные режимы редактирования элемента InkCanvas в действии

Элемент управления ComboBox

После заполнения элемента управления ComboBox (или ListBox) есть три способа определения выбранного в них элемента. Во-первых, когда необходимо найти числовой индекс выбранного элемента, должно применяться свойство SelectedIndex (отсчет на-

чинается с нуля; значение -1 представляет отсутствие выбора). Во-вторых, если требуется получить объект, выбранный внутри списка, то подойдет свойство `SelectedItem`. В-третьих, свойство `SelectedValue` позволяет получить значение выбранного объекта (обычно с помощью вызова `ToString()`).

Последний фрагмент кода, который понадобится добавить для данной вкладки, отвечает за изменение цвета штрихов, нарисованных в `InkCanvas`. Свойство `DefaultDrawingAttributes` элемента `InkCanvas` возвращает объект `DrawingAttributes`, который позволяет конфигурировать многочисленные аспекты пера, включая его размер и цвет (помимо других настроек). Модифицируем код C# следующей реализацией метода `ColorChanged()`:

```
private void ColorChanged(object sender, SelectionChangedEventArgs e)
{
    // Получить выбранный элемент в раскрывающемся списке.
    string colorToUse =
        (this.comboColors.SelectedItem as ComboBoxItem)?.Content.ToString();
    // Изменить цвет, используемый для визуализации штрихов.
    this.MyInkCanvas.DefaultDrawingAttributes.Color =
        (Color)ColorConverter.ConvertFromString(colorToUse);
}
```

Вспомните, что `ComboBox` содержит коллекцию `ComboBoxItems`. В сгенерированной разметке XAML присутствует такое определение:

```
<ComboBox x:Name="comboColors" Width="100" SelectionChanged="ColorChanged">
    <ComboBoxItem Content="Red"/>
    <ComboBoxItem Content="Green"/>
    <ComboBoxItem Content="Blue"/>
</ComboBox>
```

В результате обращения к свойству `SelectedItem` получается выбранный элемент `ComboBoxItem`, который хранится как экземпляр общего типа `Object`. После приведения `Object` к `ComboBoxItem` извлекается значение `Content`, которое будет строкой `Red`, `Green` или `Blue`. Эта строка затем преобразуется в объект `Color` с применением удобного служебного класса `ColorConverter`. Снова запустим программу. Теперь должна появиться возможность переключения между цветами при визуализации изображения.

Обратите внимание, что элементы управления `ComboBox` и `ListBox` также могут иметь сложное содержимое, а не только список текстовых данных. Чтобы получить представление о некоторых возможностях, откроем редактор XAML для окна и изменим определение элемента управления `ComboBox`, поместив в него набор элементов `StackPanel`, каждый из которых содержит `Ellipse` и `Label` (свойство `Width` элемента `ComboBox` установлено в 175):

```
<ComboBox x:Name="comboColors" Width="175" SelectionChanged="ColorChanged">
    <StackPanel Orientation="Horizontal" Tag="Red">
        <Ellipse Fill="Red" Height="50" Width="50"/>
        <Label FontSize="20" HorizontalAlignment="Center"
            VerticalAlignment="Center" Content="Red"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal" Tag="Green">
        <Ellipse Fill="Green" Height="50" Width="50"/>
        <Label FontSize="20" HorizontalAlignment="Center"
            VerticalAlignment="Center" Content="Green"/>
    </StackPanel>
```



```

<StackPanel Orientation="Horizontal" Tag="Blue">
  <Ellipse Fill="Blue" Height="50" Width="50"/>
  <Label FontSize="20" HorizontalAlignment="Center"
    VerticalAlignment="Center" Content="Blue"/>
</StackPanel>
</ComboBox>

```

В определении каждого элемента `StackPanel` присваивается значение свойству `Tag`, что является быстрым и удобным способом выявления, какой стек элементов был выбран пользователем (для этого существуют и лучшие способы, но пока достаточно тако-го). С указанной поправкой необходимо изменить реализацию метода `ColorChanged()`:

```

private void ColorChanged(object sender, SelectionChangedEventArgs e)
{
    // Получить свойство Tag выбранного элемента StackPanel.
    string colorToUse = (this.comboColors.SelectedItem
        as StackPanel).Tag.ToString();
    ...
}

```

После запуска программы элемент управления `ComboBox` будет выглядеть так, как показано на рис. 25.21.

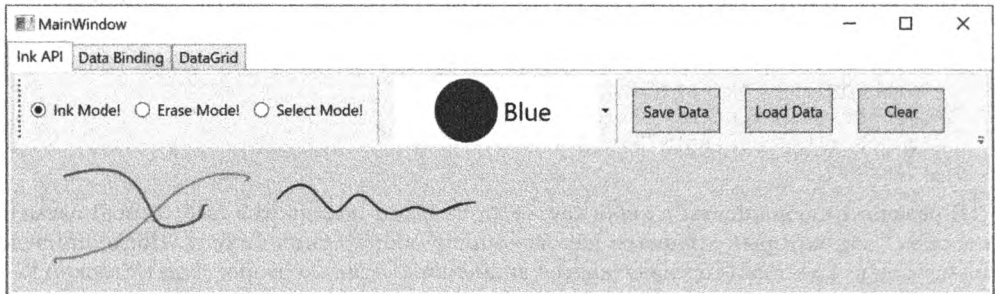


Рис. 25.21. Специальный элемент `ComboBox`, реализованный с помощью модели содержимого WPF

Сохранение, загрузка и очистка данных `InkCanvas`

Последняя часть вкладки `Ink API` позволит сохранять и загружать данные контейнера `InkCanvas`, а также очищать его содержимое, добавляя обработчики событий для кнопок в панели инструментов. Модифицируем разметку XAML для кнопок за счет добавления разметки, отвечающей за события щелчков:

```

<Button Grid.Column="0" x:Name="btnSave" Margin="10,10" Width="70"
  Content="Save Data" Click="SaveData"/>
<Button Grid.Column="1" x:Name="btnLoad" Margin="10,10" Width="70"
  Content="Load Data" Click="LoadData"/>
<Button Grid.Column="2" x:Name="btnClear" Margin="10,10" Width="70"
  Content="Clear" Click="Clear"/>

```

Импортируем пространства имен `System.IO` и `System.Windows.Ink` в файл кода. Реализуем обработчики событий следующим образом:

```

private void SaveData(object sender, RoutedEventArgs e)
{
    // Сохранить все данные InkCanvas в локальном файле.
}

```

```

using (FileStream fs = new FileStream("StrokeData.bin", FileMode.Create))
{
    this.MyInkCanvas.Strokes.Save(fs);
    fs.Close();
}
}

private void LoadData(object sender, RoutedEventArgs e)
{
    // Наполнить StrokeCollection из файла.
    using (FileStream fs = new FileStream("StrokeData.bin",
        FileMode.Open, FileAccess.Read))
    {
        StrokeCollection strokes = new StrokeCollection(fs);
        this.MyInkCanvas.Strokes = strokes;
    }
}

private void Clear(object sender, RoutedEventArgs e)
{
    // Очистить все штрихи.
    this.MyInkCanvas.Strokes.Clear();
}

```

Теперь должна появиться возможность сохранения данных в файле, загрузки из файла и очистки InkCanvas от всех данных. Итак, работа с первой вкладкой элемента управления TabControl завершена, равно как и исследование интерфейса Ink API. Конечно, о технологии Ink API можно рассказать еще много чего, но теперь вы должны обладать достаточными знаниями, чтобы продолжить изучение темы самостоятельно. Далее вы узнаете, как применять привязку данных WPF.

Введение в модель привязки данных WPF

Элементы управления часто служат целью для разнообразных операций привязки данных. Выразаясь просто, *привязка данных* представляет собой действие по подключению свойств элемента управления к значениям данных, которые могут изменяться на протяжении жизненного цикла приложения. Это позволяет элементу пользовательского интерфейса отображать состояние переменной в коде. Например, привязку данных можно использовать для решения следующих задач:

- отмечать флажок элемента управления CheckBox на основе булевого свойства заданного объекта;
- отображать в элементах TextBox информацию, извлеченную из реляционной базы данных;
- подключать элемент Label к целому числу, представляющему количество файлов в папке.

При работе со встроенным механизмом привязки данных WPF важно помнить о разнице между *источником* и *местом назначения* операции привязки. Как и можно было ожидать, источником операции привязки данных являются сами данные (булевоe свойство, реляционные данные и т.д.), а местом назначения (или целью) — свойство элемента управления пользовательского интерфейса, в котором задействуется содержимое данных (вроде свойства элемента управления CheckBox или TextBox).

В дополнение к привязке традиционных данных инфраструктура WPF делает возможной привязку элементов, как утверждалось в предшествующих примерах. Это зна-

чит, что можно привязать (скажем) видимость свойства к свойству состояния отметки флажка. Такое действие было определено возможным в Windows Forms, но требовало реализации через код. Инфраструктура WPF предлагает развитую экосистему привязки данных, которая способна почти целиком поддерживаться в разметке. Она также позволяет обеспечивать синхронизацию источника и цели в случае изменения значений данных.

Построение вкладки Data Binding

В окне Document Outline заменим элемент управления Grid во второй вкладке панелью StackPanel. С применением панели инструментов и окна Properties среды Visual Studio создадим следующую начальную компоновку:

```
<TabItem x:Name="tabDataBinding" Header="Data Binding">
  <StackPanel Width="250">
    <Label Content="Move the scroll bar to see the current value"/>

    <!--Значение линейки прокрутки является источником этой привязки данных-->
    <ScrollBar x:Name="mySB" Orientation="Horizontal" Height="30"
      Minimum = "1" Maximum = "100" LargeChange="1" SmallChange="1"/>

    <!-- Содержимое метки будет привязано к линейке прокрутки -->
    <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
      BorderThickness="2" Content = "0"/>
  </StackPanel>
</TabItem>
```

Обратите внимание, что объект ScrollBar (названный здесь mySB) сконфигурирован с диапазоном от 1 до 100. Цель заключается в том, чтобы при изменении положения ползунка линейки прокрутки (либо по щелчку на стрелке влево или вправо) элемент Label автоматически обновлялся текущим значением. В настоящий момент значение свойства Content элемента управления Label установлено в "0"; тем не менее, мы изменим его посредством операции привязки данных.

Установка привязки данных

Механизмом, обеспечивающим определение привязки в разметке XAML, является расширение разметки {Binding}. Хотя привязки можно определять посредством Visual Studio, это столь же легко делать прямо в разметке. Отредактируем разметку XAML свойства Content элемента Label по имени labelSBThumb следующим образом:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
  BorderThickness="2" Content = "{Binding Value, ElementName=mySB}"/>
```

Обратите внимание на значение, присвоенное свойству Content элемента Label. Конструкция {Binding} обозначает операцию привязки данных. Значение ElementName представляет источник операции привязки данных (объект ScrollBar), а Path указывает свойство, к которому осуществляется привязка (свойство Value линейки прокрутки).

Запустив программу снова, можно обнаружить, что содержимое метки обновляется на основе значения линейки прокрутки по мере перемещения ползунка.

Свойство DataContext

Для определения операции привязки данных в XAML может использоваться альтернативный формат, при котором допускается разбивать значения, указанные расширением разметки {Binding}, за счет явной установки свойства DataContext в источник операции привязки:

```

<!-- Разбиение объекта и значения посредством DataContext -->
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
      BorderThickness="2"
      DataContext = "{Binding ElementName=mySB}"
      Content = "{Binding Path=Value}" />

```

В текущем примере вывод будет идентичным. С учетом этого вполне вероятно вас интересует, в каких случаях необходимо устанавливать свойство `DataContext` явно. Поступать так может быть удобно из-за того, что подэлементы способны наследовать свои значения в дереве разметки.

Подобным образом можно легко устанавливать один и тот же источник данных для семейства элементов управления, не повторяя избыточные XAML-фрагменты `"{Binding ElementName=X, Path=Y}"` во множестве элементов управления. Например, пусть в панель `StackPanel` вкладки добавлен новый элемент `Button` (вскоре вы увидите, почему он имеет настолько большой размер):

```
<Button Content="Click" Height="200"/>
```

Чтобы сгенерировать привязки данных для множества элементов управления, можно было бы применить `Visual Studio`, но взамен мы введем модифицированную разметку в редакторе XAML:

```

<!--Обратите внимание, что StackPanel устанавливает свойство DataContext-->
<StackPanel Background="#FFE5E5" DataContext = "{Binding ElementName=mySB}">
  <Label Content="Move the scroll bar to see the current value"/>
  <ScrollBar Orientation="Horizontal" Height="30" Name="mySB"
    Maximum = "100" LargeChange="1" SmallChange="1"/>
  <!-- Теперь оба элемента пользовательского интерфейса работают
    со значением линейки прокрутки уникальными путями -->
  <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
    BorderThickness="2" Content = "{Binding Path=Value}"/>
  <Button Content="Click" Height="200" FontSize = "{Binding Path=Value}"/>
</StackPanel>

```

Здесь свойство `DataContext` панели `StackPanel` устанавливается напрямую. Таким образом, при перемещении ползунка не только отображается текущее значение в элементе `Label`, но также увеличивается размер шрифта элемента `Button` в соответствие с тем же значением (на рис. 25.22 показан возможный вывод).

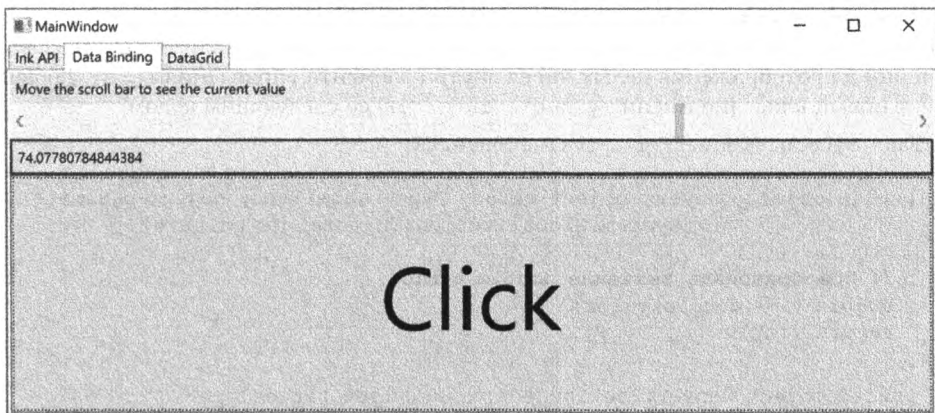


Рис. 25.22. Привязка значения `ScrollBar` к элементам `Label` и `Button`

Форматирование привязанных данных

Вместо ожидаемого целого числа для представления положения ползунка тип `ScrollBar` использует значение `double`. Следовательно, по мере перемещения ползунка внутри элемента `Label` будут отображаться разнообразные значения с плавающей точкой (вроде 61.0576923076923), которые выглядят не слишком интуитивно понятными для конечного пользователя, почти наверняка ожидающего увидеть целые числа (такие как 61, 62, 63 и т.д.).

При желании форматировать данные можно добавить свойство `ContentStringFormat`, передав ему специальную строку и спецификатор формата `.NET`:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
      BorderThickness="2" Content = "{Binding Path=Value}"
      ContentStringFormat="The value is: {0:F0}"/>
```

Если в спецификации форматирования отсутствует какой-либо текст, тогда спецификатор формата понадобится предварить пустым набором фигурных скобок, который является управляющей последовательностью для XAML. Такой прием уведомляет процессор о том, что следующие за `{}` символы представляют собой литералы, а не, скажем, конструкцию привязки. Вот обновленная разметка XAML:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
      BorderThickness="2" Content = "{ {Binding Path=Value}"
      ContentStringFormat="{0:F0}"/>
```

На заметку! При привязке свойства `Text` элемента управления пару “имя-значение” объекта `StringFormat` можно добавлять прямо в конструкции привязки. Она должна быть отдельной только для свойств `Content`.

Преобразование данных с использованием интерфейса `IValueConverter`

Если требуется нечто большее, чем просто форматирование данных, тогда можно создать специальный класс, реализующий интерфейс `IValueConverter` из пространства имен `System.Windows.Data`. В интерфейсе `IValueConverter` определены два члена, позволяющие выполнять преобразование между источником и целью (в случае двунаправленной привязки). После определения такой класс можно применять для дальнейшего уточнения процесса привязки данных.

Вместо использования свойства форматирования можно применять преобразователь значений для отображения целых чисел внутри элемента управления `Label`. Добавим в проект новый класс (по имени `MyDoubleConverter`) со следующим кодом:

```
class MyDoubleConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
        System.Globalization.CultureInfo culture)
    {
        // Преобразовать значение double в int.
        double v = (double)value;
        return (int)v;
    }

    public object ConvertBack(object value, Type targetType,
        object parameter, System.Globalization.CultureInfo culture)
    {

```

```
// Поскольку заботиться о "двунаправленной" привязке
// не нужно, просто вернуть значение value.
return value;
}
}
```

Метод `Convert()` вызывается при передаче значения от источника (`ScrollBar`) к цели (свойство `Content` элемента `Label`). Хотя он принимает много входных аргументов, для такого преобразования понадобится манипулировать только входным аргументом типа `object`, который представляет текущее значение `double`. Данный тип можно использовать для приведения к целому и возврата нового числа.

Метод `ConvertBack()` будет вызываться, когда значение передается от цели к источнику (если включен двунаправленный режим привязки). Здесь мы просто возвращаем значение `value`. Это позволяет вводить в `TextBox` значение с плавающей точкой (например, 99.9) и автоматически преобразовывать его в целочисленное значение (99), когда пользователь перемещает фокус из элемента управления. Такое "бесплатное" преобразование происходит из-за того, что метод `Convert()` будет вызываться еще раз после вызова `ConvertBack()`. Если просто вернуть `null` из `ConvertBack()`, то синхронизация привязки будет выглядеть нарушенной, т.к. элемент `TextBox` по-прежнему будет отображать число с плавающей точкой. Чтобы применить построенный преобразователь в разметке, сначала нужно создать локальный ресурс, представляющий только что законченный класс. Не переживайте по поводу механики добавления ресурсов; тема будет детально раскрыта в нескольких последующих главах. Поместим показанную ниже разметку сразу после открывающего дескриптора `Window`:

```
<Window.Resources>
  <local:MyDoubleConverter x:Key="DoubleConverter"/>
</Window.Resources>
```

Далее обновим конструкцию привязки для элемента управления `Label`:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2"
  Content = "{Binding Path=Value,Converter={StaticResource DoubleConverter}}" />
```

После запуска приложения теперь можно видеть только целые числа.

Установление привязок данных в коде

Специальный преобразователь данных можно также регистрировать в коде. Начнем с очистки текущего определения элемента управления `Label` внутри вкладки `Data Binding`, чтобы расширение разметки `{Binding}` больше не использовалось:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2" />
```

Добавим оператор `using` для `System.Windows.Data` и в конструкторе окна вызовем новый закрытый вспомогательный метод по имени `SetBindings()`, код которого показан ниже:

```
private void SetBindings()
{
  // Создать объект Binding.
  Binding b = new Binding();
  // Зарегистрировать преобразователь, источник и путь.
  b.Converter = new MyDoubleConverter();
  b.Source = this.mySB;
  b.Path = new PropertyPath("Value");
  // Вызвать метод SetBinding() объекта Label.
  this.labelSBThumb.SetBinding(Label.ContentProperty, b);
}
```

Единственная часть метода `SetBindings()`, которая может выглядеть несколько необычной — вызов `SetBinding()`. Обратите внимание, что первый параметр обращается к статическому, доступному только для чтения полю `ContentProperty` класса `Label`. Как вы узнаете далее в главе, такая конструкция называется *свойством зависимости*. Пока просто имейте в виду, что при установке привязки в коде первый аргумент почти всегда требует указания имени класса, нуждающегося в привязке (`Label` в рассматриваемом случае), за которым следует обращение к внутреннему свойству с добавлением к его имени суффикса `Property`. Запустив приложение, можно удостовериться в том, что элемент `Label` отображает только целые числа.

Построение вкладки `DataGrid`

В предыдущем примере привязки данных иллюстрировался способ конфигурирования двух (или большего количества) элементов управления для участия в операции привязки данных. Наряду с тем, что это удобно, возможно также привязывать данные из файлов XML, базы данных и объектов в памяти. Чтобы завершить текущий пример, спроектируем финальную вкладку элемента управления `DataGrid`, которая будет отображать информацию, извлеченную из таблицы `Inventory` базы данных `AutoLot`.

Как и с другими вкладками, начнем с замены текущего элемента `Grid` панелью `StackPanel`, напрямую обновив разметку XAML в Visual Studio. Затем внутри нового элемента `StackPanel` определим элемент управления `DataGrid` по имени `gridInventory`:

```
<TabItem x:Name="tabDataGrid" Header="DataGrid">
  <StackPanel>
    <DataGrid x:Name="gridInventory" Height="288"/>
  </StackPanel>
</TabItem>
```

С помощью диспетчера пакетов NuGet добавим в проект инфраструктуру Entity Framework. Щелчком правой кнопкой мыши на решении, выберем в контекстном меню пункт `Add⇒Existing Project (Добавить⇒Существующий проект)` и добавим проект `AutoLotDAL` из главы 22. Добавим ссылку на проект `AutoLotDAL` из проекта `WpfControlsAndAPIs`. Добавим строку подключения в файл `App.config`. Вот пример строки подключения:

```
<connectionStrings>
  <add name="AutoLotConnection" connectionString="data source=(LocalDb)\
MSSQLLocalDB;initial catalog=AutoLot;integrated security=True;MultipleActive
ResultSets=True;App=EntityFramework"
  providerName="System.Data.SqlClient" />
</connectionStrings>
```

Откроем файл кода окна, добавим последний вспомогательный метод по имени `ConfigureGrid()` и вызовем его в конструкторе. Предполагая, что пространство имен `AutoLotDAL` было импортировано, останется лишь добавить несколько строк кода:

```
using AutoLotDAL.Repos;
private void ConfigureGrid()
{
    using (var repo = new InventoryRepo())
    {
        // Построить запрос LINQ, который извлекает некоторые данные из таблицы Inventory.
        gridInventory.ItemsSource =
            repo.GetAll().Select(x => new { x.Id, x.Make, x.Color, x.PetName });
    }
}
```

Запрос LINQ создает новый анонимный объект, который исключает свойство `Timestamp`, т.к. указанное поле не содержит значение, предназначенное для отображения пользователю. Запустив проект, можно увидеть данные, заполняющие сетку. При желании сделать сетку более привлекательной можно применить окно `Properties` в `Visual Studio` для редактирования свойств сетки, чтобы улучшить ее внешний вид.

На этом текущий пример завершен. В последующих главах вы увидите в действии другие элементы управления, но к настоящему моменту вы должны чувствовать себя увереннее с процессом построения пользовательских интерфейсов в `Visual Studio`, а также при работе с разметкой XAML и кодом C#.

Исходный код. Проект `WpfControlsAndAPIs` доступен в подкаталоге `Chapter_25`.

Роль свойств зависимости

Подобно любому API-интерфейсу .NET внутри инфраструктуры WPF используется каждый член системы типов .NET (классы, структуры, интерфейсы, делегаты, перечисления) и каждый член типа (свойства, методы, события, константные данные, поля только для чтения и т.д.). Однако WPF также поддерживает уникальную программную концепцию под названием *свойство зависимости*.

Как и "нормальное" свойство .NET (которое в литературе, посвященной WPF, часто называют *свойством CLR*), свойство зависимости можно устанавливать декларативно с помощью разметки XAML или программно в файле кода. Кроме того, свойства зависимости (подобно свойствам CLR) в конечном итоге предназначены для инкапсуляции полей данных класса и могут быть сконфигурированы как доступные только для чтения, только для записи или для чтения и записи.

Вы будете практически всегда пребывать в блаженном неведении относительно того, что фактически устанавливаете (или читаете) свойство зависимости, а не свойство CLR. Например, свойства `Height` и `Width`, которые элементы управления WPF наследуют от класса `FrameworkElement`, а также член `Content`, унаследованный от класса `ControlContent`, на самом деле являются свойствами зависимости:

```
<!-- Установить три свойства зависимости -->
<Button x:Name = "btnMyButton" Height = "50" Width = "100" Content = "OK"/>
```

С учетом всех указанных сходств возникает вопрос: зачем нужно было определять в WPF новый термин для такой знакомой концепции? Ответ кроется в способе реализации свойства зависимости внутри класса. Пример кодирования будет показан ниже, а на высоком уровне все свойства зависимости создаются в следующей манере.

Класс, который определяет свойство зависимости, должен иметь в своей цепочке наследования класс `DependencyObject`.

Одиночное свойство зависимости представляется как открытое, статическое, допускающее только чтение поле типа `DependencyProperty`. По соглашению это поле именуется путем снабжения имени оболочки CLR (см. последний пункт списка) суффиксом в виде слова `Property`.

Переменная типа `DependencyProperty` регистрируется посредством вызова статического метода `DependencyProperty.Register()`, который обычно происходит в статическом конструкторе или встраивается в объявление переменной.

В классе будет определено дружественное к XAML свойство CLR, которое вызывает методы, предоставляемые классом `DependencyObject`, для получения и установки значения.

После реализации свойства зависимости предлагают несколько мощных средств, которые применяются разнообразными технологиями WPF, в том числе привязкой дан-

ных, службами анимации, стилями, шаблонами и т.д. Мотивацией создания свойств зависимости было желание предоставить способ вычисления значений свойств на основе значений из других источников. Ниже приведен список основных преимуществ, которые выходят далеко за рамки простой инкапсуляции данных, обеспечиваемой свойствами CLR.

Свойства зависимости могут наследовать свои значения от определения XAML родительского элемента. Например, если в открывающем дескрипторе Window определено значение для атрибута `FontSize`, то все элементы управления внутри Window по умолчанию будут иметь тот же самый размер шрифта.

Свойства зависимости поддерживают возможность получать значения, которые установлены элементами внутри их области определения XAML, например, в случае установки элементом `Button` свойства `Dock` родительского контейнера `DockPanel`. (Вспомните, что *присоединяемые свойства* делают именно это, поскольку являются разновидностью свойств зависимости.)

Свойства зависимости позволяют инфраструктуре WPF вычислять значение на основе множества внешних значений, что может быть важно для служб анимации и привязки данных.

Свойства зависимости предоставляют поддержку инфраструктуры для триггеров WPF (также довольно часто используемых при работе с анимацией и привязкой данных).

Имейте в виду, что во многих случаях вы будете взаимодействовать с существующим свойством зависимости в манере, идентичной работе с обычным свойством CLR (благодаря оболочке XAML). В предыдущем разделе, посвященном привязке данных, вы узнали, что если необходимо установить привязку данных в коде, то должен быть вызван метод `SetBinding()` на целевом объекте операции и указано *свойство зависимости*, с которым будет работать привязка:

```
private void SetBindings()
{
    Binding b = new Binding();
    b.Converter = new MyDoubleConverter();
    b.Source = this.mySB;
    b.Path = new PropertyPath("Value");

    // Указать свойство зависимости.
    this.labelSBThumb.SetBinding(Label.ContentProperty, b);
}
```

Вы увидите похожий код в главе 27 во время исследования запуска анимации в коде:

```
// Указать свойство зависимости.
rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);
```

Потребность в построении специального свойства зависимости возникает только во время разработки собственного элемента управления WPF. Например, когда создается класс `UserControl` с четырьмя специальными свойствами, которые должны тесно интегрироваться с API-интерфейсом WPF, они должны быть реализованы с применением логики свойств зависимости.

В частности, если нужно, чтобы свойство было целью операции привязки данных или анимации, если оно обязано уведомлять о своем изменении, если свойство должно быть в состоянии работать в качестве установщика в стиле WPF или получать свои значения от родительского элемента, то возможностей обычного свойства CLR окажется не достаточно. В случае использования обычного свойства другие программисты действительно могут получать и устанавливать его значение, но если они попытают-

ся применить такое свойство внутри контекста службы WPF, то оно не будет работать ожидаемым образом. Поскольку заранее нельзя узнать, как другие пожелают взаимодействовать со свойствами специальных классов `UserControl`, нужно выработать в себе привычку при построении специальных элементов управления *всегда* определять свойства зависимости.

Исследование существующего свойства зависимости

Прежде чем вы научитесь создавать специальные свойства зависимости, давайте рассмотрим внутреннюю реализацию свойства `Height` класса `FrameworkElement`. Ниже приведен соответствующий код (с комментариями):

```
// FrameworkElement "является" DependencyObject.
public class FrameworkElement : UIElement, IFrameworkInputElement,
    IInputElement, ISupportInitialize, IHaveResources, IQueryAmbient
{
    ...
    // Статическое поле только для чтения типа DependencyProperty.
    public static readonly DependencyProperty HeightProperty;

    // Поле DependencyProperty часто регистрируется
    // в статическом конструкторе класса.
    static FrameworkElement()
    {
        ...
        HeightProperty = DependencyProperty.Register(
            "Height",
            typeof(double),
            typeof(FrameworkElement),
            new FrameworkPropertyMetadata((double) 1.0 / (double) 0.0,
                FrameworkPropertyMetadataOptions.AffectsMeasure,
                new PropertyChangedCallback(FrameworkElement.OnTransformDirty)),
            new ValidateValueCallback(FrameworkElement.IsWidthHeightValid));
    }

    // Оболочка CLR, реализованная с использованием
    // унаследованных методов GetValue()/SetValue().
    public double Height
    {
        get { return (double) base.GetValue(HeightProperty); }
        set { base.SetValue(HeightProperty, value); }
    }
}
```

Как видите, по сравнению с обычными свойствами CLR свойства зависимости требуют немалого объема дополнительного кода. В реальности зависимость может оказаться даже еще более сложной, чем показано здесь (к счастью, многие реализации проще свойства `Height`).

В первую очередь вспомните, что если в классе необходимо определить свойство зависимости, то он должен иметь в своей цепочке наследования `DependencyObject`, т.к. именно этот класс определяет методы `GetValue()` и `SetValue()`, применяемые в оболочке CLR. Из-за того, что класс `FrameworkElement` "является" `DependencyObject`, указанное требование удовлетворено.

Далее вспомните, что сущность, где действительно хранится значение свойства (значение `double` в случае `Height`), представляется как открытое, статическое, допускающее только чтение поле типа `DependencyProperty`. По соглашению имя этого свойства

должно всегда формироваться из имени связанной оболочки CLR с добавлением суффикса `Property`:

```
public static readonly DependencyProperty HeightProperty;
```

Учитывая, что свойства зависимости объявляются как статические поля, они обычно создаются (и регистрируются) внутри статического конструктора класса. Объект `DependencyProperty` создается посредством вызова статического метода `DependencyProperty.Register()`. Данный метод имеет множество перегруженных версий, но в случае свойства `Height` он вызывается следующим образом:

```
HeightProperty = DependencyProperty.Register(
    "Height",
    typeof(double),
    typeof(FrameworkElement),
    new FrameworkPropertyMetadata((double)0.0,
        FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(FrameworkElement.OnTransformDirty)),
    new ValidateValueCallback(FrameworkElement.IsWidthHeightValid));
```

Первым аргументом, передаваемым методу `DependencyProperty.Register()`, является имя обычного свойства CLR класса (`Height`), а второй аргумент содержит информацию о типе данных, который его инкапсулирует (`double`). Третий аргумент указывает информацию о типе класса, которому принадлежит свойство (`FrameworkElement`). Хотя такие сведения могут показаться избыточными (в конце концов, поле `HeightProperty` уже определено внутри класса `FrameworkElement`), это очень продуманный аспект WPF, поскольку он позволяет одному классу регистрировать свойства в другом классе (даже если его определение было запечатано).

Четвертый аргумент, передаваемый методу `DependencyProperty.Register()` в рассмотренном примере, представляет собой то, что действительно делает свойства зависимости уникальными. Здесь передается объект `FrameworkPropertyMetadata`, который описывает разнообразные детали относительно того, как инфраструктура WPF должна обрабатывать данное свойство в плане уведомлений с помощью обратных вызовов (если свойству необходимо извещать других, когда его значение изменяется). Кроме того, объект `FrameworkPropertyMetadata` указывает различные параметры (представленные перечислением `FrameworkPropertyMetadataOptions`), которые управляют тем, на что свойство воздействует (работает ли оно с привязкой данных, может ли наследоваться и т.д.). В данном случае аргументы конструктора `FrameworkPropertyMetadata` можно описать так:

```
new FrameworkPropertyMetadata(
    // Стандартное значение свойства.
    (double)0.0,
    // Параметры метаданных.
    FrameworkPropertyMetadataOptions.AffectsMeasure,
    // Делегат, который указывает на метод, вызываемый при изменении свойства.
    new PropertyChangedCallback(FrameworkElement.OnTransformDirty)
)
```

Поскольку последний аргумент конструктора `FrameworkPropertyMetadata` является делегатом, обратите внимание, что он указывает на статический метод `OnTransformDirty()` класса `FrameworkElement`. Код метода `OnTransformDirty()` здесь не приводится, но имейте в виду, что при создании специального свойства зависимости всегда можно указывать делегат `PropertyChangeCallback`, нацеленный на метод, который будет вызываться в случае изменения значения свойства.

Обратимся к финальному параметру метода `DependencyProperty.Register()` — второму делегату типа `ValidateValueCallback`, указывающему на метод класса `FrameworkElement`, который вызывается для проверки достоверности значения, присваиваемого свойству:

```
new ValidateValueCallback(FrameworkElement.IsWidthHeightValid)
```

Метод `IsWidthHeightValid()` содержит логику, которую обычно ожидают найти в блоке установки значения свойства (как более подробно объясняется в следующем разделе):

```
private static bool IsWidthHeightValid(object value)
{
    double num = (double) value;
    return ((!DoubleUtil.IsNaN(num) && (num >= 0.0))
        && !double.IsPositiveInfinity(num));
}
```

После того, как объект `DependencyProperty` зарегистрирован, остается упаковать поле в обычное свойство CLR (`Height` в рассматриваемом случае). Тем не менее, обратите внимание, что блоки `get` и `set` не просто возвращают или устанавливают значение `double` переменной-члена уровня класса, а делают это косвенно с использованием методов `GetValue()` и `SetValue()` базового класса `System.Windows.DependencyObject`:

```
public double Height
{
    get { return (double) base.GetValue(HeightProperty); }
    set { base.SetValue(HeightProperty, value); }
}
```

Важные замечания относительно оболочек свойств CLR

Итак, подводя итоги сказанному до сих пор, свойства зависимости выглядят как обычные свойства, когда вы извлекаете или устанавливаете их значения в разметке XAML либо в коде, но “за кулисами” они реализованы с помощью гораздо более замысловатых приемов кодирования. Вспомните, что основным назначением этого процесса является построение специального элемента управления, имеющего специальные свойства, которые должны быть интегрированы со службами WPF, требующими взаимодействия через свойства зависимости (например, с анимацией, привязкой данных и стилями).

Несмотря на то что часть реализации свойства зависимости предусматривает определение оболочки CLR, вы никогда не должны помещать логику проверки достоверности в блок `set`. К тому же оболочка CLR свойства зависимости не должна делать ничего кроме вызовов `GetValue()` или `SetValue()`.

Исполняющая среда WPF сконструирована таким образом, что если написать разметку XAML, которая выглядит как установка свойства, например:

```
<Button x:Name="myButton" Height="100" .../>
```

то исполняющая среда вообще обойдет блок установки свойства `Height` и напрямую вызовет метод `SetValue()`! Причина такого необычного поведения связана с простым приемом оптимизации. Если бы исполняющая среда WPF обращалась к блоку установки свойства `Height`, то ей пришлось бы во время выполнения выяснять посредством рефлексии, где находится поле `DependencyProperty` (указанное в первом аргументе `SetValue()`), ссылаться на него в памяти и т.д. То же самое остается справедливым и при написании разметки XAML, которая извлекает значение свойства `Height` — метод `GetValue()` будет вызываться напрямую.

Но раз так, тогда зачем вообще строить оболочку CLR? Дело в том, что XAML в WPF не позволяет вызывать функции в разметке, поэтому следующий фрагмент приведет к ошибке:

```
<!-- Ошибка! Вызывать методы в XAML-разметке WPF нельзя! -->
<Button x:Name="myButton" this.SetValue("100") .../>
```

На самом деле установку или получение значения в разметке с применением оболочки CLR следует считать способом сообщения исполняющей среде WPF о необходимости вызова методов `GetValue()`/`SetValue()`, т.к. напрямую вызывать их в разметке невозможно. А что, если обратиться к оболочке CLR в коде, как показано ниже?

```
Button b = new Button();
b.Height = 10;
```

В таком случае, если блок `set` свойства `Height` содержит какой-то код помимо вызова `SetValue()`, то он *должен* выполняться, потому что оптимизация синтаксического анализатора XAML в WPF не задействуется.

Запомните основное правило: при регистрации свойства зависимости используйте делегат `ValidateValueCallback` для указания на метод, который выполняет проверку достоверности данных. Такой подход гарантирует корректное поведение независимо от того, что применяется для получения/установки свойства зависимости — разметка XAML или код.

Построение специального свойства зависимости

Если к настоящему моменту вы слегка запутались, то такая реакция совершенно нормальна. Создание свойств зависимости может требовать некоторого времени на привыкание. Как бы то ни было, но это часть процесса построения многих специальных элементов управления WPF, так что давайте рассмотрим, каким образом создается свойство зависимости.

Начнем с создания нового проекта приложения WPF по имени `CustomDepProp`. Выберем в меню `Project (Проект)` пункт `Add New Item (Добавить новый элемент)` и, указав `User Control (WPF)` (Пользовательский элемент управления (WPF)) для типа элемента, создадим элемент с именем `ShowNumberControl.xaml`.

На заметку! Более подробные сведения о классе `UserControl` в WPF ищите в главе 27, а пока просто следуйте указаниям по мере проработки примера.

Подобно окну типы `UserControl` в WPF имеют файл XAML и связанный файл кода. Модифицируем разметку XAML пользовательского элемента управления, чтобы определить простой элемент `Label` внутри `Grid`:

```
<UserControl x:Class="CustomDepProp.ShowNumberControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:CustomDepProp"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <Label x:Name="numberDisplay" Height="50" Width="200"
            Background="LightBlue"/>
    </Grid>
</UserControl>
```

В файле кода для данного элемента создадим обычное свойство .NET, которое упаковывает поле типа `int` и устанавливает новое значение для свойства `Content` элемента `Label`:

```
public partial class ShowNumberControl : UserControl
{
    public ShowNumberControl()
    {
        InitializeComponent();
    }

    // Обычное свойство .NET.
    private int _currNumber = 0;
    public int CurrentNumber
    {
        get => _currNumber;
        set
        {
            _currNumber = value;
            numberDisplay.Content = CurrentNumber.ToString();
        }
    }
}
```

Теперь обновим определение XAML, объявив экземпляр специального элемента управления внутри диспетчера компоновки `StackPanel`. Поскольку специальный элемент управления не входит в состав основных сборок WPF, понадобится определить специальное пространство имен XML, которое отображается на него. Вот требуемая разметка:

```
<Window x:Class="CustomDepProp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:myCtrls="clr-namespace:CustomDepProp"
        xmlns:local="clr-namespace:CustomDepProp"
        mc:Ignorable="d"
        Title="Simple Dependency Property App" Height="150" Width="250"
        WindowStartupLocation="CenterScreen">
    <StackPanel>
        <myCtrls:ShowNumberControl x:Name="myShowNumberCtrl" CurrentNumber="100"/>
    </StackPanel>
</Window>
```

Похоже, что визуальный конструктор Visual Studio корректно отображает значение, установленное в свойстве `CurrentNumber` (рис. 25.23).

Однако что, если к свойству `CurrentNumber` необходимо применить объект анимации, который обеспечит изменение значения свойства от 100 до 200 в течение 10 секунд? Если это желательно сделать в разметке, тогда область `myCtrls:ShowNumberControl` можно изменить следующим образом:

```
<myCtrls:ShowNumberControl x:Name="myShowNumberCtrl" CurrentNumber="100">
    <myCtrls:ShowNumberControl.Triggers>
        <EventTrigger RoutedEvent = "myCtrls:ShowNumberControl.Loaded">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard TargetProperty = "CurrentNumber">
                        <Int32Animation From = "100" To = "200" Duration = "0:0:10"/>
                    </Storyboard>
                </EventTrigger.Actions>
            </EventTrigger>
        </myCtrls:ShowNumberControl.Triggers>
    </myCtrls:ShowNumberControl>
```

```

        </Storyboard>
    </BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
</myCtrls:ShowNumberControl.Triggers>
</myCtrls:ShowNumberControl>

```

После запуска приложения объект анимации не сможет найти подходящую цель и потому будет проигнорирован. Причина в том, что свойство `CurrentNumber` не было зарегистрировано как свойство зависимости! Чтобы устранить проблему, возвратимся в файл кода для специального элемента управления и полностью закомментируем текущую логику свойства (включая закрытое поддерживающее поле). Теперь поместим курсор внутрь области определения класса и введем фрагмент кода `propdp`. После ввода `propdp` два раза нажмем клавишу `<Tab>`. Фрагмент кода развернется в базовый шаблон свойства зависимости:

```

public int MyProperty
{
    get { return (int)GetValue(MyPropertyProperty); }
    set { SetValue(MyPropertyProperty, value); }
}

// Использовать DependencyProperty в качестве поддерживающего хранилища
// для MyProperty.
// Это сделает возможной анимацию, применение стилей, привязку и т.д.
public static readonly DependencyProperty MyPropertyProperty =
    DependencyProperty.Register("MyProperty", typeof(int),
        typeof(ownerclass), new PropertyMetadata(0));

```

Модифицируем шаблон, как показано ниже:

```

public int CurrentNumber
{
    get => (int)GetValue(CurrentNumberProperty);
    set => SetValue(CurrentNumberProperty, value);
}

public static readonly DependencyProperty CurrentNumberProperty =
    DependencyProperty.Register("CurrentNumber",
        typeof(int),
        typeof>ShowNumberControl),
        new UIPropertyMetadata(0));

```

Работа похожа на ту, что делалась в реализации свойства `Height`; тем не менее, фрагмент кода `propdp` регистрирует свойство непосредственно в теле, а не в статическом конструкторе (что хорошо). Также обратите внимание, что объект `UIPropertyMetadata` используется для определения стандартного целочисленного значения (0) вместо более сложного объекта `FrameworkPropertyMetadata`. В итоге получается простейшая версия `CurrentNumber` как свойства зависимости.

Добавление процедуры проверки достоверности данных

Несмотря на наличие свойства зависимости по имени `CurrentNumber`, анимация пока еще не наблюдается. Следующей корректировкой будет указание функции, вызываемой для выполнения проверки достоверности данных. В данном примере мы предположим, что нужно обеспечить нахождение значения свойства `CurrentNumber` в диапазоне между 0 и 500.

Добавим в метод `DependencyProperty.Register()` последний аргумент типа `ValidateValueCallback`, указывающий на метод по имени `ValidateCurrentNumber`.

Здесь `ValidateValueCallback` является делегатом, который может указывать только на методы, возвращающие тип `bool` и принимающие единственный аргумент типа `object`. Экземпляр `object` представляет присваиваемое новое значение. Реализация `ValidateCurrentNumber` должна возвращать `true`, если входное значение находится в ожидаемом диапазоне, и `false` в противном случае:

```
public static readonly DependencyProperty CurrentNumberProperty =
    DependencyProperty.Register("CurrentNumber",
        typeof(int),
        typeof>ShowNumberControl),
        new UIPropertyMetadata(100),
        new ValidateValueCallback(ValidateCurrentNumber));

public static bool ValidateCurrentNumber(object value)
{
    // Простое бизнес-правило: значение должно находиться в диапазоне между 0 и 500.
    return Convert.ToInt32(value) >= 0 && Convert.ToInt32(value) <= 500;
}
```

Реагирование на изменение свойства

Итак, допустимое число уже есть, но анимация по-прежнему отсутствует. Последнее изменение, которое потребуется внести — передать во втором аргументе конструктора `UIPropertyMetadata` объект `PropertyChangedCallback`. Данный делегат может указывать на любой метод, принимающий `DependencyObject` в первом параметре и `DependencyPropertyChangedEventArgs` во втором. Модифицируем код следующим образом:

```
// Обратите внимание на второй параметр конструктора UIPropertyMetadata.
public static readonly DependencyProperty CurrentNumberProperty =
    DependencyProperty.Register("CurrentNumber", typeof(int), typeof>ShowNumberControl),
    new UIPropertyMetadata(100,
        new PropertyChangedCallback(CurrentNumberChanged)),
    new ValidateValueCallback(ValidateCurrentNumber));
```

Конечной целью внутри метода `CurrentNumberChanged()` будет изменение свойства `Content` объекта `Label` на новое значение, присвоенное свойству `CurrentNumber`. Однако мы сталкиваемся с серьезной проблемой: метод `CurrentNumberChanged()` является статическим, т.к. он должен работать со статическим объектом `DependencyProperty`. Как тогда получить доступ к объекту `Label` для текущего экземпляра `ShowNumberControl`? Нужная ссылка содержится в первом параметре `DependencyObject`. Новое значение можно найти с применением входных аргументов события. Ниже показан необходимый код, который будет изменять свойство `Content` объекта `Label`:

```
private static void CurrentNumberChanged(DependencyObject depObj,
    DependencyPropertyChangedEventArgs args)
{
    // Привести DependencyObject к ShowNumberControl.
    ShowNumberControl c = (ShowNumberControl)depObj;

    // Получить элемент управления Label в ShowNumberControl.
    Label theLabel = c.numberDisplay;

    // Установить для Label новое значение.
    theLabel.Content = args.NewValue.ToString();
}
```


Видите, насколько долгий путь пришлось пройти, чтобы всего лишь изменить содержимое метки. Преимущество заключается в том, что теперь свойство зависимости `CurrentNumber` может быть целью для стиля WPF, объекта анимации, операции привязки данных и т.д. Снова запустив приложение, легко заметить, что значение изменяется во время выполнения.

На этом обзор свойств зависимости WPF завершен. Хотя теперь вы должны гораздо лучше понимать, что они позволяют делать, и как создавать собственные свойства подобного рода, имейте в виду, что многие детали здесь не были раскрыты.

Если вам однажды понадобится создавать множество собственных элементов управления, поддерживающих специальные свойства, то загляните в подраздел "Properties" ("Свойства") узла "WPF Fundamentals" ("Основы WPF") в документации .NET Framework 4.7 SDK. Там вы найдете намного больше примеров построения свойств зависимости, присоединяемых свойств, разнообразных способов конфигурирования метаданных и массу других подробных сведений.

Исходный код. Проект `CustomDepProp` доступен в подкаталоге `Chapter_25`.

Резюме

В главе рассматривались некоторые аспекты элементов управления WPF, начиная с обзора набора инструментов для элементов управления и роли диспетчеров компоновки (панелей). Первый пример был посвящен построению простого приложения текстового процессора. В нем демонстрировалось использование интегрированной в WPF функциональности проверки правописания, а также создание главного окна с системой меню, строкой состояния и панелью инструментов.

Более важно то, что вы научились строить команды WPF. Эти независимые от элементов управления события можно присоединять к элементу пользовательского интерфейса или входному жесту для автоматического наследования готовой функциональности (например, операций с буфером обмена).

Кроме того, вы узнали немало сведений о построении пользовательских интерфейсов в XAML и попутно ознакомились с интерфейсом Ink API, предлагаемым WPF. Вы также получили представление об операциях привязки данных WPF, включая использование класса `DataGrid` из WPF для отображения информации из специальной базы данных `AutoLot`.

Наконец, вы выяснили, что инфраструктура WPF добавляет уникальный аспект к традиционным программным примитивам .NET, в частности к свойствам и событиям. Как было показано, механизм свойств зависимости позволяет строить свойство, которое может интегрироваться с набором служб WPF (анимации, привязки данных, стили и т.д.). В качестве связанного замечания: механизм маршрутизируемых событий предоставляет событию способ распространяться вверх или вниз по дереву разметки.

ГЛАВА 26

Службы визуализации графики WPF

В настоящей главе мы рассмотрим возможности графической визуализации WPF. Вы увидите, что инфраструктура WPF предоставляет три отдельных способа визуализации графических данных: фигуры, рисунки и визуальные объекты. Разобравшись в преимуществах и недостатках каждого подхода, мы приступим к исследованию мира интерактивной двумерной графики с использованием классов из пространства имен `System.Windows.Shapes`. Затем будет показано, как с помощью рисунков и геометрических объектов визуализировать двумерные данные в легковесной манере. И, наконец, вы узнаете, каким образом добиться от визуального уровня максимальной функциональности и производительности.

Попутно будут затронуты многие связанные темы, такие как создание специальных кистей и перьев, применение графических трансформаций к визуализации и выполнение операций проверки попадания. В частности вы увидите, как можно упростить решение задач кодирования графики с помощью интегрированных инструментов Visual Studio и дополнительного средства под названием Inkscape.

На заметку! Графика является ключевым аспектом разработки WPF. Даже если вы не строите приложение с интенсивной графикой (вроде видеоигры или мультимедийного приложения), то рассматриваемые в главе темы критически важны при работе с такими службами, как шаблоны элементов управления, анимация и настройка привязки данных.

Службы графической визуализации WPF

Инфраструктура WPF использует особую разновидность графической визуализации, которая известна под названием *графика режима сохранения* (retained mode). Попросту говоря, это означает, что после применения разметки XAML или процедурного кода для генерирования графической визуализации инфраструктура WPF несет ответственность за сохранение визуальных элементов и обеспечение их корректной перерисовки и обновления в оптимальной манере. Таким образом, визуализируемые графические данные присутствуют постоянно, даже когда конечный пользователь скрывает изображение, изменяя размер окна или сворачивая его, перекрывая одно окно другим и т.д.

По разительному контрасту предшествующие версии API-интерфейсов графической визуализации от Microsoft (включая GDI+ в Windows Forms) были графическими системами *прямого режима* (immediate mode). В такой модели ответственность за корректное “запоминание” и обновление визуализируемых элементов на протяжении времени жизни приложения возлагалась на программиста. Например, в приложении Windows

Forms визуализация фигуры вроде прямоугольника предусматривала обработку события Paint (или переопределение виртуального метода OnPaint()), получение объекта Graphics для рисования прямоугольника и, что важнее всего, добавление инфраструктуры, обеспечивающей сохранение изображения в ситуации, когда пользователь изменил размеры окна (например, за счет создания переменных-членов для представления позиции прямоугольника и вызова метода Invalidate() во многих местах кода).

Переход от графики прямого режима к графике режима сохранения — действительно хорошее решение, т.к. программистам приходится писать и сопровождать гораздо меньший объем рутинного кода для поддержки графики. Однако речь не идет о том, что API-интерфейс графики WPF полностью отличается от более ранних инструментальных наборов визуализации. Например, как и GDI+, инфраструктура WPF поддерживает разнообразные типы объектов кистей и перьев, приемы проверки попадания, области отсечения, графические трансформации и т.д. Поэтому если у вас есть опыт работы с GDI+ (или GDI на языке C/C++), то вы уже имеете неплохое представление о способе выполнения базовой визуализации в WPF.

Варианты графической визуализации WPF

Как и с другими аспектами разработки приложений WPF, существует выбор из нескольких способов выполнения графической визуализации после принятия решения делать это посредством разметки XAML или процедурного кода C# (либо их комбинации). В частности, инфраструктура WPF предлагает следующие три индивидуальных подхода к визуализации графических данных.

- *Фигуры.* Инфраструктура WPF предоставляет пространство имен System.Windows.Shapes, в котором определено небольшое количество классов для визуализации двумерных геометрических объектов (прямоугольников, эллипсов, многоугольников и т.п.). Хотя такие типы просты в использовании и очень мощные, в случае непродуманного применения они могут привести к значительным накладным расходам памяти.
- *Рисунки и геометрические объекты.* Второй способ визуализации графических данных в WPF предполагает работу с классами, производными от абстрактного класса System.Windows.Media.Drawing. Используя классы, подобные GeometryDrawing или ImageDrawing (в дополнение к различным геометрическим объектам), можно визуализировать графические данные в более легковесной (но менее функциональной) манере.
- *Визуальные объекты.* Самый быстрый и легковесный способ визуализации графических данных в WPF предусматривает работу с визуальным уровнем, который доступен только через код C#. С применением классов, производных от System.Windows.Media.Visual, можно взаимодействовать непосредственно с графической подсистемой WPF.

Причина предоставления разных способов решения той же самой задачи (т.е. визуализации графических данных) связана с расходом памяти и в конечном итоге с производительностью приложения. Поскольку WPF является системой, интенсивно использующей графику, нет ничего необычного в том, что приложению требуется визуализировать сотни или даже тысячи различных изображений на поверхности окна, и выбор реализации (фигуры, рисунки или визуальные объекты) может оказать огромное влияние.

Важно понимать, что при построении приложения WPF высока вероятность использования всех трех подходов. В качестве эмпирического правила запомните: если нужен умеренный объем *интерактивных* графических данных, которыми может манипулиро-

вать пользователь (принимающих ввод от мыши, отображающих всплывающие подсказки и т.д.), то следует применять члены из пространства имен `System.Windows.Shapes`.

Напротив, рисунки и геометрические объекты лучше подходят, когда необходимо моделировать сложные и по большей части не интерактивные векторные графические данные с использованием разметки XAML или кода C#. Хотя рисунки и геометрические объекты способны реагировать на события мыши, а также поддерживают проверку попадания и операции перетаскивания, для выполнения таких действий обычно придется писать больше кода.

Наконец, если требуется самый быстрый способ визуализации значительных объемов графических данных, то должен быть выбран визуальный уровень. Например, предположим, что инфраструктура WPF применяется для построения научного приложения, которое должно отображать тысячи точек на графике данных. За счет использования визуального уровня точки на графике можно визуализировать оптимальным способом. Как будет показано далее в главе, визуальный уровень доступен только из кода C#, но не из разметки XAML.

Независимо от выбранного подхода (фигуры, рисунки и геометрические объекты или визуальные объекты) всегда будут применяться распространенные графические примитивы, такие как кисти (для заполнения ограниченных областей), перья (для рисования контуров) и объекты трансформаций (которые видоизменяют данные). Начнем исследование с классов из пространства имен `System.Windows.Shapes`.

На заметку! Инфраструктура WPF поставляется также с полнофункциональным API-интерфейсом, который можно использовать для визуализации и манипулирования трехмерной графикой, но в книге он не рассматривается. Если вас интересует внедрение трехмерной графики в свои приложения, тогда обратитесь в документацию .NET Framework 4.7 SDK.

Визуализация графических данных с использованием фигур

Члены пространства имен `System.Windows.Shapes` предлагают наиболее прямолинейный, интерактивный и самый затратный в плане расхода памяти способ визуализации двумерного изображения. Это небольшое пространство имен (расположенное в сборке `PresentationFramework.dll`) состоит всего из шести запечатанных классов, которые расширяют абстрактный базовый класс `Shape: Ellipse, Rectangle, Line, Polygon, Polyline` и `Path`.

Абстрактный класс `Shape` унаследован от класса `FrameworkElement`, который сам унаследован от `UIElement`. В указанных классах определены члены для работы с изменением размеров, всплывающими подсказками, курсорами мыши и т.п. Благодаря такой цепочке наследования при визуализации графических данных с применением классов, производных от `Shape`, объекты получаются почти такими же функциональными (с точки зрения взаимодействия с пользователем), как элементы управления WPF.

Скажем, для выяснения, щелкнул ли пользователь на визуализированном изображении, достаточно обработать событие `MouseDown`. Например, если написать следующую разметку XAML для объекта `Rectangle` внутри элемента управления `Grid` начального окна `Window`:

```
<Rectangle x:Name="myRect" Height="30" Width="30"
    Fill="Green" MouseDown="myRect_MouseDown"/>
```

то можно реализовать обработчик события `MouseDown`, который изменяет цвет фона прямоугольника в результате щелчка на нем:

```
private void myRect_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Изменить цвет прямоугольника в результате щелчка на нем.
    myRect.Fill = Brushes.Pink;
}
```

В отличие от других инструментальных наборов, предназначенных для работы с графикой, вам не придется писать громоздкий код инфраструктуры, в котором вручную сопоставляются координаты мыши с геометрическим объектом, выясняется попадание курсора внутрь границ, выполняется визуализация в неотображаемый буфер и т.д. Члены пространства имен `System.Windows.Shapes` просто реагируют на зарегистрированные вами события подобно типичному элементу управления WPF (`Button` и т.д.).

Недостаток всей готовой функциональности связан с тем, что фигуры потребляют довольно много памяти. Если строится научное приложение, которое рисует тысячи точек на экране, то использование фигур будет неудачным выбором (по существу таким же расточительным в плане памяти, как визуализация тысяч объектов `Button`). Тем не менее, когда нужно сгенерировать интерактивное двумерное векторное изображение, фигуры оказываются прекрасным вариантом.

Помимо функциональности, унаследованной от родительских классов `UIElement` и `FrameworkElement`, в классе `Shape` определено множество собственных членов, наиболее полезные из которых кратко описаны в табл. 26.1.

Таблица 26.1. Ключевые свойства базового класса `Shape`

Свойства	Описание
<code>DefiningGeometry</code>	Возвращает объект <code>Geometry</code> , который представляет общие размеры текущей фигуры. Этот объект содержит <i>только</i> точки, применяемые для визуализации данных, и не имеет никаких следов функциональности из класса <code>UIElement</code> или <code>FrameworkElement</code>
<code>Fill</code>	Позволяет указать объект кисти для заполнения внутренней области фигуры
<code>GeometryTransform</code>	Позволяет применять трансформацию к фигуре <i>до</i> ее визуализации на экране. Унаследованное (от <code>UIElement</code>) свойство <code>RenderTransform</code> применяет трансформацию <i>после</i> визуализации фигуры на экране
<code>Stretch</code>	Описывает, каким образом расположить фигуру в выделенном ей пространстве, например, по ее позиции внутри диспетчера компоновки. Это управляется с использованием соответствующего переключения <code>System.Windows.Media.Stretch</code>
<code>Stroke</code>	Определяет объект кисти или в ряде случаев объект пера (который на самом деле является замаскированным объектом кисти), применяемый для рисования границы фигуры
<code>StrokeDashArray</code> , <code>StrokeEndLineCap</code> , <code>StrokeStartLineCap</code> , <code>StrokeThickness</code>	Эти (и другие) свойства, связанные со штрихами, управляют тем, как сконфигурированы линии при рисовании границ фигуры. В большинстве случаев данные свойства будут конфигурировать кисть, используемую для рисования границы или линии

На заметку! Если вы забудете установить свойства `Fill` и `Stroke`, то WPF предоставит “невидимые” кисти, вследствие чего фигура не будет видна на экране!

Добавление прямоугольников, эллипсов и линий на поверхность Canvas

Мы построим приложение WPF, которое способно визуализировать фигуры, с применением XAML и C#, и попутно исследуем процесс проверки попадания. Создадим новый проект приложения WPF по имени `RenderingWithShapes` и изменим заголовок главного окна в `MainWindow.xaml` на `Fun with Shapes!`. Модифицируем первоначальную разметку XAML для элемента `Window`, заменив `Grid` панелью `DockPanel`, которая содержит (пока пустые) элементы `ToolBar` и `Canvas`. Обратите внимание, что каждому содержащемуся элементу посредством свойства `Name` назначается подходящее имя.

```
<DockPanel LastChildFill="True">
  <ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
  </ToolBar>
  <Canvas Background="LightBlue" Name="canvasDrawingArea"/>
</DockPanel>
```

Заполним элемент `ToolBar` набором объектов `RadioButton`, каждый из которых содержит объект специфического класса, производного от `Shape`. Легко заметить, что каждому элементу `RadioButton` назначается то же самое групповое имя `GroupName` (чтобы обеспечить взаимное исключение) и также подходящее индивидуальное имя.

```
<ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
  <RadioButton Name="circleOption" GroupName="shapeSelection">
    <Ellipse Fill="Green" Height="35" Width="35" />
  </RadioButton>
  <RadioButton Name="rectOption" GroupName="shapeSelection">
    <Rectangle Fill="Red" Height="35" Width="35" RadiusY="10" RadiusX="10" />
  </RadioButton>
  <RadioButton Name="lineOption" GroupName="shapeSelection">
    <Line Height="35" Width="35" StrokeThickness="10" Stroke="Blue"
      X1="10" Y1="10" Y2="25" X2="25"
      StrokeStartLineCap="Triangle" StrokeEndLineCap="Round" />
  </RadioButton>
</ToolBar>
```

Как видите, объявление объектов `Rectangle`, `Ellipse` и `Line` в разметке XAML довольно прямолинейно и требует лишь минимальных комментариев. Вспомните, что свойство `Fill` позволяет указать кисть для рисования внутренностей фигуры. Когда нужна кисть сплошного цвета, можно просто задать жестко закодированную строку известных значений, а соответствующий преобразователь типа сгенерирует корректный объект. Интересная характеристика типа `Rectangle` связана с тем, что в нем определены свойства `RadiusX` и `RadiusY`, позволяющие визуализировать скругленные углы.

Объект `Line` представлен своими начальной и конечной точками с использованием свойств `X1`, `X2`, `Y1` и `Y2` (учитывая, что *высота* и *ширина* при описании линии имеют мало смысла). Здесь устанавливается несколько дополнительных свойств, которые управляют тем, как визуализируются начальная и конечная точки объекта `Line`, а также настраивают параметры штриха. На рис. 26.1 показана визуализированная панель инструментов в визуальном конструкторе WPF среды Visual Studio.

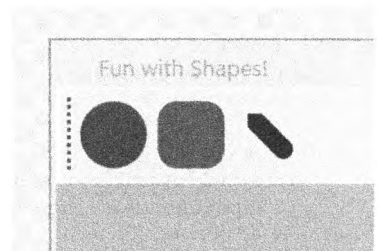


Рис. 26.1. Применение объектов `Shape` в качестве содержимого для набора элементов `RadioButton`

С помощью окна Properties (Свойства) среды Visual Studio создадим обработчик события `MouseLeftButtonDown` для `Canvas` и обработчик события `Click` для каждого элемента `RadioButton`. Цель заключается в том, чтобы в коде C# визуализировать выбранную фигуру (круг, квадрат или линию), когда пользователь щелкает внутри `Canvas`. Первым делом определим следующее вложенное перечисление (и соответствующую переменную-член) внутри класса, производного от `Window`:

```
public partial class MainWindow : Window
{
    private enum SelectedShape
    { Circle, Rectangle, Line }

    private SelectedShape _currentShape;
    ...
}
```

В каждом обработчике `Click` установим переменную-член `currentShape` в корректное значение `SelectedShape`:

```
private void circleOption_Click(object sender, RoutedEventArgs e)
{
    _currentShape = SelectedShape.Circle;
}

private void rectOption_Click(object sender, RoutedEventArgs e)
{
    _currentShape = SelectedShape.Rectangle;
}

private void lineOption_Click(object sender, RoutedEventArgs e)
{
    _currentShape = SelectedShape.Line;
}
```

Посредством обработчика события `MouseLeftButtonDown` элемента `Canvas` будет визуализироваться подходящая фигура (предопределенного размера) в начальной точке, которая соответствует позиции X,Y курсора мыши. Ниже приведена полная реализация с последующим анализом:

```
private void canvasDrawingArea_MouseLeftButtonDown(object sender,
                                                    MouseButtonEventArgs e)
{
    Shape shapeToRender = null;
    // Сконфигурировать корректную фигуру для рисования.
    switch (_currentShape)
    {
        case SelectedShape.Circle:
            shapeToRender = new Ellipse() { Fill = Brushes.Green, Height = 35, Width = 35 };
            break;
        case SelectedShape.Rectangle:
            shapeToRender = new Rectangle()
            { Fill = Brushes.Red, Height = 35, Width = 35, RadiusX = 10, RadiusY = 10 };
            break;
        case SelectedShape.Line:
            shapeToRender = new Line()
            {
                Stroke = Brushes.Blue,
                StrokeThickness = 10,
                X1 = 0, X2 = 50, Y1 = 0, Y2 = 50,
            }
    }
}
```

```

        StrokeStartLineCap= PenLineCap.Triangle,
        StrokeEndLineCap = PenLineCap.Round
    };
    break;
default:
    return;
}

// Установить верхний левый угол для рисования на холсте.
Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);

// Нарисовать фигуру.
canvasDrawingArea.Children.Add(shapeToRender);
}

```

На заметку! Возможно, вы заметили, что объекты `Ellipse`, `Rectangle` и `Line`, создаваемые в методе `canvasDrawingArea_MouseLeftButtonDown()`, имеют те же настройки свойств, что и соответствующие определения XAML. Как и можно было ожидать, код можно упростить, но это требует понимания объектных ресурсов WPF, которые будут рассматриваться в главе 29.

В коде проверяется переменная-член `_currentShape` с целью создания корректного объекта, производного от `Shape`. Затем устанавливаются координаты левого верхнего угла внутри `Canvas` с использованием входного объекта `MouseButtonEventArgs`. Наконец, в коллекцию объектов `UIElement`, поддерживаемую `Canvas`, добавляется новый производный от `Shape` объект. Если запустить программу прямо сейчас, то она должна позволить щелкать левой кнопкой мыши где угодно на холсте и визуализировать в позиции щелчка выбранную фигуру.

Удаление прямоугольников, эллипсов и линий с поверхности Canvas

Имея в распоряжении элемент `Canvas` с коллекцией объектов, может возникнуть вопрос: как динамически удалить элемент, скажем, в ответ на щелчок пользователя правой кнопкой мыши на фигуре? Это делается с помощью класса `VisualTreeHelper` из пространства имен `System.Windows.Media`. Роль “визуальных деревьев” и “логических деревьев” более подробно объясняется в главе 27, а пока обрабатываем событие `MouseRightButtonDown` объекта `Canvas` и реализуем соответствующий обработчик:

```

private void canvasDrawingArea_MouseRightButtonDown(object sender,
                                                    MouseButtonEventArgs e)
{
    // Сначала получить координаты X,Y позиции, где пользователь выполнил щелчок.
    Point pt = e.GetPosition((Canvas)sender);

    // Использовать метод HitTest() класса VisualTreeHelper, чтобы
    // выявить, щелкнул ли пользователь на элементе внутри Canvas.
    HitTestResult result = VisualTreeHelper.HitTest(canvasDrawingArea, pt);

    // Если переменная result не равна null, то щелчок произведен на фигуре.
    if (result != null)
    {
        // Получить фигуру, на которой совершен щелчок, и удалить ее из Canvas.
        canvasDrawingArea.Children.Remove(result.VisualHit as Shape);
    }
}

```


Метод `canvasDrawingArea_MouseRightButtonDown()` начинается с получения точных координат X,Y позиции, где пользователь щелкнул внутри Canvas, и проверки попадания посредством статического метода `VisualTreeHelper.HitTest()`. Возвращаемое значение — объект `HitTestResult` — будет установлено в `null`, если пользователь выполнил щелчок не на `UIElement` внутри Canvas. Если значение `HitTestResult` не равно `null`, тогда с помощью свойства `VisualHit` можно получить объект `UIElement`, на котором был совершен щелчок, и привести его к типу, производному от `Shape` (вспомните, что Canvas может содержать любой `UIElement`, а не только фигуры). Детали, связанные с “визуальным деревом”, будут изложены в главе 27.

На заметку! По умолчанию метод `VisualTreeHelper.HitTest()` возвращает объект `UIElement` самого верхнего уровня, на котором совершен щелчок, и не предоставляет информацию о других объектах, расположенных под ним (т.е. перекрытых в Z-порядке).

В результате внесенных модификаций должна появиться возможность добавления фигуры на Canvas щелчком левой кнопкой мыши и ее удаления щелчком правой кнопкой мыши.

К настоящему моменту мы применяли объекты типов, производных от `Shape`, для визуализации содержимого элементов `RadioButton` с использованием разметки XAML и заполняли Canvas в коде C#. Во время исследования роли кистей и графических трансформаций в данный пример будет добавлена дополнительная функциональность. К слову, в другом примере главы будут иллюстрироваться приемы перетаскивания на объектах `UIElement`. А пока давайте рассмотрим оставшиеся члены пространства имен `System.Windows.Shapes`.

Работа с элементами `Polyline` и `Polygon`

В текущем примере используются только три класса, производных от `Shape`. Остальные дочерние классы (`Polyline`, `Polygon` и `Path`) чрезвычайно трудно корректно визуализировать без инструментальной поддержки (такой как инструмент Microsoft Blend, сопровождающий Visual Studio и предназначенный для разработчиков WPF, или другие инструменты, которые могут создавать векторную графику) — просто потому, что они требуют определения большого количества точек для своего выходного представления. Вскоре вы узнаете о роли инструмента Microsoft Blend, а пока займемся кратким обзором оставшихся типов `Shapes`.

Тип `Polyline` позволяет определить коллекцию координат X,Y (через свойство `Points`) для рисования последовательности линейных сегментов, не требующих замыкания. Тип `Polygon` похож, но запрограммирован так, что всегда замыкает контур, соединяя начальную точку с конечной, и заполняет внутреннюю область с помощью указанной кисти. Предположим, что в редакторе `Kaxaml` создан следующий элемент `StackPanel`:

```
<!-- Элемент Polyline не замыкает автоматически конечные точки -->
<Polyline Stroke="Red" StrokeThickness="20" StrokeLineJoin="Round"
  Points="10,10 40,40 10,90 300,50"/>
<!-- Элемент Polygon всегда замыкает конечные точки -->
<Polygon Fill="AliceBlue" StrokeThickness="5" Stroke="Green"
  Points="40,10 70,80 10,50" />
```

На рис. 26.2 показан визуализированный вывод в `Kaxaml`.

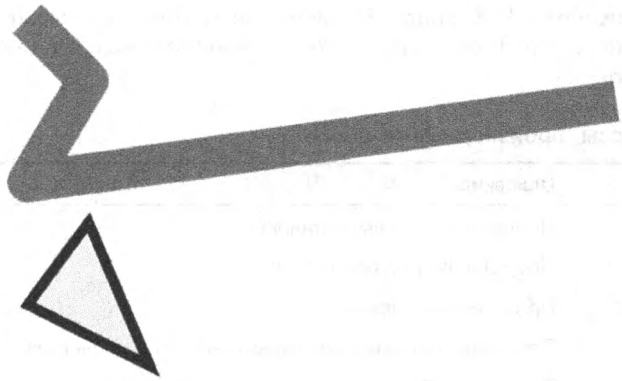


Рис. 26.2. Элементы Polyline и Polygon

Работа с элементом Path

Применяя только типы `Rectangle`, `Ellipse`, `Polygon`, `Polyline` и `Line`, нарисовать детализированное двумерное векторное изображение было бы исключительно трудно, т.к. упомянутые примитивы не позволяют легко фиксировать графические данные, подобные кривым, объединениям перекрывающихся данных и т.д. Последний производный от `Shape` класс, `Path`, предоставляет возможность определения сложных двумерных графических данных в виде коллекции независимых геометрических объектов. После того, как коллекция таких геометрических объектов определена, ее можно присвоить свойству `Data` класса `Path`, где она будет использоваться для визуализации сложного двумерного изображения.

Свойство `Data` получает объект производного от `System.Windows.Media.Geometry` класса, который содержит ключевые члены, кратко описанные в табл. 26.2.

Таблица 26.2. Избранные члены класса `System.Windows.Media.Geometry`

Член	Описание
<code>Bounds</code>	Устанавливает текущий ограничивающий прямоугольник, который содержит геометрический объект
<code>FillContains()</code>	Выясняет, находится ли заданный объект <code>Point</code> (или другой объект <code>Geometry</code>) внутри границ определенного класса, производного от <code>Geometry</code> . Это полезно при вычислениях для проверки попадания
<code>GetArea()</code>	Возвращает общую область, занятую объектом производного от <code>Geometry</code> типа
<code>GetRenderBounds()</code>	Возвращает объект <code>Rect</code> , содержащий наименьший из возможных прямоугольник, который может быть применен для визуализации объекта класса, производного от <code>Geometry</code>
<code>Transform</code>	Назначает геометрическому объекту экземпляр класса <code>Transform</code> для изменения визуализации

Классы, которые расширяют класс `Geometry` (табл. 26.3), выглядят очень похожими на свои аналоги, производные от `Shape`. Например, класс `EllipseGeometry` имеет члены, подобные членам класса `Ellipse`. Крупное отличие связано с тем, что производные от `Geometry` классы не знают, каким образом визуализировать себя напрямую.

поскольку они не являются `UIElement`. Взамен классы, производные от `Geometry`, представляют всего лишь коллекцию данных о точках, которая указывает объекту `Path`, как их визуализировать.

Таблица 26.3. Классы, производные от `Geometry`

Класс	Описание
<code>LineGeometry</code>	Представляет прямую линию
<code>RectangleGeometry</code>	Представляет прямоугольник
<code>EllipseGeometry</code>	Представляет эллипс
<code>GeometryGroup</code>	Позволяет группировать вместе несколько объектов <code>Geometry</code>
<code>CombinedGeometry</code>	Позволяет объединять два разных объекта <code>Geometry</code> в единую фигуру
<code>PathGeometry</code>	Представляет фигуру, образованную из линий и кривых

В показанной далее разметке для элемента `Path` используется несколько производных от `Geometry` типов. Обратите внимание, что свойство `Data` объекта `Path` устанавливается в объект `GeometryGroup`, который содержит объекты других производных от `Geometry` классов, таких как `EllipseGeometry`, `RectangleGeometry` и `LineGeometry`. Результат представлен на рис. 26.3.

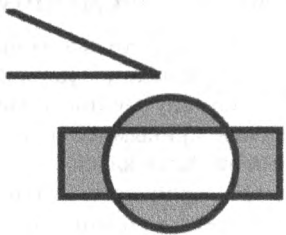


Рис. 26.3. Элемент `Path`, содержащий разнообразные объекты `Geometry`

```
<!-- Элемент Path содержит набор объектов
      Geometry, установленный в свойстве Data -->
<Path Fill = "Orange" Stroke = "Blue"
      StrokeThickness = "3">
  <Path.Data>
    <GeometryGroup>
      <EllipseGeometry Center = "75,70"
        RadiusX = "30" RadiusY = "30" />
      <RectangleGeometry Rect = "25,55 100 30" />
      <LineGeometry StartPoint="0,0" EndPoint="70,30" />
      <LineGeometry StartPoint="70,30" EndPoint="0,30" />
    </GeometryGroup>
  </Path.Data>
</Path>
```

Изображение на рис. 26.3 может быть визуализировано с применением показанных ранее классов `Line`, `Ellipse` и `Rectangle`. Однако это потребовало бы помещения различных объектов `UIElement` в память. Когда для моделирования точек рисуемого изображения используются геометрические объекты, а затем коллекция геометрических объектов помещается в контейнер, который способен визуализировать данные (`Path` в рассматриваемом случае), то тем самым сокращается расход памяти.

Теперь вспомните, что класс `Path` имеет ту же цепочку наследования, что и любой член пространства имен `System.Windows.Shapes`, а потому обладает возможностью отправлять такие же уведомления о событиях, как другие элементы `UIElement`. Следовательно, если определить тот же самый элемент `<Path>` в проекте Visual Studio, тогда выяснить, что пользователь щелкнул в любом месте линии, можно будет за счет обработки события мыши (не забывайте, что редактор `Кахам! не разрешает обрабатывать события для написанной разметки`).

“Мини-язык” моделирования путей

Из всех классов, перечисленных в табл. 26.3, класс `PathGeometry` наиболее сложен для конфигурирования в терминах XAML и кода. Причина объясняется тем фактом, что каждый сегмент `PathGeometry` состоит из объектов, содержащих разнообразные сегменты и фигуры (скажем, `ArcSegment`, `BezierSegment`, `LineSegment`, `PolyBezierSegment`, `PolyLineSegment`, `PolyQuadraticBezierSegment` и т.д.). Вот пример объекта `Path`, свойство `Data` которого было установлено в элемент `PathGeometry`, состоящий из различных фигур и сегментов:

```
<Path Stroke="Black" StrokeThickness="1">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="10,50">
          <PathFigure.Segments>
            <BezierSegment
              Point1="100,0"
              Point2="200,200"
              Point3="300,100"/>
            <LineSegment Point="400,100" />
            <ArcSegment
              Size="50,50" RotationAngle="45"
              IsLargeArc="True" SweepDirection="Clockwise"
              Point="200,100"/>
          </PathFigure.Segments>
        </PathFigure>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```

По правде говоря, лишь немногим программистам придется когда-либо вручную строить сложные двумерные изображения, напрямую описывая объекты производных от `Geometry` или `PathSegment` классов. Позже в главе вы узнаете, как преобразовывать векторную графику в операторы “мини-языка” моделирования путей, которые можно применять в разметке XAML.

Даже с учетом содействия со стороны упомянутых ранее инструментов объем разметки XAML, требуемой для определения сложных объектов `Path`, может быть устрашающе большим, т.к. данные состоят из полных описаний различных объектов классов, производных от `Geometry` или `PathSegment`. Для того чтобы создавать более лаконичную разметку, в классе `Path` поддерживается специализированный “мини-язык”.

Например, вместо установки свойства `Data` объекта `Path` в коллекцию объектов производных от `Geometry` и `PathSegment` классов его можно установить в одиночный строковый литерал, содержащий набор известных символов и различных значений, которые определяют фигуру, подлежащую визуализации. Ниже приведен простой пример, а его результирующий вывод показан на рис. 26.4:

```
<Path Stroke="Black" StrokeThickness="3"
  Data="M 10,75 C 70,15 250,270 300,175 H 240" />
```

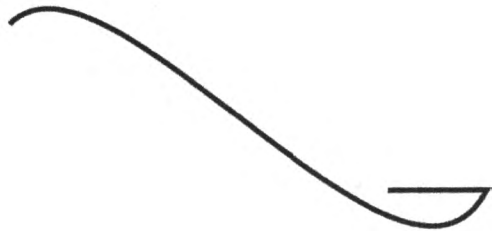


Рис. 26.4. “Мини-язык” моделирования путей позволяет компактно описывать объектную модель `Geometry/PathSegment`

Команда *M* (от *move* — переместить) принимает координаты *X,Y* позиции, которая представляет начальную точку рисования. Команда *C* (от *curve* — кривая) принимает последовательность точек для визуализации кривой (точнее кубической кривой Безье), а команда *H* (от *horizontal* — горизонталь) рисует горизонтальную линию.

И снова следует отметить, что вам очень редко придется вручную строить или анализировать строковый литерал, содержащий инструкции мини-языка моделирования путей. Тем не менее, цель в том, чтобы разметка XAML, генерируемая специализированными инструментами, не казалась совершенно непонятной. Если вас интересуют детали этой конкретной грамматики, тогда обращайтесь в раздел “Path Markup Syntax” (“Синтаксис разметки путей”) документации .NET Framework 4.7 SDK.

Кисти и перья WPF

Каждый способ графической визуализации (фигуры, рисование и геометрические объекты, а также визуальные объекты) интенсивно использует кисти, которые позволяют управлять заполнением внутренней области двумерной фигуры. В WPF предоставляются шесть разных типов кистей, и все они расширяют класс `System.Windows.Media.Brush`. Несмотря на то что `Brush` является абстрактным классом, его потомки, описанные в табл. 26.4, могут применяться для заполнения области содержимым почти любого мыслимого вида.

Таблица 26.4. Классы, производные от `Brush`

Класс	Описание
<code>DrawingBrush</code>	Заполняет область с помощью объекта производного от <code>Drawing</code> класса (<code>GeometryDrawing</code> , <code>ImageDrawing</code> или <code>VideoDrawing</code>)
<code>OmageBrush</code>	Заполняет область изображением (представленным посредством объекта <code>ImageSource</code>)
<code>LinearGradientBrush</code>	Заполняет область линейным градиентом
<code>RadialGradientBrush</code>	Заполняет область радиальным градиентом
<code>SolidColorBrush</code>	Заполняет область сплошным цветом, указанным в свойстве <code>Color</code>
<code>VisualBrush</code>	Заполняет область с помощью объекта производного от <code>Visual</code> класса (<code>DrawingVisual</code> , <code>Viewport3DVisual</code> и <code>ContentVisual</code>)

Классы `DrawingBrush` и `VisualBrush` позволяют строить кисть на основе существующего класса, производного от `Drawing` или `Visual`. Такие классы кистей используются при работе с двумя другими способами визуализации графики WPF (рисунками или визуальными объектами) и будут объясняться далее в главе.

Класс `ImageBrush` позволяет строить кисть, отображающую данные изображения из внешнего файла или встроенного ресурса приложения, который указан в его свойстве `ImageSource`. Оставшиеся типы кистей (`LinearGradientBrush` и `RadialGradientBrush`) довольно просты в применении, хотя требуемая разметка XAML может оказаться многословной. К счастью, в среде Visual Studio поддерживаются интегрированные редакторы кистей, которые облегчают задачу генерации стилизованных кистей.

Конфигурирование кистей с использованием Visual Studio

Давайте обновим приложение WPF для рисования `RenderingShapes`, чтобы использовать в нем более интересные кисти. В трех фигурах, которые были задействованы до сих пор при визуализации данных в панели инструментов, применяются простые

сплошные цвета, так что их значения можно зафиксировать с помощью простых строковых литералов. Чтобы сделать задачу чуть более интересной, теперь мы будем использовать интегрированный редактор кистей. Удостоверившись в том, что редактор XAML начального окна открыт в IDE-среде, выберем элемент `Ellipse`. В окне `Properties` отыщем категорию `Brush` (Кисть) и щелкнем на свойстве `Fill` (рис. 26.5).

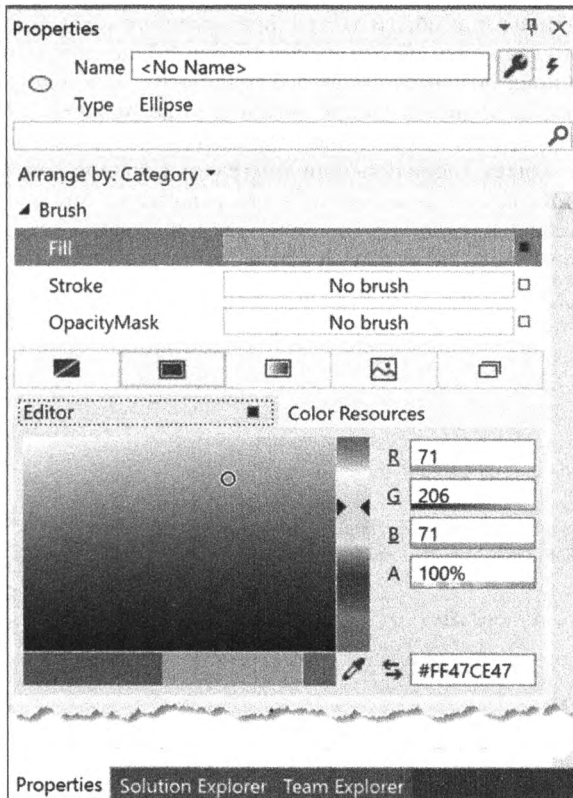


Рис. 26.5. Любое свойство, которое требует кисти, может быть сконфигурировано с помощью интегрированного редактора кистей

В верхней части редактора кистей находится набор свойств, которые являются “совместимыми с кистью” для выбранного элемента (т.е. `Fill`, `Stroke` и `OpacityMask`). Под ними расположен набор вкладок, которые позволяют конфигурировать разные типы кистей, включая текущую кисть со сплошным цветом. Для управления цветом текущей кисти можно применять инструмент выбора цвета, а также ползунки `ARGB` (alpha, red, green, blue — прозрачность, красный, зеленый, синий). С помощью этих ползунков и связанной с ними области выбора цвета можно создавать сплошной цвет любого вида. Используем указанные инструменты для изменения цвета в свойстве `Fill` элемента `Ellipse` и посмотрим результирующую разметку XAML. Как видите, цвет сохраняется в виде шестнадцатеричного значения:

```
<Ellipse Fill="#FF47CE47" Height="35" Width="35" />
```

Что более интересно, тот же самый редактор позволяет конфигурировать и градиентные кисти, которые применяются для определения последовательностей цветов и точек перехода цветов. Вспомните, что редактор кистей предлагает набор вкладок, пер-

вая из которых позволяет установить *пустую кисть* для отсутствующего визуализированного вывода. Остальные четыре дают возможность установить кисть сплошного цвета (как только что было показано), градиентную кисть, мозаичную кисть и кисть с изображением.

Щелкнем на вкладке градиентной кисти; редактор отобразит несколько новых настроек (рис. 26.6). Три кнопки в нижнем левом углу позволяют выбрать линейный градиент, радиальный градиент или обратить градиентные переходы. Полоса внизу покажет текущий цвет каждого градиентного перехода, который будет представлен специальным ползунком. Перетаскивая ползунок по полосе градиента, можно управлять смещением градиента. Кроме того, щелкая на конкретном ползунке, можно изменять цвет определенного градиентного перехода с помощью селектора цвета. Наконец, щелчок прямо на полосе градиента позволяет добавлять дополнительные градиентные переходы.

Потратьте некоторое время на освоение этого редактора. Мы построим радиальную градиентную кисть, содержащую три градиентных перехода, и установим их цвета. На рис. 26.6 показан пример кисти, использующей три разных оттенка зеленого цвета.

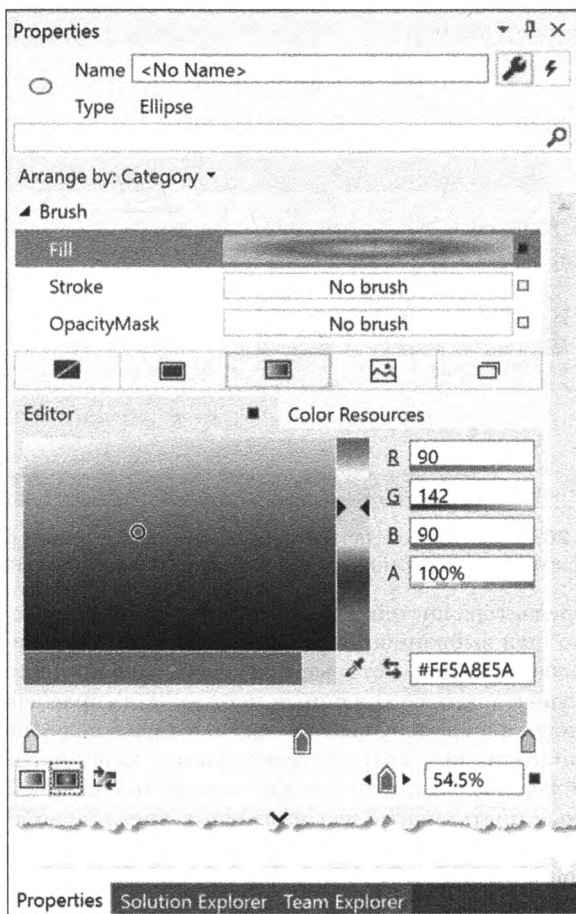


Рис. 26.6. Редактор кистей Visual Studio позволяет строить базовые градиентные кисти

В результате IDE-среда обновит разметку XAML, добавив набор специальных кистей и присвоив их совместимым с кистями свойствам (свойство `Fill` элемента `Ellipse` в рассматриваемом примере) с применением синтаксиса "свойство-элемент":

```
<Ellipse Height="35" Width="35">
  <Ellipse.Fill>
    <RadialGradientBrush>
      <GradientStop Color="#FF77F177" Offset="0"/>
      <GradientStop Color="#FF11E611" Offset="1"/>
      <GradientStop Color="#FF5A8E5A" Offset="0.545"/>
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

Конфигурирование кистей в коде

Теперь, когда мы построили специальную кисть для определения XAML элемента `Ellipse`, соответствующий код C# устарел в том, что он по-прежнему будет визуализировать круг со сплошным зеленым цветом. Для восстановления синхронизации модифицируем нужный оператор `case`, чтобы использовать только что созданную кисть. Ниже показано необходимое обновление, которое выглядит более сложным, чем можно было ожидать, т.к. шестнадцатеричное значение преобразуется в подходящий объект `Color` посредством класса `System.Windows.Media.ColorConverter` (результат изменения представлен на рис. 26.7):

```
case SelectedShape.Circle:
    shapeToRender = new Ellipse() { Height = 35, Width = 35 };
    // Создать кисть RadialGradientBrush в коде.
    RadialGradientBrush brush = new RadialGradientBrush();
    brush.GradientStops.Add(new GradientStop(
        (Color)ColorConverter.ConvertFromString("#FF77F177"), 0));
    brush.GradientStops.Add(new GradientStop(
        (Color)ColorConverter.ConvertFromString("#FF11E611"), 1));
    brush.GradientStops.Add(new GradientStop(
        (Color)ColorConverter.ConvertFromString("#FF5A8E5A"), 0.545));
    shapeToRender.Fill = brush;
    break;
```

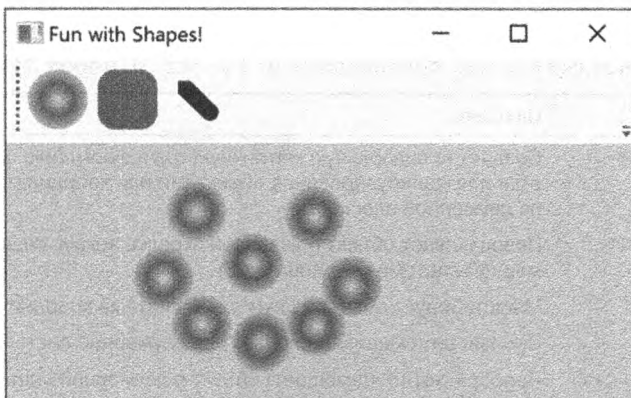


Рис. 26.7. Рисование более интересных кругов

Кстати, объекты GradientStop можно строить, указывая простой цвет в качестве первого параметра конструктора с применением перечисления Colors, которое дает сконфигурированный объект Color:

```
GradientStop g = new GradientStop(Colors.Aquamarine, 1);
```

Если требуется более тонкий контроль, то можно передавать объект Color, сконфигурированный в коде, например:

```
Color myColor = new Color() { R = 200, G = 100, B = 20, A = 40 };
GradientStop g = new GradientStop(myColor, 34);
```

Разумеется, использование перечисления Colors и класса Color не ограничивается градиентными кистями. Их можно применять всякий раз, когда необходимо представить значение цвета в коде.

Конфигурирование перьев

В сравнении с кистями *перо* представляет собой объект для рисования границ геометрических объектов или в случае класса Line либо PolyLine — самого линейного геометрического объекта. В частности, класс Pen позволяет рисовать линию указанной толщины, представленной значением типа double. Вдобавок объект Pen может быть сконфигурирован с помощью того же самого вида свойств, что и в классе Shape, таких как начальный и конечный концы пера, шаблоны точек-тире и т.д. Например, для определения атрибутов пера к определению фигуры можно добавить следующую разметку:

```
<Pen Thickness="10" LineJoin="Round" EndLineCap="Triangle" StartLineCap="Round"/>
```

Во многих случаях создавать объект Pen непосредственно не придется, потому что это делается косвенно, когда присваиваются значения свойствам вроде StrokeThickness производного от Shape типа (а также других типов UIElement). Однако строить специальный объект Pen удобно при работе с типами, производными от Drawing (которые рассматриваются позже в главе). Среда Visual Studio не располагает редактором перьев как таковым, но позволяет конфигурировать все свойства, связанные со штрихами, для выбранного элемента с использованием окна Properties.

Применение графических трансформаций

В завершение обсуждения фигур рассмотрим тему *трансформаций*. Инфраструктура WPF поставляется с многочисленными классами, которые расширяют абстрактный базовый класс System.Windows.Media.Transform. В табл. 26.5 кратко описаны основные классы, производные от Transform.

Таблица 26.5. Основные классы, производные от System.Windows.Media.Transform

Класс	Описание
MatrixTransform	Создает произвольную матричную трансформацию, которая используется для манипулирования объектами или координатными системами на двумерной плоскости
RotateTransform	Поворачивает объект по часовой стрелке вокруг указанной точки в двумерной системе координат (X,Y)
ScaleTransform	Масштабирует объект в двумерной системе координат (X,Y)
SkewTransform	Производит сдвиг объекта в двумерной системе координат (X,Y)
TranslateTransform	Преобразует (перемещает) объект в двумерной системе координат (X,Y)
TransformGroup	Представляет комбинированный объект Transform, состоящий из других объектов Transform

Трансформации могут применяться к любым объектам `UIElement` (например, к объектам производных от `Shape` классов, а также к элементам управления `Button`, `TextBox` и т.п.). Используя классы трансформаций, можно визуализировать графические данные под заданным углом, сжимать изображение на поверхности и растягивать, сжимать либо поворачивать целевой элемент разными способами.

На заметку! Хотя объекты трансформаций могут применяться повсеместно, вы сочтете их наиболее удобными при работе с анимацией WPF и специальными шаблонами элементов управления. Как будет показано далее в главе, анимацию WPF можно использовать для включения в специальный элемент управления визуальных подсказок, предназначенных конечному пользователю.

Назначать целевому объекту (`Button`, `Path` и т.д.) трансформацию (либо целый набор трансформаций) можно с помощью двух общих свойств, `LayoutTransform` и `RenderTransform`.

Свойство `LayoutTransform` удобно тем, что трансформация происходит *перед* визуализацией элементов в диспетчере компоновки и потому не влияет на операции Z-упорядочивания (т.е. трансформируемые данные изображений не перекрываются).

С другой стороны, трансформация из свойства `RenderTransform` иницируется после того, как элементы попали в свои контейнеры, поэтому вполне возможно, что элементы будут трансформированы с перекрытием друг друга в зависимости от того, как они организованы в контейнере.

Первый взгляд на трансформации

Вскоре мы добавим к проекту `RenderingWithShapes` некоторую трансформирующую логику. Чтобы увидеть объект трансформации в действии, откроем редактор `Кахамл`, определим внутри корневого элемента `Page` или `Window` простой элемент `StackPanel` и установим свойство `Orientation` в `Horizontal`. Далее добавим следующий элемент `Rectangle`, который будет нарисован под углом в 45 градусов с применением объекта `RotateTransform`:

```
<!-- Элемент Rectangle с трансформацией поворотом -->
<Rectangle Height ="100" Width ="40" Fill ="Red">
  <Rectangle.LayoutTransform>
    <RotateTransform Angle ="45"/>
  </Rectangle.LayoutTransform>
</Rectangle>
```

Здесь элемент `Button` сжимается на поверхности на 20 градусов посредством трансформации `SkewTransform`:

```
<!-- Элемент Button с трансформацией сжатием -->
<Button Content ="Click Me!" Width="95" Height="40">
  <Button.LayoutTransform>
    <SkewTransform AngleX ="20" AngleY ="20"/>
  </Button.LayoutTransform>
</Button>
```

Для полноты картины ниже приведен элемент `Ellipse`, масштабированный на 20% с помощью трансформации `ScaleTransform` (обратите внимание на значения, установленные в свойствах `Height` и `Width`), а также элемент `TextBox`, к которому применена группа объектов трансформации:

```

<!-- Элемент Ellipse, масштабированный на 20% -->
<Ellipse Fill="Blue" Width="5" Height="5">
  <Ellipse.LayoutTransform>
    <ScaleTransform ScaleX="20" ScaleY="20"/>
  </Ellipse.LayoutTransform>
</Ellipse>
<!-- Элемент TextBox, повернутый и скошенный -->
<TextBox Text="Me Too!" Width="50" Height="40">
  <TextBox.LayoutTransform>
    <TransformGroup>
      <RotateTransform Angle="45"/>
      <SkewTransform AngleX="5" AngleY="20"/>
    </TransformGroup>
  </TextBox.LayoutTransform>
</TextBox>

```

Следует отметить, что в случае применения трансформации выполнять какие-либо ручные вычисления для реагирования на проверку попадания, перемещение фокуса ввода и аналогичные действия не придется. Графический механизм WPF самостоятельно решает такие задачи. Например, на рис. 26.8 можно видеть, что элемент `TextBox` по-прежнему реагирует на клавиатурный ввод.

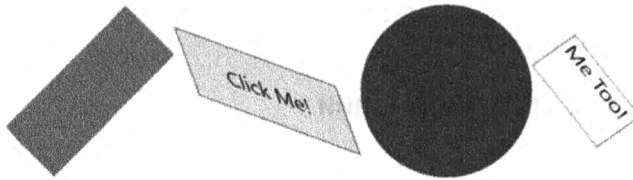


Рис. 26.8. Результат применения объектов графических трансформаций

Трансформация данных Canvas

Теперь давайте внедрим в пример `RenderingWithShapes` логику трансформации. Помимо применения объектов трансформации к одиночному элементу (`Rectangle`, `TextBox` и т.д.) их можно также применять к диспетчеру компоновки, чтобы трансформировать все внутренние данные. Например, всю панель `DockPanel` главного окна можно было бы визуализировать под углом:

```

<DockPanel LastChildFill="True">
  <DockPanel.LayoutTransform>
    <RotateTransform Angle="45"/>
  </DockPanel.LayoutTransform>
  ...
</DockPanel>

```

В рассматриваемом примере это несколько чрезмерно, так что давайте добавим последнюю (менее радикальную) возможность, которая позволит пользователю зеркально отобразить целый контейнер `Canvas` и всю содержащуюся в нем графику. Начнем с добавления в `ToolBar` финального элемента `ToggleButton` со следующим определением:

```

<ToggleButton Name="flipCanvas" Click="flipCanvas_Click" Content="Flip Canvas!"/>

```

Внутри обработчика события `Click` для нового элемента `ToggleButton` создадим объект `RotateTransform` и подключим его к объекту `Canvas` через свойство `LayoutTransform`, если элемент `ToggleButton` отмечен. Если же элемент `ToggleButton` не отмечен, тогда мы удалим трансформацию, установив свойство `LayoutTransform` в `null`.

```
private void flipCanvas_Click(object sender, RoutedEventArgs e)
{
    if (flipCanvas.IsChecked == true)
    {
        RotateTransform rotate = new RotateTransform(-180);
        canvasDrawingArea.LayoutTransform = rotate;
    }
    else
    {
        canvasDrawingArea.LayoutTransform = null;
    }
}
```

Запустив приложение, добавим несколько графических фигур в область Canvas, следя за тем, чтобы они находились впритык к ее краям. После щелчка на новой кнопке обнаружится, что фигуры выходят за границы Canvas (рис. 26.9). Причина в том, что не был определен прямоугольник отсечения.

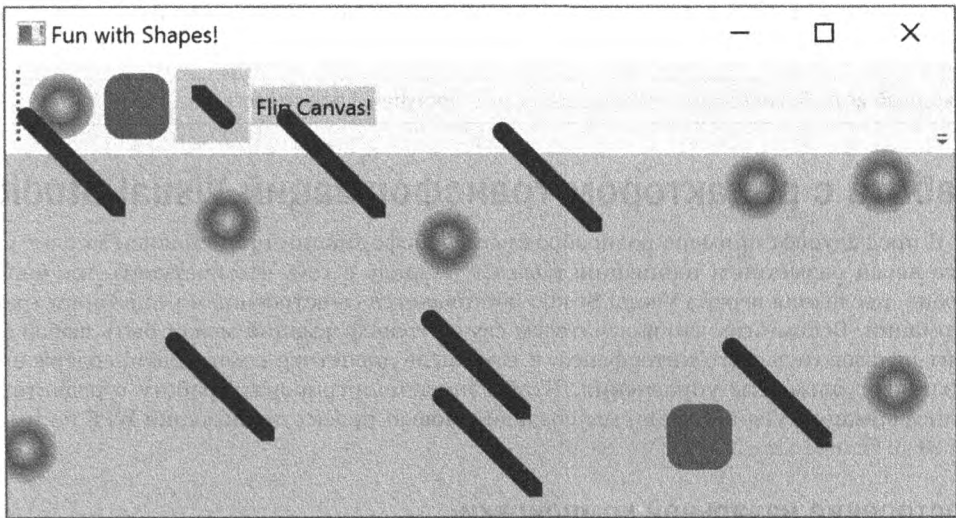


Рис. 26.9. После трансформации фигуры выходят за границы Canvas

Исправить проблему легко. Вместо того чтобы вручную писать сложную логику отсечения, просто установим свойство `ClipToBounds` элемента Canvas в `true`, предотвратив визуализацию дочерних элементов вне границ родительского элемента. После запуска приложения можно заметить, что графические данные больше не покидают границ отведенной области.

```
<Canvas ClipToBounds = "True" ... >
```

Последняя крошечная модификация, которую понадобится внести, связана с тем фактом, что когда пользователь зеркально отображает холст, щелкая на кнопке переключения, а затем щелкает на нем для рисования новой фигуры, то точка, где был произведен щелчок, не является той позицией, куда попадут графические данные. Взамен они появятся в месте нахождения курсора мыши.

Чтобы устранить проблему, применим тот же самый объект трансформации к рисуемой фигуре перед выполнением визуализации (через `RenderTransform`). Ниже показан основной фрагмент кода:

```
private void canvasDrawingArea_MouseLeftButtonDown(object sender,
                                                    MouseButtonEventArgs e)
{
    // Для краткости код не показан.
    if (flipCanvas.IsChecked == true)
    {
        RotateTransform rotate = new RotateTransform(-180);
        shapeToRender.RenderTransform = rotate;
    }
    // Установить верхнюю левую точку для рисования на холсте.
    Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
    Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);
    // Нарисовать фигуру.
    canvasDrawingArea.Children.Add(shapeToRender);
}
```

На этом исследование пространства имен `System.Windows.Shapes`, кистей и трансформаций завершено. Прежде чем перейти к анализу роли визуализации графики с использованием рисунков и геометрических объектов, давайте посмотрим, как IDE-среда Visual Studio может упростить работу с примитивными графическими элементами.

Исходный код. Проект `RenderingWithShapes` доступен в подкаталоге `Chapter_26`.

Работа с редактором трансформаций Visual Studio

В предыдущем примере разнообразные трансформации применялись за счет ручного ввода разметки и написания кода C#. Наряду с тем, что поступать так вполне удобно, последняя версия Visual Studio поставляется со встроенным редактором трансформаций. Вспомните, что получателем служб трансформаций может быть любой элемент пользовательского интерфейса, в том числе диспетчер компоновки, содержащий различные элементы управления. Чтобы продемонстрировать работу с редактором трансформаций Visual Studio, мы создадим новый проект приложения WPF по имени `FunWithTransforms`.

Построение начальной компоновки

Первым делом разделим первоначальный элемент `Grid` на две колонки с применением встроенного редактора сетки (точные размеры колонок роли не играют). Далее отыщем в панели инструментов элемент управления `StackPanel` и добавим его так, чтобы он занял все пространство первой колонки `Grid`; затем добавим в панель `StackPanel` три элемента управления `Button`:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <StackPanel Grid.Row="0" Grid.Column="0">
    <Button Name="btnSkew" Content="Skew" Click="Skew" />
    <Button Name="btnRotate" Content="Rotate" Click="Rotate" />
    <Button Name="btnFlip" Content="Flip" Click="Flip" />
  </StackPanel>
</Grid>
```

Добавим обработчики для кнопок:

```
private void Skew(object sender, RoutedEventArgs e)
{
}
private void Rotate(object sender, RoutedEventArgs e)
{
}
private void Flip(object sender, RoutedEventArgs e)
{
}
```

Для завершения пользовательского интерфейса создадим во второй колонке элемента Grid произвольную графику (используя любой прием, представленный ранее в главе). Вот разметка, применяемая в данном примере:

```
<Canvas x:Name="myCanvas" Grid.Column="1" Grid.Row="0">
  <Ellipse HorizontalAlignment="Left" VerticalAlignment="Top"
    Height="186" Width="92" Stroke="Black"
    Canvas.Left="20" Canvas.Top="31">
    <Ellipse.Fill>
      <RadialGradientBrush>
        <GradientStop Color="#FF951ED8" Offset="0.215"/>
        <GradientStop Color="#FF2FECB0" Offset="1"/>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
  <Ellipse HorizontalAlignment="Left" VerticalAlignment="Top"
    Height="101" Width="110" Stroke="Black"
    Canvas.Left="122" Canvas.Top="126">
    <Ellipse.Fill>
      <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
        <GradientStop Color="#FFB91DDC" Offset="0.355"/>
        <GradientStop Color="#FFB0381D" Offset="1"/>
      </LinearGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
</Canvas>
```

Окончательная компоновка показана на рис. 26.10.

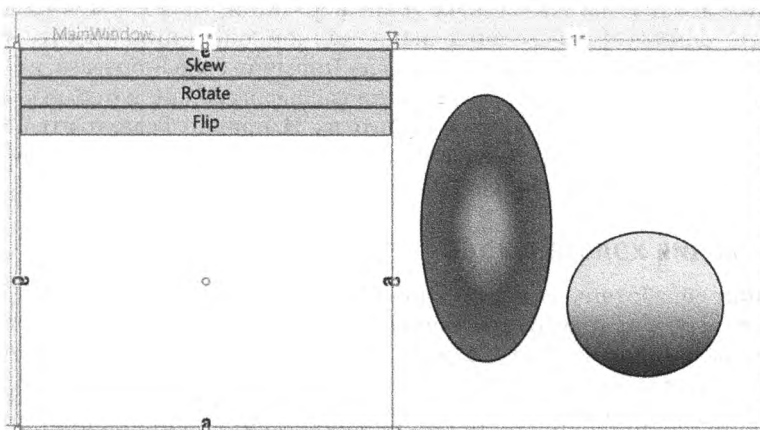


Рис. 26.10. Окончательная компоновка в примере трансформации

Применение трансформаций на этапе проектирования

Как упоминалось ранее, IDE-среда Visual Studio предоставляет встроенный редактор трансформаций, который можно найти в окне Properties. Раскроем раздел Transform (Трансформация), чтобы отобразить области RenderTransform и LayoutTransform редактора (рис. 26.11).

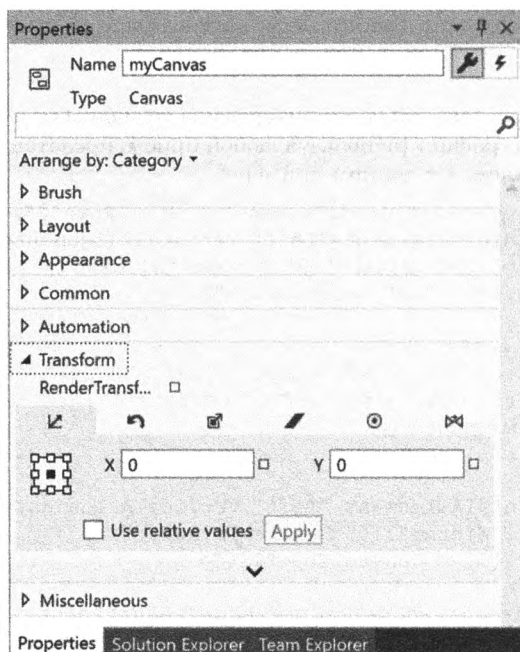


Рис. 26.11. Редактор трансформаций

Подобно разделу Brush раздел Transform предлагает несколько вкладок, предназначенных для конфигурирования разнообразных типов графической трансформации текущего выбранного элемента. В табл. 26.6 описаны варианты трансформации, доступные на этих вкладках (в порядке слева направо).

Опробуйте каждую из описанных трансформаций, используя в качестве цели специальную фигуру (для отмены выполненной операции просто нажимайте <Ctrl+Z>). Как и многие другие аспекты раздела Transform окна Properties, каждая трансформация имеет уникальный набор параметров конфигурации, которые должны стать вполне понятными, как только вы просмотрите их. Например, редактор трансформации Skew позволяет устанавливать значения скоса X и Y, а редактор трансформации Flip дает возможность зеркально отображать относительно оси X или Y и т.д.

Трансформация холста в коде

Реализации обработчиков для всех кнопок будут более или менее похожими. Мы сконфигурируем объект трансформации и присвоим его объекту myCanvas. Затем после запуска приложения можно будет щелкать на кнопке, чтобы просматривать результат применения трансформации. Ниже приведен полный код обработчиков (обратите внимание на установку свойства LayoutTransform, что позволяет данным фигуры позиционироваться относительно родительского контейнера):

```
private void Flip(object sender, System.Windows.RoutedEventArgs e)
{
    myCanvas.LayoutTransform = new ScaleTransform(-1, 1);
}
private void Rotate(object sender, System.Windows.RoutedEventArgs e)
{
    myCanvas.LayoutTransform = new RotateTransform(180);
}
private void Skew(object sender, System.Windows.RoutedEventArgs e)
{
    myCanvas.LayoutTransform = new SkewTransform(40, -20);
}
```

Исходный код. Проект FunWithTransforms доступен в подкаталоге Chapter_26.

Таблица 26.6. Варианты трансформации

Трансформация	Описание
Translate (Трансляция)	Позволяет сдвинуть местоположение элемента в позицию X,Y
Rotate (Поворот)	Позволяет повернуть элемент на угол до 360 градусов
Scale (Масштабирование)	Позволяет увеличить или уменьшить элемент на масштабный коэффициент в направлениях X и Y
Skew (Перекашивание)	Позволяет скосить ограничивающий прямоугольник, содержащий выбранный элемент, на масштабный коэффициент в направлениях X и Y
Center Point (Центральная точка)	При повороте или зеркальном отображении объекта элемент перемещается относительно фиксированной точки, которая называется центральной точкой объекта. По умолчанию центральная точка объекта расположена в центре объекта; тем не менее, эта трансформация позволяет изменять центральную точку объекта для выполнения поворота или зеркального отображения относительно другой точки
Flip (Зеркальное отображение)	Позволяет зеркально отобразить выбранный элемент на основе координаты X или Y центральной точки

Визуализация графических данных с использованием рисунков и геометрических объектов

Несмотря на то что типы Shape позволяют генерировать интерактивную двумерную поверхность любого вида, из-за насыщенной цепочки наследования они потребляют довольно много памяти. И хотя класс Path может помочь снизить накладные расходы за счет применения включенных геометрических объектов (вместо крупной коллекции других фигур), инфраструктура WPF предоставляет развитый API-интерфейс рисования и геометрии, который визуализирует еще более легкие двумерные векторные изображения.

Входной точкой в этот API-интерфейс является абстрактный класс System.Windows.Media.Drawing (из сборки PresentationCore.dll), который сам по себе всего лишь определяет ограничивающий прямоугольник для хранения результатов визуализации.

Инфраструктура WPF предлагает разнообразные классы, расширяющие `Drawing`, каждый из которых представляет отдельный способ рисования содержимого (табл. 26.7).

Таблица 26.7. Классы, производные от `Drawing`

Класс	Описание
<code>DrawingGroup</code>	Используется для комбинирования коллекции отдельных объектов, производных от <code>Drawing</code> , в единую составную визуализацию
<code>GeometryDrawing</code>	Применяется для визуализации двумерных фигур в очень легковесной манере
<code>GlyphRunDrawing</code>	Используется для визуализации текстовых данных с применением служб графической визуализации WPF
<code>ImageDrawing</code>	Используется для визуализации файла изображения, или набора геометрических объектов, внутри ограничивающего прямоугольника
<code>VideoDrawing</code>	Применяется для воспроизведения аудио- или видеофайла. Этот тип может полноценно использоваться только в процедурном коде. Для воспроизведения видео в разметке XAML гораздо лучше подходит тип <code>MediaPlayer</code>

Будучи более легковесными, производные от `Drawing` типы не обладают встроенной возможностью обработки событий, т.к. они не являются `UIElement` или `FrameworkElement` (хотя допускают программную реализацию логики проверки попадания).

Другое ключевое отличие между типами, производными от `Drawing`, и типами, производными от `Shape`, состоит в том, что производные от `Drawing` типы не умеют визуализировать себя, поскольку не унаследованы от `UIElement`! Для отображения содержимого производные типы должны помещаться в какой-то контейнерный объект (в частности `DrawingImage`, `DrawingBrush` или `DrawingVisual`).

Класс `DrawingImage` позволяет помещать рисунки и геометрические объекты внутрь элемента управления `Image` из WPF, который обычно применяется для отображения данных из внешнего файла. Класс `DrawingBrush` дает возможность строить кисть на основе рисунков и геометрических объектов, которая предназначена для установки свойства, требующего кисть. Наконец, класс `DrawingVisual` используется только на "визуальном" уровне графической визуализации, полностью управляемом из кода C#.

Хотя работать с рисунками немного сложнее, чем с простыми фигурами, отделение графической композиции от графической визуализации делает типы, производные от `Drawing`, гораздо более легковесными, чем производные от `Shape` типы, одновременно сохраняя их ключевые службы.

Построение кисти `DrawingBrush` с использованием геометрических объектов

Ранее в главе элемент `Path` заполнялся группой геометрических объектов примерно так:

```
<Path Fill = "Orange" Stroke = "Blue" StrokeThickness = "3">
  <Path.Data>
    <GeometryGroup>
      <EllipseGeometry Center = "75,70" RadiusX = "30" RadiusY = "30" />
      <RectangleGeometry Rect = "25,55 100 30" />
    </GeometryGroup>
  </Path.Data>
</Path>
```

```

    <LineGeometry StartPoint="0,0" EndPoint="70,30" />
    <LineGeometry StartPoint="70,30" EndPoint="0,30" />
  </GeometryGroup>
</Path.Data>
</Path>

```

Поступая подобным образом, мы достигаем интерактивности Path при чрезвычайной легковесности, присущей геометрическим объектам. Однако если необходимо визуализировать аналогичный вывод и отсутствует потребность в любой (готовой) интерактивности, тогда тот же самый элемент <GeometryGroup> можно поместить внутрь DrawingBrush:

```

<DrawingBrush>
  <DrawingBrush.Drawing>
    <GeometryDrawing>
      <GeometryDrawing.Geometry>
        <GeometryGroup>
          <EllipseGeometry Center = "75,70"
            RadiusX = "30" RadiusY = "30" />
          <RectangleGeometry Rect = "25,55 100 30" />
          <LineGeometry StartPoint="0,0" EndPoint="70,30" />
          <LineGeometry StartPoint="70,30" EndPoint="0,30" />
        </GeometryGroup>
      </GeometryDrawing.Geometry>
      <!-- Специальное перо для рисования границ -->
      <GeometryDrawing.Pen>
        <Pen Brush="Blue" Thickness="3"/>
      </GeometryDrawing.Pen>
      <!-- Специальная кисть для заполнения внутренней области -->
      <GeometryDrawing.Brush>
        <SolidColorBrush Color="Orange"/>
      </GeometryDrawing.Brush>
    </GeometryDrawing>
  </DrawingBrush.Drawing>
</DrawingBrush>

```

При помещении группы геометрических объектов внутрь DrawingBrush также понадобится установить объект Pen, применяемый для рисования границ, потому что свойство Stroke больше не наследуется от базового класса Shape. Здесь был создан элемент Pen с теми же настройками, которые использовались в значениях Stroke и StrokeThickness из предыдущего примера Path.

Кроме того, поскольку свойство Fill больше не наследуется от класса Shape, нужно также применять синтаксис “элемент-свойство” для определения объекта кисти, предназначенного элементу DrawingGeometry, со сплошным оранжевым цветом, как в предыдущих настройках Path.

Рисование с помощью DrawingBrush

Теперь объект DrawingBrush можно использовать для установки значения любого свойства, требующего объекта кисти. Например, после подготовки следующей разметки в редакторе Кахамл посредством синтаксиса “элемент-свойство” можно рисовать изображение по всей поверхности Page:

```

<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

```

```

<Page.Background>
  <DrawingBrush>
    <!-- Тот же самый объект DrawingBrush, что и ранее -->
  </DrawingBrush>
</Page.Background>
</Page>

```

Или же элемент `DrawingBrush` можно применять для установки другого совместимого с кистью свойства, такого как свойство `Background` элемента `Button`:

```

<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Button Height="100" Width="100">
    <Button.Background>
      <DrawingBrush>
        <!-- Тот же самый объект DrawingBrush, что и ранее -->
      </DrawingBrush>
    </Button.Background>
  </Button>
</Page>

```

Независимо от того, какое совместимое с кистью свойство устанавливается с использованием специального объекта `DrawingBrush`, визуализация двумерного графического изображения в итоге получается с намного меньшими накладными расходами, чем в случае визуализации того же изображения посредством фигур.

Включение типов `Drawing` в `DrawingImage`

Тип `DrawingImage` позволяет подключать рисованный геометрический объект к элементу управления `Image` из WPF. Взгляните на следующую разметку:

```

<Image>
  <Image.Source>
    <DrawingImage>
      <DrawingImage.Drawing>
        <!-- Тот же самый объект DrawingBrush, что и ранее -->
      </DrawingImage.Drawing>
    </DrawingImage>
  </Image.Source>
</Image>

```

В данном случае элемент `GeometryDrawing` был помещен внутрь элемента `DrawingImage`, а не `DrawingBrush`. С применением элемента `DrawingImage` можно установить свойство `Source` элемента управления `Image`.

Работа с векторными изображениями

Вы наверняка согласитесь с тем, что художнику будет довольно трудно создавать сложное векторное изображение с использованием инструментов и приемов, предоставляемых средой `Visual Studio`. В распоряжении художников есть собственные наборы инструментов, которые позволяют производить замечательную векторную графику. Изобразительными возможностями подобного рода не обладает ни IDE-среда `Visual Studio`, ни сопровождающий ее инструмент `Microsoft Blend`. Перед тем, как векторные изображения можно будет импортировать в приложение WPF, они должны быть преобразованы в выражения путей. После этого можно программировать с применением сгенерированной объектной модели, используя `Visual Studio`.

На заметку! Используемое изображение (LaserSign.svg) и экспортированные данные путей (LaserSign.xaml) можно найти в подкаталоге Chapter_26 загружаемого кода примеров. Изображение взято из статьи Википедии по адресу https://ru.wikipedia.org/wiki/Символы_опасности.

Преобразование файла с векторной графикой в файл XAML

Прежде чем можно будет импортировать сложные графические данные (такие как векторная графика) в приложение WPF, графику понадобится преобразовать в данные путей. Чтобы проиллюстрировать, как это делается, возьмем пример файла изображения .svg с упомянутым выше знаком опасности лазерного излучения. Затем загрузим и установим инструмент с открытым кодом под названием Inkscape (из веб-сайта www.inkscape.org). С помощью Inkscape откроем файл LaserSign.svg из подкаталога Chapter_26. Может быть запрошена модернизация формата. Установим настройки, как показано на рис. 26.12.

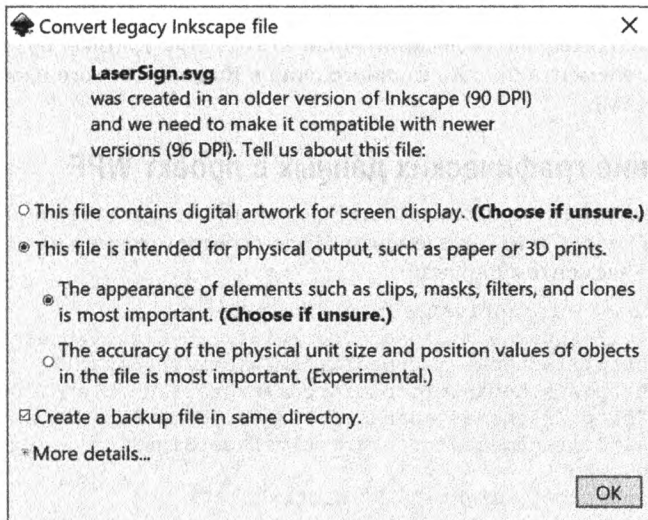


Рис. 26.12. Модернизация файла SVG до самого последнего формата в Inkscape

Следующие шаги поначалу покажутся несколько странными, но на самом деле они представляют собой простой способ преобразования векторных изображений в разметку XAML. Когда изображение приобрело желаемый вид, необходимо выбрать пункт меню File⇒Print (Файл⇒Печать). Далее потребуется указать Microsoft XPS Document Writer в качестве целевого принтера и щелкнуть на кнопке Print (Печать). В открывшемся окне следует ввести имя файла и выбрать место, где он должен быть сохранен, после чего щелкнуть на кнопке Save (Сохранить). В результате получается файл *.xps (или *.oxps).

На заметку! В зависимости от нескольких переменных среды в конфигурации системы сгенерированный файл будет иметь либо расширение .xps, либо расширение .oxps. В любом случае дальнейший процесс идентичен.

Форматы *.xps и *.oxps в действительности представляют собой архивы ZIP. Переименовав расширение в .zip, файл можно открыть в проводнике файлов (или в предпочитаемой утилите архивации). Файл содержит иерархию папок, приведенную на рис. 26.13.

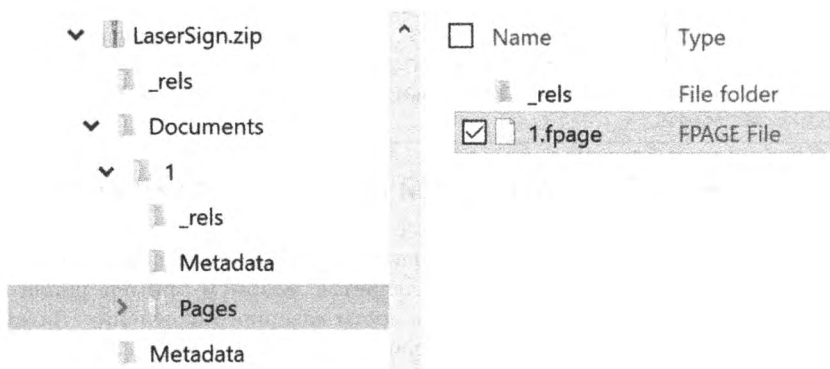


Рис. 26.13. Иерархия папок в файле *.xps или *.oxps

Необходимый файл находится в папке Pages (Documents/1/Pages) и называется 1.fpage. Откроем его в текстовом редакторе и скопируем в буфер все данные кроме открывающего и закрывающего дескрипторов FixedPage. Данные путей затем можно поместить внутрь элемента Canvas главного окна в Kaxaml. В итоге изображение будет показано в окне XAML.

Импортирование графических данных в проект WPF

Создадим новый проект приложения WPF по имени InteractiveLaserSign. Изменим значения свойств Height и Width элемента Window соответственно на 625 и 675 и заменим элемент Grid элементом Canvas:

```
<Window x:Class="InteractiveLaserSign.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:InteractiveLaserSign"
  mc:Ignorable="d"
  Title="MainWindow" Height="625" Width="675">
  <Canvas>
  </Canvas>
</Window>
```

Скопируем полную разметку XAML из файла 1.fpage (исключая внешний дескриптор FixedPage) и вставим ее в элемент управления Canvas внутри MainWindow. Просмотрев окно в режиме проектирования, легко удостовериться в том, что знак опасности лазерного излучения успешно воспроизводится в приложении.

Заглянув в окно Document Outline, можно заметить, что каждая часть изображения представлена как XAML-элемент Path. Если изменить размеры элемента Window, то качество изображения останется тем же самым безотносительно к тому, насколько большим сделано окно. Причина в том, что изображения, представленные с помощью элементов Path, визуализируются с применением механизма рисования и математики, а не за счет манипулирования пикселями.

Взаимодействие с изображением

Вспомните, что маршрутизируемое событие распространяется туннельным и пузырьковым образом, поэтому щелчок на любом элементе Path внутри Canvas может

быть обработан обработчиком событий щелчка на Canvas. Модифицируем разметку Canvas следующим образом:

```
<Canvas MouseLeftButtonDown="Canvas_MouseLeftButtonDown">
```

Добавим обработчик событий с таким кодом:

```
private void Canvas_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    if (e.OriginalSource is Path p)
    {
        p.Fill = new SolidColorBrush(Colors.Red);
    }
}
```

Запустим приложение и щелкнем на линиях, чтобы увидеть эффекты.

Теперь вы понимаете процесс генерации данных путей для сложной графики и знаете, как взаимодействовать с графическими данными в коде. Вы наверняка согласитесь, что наличие у профессиональных художников возможности генерировать сложные графические данные и экспортировать их в виде разметки XAML исключительно важна. После того как графические данные сохранены в файле XAML, разработчики могут импортировать разметку и писать код для взаимодействия с объектной моделью.

Исходный код. Проект *InteractiveLaserSign* доступен в подкаталоге *Chapter_26*.

Визуализация графических данных с использованием визуального уровня

Последний вариант визуализации графических данных с помощью WPF называется *визуальным уровнем*. Ранее уже упоминалось, что доступ к нему возможен только из кода (он не дружелюбен по отношению к разметке XAML). Несмотря на то что подавляющее большинство приложений WPF будут хорошо работать с применением фигур, рисунков и геометрических объектов, визуальный уровень обеспечивает самый быстрый способ визуализации крупных объемов графических данных. Визуальный уровень также может быть полезен, когда необходимо визуализировать единственное изображение в крупной области. Например, если требуется заполнить фон окна простым статическим изображением, тогда визуальный уровень будет наиболее быстрым способом решения такой задачи. Кроме того, он удобен, когда нужно очень быстро менять фон окна в зависимости от ввода пользователя или чего-нибудь еще.

Давайте построим небольшую программу, иллюстрирующую основы использования визуального уровня.

Базовый класс *Visual* и производные дочерние классы

Абстрактный класс `System.Windows.Media.Visual` предлагает минимальный набор служб (визуализацию, проверку попадания, трансформации) для визуализации графики, но не предоставляет поддержку дополнительных невидимых служб, которые могут приводить к разбуханию кода (события ввода, службы компоновки, стили и привязка данных). Класс *Visual* является абстрактным базовым классом. Для выполнения действительных операций визуализации должен применяться один из его производных классов. В WPF определено несколько подклассов *Visual*, в том числе *DrawingVisual*, *Viewport3DVisual* и *ContainerVisual*.

В рассматриваемом далее примере мы сосредоточимся только на `DrawingVisual` — легковесном классе рисования, который используется для визуализации фигур, изображений или текста.

Первый взгляд на класс `DrawingVisual`

Чтобы визуализировать данные на поверхности с применением класса `DrawingVisual`, понадобится выполнить следующие основные шаги:

- получить объект `DrawingContext` из `DrawingVisual`;
- использовать объект `DrawingContext` для визуализации графических данных.

Эти два шага представляют абсолютный минимум, необходимый для визуализации каких-то данных на поверхности. Тем не менее, когда нужно, чтобы визуализируемые графические данные реагировали на вычисления при проверке попадания (что важно для добавления взаимодействия с пользователем), потребуется также выполнить дополнительные шаги:

- обновить логическое и визуальное деревья, поддерживаемые контейнером, на котором производится визуализация;
- переопределить два виртуальных метода из класса `FrameworkElement`, позволив контейнеру получать созданные визуальные данные.

Давайте исследуем последние два шага более подробно. Чтобы продемонстрировать применение класса `DrawingVisual` для визуализации двумерных данных, создадим в Visual Studio новый проект приложения WPF по имени `RenderingWithVisuals`. Нашей первой целью будет использование класса `DrawingVisual` для динамического присваивания данных элементу управления `Image` из WPF. Начнем со следующего обновления разметки XAML окна для обработки события `Loaded`:

```
<Window x:Class="RenderingWithVisuals.MainWindow"
    <!-- Для краткости разметка не показана -->
    Title="Fun With Visual Layer" Height="350" Width="525"
    Loaded="MainWindow_Loaded">
```

Заменяем элемент `Grid` панелью `StackPanel` и добавим в нее элемент `Image`:

```
<StackPanel Background="AliceBlue" Name="myStackPanel">
    <Image Name="myImage" Height="80"/>
</StackPanel>
```

Элемент управления `Image` пока не имеет значения в свойстве `Source`, т.к. оно будет устанавливаться во время выполнения. С событием `Loaded` связана работа по построению графических данных в памяти с применением объекта `DrawingBrush`. Вот реализация обработчика события `Loaded`:

```
private void MainWindow_Loaded(object sender, RoutedEventArgs e)
{
    const int TextFontSize = 30;
    // Создать объект System.Windows.Media.FormattedText.
    FormattedText text = new FormattedText("Hello Visual Layer!",
        new System.Globalization.CultureInfo("en-us"),
        FlowDirection.LeftToRight,
        new Typeface(this.FontFamily, FontStyles.Italic, FontWeights.DemiBold,
            FontStretches.UltraExpanded),
        TextFontSize,
        Brushes.Green,
        null,
        VisualTreeHelper.GetDpi(this).PixelsPerDip);
```

```
// Создать объект DrawingVisual и получить объект DrawingContext.
DrawingVisual drawingVisual = new DrawingVisual();
using(DrawingContext drawingContext = drawingVisual.RenderOpen())
{
    // Вызвать любой из методов DrawingContext для визуализации данных.
    drawingContext.DrawRoundedRectangle(Brushes.Yellow,
        new Pen(Brushes.Black, 5),
        new Rect(5, 5, 450, 100), 20, 20);
    drawingContext.DrawText(text, new Point(20, 20));
}
// Динамически создать битовое изображение,
// используя данные в объекте DrawingVisual.
RenderTargetBitmap bmp = new RenderTargetBitmap(500, 100, 100, 90,
    PixelFormats.Pbgra32);
bmp.Render(drawingVisual);
// Установить источник для элемента управления Image.
myImage.Source = bmp;
}
```

В коде задействовано несколько новых классов WPF, которые будут кратко описаны ниже (полные сведения можно получить в документации .NET Framework 4.7 SDK). Метод начинается с создания нового объекта `FormattedText`, который представляет текстовую часть конструируемого изображения в памяти. Как видите, конструктор позволяет указывать многочисленные атрибуты, в том числе размер шрифта, семейство шрифтов, цвет переднего плана и сам текст.

Затем через вызов метода `RenderOpen()` на экземпляре `DrawingVisual` получается необходимый объект `DrawingContext`. Здесь в `DrawingVisual` визуализируется цветной прямоугольник со скругленными углами, за которым следует форматированный текст. В обоих случаях графические данные помещаются в `DrawingVisual` с применением жестко закодированных значений, что не слишком хорошо в производственном приложении, но вполне подходит для такого простого теста.

На заметку! Обязательно просмотрите описание класса `DrawingContext` в документации .NET Framework 4.7 SDK, чтобы ознакомиться со всеми членами, связанными с визуализацией. Если в прошлом вы работали с объектом `Graphics` из `Windows Forms`, то `DrawingContext` должен выглядеть похожим.

Несколько последних операторов отображают `DrawingVisual` на объект `RenderTargetBitmap`, который является членом пространства имен `System.Windows.Media.Imaging`. Этот класс принимает визуальный объект и трансформирует его в растровое изображение, находящееся в памяти. Затем устанавливается свойство `Source` элемента управления `Image` и получается вывод, показанный на рис. 26.14.

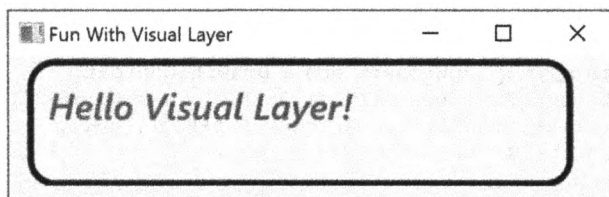


Рис. 26.14. Использование визуального уровня для визуализации растрового изображения, расположенного в памяти

На заметку! Пространство имен `System.Windows.Media.Imaging` содержит дополнительные классы кодирования, которые позволяют сохранять находящийся в памяти объект `RenderTargetBitmap` в физический файл с разнообразными форматами. Детали ищите в описании `JpegBitmapEncoder` и связанных с ним классов.

Визуализация графических данных в специальном диспетчере компоновки

Хотя применение `DrawingVisual` для рисования на фоне элемента управления WPF представляет интерес, возможно чаще придется строить специальный диспетчер компоновки (`Grid`, `StackPanel`, `Canvas` и т.д.), который внутренне использует визуальный уровень для визуализации своего содержимого. После создания такого специального диспетчера компоновки его можно подключить к обычному элементу `Window` (а также `Page` или `UserControl`) и позволить части пользовательского интерфейса использовать высоко оптимизированный агент визуализации, в то время как для визуализации не критичных графических данных будут применяться фигуры и рисунки.

Если дополнительная функциональность, предлагаемая специализированным диспетчером компоновки, не требуется, то можно просто расширить класс `FrameworkElement`, который обладает необходимой инфраструктурой, позволяющей содержать также и визуальные элементы. В целях иллюстрации вставим в проект новый класс по имени `CustomVisualFrameworkElement`. Унаследуем его от `FrameworkElement` и импортируем пространства имен `System.Windows`, `System.Windows.Input` и `System.Windows.Media`.

Класс `CustomVisualFrameworkElement` будет поддерживать переменную-член типа `VisualCollection`, которая содержит два фиксированных объекта `DrawingVisual` (конечно, в эту коллекцию можно было бы добавлять члены с помощью мыши, но мы решили сохранить пример простым). Модифицируем код класса следующим образом:

```
public class CustomVisualFrameworkElement : FrameworkElement
{
    // Коллекция всех визуальных объектов.
    VisualCollection theVisuals;
    public CustomVisualFrameworkElement()
    {
        // Заполнить коллекцию VisualCollection несколькими объектами DrawingVisual.
        // Аргумент конструктора представляет владельца визуальных объектов.
        theVisuals = new VisualCollection(this) { AddRect(), AddCircle() };
    }
    private Visual AddCircle()
    {
        DrawingVisual drawingVisual = new DrawingVisual();
        // Получить объект DrawingContext для создания нового содержимого.
        using (DrawingContext drawingContext = drawingVisual.RenderOpen())
        {
            // Создать круг и нарисовать его в DrawingContext.
            Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
            drawingContext.DrawEllipse(Brushes.DarkBlue, null,
                new Point(70, 90), 40, 50);
        }
        return drawingVisual;
    }
}
```

```
private Visual AddRect()
{
    DrawingVisual drawingVisual = new DrawingVisual();
    using (DrawingContext drawingContext = drawingVisual.RenderOpen())
    {
        Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
        drawingContext.DrawRectangle(Brushes.Tomato, null, rect);
    }
    return drawingVisual;
}
}
```

Прежде чем специальный элемент `FrameworkElement` можно будет использовать внутри `Window`, потребуется переопределить два упомянутых ранее ключевых виртуальных члена, которые вызываются внутренней инфраструктурой WPF во время процесса визуализации. Метод `GetVisualChild()` возвращает из коллекции дочерних элементов дочерний элемент по указанному индексу. Свойство `VisualChildrenCount`, допускающее только чтение, возвращает количество визуальных дочерних элементов внутри визуальной коллекции. Оба члена легко реализовать, т.к. всю реальную работу можно делегировать переменной-члену типа `VisualCollection`:

```
protected override int VisualChildrenCount => theVisuals.Count;
protected override Visual GetVisualChild(int index)
{
    // Значение должно быть больше нуля, поэтому разумно это проверить.
    if (index < 0 || index >= theVisuals.Count)
    {
        throw new ArgumentOutOfRangeException();
    }
    return theVisuals[index];
}
```

Теперь мы располагаем достаточной функциональностью, чтобы протестировать наш специальный класс. Модифицируем описание XAML элемента `Window`, добавив в существующий контейнер `StackPanel` один объект `CustomVisualFrameworkElement`. Это потребует создания специального пространства имен XML, которое отображается на пространство имен `.NET`.

```
<Window x:Class="RenderingWithVisuals.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:RenderingWithVisuals"
    Title="Fun with the Visual Layer" Height="350" Width="525"
    Loaded="Window_Loaded" WindowStartupLocation="CenterScreen">
    <StackPanel Background="AliceBlue" Name="myStackPanel">
        <Image Name="myImage" Height="80"/>
        <local:CustomVisualFrameworkElement/>
    </StackPanel>
</Window>
```

Результат выполнения программы показан на рис. 26.15.

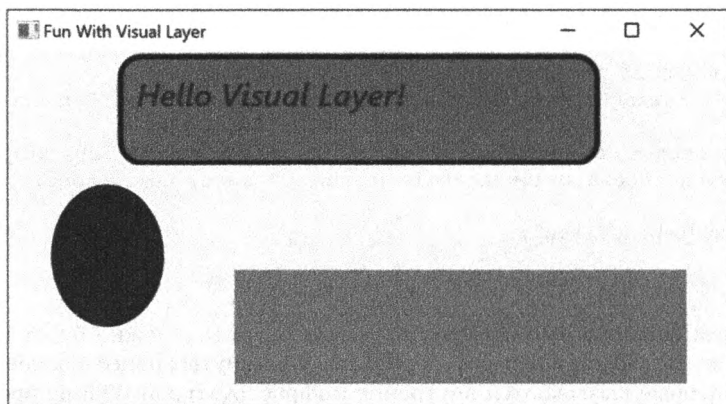


Рис. 26.15. Использование визуального уровня для визуализации данных в специальном элементе `FrameworkElement`

Реагирование на операции проверки попадания

Поскольку класс `DrawingVisual` не располагает инфраструктурой `UIElement` или `FrameworkElement`, необходимо программно добавить возможность реагирования на операции проверки попадания. Благодаря концепции *логического* и *визуального* деревьев на визуальном уровне делать это очень просто. Оказывается, что в результате написания блока XAML по существу строится логическое дерево элементов. Однако с каждым логическим деревом связано намного более развитое описание, известное как визуальное дерево, которое содержит низкоуровневые инструкции визуализации.

Упомянутые деревья подробно рассматриваются в главе 27, а сейчас достаточно знать, что до тех пор, пока специальные визуальные объекты не будут зарегистрированы в таких структурах данных, выполнять операции проверки попадания невозможно. К счастью, контейнер `VisualCollection` обеспечивает регистрацию автоматически (вот почему в аргументе конструктора необходимо передавать ссылку на специальный элемент `FrameworkElement`).

Изменим код класса `CustomVisualFrameworkElement` для обработки события `MouseDown` в конструкторе класса с применением стандартного синтаксиса C#:

```
this.MouseDown += CustomVisualFrameworkElement_MouseDown;
```

Реализация данного обработчика будет вызывать метод `VisualTreeHelper.HitTest()` с целью выяснения, находится ли курсор мыши внутри границ одного из визуальных объектов. Для этого в одном из параметров метода `HitTest()` указывается делегат `HitTestResultCallback`, который будет выполнять вычисления. Добавим в класс `CustomVisualFrameworkElement` следующие методы:

```
void CustomVisualFrameworkElement_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Выяснить, где пользователь выполнил щелчок.
    Point pt = e.GetPosition((UIElement)sender);

    // Вызвать вспомогательную функцию через делегат, чтобы
    // посмотреть, был ли совершен щелчок на визуальном объекте.
    VisualTreeHelper.HitTest(this, null,
        new HitTestResultCallback(myCallback), new PointHitTestParameters(pt));
}
```

```

public HitTestResultBehavior myCallback(HitTestResult result)
{
    // Если щелчок был совершен на визуальном объекте, то
    // переключиться между скошенной и нормальной визуализацией.
    if (result.VisualHit.GetType() == typeof(DrawingVisual))
    {
        if (((DrawingVisual)result.VisualHit).Transform == null)
        {
            ((DrawingVisual)result.VisualHit).Transform = new SkewTransform(7, 7);
        }
        else
        {
            ((DrawingVisual)result.VisualHit).Transform = null;
        }
    }
    // Сообщить методу HitTest() о прекращении углубления в визуальное дерево.
    return HitTestResultBehavior.Stop;
}

```

Запустим программу снова. Теперь должна появиться возможность щелкать на любом из отображенных визуальных объектов и наблюдать за выполнением трансформации. Наряду с тем, что рассмотренный пример взаимодействия с визуальным уровнем WPF очень прост, не забывайте, что здесь можно использовать те же самые кисти, трансформации, перья и диспетчеры компоновки, которые обычно применяются в разметке XAML. Таким образом, вы уже знаете довольно много о работе с классами, производными от Visual.

Исходный код. Проект `RenderingWithVisuals` доступен в подкаталоге `Chapter_26`.

На этом исследование служб графической визуализации WPF завершено. Несмотря на раскрытие ряда интересных тем, на самом деле мы лишь слегка затронули обширную область графических возможностей инфраструктуры WPF. Дальнейшее изучение фигур, рисунков, кистей, трансформаций и визуальных объектов вы можете продолжить самостоятельно (в оставшихся главах, посвященных WPF, еще встретятся дополнительные детали).

Резюме

Поскольку Windows Presentation Foundation является настолько насыщенной графической инфраструктурой для построения графических пользовательских интерфейсов, не должно вызывать удивления наличие нескольких способов визуализации графического вывода. Глава начиналась с рассмотрения трех подходов к визуализации (фигуры, рисунки и визуальные объекты), а также разнообразных примитивов визуализации, таких как кисти, перья и трансформации.

Вспомните, что когда необходимо строить интерактивную двумерную визуализацию, то фигуры делают такой процесс очень простым. С другой стороны, статические, не интерактивные изображения могут визуализироваться в более оптимальной манере с использованием рисунков и геометрических объектов, а визуальный уровень (доступный только в коде) обеспечит максимальный контроль и производительность.

ГЛАВА 27

Ресурсы, анимация, стили и шаблоны WPF

В настоящей главе будут представлены три важных (и взаимосвязанных) темы, которые позволят углубить понимание API-интерфейса Windows Presentation Foundation (WPF). Первым делом вы изучите роль *логических ресурсов*. Вы увидите, что система логических ресурсов (также называемых *объектными ресурсами*) представляет собой способ ссылки на часто используемые объекты внутри приложения WPF. Хотя логические ресурсы нередко реализуются в разметке XAML, они могут быть определены и в процедурном коде.

Далее вы узнаете, как определять, выполнять и управлять анимационной последовательностью. Вопреки тому, что можно было подумать, применение анимации WPF не ограничивается видеоиграми или мультимедиа-приложениями. В API-интерфейсе WPF анимация может использоваться, например, для подсветки кнопки, когда она получает фокус, или увеличения размера выбранной строки в *DataGrid*. Понимание анимации является ключевым аспектом построения специальных шаблонов элементов управления (как вы увидите позже в главе).

Затем будет объяснена роль стилей и шаблонов WPF. Подобно веб-странице, в которой применяются стили CSS или механизм тем ASP.NET, приложение WPF может определять общий вид и поведение для набора элементов управления. Такие стили можно определять в разметке и сохранять их в виде объектных ресурсов для последующего использования, а также динамически применять во время выполнения. В последнем примере вы научитесь строить специальные шаблоны элементов управления.

Система ресурсов WPF

Нашей первой задачей будет исследование темы встраивания и доступа к ресурсам приложения. Инфраструктура WPF поддерживает два вида ресурсов. Первый из них — *двоичные ресурсы*; эта категория обычно включает элементы, которые большинство программистов считают ресурсами в традиционном смысле (встроенные файлы изображений или звуковых клипов, значки, используемые приложением, и т.д.).

Вторая категория, называемая *объектными ресурсами* или *логическими ресурсами*, представляет именованные объекты .NET, которые можно упаковывать и многократно применять повсюду в приложении. Несмотря на то что упаковывать в виде объектного ресурса разрешено любой объект .NET, логические ресурсы особенно удобны при работе с графическими данными произвольного рода, поскольку можно определить часто используемые графические примитивы (кисти, перья, анимации и т.д.) и ссылаться на них по мере необходимости.

Работа с двоичными ресурсами

Прежде чем перейти к теме объектных ресурсов, давайте кратко проанализируем, как упаковывать *двоичные ресурсы* вроде значков и файлов изображений (например, логотипов компании либо изображений для анимации) внутри приложений. Создадим в Visual Studio новый проект приложения WPF по имени `BinaryResourcesApp`. Модифицируем разметку начального окна для обработки события `Loaded` элемента `Window` и применения `DockPanel` в качестве корня компоновки:

```
<Window x:Class="BinaryResourcesApp.MainWindow"
    <!-- Для краткости разметка не показана -->
    Title="Fun with Binary Resources" Height="500" Width="649"
    Loaded="MainWindow_OnLoaded">
    <DockPanel LastChildFill="True">
    </DockPanel>
</Window>
```

Предположим, что приложение должно отображать внутри части окна один из трех файлов изображений, основываясь на пользовательском вводе. Элемент управления `Image` из WPF может использоваться для отображения не только типичного файла изображения (*.bmp, *.gif, *.ico, *.jpg, *.png, *.wdp или *.tiff), но также данных объекта `DrawingImage` (как было показано в главе 26). Построим пользовательский интерфейс окна, который поддерживает диспетчер компоновки `DockPanel`, содержащий простую панель инструментов с кнопками `Next` (Вперед) и `Previous` (Назад). Под панелью инструментов расположен элемент управления `Image`, свойство `Source` которого в текущий момент не установлено:

```
<DockPanel LastChildFill="True">
    <ToolBar Height="60" Name="picturePickerToolBar" DockPanel.Dock="Top">
        <Button x:Name="btnPreviousImage" Height="40"
            Width="100" BorderBrush="Black"
            Margin="5" Content="Previous" Click="btnPreviousImage_Click"/>
        <Button x:Name="btnNextImage" Height="40" Width="100" BorderBrush="Black"
            Margin="5" Content="Next" Click="btnNextImage_Click"/>
    </ToolBar>

    <!-- Этот элемент Image будет заполняться в коде -->
    <Border BorderThickness="2" BorderBrush="Green">
        <Image x:Name="imageHolder" Stretch="Fill" />
    </Border>
</DockPanel>
```

Добавим следующие пустые обработчики событий (в случае работы с Visual Studio они уже присутствуют в коде):

```
private void MainWindow_OnLoaded(object sender, RoutedEventArgs e)
{
}

private void btnPreviousImage_Click(object sender, RoutedEventArgs e)
{
}

private void btnNextImage_Click(object sender, RoutedEventArgs e)
{
}
```

Когда окно загружается, изображения добавятся в коллекцию, по которой будет совершаться проход с помощью кнопок `Next` и `Previous`. Теперь, располагая инфраструктурой приложения, займемся исследованием разных вариантов ее реализации.

Включение в проект несвязанных файлов ресурсов

Один из вариантов предусматривает поставку файлов изображений в виде набора несвязанных файлов в каком-то подкаталоге внутри пути установки приложения. Начнем с добавления в проект новой папки (по имени Images). Добавим в папку несколько изображений, щелкнув правой кнопкой мыши внутри данной папки и выбрав в контекстном меню пункт Add⇒Existing Item (Добавить⇒Существующий элемент). В открывшемся диалоговом окне Add Existing Item (Добавление существующего элемента) изменим фильтр файлов на *.*, чтобы стали видны файлы изображений. Вы можете добавлять собственные файлы изображений или задействовать три файла изображений с именами Deer.jpg, Dogs.jpg и Welcome.jpg из загружаемого кода примеров.

Конфигурирование несвязанных ресурсов

Для копирования содержимого папки \Images в папку \bin\Debug при построении проекта выберем все изображения в окне Solution Explorer, щелкнем правой кнопкой мыши и выберем в контекстном меню пункт Properties (Свойства), чтобы открыть окно Properties (Свойства). Установим свойство Build Action (Действие сборки) в Content (Содержимое), а свойство Copy Output Directory (Копировать в выходной каталог) в Copy always (Копировать всегда), как показано на рис. 27.1.

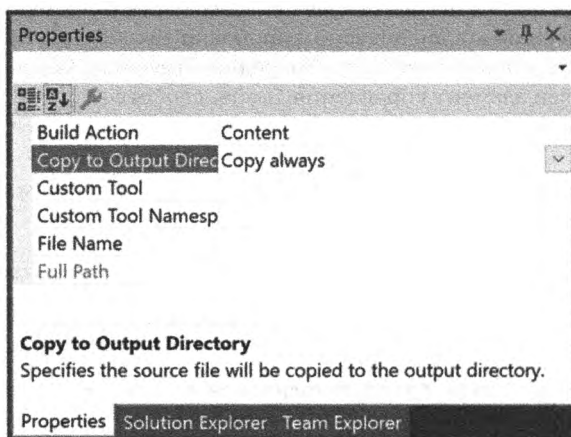


Рис. 27.1. Конфигурирование данных изображений для копирования в выходной каталог

На заметку! Для свойства Copy Output Directory можно было бы также выбрать вариант Copy if Newer (Копировать, если новее), что позволит сократить время копирования при построении крупных проектов с большим объемом содержимого. В рассматриваемом примере варианта Copy always вполне достаточно.

После построения проекта появится возможность щелкнуть на кнопке Show all Files (Показать все файлы) в окне Solution Explorer и просмотреть скопированную папку \Images внутри \bin\Debug (может также потребоваться щелкнуть на кнопке Refresh (Обновить)).

Программная загрузка изображения

Инфраструктура WPF предоставляет класс по имени BitmapImage, определенный в пространстве имен System.Windows.Media.Imaging. Он позволяет загружать данные

из файла изображения, местоположение которого представлено объектом `System.Uri`. Добавим поле типа `List<BitmapImage>` для хранения всех изображений, а также поле типа `int` для хранения индекса изображения, показанного в текущий момент.

В обработчике события `Loaded` окна заполним список изображений и установим свойство `Source` элемента управления `Image` в первое изображение из списка:

```
private void MainWindow_OnLoaded(object sender, RoutedEventArgs e)
{
    try
    {
        string path = Environment.CurrentDirectory;
        // Загрузить эти изображения во время загрузки окна.
        _images.Add(new BitmapImage(new Uri($"{path}\\Images\\Deer.jpg")));
        _images.Add(new BitmapImage(new Uri($"{path}\\Images\\Dogs.jpg")));
        _images.Add(new BitmapImage(new Uri($"{path}\\Images\\Welcome.jpg")));
        // Показать первое изображение в списке.
        imageHolder.Source = _images[_currImage];
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Реализуем обработчики для кнопок `Previous` и `Next`, чтобы обеспечить проход по изображениям. Когда пользователь добирается до конца списка, производится переход в начало и наоборот.

```
private void btnPreviousImage_Click(object sender, RoutedEventArgs e)
{
    if (--_currImage < 0)
    {
        _currImage = _images.Count - 1;
    }
    imageHolder.Source = _images[_currImage];
}
private void btnNextImage_Click(object sender, RoutedEventArgs e)
{
    if (++_currImage >= _images.Count)
    {
        _currImage = 0;
    }
    imageHolder.Source = _images[_currImage];
}
```

Теперь можно запустить программу и переключаться между всеми изображениями.

Встраивание ресурсов приложения

Если файлы изображений взамен необходимо встроить прямо в сборку .NET как двоичные ресурсы, тогда выберем файлы изображений в окне `Solution Explorer` (из папки `\Images`, а не `\bin\Debug\Images`) и установим свойство `Build Action` в `Resource` (Ресурс), а свойство `Copy to Output Directory` — в `Do not copy` (Не копировать), как показано на рис. 27.2.

В меню `Build` (Сборка) среды `Visual Studio` выберем пункт `Clean Solution` (Очистить решение), чтобы очистить текущее содержимое папки `\bin\Debug\Images`, и повторно построим проект. Обновим окно `Solution Explorer` и удостоверимся в том, что данные в каталоге `\bin\Debug\Images` отсутствуют.

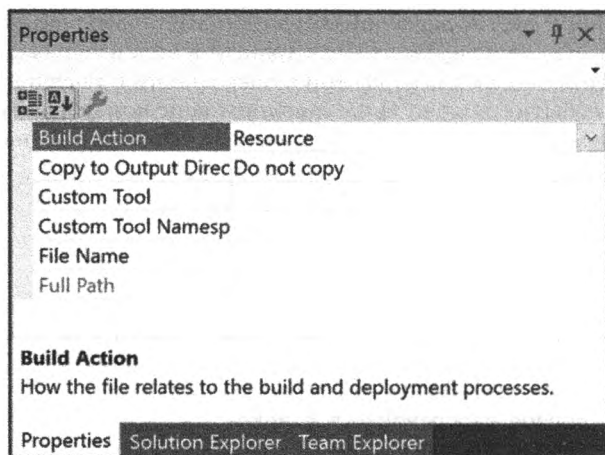


Рис. 27.2. Конфигурирование изображений как встроенных ресурсов

При текущих параметрах сборки графические данные больше не копируются в выходную папку, а встраиваются в саму сборку. Прием обеспечивает наличие ресурсов, но также приводит к увеличению размера скомпилированной сборки.

Нужно модифицировать код для загрузки изображений в список, извлекая их из скомпилированной сборки:

```
// Извлечь из сборки и затем загрузить изображения
_images.Add(new BitmapImage(new Uri(@"Images/Deer.jpg", UriKind.Relative)));
_images.Add(new BitmapImage(new Uri(@"Images/Dogs.jpg", UriKind.Relative)));
_images.Add(new BitmapImage(new Uri(@"Images/Welcome.jpg", UriKind.Relative)));
```

В таком случае больше не придется определять путь установки и можно просто задавать ресурсы по именам, которые учитывают название исходного подкаталога. Также обратите внимание, что при создании объектов `Uri` указывается значение `Relative` перечисления `UriKind`. В данный момент исполняемая программа представляет собой автономную сущность, которая может быть запущена из любого местоположения на машине, т.к. все скомпилированные данные находятся внутри сборки.

Исходный код. Проект `BinaryResourcesApp` доступен в подкаталоге `Chapter_27`.

Работа с объектными (логическими) ресурсами

При построении приложения WPF часто приходится определять большой объем разметки XAML для использования во многих местах окна или возможно во множестве окон или проектов. Например, пусть создана *безупречная* кисть с линейным градиентом, определение которой в разметке занимает 10 строк. Теперь кисть необходимо применить в качестве фонового цвета для каждого элемента `Button` в проекте, состоящем из 8 окон, т.е. всего получается 16 элементов `Button`.

Худшее, что можно было бы предпринять — копировать и вставлять одну и ту же разметку XAML в каждый элемент управления `Button`. Очевидно, в итоге это могло бы стать настоящим кошмаром при сопровождении, т.к. всякий раз, когда нужно скорректировать внешний вид и поведение кисти, приходилось бы вносить изменения во многие места.

К счастью, *объектные ресурсы* позволяют определить фрагмент разметки XAML, назначить ему имя и сохранить в подходящем словаре для использования в будущем. Подобно двоичным ресурсам объектные ресурсы часто компилируются в сборку, где они требуются. Однако в такой ситуации нет необходимости возиться со свойством `Build Action`. При условии, что разметка XAML помещена в корректное местоположение, компилятор позаботится обо всем остальном.

Взаимодействие с объектными ресурсами является крупной частью процесса разработки приложений WPF. Вы увидите, что объектные ресурсы могут быть намного сложнее, чем специальная кисть. Допускается определять анимацию на основе XAML, трехмерную визуализацию, специальный стиль элемента управления, шаблон данных, шаблон элемента управления и многое другое, и упаковывать каждую сущность в многократно используемый ресурс.

Роль свойства `Resources`

Как уже упоминалось, для применения в приложении объектные ресурсы должны быть помещены в подходящий объект словаря. Каждый производный от `FrameworkElement` класс поддерживает свойство `Resources`, которое инкапсулирует объект `ResourceDictionary`, содержащий определенные объектные ресурсы. Объект `ResourceDictionary` может хранить элементы любого типа, потому что оперирует экземплярами `System.Object` и допускает манипуляции из разметки XAML или процедурного кода.

В WPF все элементы управления плюс элементы `Window`, `Page` (используемые при построении навигационных приложений) и `UserControl` расширяют класс `FrameworkElement`, так что почти все виджеты предоставляют доступ к `ResourceDictionary`. Более того, класс `Application`, хотя и не расширяет `FrameworkElement`, но поддерживает свойство с идентичным именем `Resources`, которое предназначено для той же цели.

Определение ресурсов уровня окна

Чтобы приступить к исследованию роли объектных ресурсов, создадим в Visual Studio новый проект приложения WPF по имени `ObjectResourcesApp` и заменим первоначальный элемент `Grid` горизонтально выровненным диспетчером компоновки `StackPanel`, внутри которого определим два элемента управления `Button` (чего вполне достаточно для пояснения роли объектных ресурсов):

```
<StackPanel Orientation="Horizontal">
    <Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20"/>
    <Button Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"/>
</StackPanel>
```

Выберем кнопку `OK` и установим в свойстве `Background` специальный тип кисти с применением интегрированного редактора кистей (который обсуждался в главе 26). Взгляните, как кисть была встроена внутрь области между дескрипторами `<Button>` и `</Button>`:

```
<Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20">
    <Button.Background>
        <RadialGradientBrush>
            <GradientStop Color="#FFC44EC4" Offset="0" />
            <GradientStop Color="#FF829CEB" Offset="1" />
            <GradientStop Color="#FF793879" Offset="0.669" />
        </RadialGradientBrush>
    </Button.Background>
</Button>
```

Чтобы разрешить использовать эту кисть также и в кнопке Cancel (Отмена), область определения RadialGradientBrush должна быть расширена до словаря ресурсов родительского элемента. Например, если переместить RadialGradientBrush в StackPanel, то обе кнопки смогут применять одну и ту же кисть, т.к. они являются дочерними элементами того же самого диспетчера компоновки. Что еще лучше, кисть можно было бы упаковать в словарь ресурсов самого окна, в результате чего ее могли бы свободно использовать все элементы содержимого окна.

Когда необходимо определить ресурс, для установки свойства Resources владельца применяется синтаксис "свойство-элемент". Кроме того, элементу ресурса назначается значение `x:Key`, которое будет использоваться другими частями окна для ссылки на объектный ресурс. Имейте в виду, что атрибуты `x:Key` и `x>Name` — не одно и то же! Атрибут `x>Name` позволяет получать доступ к объекту как к переменной-члену в файле кода, в то время как атрибут `x:Key` дает возможность ссылаться на элемент в словаре ресурсов.

Среда Visual Studio позволяет переместить ресурс на более высокий уровень с применением соответствующего окна Properties. Чтобы сделать это, сначала понадобится идентифицировать свойство, имеющее сложный объект, который необходимо упаковать в виде ресурса (свойство Background в рассматриваемом примере). Справа от свойства находится небольшой квадрат, щелчок на котором приводит к открытию всплывающего меню. Выберем в нем пункт Convert to New Resource (Преобразовать в новый ресурс), как продемонстрировано на рис. 27.3.

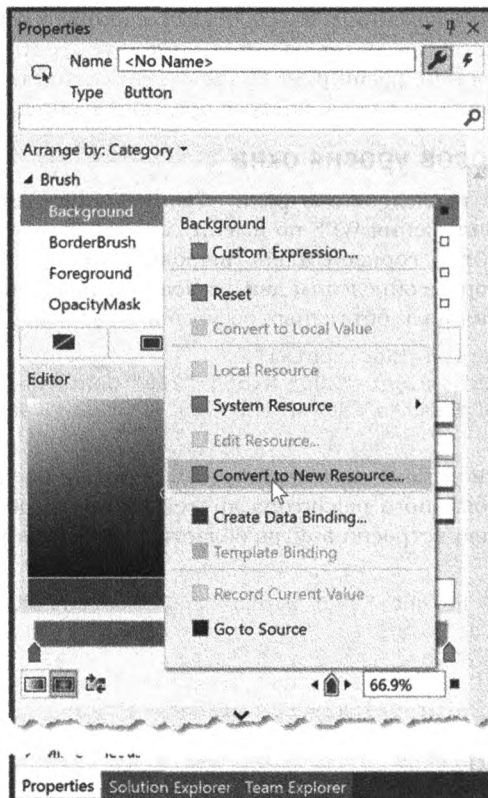


Рис. 27.3. Перемещение сложного объекта в контейнер ресурсов

Будет запрошено имя ресурса (`myBrush`) и предложено указать, куда он должен быть помещен. Оставим отмеченным переключатель **This document** (Этот документ), который выбирается по умолчанию (рис. 27.4).

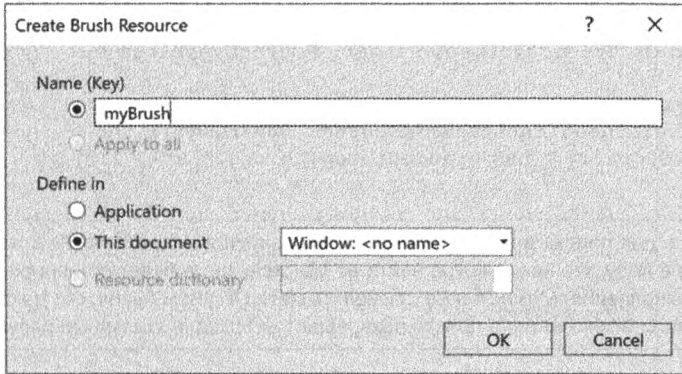


Рис. 27.4. Назначение имени объектному ресурсу

В результате определение кисти переместится внутрь дескриптора `Window.Resources`:

```
<Window.Resources>
  <RadialGradientBrush x:Key="myBrush">
    <GradientStop Color="#FFC44EC4" Offset="0" />
    <GradientStop Color="#FF829CEB" Offset="1" />
    <GradientStop Color="#FF793879" Offset="0.669" />
  </RadialGradientBrush>
</Window.Resources>
```

Свойство `Background` элемента управления `Button` обновляется для работы с новым ресурсом:

```
<Button Margin="25" Height="200" Width="200" Content="OK"
  FontSize="20" Background="{DynamicResource myBrush}"/>
```

Мастер создания ресурсов определил новый ресурс как динамический (`DynamicResource`). Динамические ресурсы рассматриваются позже, а пока изменим тип ресурса на статический (`StaticResource`):

```
<Button Margin="25" Height="200" Width="200" Content="OK"
  FontSize="20" Background="{StaticResource myBrush}"/>
```

Чтобы оценить преимущества, модифицируем свойство `Background` кнопки `Cancel` (Отмена), указав в нем тот же самый ресурс `StaticResource`, после чего можно будет видеть повторное использование в действии:

```
<Button Margin="25" Height="200" Width="200" Content="Cancel"
  FontSize="20" Background="{StaticResource myBrush}"/>
```

Расширение разметки {StaticResource}

Расширение разметки `{StaticResource}` применяет ресурс только раз (при инициализации) и он остается "подключенным" к первоначальному объекту на протяжении всей жизни приложения. Некоторые свойства (вроде градиентных переходов) будут обновляться, но в случае создания нового элемента `Brush`, например, элемент управления не обновится. Чтобы взглянуть на такое поведение в действии, добавим свойство `Name` и обработчик события `Click` к каждому элементу управления `Button`:

```
<Button Name="Ok" Margin="25" Height="200" Width="200" Content="OK"
    FontSize="20" Background="{StaticResource myBrush}" Click="Ok_OnClick"/>
<Button Name="Cancel" Margin="25" Height="200" Width="200" Content="Cancel"
    FontSize="20" Background="{StaticResource myBrush}" Click="Cancel_OnClick"/>
```

Затем поместим в обработчик события `Ok_OnClick()` следующий код:

```
private void Ok_OnClick(object sender, RoutedEventArgs e)
{
    // Получить кисть и внести изменение.
    var b = (RadialGradientBrush)Resources["myBrush"];
    b.GradientStops[1] = new GradientStop(Colors.Black, 0.0);
}
```

На заметку! Здесь для поиска ресурса по имени используется индекатор `Resources`. Тем не менее, имейте в виду, что если ресурс найти не удастся, тогда будет сгенерировано исключение времени выполнения. Можно также применять метод `TryFindResource()`, который не приводит к генерации исключения, а просто возвращает `null`, если указанный ресурс не найден.

Запустив программу и щелкнув на кнопке ОК, можно заметить, что градиенты соответствующим образом изменяются. Добавим в обработчик события `Cancel_OnClick()` такой код:

```
private void Cancel_OnClick(object sender, RoutedEventArgs e)
{
    // Поместить в ячейку myBrush совершенно новую кисть.
    Resources["myBrush"] = new SolidColorBrush(Colors.Red);
}
```

Если запустить программу снова и щелкнуть на кнопке Cancel, то выяснится, что ничего не происходит!

Расширение разметки {DynamicResource}

Для свойства также можно использовать расширение разметки `DynamicResource`. Чтобы выяснить разницу, изменим разметку для кнопки Cancel, как показано ниже:

```
<Button Name="Cancel" Margin="25" Height="200" Width="200" Content="Cancel"
    FontSize="20" Background="{DynamicResource myBrush}" Click="Cancel_OnClick"/>
```

На этот раз в результате щелчка на кнопке Cancel цвет фона для кнопки Cancel изменится, цвет фона для кнопки ОК остается прежним. Причина в том, что расширение разметки `{DynamicResource}` способно обнаруживать замену внутреннего объекта, указанного посредством ключа, новым объектом. Как и можно было предположить, такая возможность требует дополнительной инфраструктуры времени выполнения, так что `{StaticResource}` обычно следует использовать, только если не планируется заменять объектный ресурс другим объектом во время выполнения с уведомлением всех элементов, которые задействуют данный ресурс.

Ресурсы уровня приложения

Когда в словаре ресурсов окна имеются объектные ресурсы, их могут потреблять все элементы этого окна, но не другие окна приложения. Решение разделять объектные ресурсы в рамках приложения предусматривает их определение на уровне приложения, а не на уровне какого-то окна. В Visual Studio отсутствуют способы автоматизации такого действия, а потому мы просто вырежем имеющееся определение объекта кисти из области `Windows.Resource` и поместим его в область `Application.Resources` файла `App.xaml`.

Теперь любое дополнительное окно или элемент управления в приложении в состоянии работать с данным объектом кисти. Ресурсы уровня приложения доступны для выбора при установке свойства Background элемента управления (рис. 27.5).

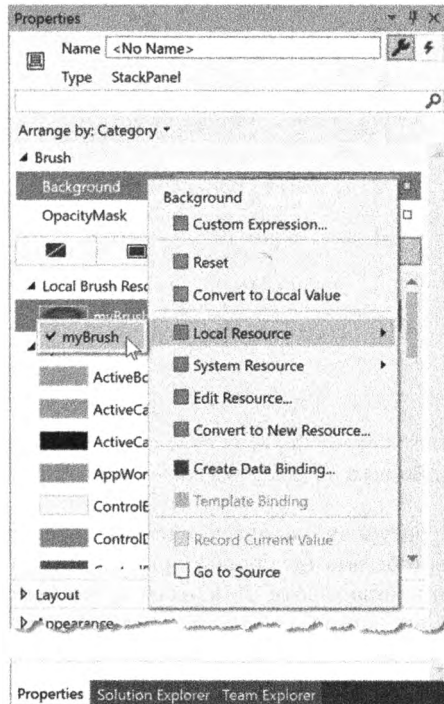


Рис. 27.5. Применение ресурсов уровня приложения

Определение объединенных словарей ресурсов

Ресурсов уровня приложения часто оказывается вполне достаточно, но они ничем не помогут, если ресурсы необходимо разделять между проектами. В таком случае понадобится определить то, что известно как *объединенный словарь ресурсов*. Считайте его библиотекой классов для ресурсов WPF; он представляет собой всего лишь файл .xaml, содержащий коллекцию ресурсов. Единственный проект может иметь любое требуемое количество таких файлов (один для кистей, один для анимации и т.д.), каждый из которых может быть добавлен в диалоговом окне Add New Item (Добавление нового элемента), открываемом через меню Project (рис. 27.6).

Вырежем текущие ресурсы из области определения Application.Resources в новом файле MyBrushes.xaml и перенесем их в словарь:

```
<ResourceDictionary
  xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
  xmlns:local="clr-namespace:ObjectResourcesApp"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <RadialGradientBrush x:Key="myBrush">
    <GradientStop Color="#FFC44EC4" Offset="0" />
    <GradientStop Color="#FF827CEB" Offset="1" />
    <GradientStop Color="#FF793879" Offset="0.669" />
  </RadialGradientBrush>
</ResourceDictionary>
```

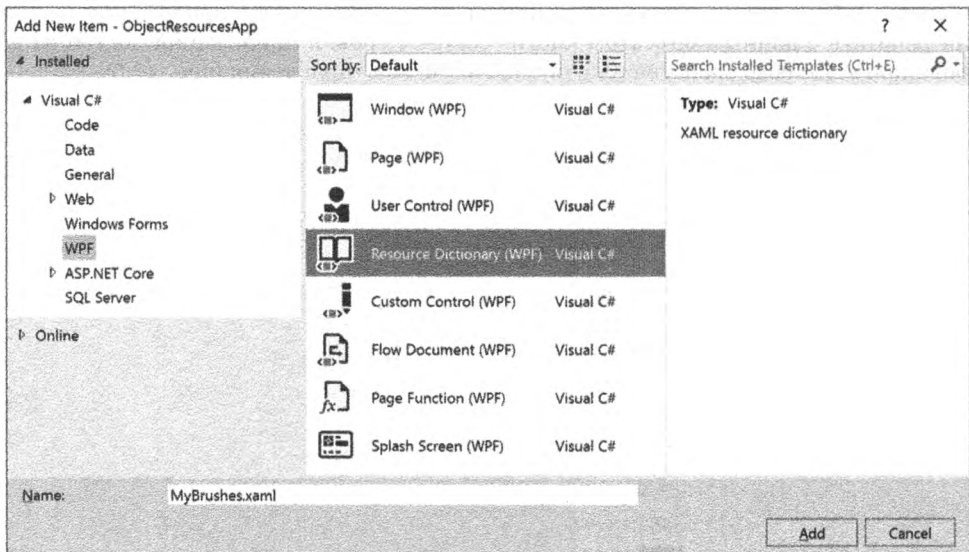


Рис. 27.6. Вставка нового объединенного словаря ресурсов

Хотя данный словарь ресурсов является частью проекта, все словари ресурсов должны быть объединены (обычно на уровне приложения) в единый словарь ресурсов, чтобы их можно было использовать. Для этого применяется следующий формат в файле App.xaml (обратите внимание, что множество словарей ресурсов объединяются за счет добавления элементов ResourceDictionary в область ResourceDictionary.MergedDictionaries):

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source = "MyBrushes.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

Проблема такого подхода в том, что каждый файл ресурсов потребует добавлять в каждый проект, нуждающийся в ресурсах. Более удачный подход к разделению ресурсов заключается в определении библиотеки классов .NET для совместного использования проектами, чем мы и займемся.

Определение сборки, включающей только ресурсы

Самый легкий способ построения сборки из одних ресурсов предусматривает создание проекта WPF User Control Library (Библиотека пользовательских элементов управления WPF). Добавим такой проект (по имени MyBrushesLibrary) в текущее решение, выбрав пункт меню Add⇒New Project (Добавить⇒Новый проект) в Visual Studio (рис. 27.7).

Теперь удалим файл UserControl1.xaml из проекта. Перетащим файл MyBrushes.xaml в проект MyBrushesLibrary и удалим его из проекта ObjectResourcesApp. Наконец, откроем файл MyBrushes.xaml в проекте MyBrushesLibrary и изменим пространство имен x:local на clr-namespace:MyBrushesLibrary.

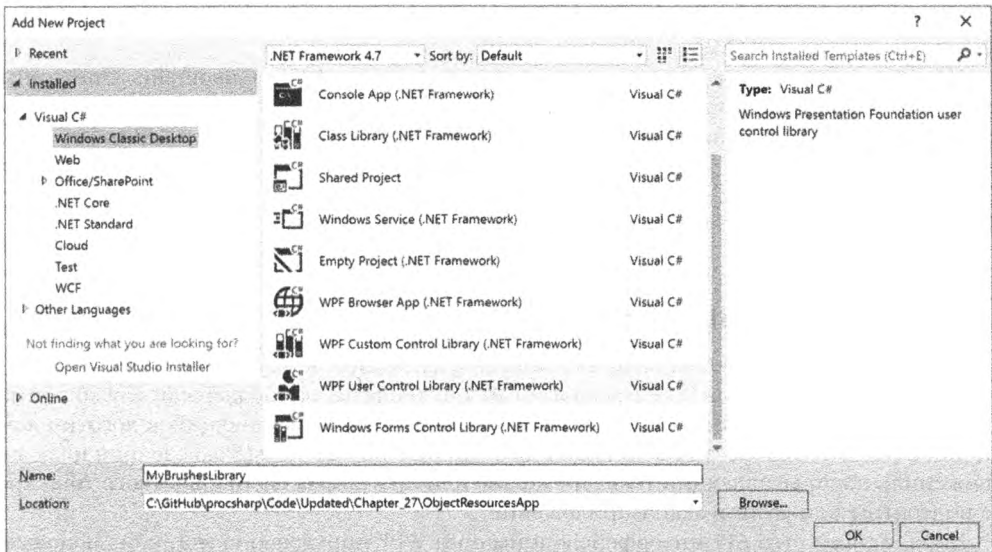


Рис. 27.7. Добавление проекта WPF User Control Library в качестве начальной точки для построения библиотеки из одних ресурсов

Вот как должно выглядеть содержимое файла `MyBrushes.xaml`:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:MyBrushesLibrary">
    <RadialGradientBrush x:Key="myBrush">
        <GradientStop Color="#FFC44EC4" Offset="0" />
        <GradientStop Color="#FF829CEB" Offset="1" />
        <GradientStop Color="#FF793879" Offset="0.669" />
    </RadialGradientBrush>
</ResourceDictionary>
```

Скомпилируем проект WPF User Control Library. Затем установим ссылку на эту библиотеку из проекта `ObjectResourcesApp` с применением диалогового окна `Add Reference` (Добавление ссылки). Объединим имеющиеся двоичные ресурсы со словарем ресурсов уровня приложения из проекта `ObjectResourcesApp`. Однако такое действие требует использования довольно забавного синтаксиса:

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <!-- Синтаксис выглядит как /ИмяСборки;Component/ИмяФайлаXamlвСборке.xaml -->
            <ResourceDictionary Source = "/MyBrushesLibrary;Component/MyBrushes.xaml"/>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

Имейте в виду, что данная строка чувствительна к пробелам. Если возле двоеточия или косых черт будут присутствовать лишние пробелы, то возникнут ошибки времени выполнения. Первая часть строки представляет собой дружественное имя внешней библиотеки (без файлового расширения). После двоеточия идет слово `Component`, а за ним

имя скомпилированного двоичного ресурса, которое будет идентичным имени исходного словаря ресурсов XAML.

Итак, знакомство с системой управления ресурсами WPF завершено. Описанные здесь приемы придется часто применять в большинстве разрабатываемых приложений (а то и во всех). Теперь давайте займемся исследованием встроенного API-интерфейса анимации WPF.

Исходный код. Проект `ObjectResourceApp` доступен в подкаталоге `Chapter_27`.

Службы анимации WPF

В дополнение к службам графической визуализации, которые рассматривались в главе 26, инфраструктура WPF предлагает API-интерфейс для поддержки служб анимации. Встретив термин *анимация*, многим на ум приходит вращающийся логотип компании, последовательность сменяющих друг друга изображений (для создания иллюзии движения), подпрыгивающий текст на экране или программа специфического типа вроде видеоигры или мультимедиа-приложения.

Наряду с тем, что API-интерфейсы анимации WPF определенно могли бы использоваться для упомянутых выше целей, анимация может применяться всякий раз, когда приложению необходимо придать особый стиль. Например, можно было бы построить анимацию для кнопки на экране, чтобы она слегка увеличивалась, когда курсор мыши находится внутри ее границ (и возвращалась к прежним размерам, когда курсор покидает границы). Или же можно было бы предусмотреть анимацию для окна, обеспечив его закрытие с использованием определенного визуального эффекта, такого как постепенное исчезновение до полной прозрачности. Применением, более ориентированным на бизнес-приложения, может быть постепенное увеличение четкости отображения сообщений об ошибках на экране, улучшая восприятие пользовательского интерфейса. Фактически поддержка анимации WPF может применяться в приложениях любого рода (бизнес-приложениях, мультимедиа-программах, видеоиграх и т.д.) всякий раз, когда нужно создать более привлекательное впечатление у пользователей.

Как и со многими другими аспектами WPF, с построением анимации не связано ничего нового. Единственная особенность в том, что в отличие от других API-интерфейсов, которые вы могли использовать в прошлом (включая Windows Forms), разработчики не обязаны создавать необходимую инфраструктуру вручную. В WPF не придется заранее создавать фоновые потоки или таймеры, применяемые для продвижения вперед анимационной последовательности, определять специальные типы для представления анимации, очищать и перерисовывать изображения либо реализовывать утомительные математические вычисления. Подобно другим аспектам WPF анимацию можно строить целиком в разметке XAML, целиком в коде C# либо с использованием комбинации того и другого.

На заметку! В среде Visual Studio отсутствует поддержка создания анимации посредством каких-либо графических инструментов и потому необходимо вводить разметку XAML вручную. Тем не менее, поставляемый в составе Visual Studio 2017 продукт Microsoft Blend на самом деле имеет встроенный редактор анимации, который способен существенно упростить решение задач.

Роль классов анимации

Чтобы разобраться в поддержке анимации WPF, потребуется начать с рассмотрения классов анимации из пространства имен `System.Windows.Media.Animation` сборки

PresentationCore.dll. Здесь вы найдете свыше 100 разных классов, которые содержат слово *Animation* в своих именах.

Все классы такого рода могут быть отнесены к одной из трех обширных категорий. Во-первых, любой класс, который следует соглашению об именовании вида *ТипДанныхAnimation* (*ByteAnimation*, *ColorAnimation*, *DoubleAnimation*, *Int32Animation* и т.д.), позволяет работать с анимацией линейной интерполяцией. Она обеспечивает плавное изменение значения во времени от начального к конечному.

Во-вторых, классы, следующие соглашению об именовании вида *ТипДанныхAnimationUsingKeyFrames* (*StringAnimationUsingKeyFrames*, *DoubleAnimationUsingKeyFrames*, *PointAnimationUsingKeyFrames* и т.д.), представляют анимацию ключевыми кадрами, которая позволяет проходить в цикле по набору определенных значений за указанный период времени. Например, ключевые кадры можно применять для изменения надписи на кнопке, проходя в цикле по последовательности индивидуальных символов.

В-третьих, классы, которые следуют соглашению об именовании вида *ТипДанныхAnimationUsingPath* (*DoubleAnimationUsingPath*, *PointAnimationUsingPath* и т.п.), представляют анимацию на основе пути, позволяющую перемещать объекты по определенному пути. Например, в приложении глобального позиционирования (GPS) анимацию на основе пути можно использовать для перемещения элемента по кратчайшему маршруту к месту, указанному пользователем.

Вполне очевидно, упомянутые классы не применяются для того, чтобы каким-то образом напрямую предоставить анимационную последовательность переменной определенного типа данных (в конце концов, как можно было бы выполнить анимацию значения 9, используя объект *Int32Animation*?).

В качестве примера возьмем свойства *Height* и *Width* типа *Label*, которые являются свойствами зависимости, упаковывающими значение *double*. Чтобы определить анимацию, которая будет увеличивать высоту метки с течением времени, можно подключить объект *DoubleAnimation* к свойству *Height* и позволить WPF позаботиться о деталях выполнения действительной анимации. Или вот другой пример: если требуется реализовать переход цвета кисти от зеленого до желтого в течение 5 секунд, то это можно сделать с применением типа *ColorAnimation*.

Следует уяснить, что классы *Animation* могут подключаться к любому свойству зависимости заданного объекта, которое имеет соответствующий тип. Как объяснялось в главе 25, свойства зависимости являются специальной формой свойств, которую требуют многие службы WPF, включая анимацию, привязку данных и стили.

По соглашению свойство зависимости определяется как статическое, доступное только для чтения поле класса, имя которого образуется добавлением слова *Property* к нормальному имени свойства. Например, для обращения к свойству зависимости для свойства *Height* класса *Button* в коде будет использоваться *Button.HeightProperty*.

Свойства *To*, *From* и *By*

Во всех классах *Animation* определены следующие ключевые свойства, которые управляют начальным и конечным значениями, применяемыми для выполнения анимации:

- *To* — представляет конечное значение анимации;
- *From* — представляет начальное значение анимации;
- *By* — представляет общую величину, на которую анимация изменяет начальное значение.

Несмотря на тот факт, что все классы поддерживают свойства *To*, *From* и *By*, они не получают их через виртуальные члены базового класса. Причина в том, что лежа-

щие в основе типы, упакованные внутри указанных свойств, варьируются в широких пределах (целые числа, цвета, объекты Thickness и т.д.), и представление всех возможностей через единственный базовый класс привело бы к очень сложным кодовым конструкциям.

В связи со сказанным может возникнуть вопрос: почему не использовались обобщения .NET для определения единственного обобщенного класса анимации с одиночным параметром типа (скажем, `Animate<T>`)? Опять-таки, поскольку существует огромное количество типов данных (цвета, векторы, целые числа, строки и т.д.), применяемых для анимации свойств зависимости, решение оказалось бы не настолько ясным, как можно было бы ожидать (не говоря уже о том, что XAML обеспечивает лишь ограниченную поддержку обобщенных типов).

Роль базового класса **Timeline**

Хотя для определения виртуальных свойств `To`, `From` и `By` не использовался единственный базовый класс, классы `Animation` все же разделяют общий базовый класс — `System.Windows.Media.Animation.Timeline`. Данный тип предлагает набор дополнительных свойств, которые управляют темпом продвижения анимации (табл. 27.1).

Таблица 27.1. Основные свойства базового класса **Timeline**

Свойства	Описание
<code>AccelerationRatio</code> , <code>DecelerationRatio</code> , <code>SpeedRatio</code>	Эти свойства применяются для управления общим темпом анимационной последовательности
<code>AutoReverse</code>	Это свойство получает или устанавливает значение, которое указывает, должна ли временная шкала воспроизводиться в обратном направлении после завершения итерации вперед (стандартным значением является <code>false</code>)
<code>BeginTime</code>	Это свойство получает или устанавливает время запуска временной шкалы. Стандартным значением является 0, что запускает анимацию немедленно
<code>Duration</code>	Это свойство позволяет устанавливать продолжительность воспроизведения временной шкалы
<code>FileBehavior</code> , <code>RepeatBehavior</code>	Эти свойства используются для управления тем, что должно произойти после завершения временной шкалы (повторение анимации, ничего и т.д.)

Реализация анимации в коде **C#**

Мы построим окно, содержащее элемент `Button`, который обладает довольно странным поведением: когда на него наводится курсор мыши, он вращается вокруг своего левого верхнего угла. Начнем с создания в Visual Studio нового проекта приложения WPF по имени `SpinningButtonAnimationApp`. Модифицируем начальную разметку, как показано ниже (обратите внимание на обработку события `MouseEnter` кнопки):

```
<Button x:Name="btnSpinner" Height="50" Width="100" Content="I Spin!"
        MouseEnter="btnSpinner_MouseEnter" Click="btnSpinner_OnClick"/>
```

В файле отделенного кода импортируем пространство имен `System.Windows.Media.Animation` и добавим в файл `C#` следующий код:

```

private bool _isSpinning = false;
private void btnSpinner_MouseEnter(object sender, MouseEventArgs e)
{
    if (!_isSpinning)
    {
        _isSpinning = true;
        // Создать объект DoubleAnimation и зарегистрировать
        // его с событием Completed.
        var dblAnim = new DoubleAnimation();
        dblAnim.Completed += (o, s) => { _isSpinning = false; };
        // Установить начальное и конечное значения.
        dblAnim.From = 0;
        dblAnim.To = 360;
        // Создать объект RotateTransform и присвоить
        // его свойству RenderTransform кнопки.
        var rt = new RotateTransform();
        btnSpinner.RenderTransform = rt;
        // Выполнить анимацию объекта RotateTransform.
        rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);
    }
}
private void btnSpinner_OnClick(object sender, RoutedEventArgs e)
{
}

```

Первая крупная задача метода `btnSpinner_MouseEnter()` связана с конфигурированием объекта `DoubleAnimation`, который будет начинать со значения 0 и заканчивать значением 360. Обратите внимание, что для этого объекта также обрабатывается событие `Completed`, где переключается булевская переменная уровня класса, которая применяется для того, чтобы выполняющаяся анимация не была сброшена в начало.

Затем создается объект `RotateTransform`, который подключается к свойству `RenderTransform` элемента управления `Button` (`btnSpinner`). Наконец, объект `RenderTransform` информируется о начале анимации его свойства `Angle` с использованием объекта `DoubleAnimation`. Реализация анимации в коде обычно осуществляется путем вызова метода `BeginAnimation()` и передачи ему лежащего в основе свойства зависимости, к которому необходимо применить анимацию (вспомните, что по соглашению оно определено как статическое поле класса), и связанного объекта анимации.

Давайте добавим в программу еще одну анимацию, которая заставит кнопку после щелчка плавно становиться невидимой. Для начала создадим обработчик события `Click` кнопки `btnSpinner` с приведенным ниже кодом:

```

private void btnSpinner_Click(object sender, RoutedEventArgs e)
{
    var dblAnim = new DoubleAnimation
    {
        From = 1.0,
        To = 0.0
    };
    btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}

```

В коде обработчика события `btnSpinner_Click()` изменяется свойство `Opacity`, чтобы постепенно скрыть кнопку из виду. Однако в настоящий момент это затруднительно, потому что кнопка вращается слишком быстро. Как можно управлять ходом анимации? Ответ на вопрос ищите ниже.

Управление темпом анимации

По умолчанию анимация будет занимать приблизительно одну секунду для перехода между значениями, которые присвоены свойствам `From` и `To`. Следовательно, кнопка располагает одной секундой, чтобы повернуться на 360 градусов, и в то же время в течение одной секунды она постепенно скроется из виду (после щелчка на ней).

Определить другой период времени для перехода анимации можно посредством свойства `Duration` объекта анимации, которому присваивается объект `Duration`. Обычно промежуток времени устанавливается путем передачи объекта `TimeSpan` конструктору класса `Duration`. Взгляните на показанное далее изменение, в результате которого кнопке будет выделено четыре секунды на вращение:

```
private void btnSpinner_MouseEnter(object sender, MouseEventArgs e)
{
    if (!_isSpinning)
    {
        _isSpinning = true;
        // Создать объект DoubleAnimation и зарегистрировать
        // его с событием Completed.
        var dblAnim = new DoubleAnimation();
        dblAnim.Completed += (o, s) => { _isSpinning = false; };
        // На завершение поворота кнопке отводится четыре секунды.
        dblAnim.Duration = new Duration(TimeSpan.FromSeconds(4));
        ...
    }
}
```

Благодаря такой модификации появится реальный шанс щелкнуть на кнопке во время ее вращения, после чего она плавно исчезнет.

На заметку! Свойство `BeginTime` класса `Animation` также принимает объект `TimeSpan`. Вспомните, что данное свойство можно устанавливать для указания времени ожидания перед запуском анимационной последовательности.

Запуск в обратном порядке и циклическое выполнение анимации

За счет установки в `true` свойства `AutoReverse` объектам `Animation` указывается о необходимости запуска анимации в обратном порядке по ее завершении. Например, если необходимо, чтобы кнопка снова стала видимой после исчезновения, можно написать следующий код:

```
private void btnSpinner_OnClick(object sender, RoutedEventArgs e)
{
    DoubleAnimation dblAnim = new DoubleAnimation
    {
        From = 1.0,
        To = 0.0
    };
    // После завершения запустить в обратном порядке.
    dblAnim.AutoReverse = true;
    btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}
```

Если нужно, чтобы анимация повторялась несколько раз (или никогда не прекращалась), тогда можно воспользоваться свойством `RepeatBehavior`, общим для всех классов `Animation`. Передавая конструктору простое числовое значение, можно указать жестко

закодированное количество повторений. С другой стороны, если передать конструктору объект `TimeSpan`, то можно задать время, в течение которого анимация должна повторяться. Наконец, чтобы выполнять анимацию бесконечно, свойство `RepeatBehavior` можно установить в `RepeatBehavior.Forever`. Рассмотрим следующие способы изменения поведения повтора одного из двух объектов `DoubleAnimation`, применяемых в примере:

```
// Повторять бесконечно.
dblAnim.RepeatBehavior = RepeatBehavior.Forever;

// Повторять три раза.
dblAnim.RepeatBehavior = new RepeatBehavior(3);

// Повторять в течение 30 секунд.
dblAnim.RepeatBehavior = new RepeatBehavior(TimeSpan.FromSeconds(30));
```

Исследование приемов добавления анимации к аспектам какого-то объекта с использованием кода C# и API-интерфейса анимации WPF завершено. Теперь посмотрим, как делать то же самое с помощью разметки XAML.

Исходный код. Проект `SpinningButtonAnimationApp` доступен в подкаталоге `Chapter_27`.

Реализация анимации в разметке XAML

Реализация анимации в разметке подобна ее реализации в коде, по крайней мере, для простых анимационных последовательностей. Когда необходимо создать более сложную анимацию, которая включает изменение значений множества свойств одновременно, объем разметки может заметно увеличиться. Даже в случае применения какого-то инструмента для генерирования анимации, основанной на разметке XAML, важно знать основы представления анимации в XAML, поскольку тогда облегчается задача модификации и настройки сгенерированного инструментом содержимого.

На заметку! В подкаталоге `XamlAnimations` внутри `Chapter_27` есть несколько файлов XAML. Скопируйте их содержимое в редактор `Kaхамl`, чтобы просмотреть результаты.

Большей частью создание анимации подобно всему тому, что вы уже видели: по-прежнему производится конфигурирование объекта `Animation`, который затем ассоциируется со свойством объекта. Тем не менее, крупное отличие связано с тем, что разметка XAML не является дружественной к вызовам методов. В результате вместо вызова `BeginAnimation()` используется *раскадровка* как промежуточный уровень.

Давайте рассмотрим полный пример анимации, определенной в терминах XAML, и подробно ее проанализируем. Приведенное далее определение XAML будет отображать окно, содержащее единственную метку. После того как объект `Label` загрузился в память, он начинает анимационную последовательность, во время которой размер шрифта увеличивается от 12 до 100 точек за период в четыре секунды. Анимация будет повторяться столько времени, сколько объект остается загруженным в память. Разметка находится в файле `GrowLabelFont.xaml`, так что его содержимое необходимо скопировать в редактор `Kaхамl`, нажать клавишу <F5> и понаблюдать за поведением.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="200" Width="600" WindowStartupLocation="CenterScreen"
  Title="Growing Label Font!">
```

```

<StackPanel>
  <Label Content = "Interesting...">
    <Label.Triggers>
      <EventTrigger RoutedEvent = "Label.Loaded">
        <EventTrigger.Actions>
          <BeginStoryboard>
            <Storyboard TargetProperty = "FontSize">
              <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                RepeatBehavior = "Forever"/>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Label.Triggers>
  </Label>
</StackPanel>
</Window>

```

А теперь подробно разберем пример.

Роль раскадровок

Двигаясь от самого глубоко вложенного элемента наружу, первым мы встречаем элемент `<DoubleAnimation>`, обращающийся к тем же самым свойствам, которые устанавливались в процедурном коде (`From`, `To`, `Duration` и `RepeatBehavior`):

```

<DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
  RepeatBehavior = "Forever"/>

```

Как упоминалось ранее, элементы `Animation` помещаются внутрь элемента `Storyboard`, применяемого для отображения объекта анимации на заданное свойство родительского типа через свойство `TargetProperty`, которым в данном случае является `FontSize`. Элемент `Storyboard` всегда находится внутри родительского элемента по имени `BeginStoryboard`:

```

<BeginStoryboard>
  <Storyboard TargetProperty = "FontSize">
    <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
      RepeatBehavior = "Forever"/>
  </Storyboard>
</BeginStoryboard>

```

Роль триггеров событий

После того как элемент `BeginStoryboard` определен, должно быть указано действие какого-то вида, которое приведет к запуску анимации. Инфраструктура WPF предлагает несколько разных способов реагирования на условия времени выполнения в разметке, один из которых называется *триггером*. С высокоуровневой точки зрения триггер можно считать способом реагирования на событие в разметке XAML без необходимости в написании процедурного кода.

Обычно когда ответ на событие реализуется в C#, пишется специальный код, который будет выполнен при поступлении события. Однако триггер — всего лишь способ получить уведомление о том, что некоторое событие произошло (загрузка элемента в память, наведение на него курсора мыши, получение им фокуса и т.д.).

Получив уведомление о появлении события, можно запускать раскадровку. В показанном ниже примере мы реагируем на факт загрузки элемента `Label` в память.

Поскольку нас интересует событие `Loaded` элемента `Label`, элемент `EventTrigger` помещается в коллекцию триггеров элемента `Label`:

```
<Label Content = "Interesting...">
  <Label.Triggers>
    <EventTrigger RoutedEvent = "Label.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "FontSize">
            <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                               RepeatBehavior = "Forever"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Label.Triggers>
</Label>
```

Рассмотрим еще один пример определения анимации в XAML, на этот раз анимации ключевыми кадрами.

Анимация с использованием дискретных ключевых кадров

В отличие от объектов анимации линейной интерполяцией, обеспечивающих только перемещение между начальной и конечной точками, объекты анимации *ключевыми кадрами* позволяют создавать коллекции специальных значений, которые должны достигаться в определенные моменты времени.

Чтобы проиллюстрировать применение типа дискретного ключевого кадра, предположим, что необходимо построить элемент управления `Button`, который выполняет анимацию своего содержимого так, что на протяжении трех секунд появляется значение `OK!` по одному символу за раз. Представленная далее разметка находится в файле `StringAnimation.xaml`. Ее можно скопировать в редактор `Кахамл` и просмотреть результаты.

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="100" Width="300"
  WindowStartupLocation="CenterScreen" Title="Animate String Data!">
  <StackPanel>
    <Button Name="myButton" Height="40"
      FontSize="16pt" FontFamily="Verdana" Width = "100">
      <Button.Triggers>
        <EventTrigger RoutedEvent="Button.Loaded">
          <BeginStoryboard>
            <Storyboard>
              <StringAnimationUsingKeyFrames RepeatBehavior = "Forever"
                Storyboard.TargetProperty="Content"
                Duration="0:0:3">
                <DiscreteStringKeyFrame Value="" KeyTime="0:0:0" />
                <DiscreteStringKeyFrame Value="O" KeyTime="0:0:1" />
                <DiscreteStringKeyFrame Value="OK" KeyTime="0:0:1.5" />
                <DiscreteStringKeyFrame Value="OK!" KeyTime="0:0:2" />
              </StringAnimationUsingKeyFrames>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Button.Triggers>
    </Button>
  </StackPanel>
</Window>
```



```

        </Button.Triggers>
    </Button>
</StackPanel>
</Window>

```

Первым делом обратите внимание, что для кнопки определяется триггер события, который обеспечивает запуск раскадровки при загрузке кнопки в память. Класс `StringAnimationUsingKeyFrames` отвечает за изменение содержимого кнопки через значение `Storyboard.TargetProperty`.

Внутри элемента `StringAnimationUsingKeyFrames` определены четыре элемента `DiscreteStringKeyFrame`, которые изменяют свойство `Content` на протяжении двух секунд (длительность, установленная объектом `StringAnimationUsingKeyFrames`, составляет в сумме три секунды, поэтому между финальным символом ' и следующим появлением 0 будет заметна небольшая пауза).

Теперь, когда вы получили некоторое представление о том, как строятся анимации в коде C# и разметке XAML, давайте выясним роль стилей WPF, которые интенсивно задействуют графику, объектные ресурсы и анимацию.

Исходный код. Несвязанные файлы XAML доступны в подкаталоге `XamlAnimations` внутри `Chapter_27`.

Роль стилей WPF

При построении пользовательского интерфейса приложения WPF нередко требуется обеспечить общий вид и поведение для целого семейства элементов управления. Например, может понадобиться сделать так, чтобы все типы кнопок имели ту же самую высоту, ширину, цвет и размер шрифта для своего строкового содержимого. Хотя решить задачу можно было бы установкой идентичных значений в индивидуальных свойствах, такой подход затрудняет внесение изменений, потому что при каждом изменении придется переустанавливать один и тот же набор свойств во множестве объектов.

К счастью, инфраструктура WPF предлагает простой способ ограничения внешнего вида и поведения связанных элементов управления с использованием *стилей*. Выразаясь просто, стиль WPF — это объект, который поддерживает коллекцию пар “свойство-значение”. С точки зрения программирования отдельный стиль представляется с помощью класса `System.Windows.Style`. Класс `Style` имеет свойство по имени `Setters`, которое открывает доступ к строго типизированной коллекции объектов `Setter`. Именно объект `Setter` обеспечивает возможность определения пар “свойство-значение”.

В дополнение к коллекции `Setters` класс `Style` также определяет несколько других важных членов, которые позволяют встраивать триггеры, ограничивать место применения стиля и даже создавать новый стиль на основе существующего (воспринимайте такой прием как “наследование стилей”). Ниже перечислены наиболее важные члены класса `Style`:

- `Triggers` — открывает доступ к коллекции объектов триггеров, которая делает возможной фиксацию условий возникновения разнообразных событий в стиле;
- `BasedOn` — разрешает строить новый стиль на основе существующего;
- `TargetType` — позволяет ограничивать место применения стиля.

Определение и применение стиля

Почти в каждом случае объект `Style` упаковывается как объектный ресурс. Подобно любому объектному ресурсу его можно упаковывать на уровне окна или на уровне приложения, а также внутри выделенного словаря ресурсов (что замечательно, поскольку делает объект `Style` легко доступным во всех местах приложения). Вспомните, что цель заключается в определении объекта `Style`, который наполняет (минимум) коллекцию `Setters` набором пар "свойство-значение".

Давайте построим стиль, который фиксирует базовые характеристики шрифта элемента управления в нашем приложении. Начнем с создания в Visual Studio нового проекта приложения WPF по имени `WpfStyles`. Откроем файл `App.xaml` и определим следующий именованный стиль:

```
<Application.Resources>
  <Style x:Key="BasicControlStyle">
    <Setter Property="Control.FontSize" Value="14"/>
    <Setter Property="Control.Height" Value="40"/>
    <Setter Property="Control.Cursor" Value="Hand"/>
  </Style>
</Application.Resources>
```

Обратите внимание, что объект `BasicControlStyle` добавляет во внутреннюю коллекцию три объекта `Setter`. Теперь применим получившийся стиль к нескольким элементам управления в главном окне. Из-за того, что стиль является объектным ресурсом, элементы управления, которым он необходим, по-прежнему должны использовать расширение разметки `{StackResource}` или `{DynamicResource}` для нахождения стиля. Когда они находят стиль, то устанавливают элемент ресурса в идентично именованное свойство `Style`. Заменяем стандартный элемент управления `Grid` следующей разметкой:

```
<StackPanel>
  <Label x:Name="lblInfo" Content="This style is boring..."
    Style="{StaticResource BasicControlStyle}" Width="150"/>
  <Button x:Name="btnTestButton" Content="Yes, but we are reusing settings!"
    Style="{StaticResource BasicControlStyle}" Width="250"/>
</StackPanel>
```

Если просмотреть элемент `Window` в визуальном конструкторе Visual Studio (или запустить приложение), то обнаружится, что оба элемента управления поддерживают те же самые курсор, высоту и размер шрифта.

Переопределение настроек стиля

В то время как оба элемента управления подчиняются стилю, после применения стиля к элементу управления вполне допустимо изменять некоторые из определенных настроек. Например, элемент `Button` теперь использует курсор `Help` (вместо курсора `Hand`, определенного в стиле):

```
<Button x:Name="btnTestButton" Content="Yes, but we are reusing settings!"
  Cursor="Help" Style="{StaticResource BasicControlStyle}" Width="250" />
```

Стили обрабатываются перед настройками индивидуальных свойств элемента управления, к которому применен стиль; следовательно, элементы управления могут "переопределять" настройки от случая к случаю.

Влияние атрибута TargetType на стили

В настоящий момент наш стиль определен так, что его может задействовать любой элемент управления (и он должен делать это явно, устанавливая свое свойство Style), поскольку каждое свойство уточнено посредством класса Control. Для программы, определяющей десятки настроек, в результате получился бы значительный объем повторяющегося кода. Один из способов несколько улучшить ситуацию предусматривает использование атрибута TargetType. Добавление атрибута TargetType к открывающему дескриптору Style позволяет точно указать, где стиль может быть применен (в данном примере внутри файла App.xaml):

```
<Style x:Key = "BasicControlStyle" TargetType="Control">
  <Setter Property = "FontSize" Value = "14"/>
  <Setter Property = "Height" Value = "40"/>
  <Setter Property = "Cursor" Value = "Hand"/>
</Style>
```

На заметку! При построении стиля, использующего базовый класс, нет нужды беспокоиться о том, что значение присваивается свойству зависимости, которое не поддерживается производными типами. Если производный тип не поддерживает заданное свойство зависимости, то оно игнорируется.

Кое в чем прием помог, но все равно мы имеем стиль, который может применяться к любому элементу управления. Атрибут TargetType более удобен, когда необходимо определить стиль, который может быть применен только к отдельному типу элементов управления. Добавим в словарь ресурсов приложения следующий стиль:

```
<Style x:Key = "BigGreenButton" TargetType="Button">
  <Setter Property = "FontSize" Value = "20"/>
  <Setter Property = "Height" Value = "100"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Background" Value = "DarkGreen"/>
  <Setter Property = "Foreground" Value = "Yellow"/>
</Style>
```

Такой стиль будет работать только с элементами управления Button (или подклассами Button). Если применить его к несовместимому элементу, тогда возникнут ошибки разметки и компиляции. Добавим элемент управления Button, который использует новый стиль:

```
<Button x:Name="btnAnotherButton"
Content="OK!" Margin="0,10,0,0"
Style="{StaticResource BigGreenButton}"
Width="250" Cursor="Help"/>
```

Результирующий вывод представлен на рис. 27.8.

Еще один эффект от атрибута TargetType заключается в том, что стиль будет применен ко всем элементам данного типа внутри области определения стиля при условии, что свойство x:Key отсутствует.

Вот еще один стиль уровня приложения, который будет автоматически применяться ко всем элементам управления TextBox в текущем приложении:

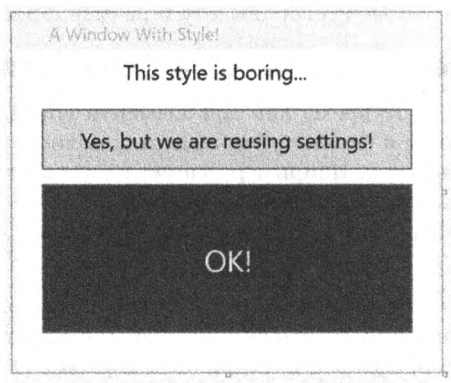


Рис. 27.8. Элементы управления с разными стилями

```

<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
    <Setter Property = "FontSize" Value = "14"/>
    <Setter Property = "Width" Value = "100"/>
    <Setter Property = "Height" Value = "30"/>
    <Setter Property = "BorderThickness" Value = "5"/>
    <Setter Property = "BorderBrush" Value = "Red"/>
    <Setter Property = "FontStyle" Value = "Italic"/>
</Style>

```

Теперь можно определять любое количество элементов управления `TextBox`, и все они автоматически получат установленный внешний вид. Если какому-то элементу управления `TextBox` не нужен такой стандартный внешний вид, тогда он может отказаться от него, установив свойство `Style` в `{x:Null}`. Например, элемент `txtTest` будет иметь неименованный стандартный стиль, а элемент `txtTest2` сделает все самостоятельно:

```

<TextBox x:Name="txtTest"/>
<TextBox x:Name="txtTest2" Style="{x:Null}" BorderBrush="Black"
    BorderThickness="5" Height="60" Width="100" Text="Ha!"/>

```

Создание подклассов существующих стилей

Новые стили можно также строить на основе существующего стиля посредством свойства `BasedOn`. Расширяемый стиль должен иметь подходящий атрибут `x:Key` в словаре, т.к. производный стиль будет ссылаться на него по имени, используя расширение разметки `{StaticResource}` или `{DynamicResource}`. Ниже представлен новый стиль, основанный на стиле `BigGreenButton`, который поворачивает элемент управления `Button` на 20 градусов:

```

<!-- Этот стиль основан на BigGreenButton -->
<Style x:Key="TiltButton" TargetType="Button"
    BasedOn = "{StaticResource BigGreenButton}">
    <Setter Property = "Foreground" Value = "White"/>
    <Setter Property = "RenderTransform">
        <Setter.Value>
            <RotateTransform Angle = "20"/>
        </Setter.Value>
    </Setter>
</Style>

```

Чтобы применить новый стиль, модифицируем разметку для кнопки следующим образом:

```

<Button x:Name="btnAnotherButton" Content="OK!" Margin="0,10,0,0"
    Style="{StaticResource TiltButton}" Width="250" Cursor="Help"/>

```

Такое действие изменяет внешний вид изображения, как показано на рис. 27.9.



Рис. 27.9. Использование производного стиля

Определение стилей с триггерами

Стили WPF могут также содержать триггеры за счет упаковки объектов `Trigger` в коллекцию `Triggers` объекта `Style`. Использование триггеров в стиле позволяет определять некоторые элементы `Setter` таким образом, что они будут применяться только в случае истинности заданного условия триггера. Например, возможно требуется увеличивать размер шрифта, когда курсор мыши находится над кнопкой. Или, скажем, нужно подсветить текстовое поле, имеющее фокус, с использованием фона указанного цвета. Триггеры полезны в ситуациях подобного рода, потому что они позволяют принимать специфические действия при изменении свойства, не требуя написания явной логики C# в файле отделенного кода.

Далее приведена модифицированная разметка для стиля элементов управления типа `TextBox`, где обеспечивается установка фона желтого цвета, когда элемент `TextBox` получает фокус:

```
<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value = "14"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Height" Value = "30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>
  <!-- Следующий установщик будет применен, только
        когда текстовое поле находится в фокусе -->
  <Style.Triggers>
    <Trigger Property = "IsFocused" Value = "True">
      <Setter Property = "Background" Value = "Yellow"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

При тестировании этого стиля обнаружится, что по мере перехода с помощью клавиши `<Tab>` между элементами `TextBox` текущий выбранный `TextBox` получает фон желтого цвета (если только стиль не отключен путем присваивания `{x:Null}` свойству `Style`).

Триггеры свойств также весьма интеллектуальны в том смысле, что когда условие триггера *не истинно*, свойство автоматически получает стандартное значение. Следовательно, как только `TextBox` теряет фокус, он также автоматически принимает стандартный цвет без какой-либо работы с вашей стороны. По контрасту с ними триггеры событий (которые исследовались при рассмотрении анимации WPF) не возвращаются автоматически в предыдущее состояние.

Определение стилей с множеством триггеров

Триггеры могут быть спроектированы так, что определенные элементы `Setter` будут применяться, когда истинными должны оказаться *многие условия*. Пусть необходимо устанавливать фон элемента `TextBox` в `Yellow` только в случае, если он имеет активный фокус и курсор мыши находится внутри его границ. Для этого можно воспользоваться элементом `MultiTrigger` и определить в нем каждое условие:

```
<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value = "14"/>
  <Setter Property = "Width" Value = "100"/>
```

```

<Setter Property = "Height" Value = "30"/>
<Setter Property = "BorderThickness" Value = "5"/>
<Setter Property = "BorderBrush" Value = "Red"/>
<Setter Property = "FontStyle" Value = "Italic"/>
<!-- Следующий установщик будет применен, только когда текстовое
      поле имеет фокус И над ним находится курсор мыши -->
<Style.Triggers>
  <MultiTrigger>
    <MultiTrigger.Conditions>
      <Condition Property = "IsFocused" Value = "True"/>
      <Condition Property = "IsMouseOver" Value = "True"/>
    </MultiTrigger.Conditions>
    <Setter Property = "Background" Value = "Yellow"/>
  </MultiTrigger>
</Style.Triggers>
</Style>

```

Анимированные стили

Стили также могут содержать в себе триггеры, которые запускают анимационную последовательность. Ниже показан последний стиль, который после применения к элементам управления Button заставит их увеличиваться и уменьшаться в размерах, когда курсор мыши находится внутри границ кнопки:

```

<!-- Стиль увеличивающейся кнопки -->
<Style x:Key = "GrowingButtonStyle" TargetType="Button">
  <Setter Property = "Height" Value = "40"/>
  <Setter Property = "Width" Value = "100"/>
  <Style.Triggers>
    <Trigger Property = "IsMouseOver" Value = "True">
      <Trigger.EnterActions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "Height">
            <DoubleAnimation From = "40" To = "200"
                          Duration = "0:0:2" AutoReverse="True"/>
          </Storyboard>
        </BeginStoryboard>
      </Trigger.EnterActions>
    </Trigger>
  </Style.Triggers>
</Style>

```

Здесь коллекция Triggers наблюдает за тем, когда свойство IsMouseOver возвращает значение true. После того как это произойдет, определяется элемент Trigger.EnterActions для выполнения простой раскадровки, которая заставляет кнопку за две секунды увеличиться до значения Height, равного 200 (и затем возвратиться к значению Height, равному 40). Чтобы отслеживать другие изменения свойств, можно также добавить область Trigger.ExitActions и определить в ней любые специальные действия, которые должны быть выполнены, когда IsMouseOver изменяется на false.

Применение стилей в коде

Вспомните, что стиль может применяться также во время выполнения. Прием удобен, когда у конечных пользователей должна быть возможность выбора внешнего вида для их пользовательского интерфейса, требуется принудительно устанавливать внешний вид и поведение на основе настроек безопасности (например, стиль DisableAllButton) или еще в какой-то ситуации.

В текущем проекте было определено порядочное количество стилей, многие из которых могут применяться к элементам управления Button. Давайте переделаем пользовательский интерфейс главного окна, чтобы позволить пользователю выбирать имена имеющихся стилей в элементе управления ListBox. На основе выбранного имени будет применен соответствующий стиль. Вот финальная разметка для элемента DockPanel:

```
<DockPanel>
  <StackPanel Orientation="Horizontal" DockPanel.Dock="Top" Margin="0,0,0,50">
    <Label Content="Please Pick a Style for this Button" Height="50"/>
    <ListBox x:Name="lstStyles" Height="80" Width="150" Background="LightBlue"
      SelectionChanged="comboStyles_Changed" />
  </StackPanel>
  <Button x:Name="btnStyle" Height="40" Width="100" Content="OK!" />
</DockPanel>
```

Элемент управления ListBox (по имени lstStyles) будет динамически заполняться внутри конструктора окна:

```
public MainWindow()
{
    InitializeComponent();
    // Заполнить окно со списком всеми стилями для элементов Button.
    lstStyles.Items.Add("GrowingButtonStyle");
    lstStyles.Items.Add("TiltButton");
    lstStyles.Items.Add("BigGreenButton");
    lstStyles.Items.Add("BasicControlStyle");
}
```

Последней задачей является обработка события SelectionChanged в связанном файле кода. Обратите внимание, что в следующем коде имеется возможность извлечения текущего ресурса по имени с использованием унаследованного метода TryFindResource():

```
private void comboStyles_Changed(object sender, SelectionChangedEventArgs e)
{
    // Получить имя стиля, выбранное в окне со списком.
    var currStyle = (Style)TryFindResource(lstStyles.SelectedValue);
    if (currStyle == null) return;
    // Установить стиль для типа кнопки.
    this.btnStyle.Style = currStyle;
}
```

После запуска приложения появляется возможность выбора одного из четырех стилей кнопок на лету. На рис. 27.10 показано готовое приложение в действии.

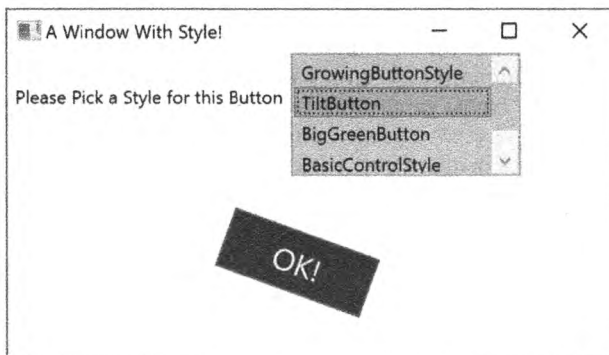


Рис. 27.10. Элементы управления с разными стилями

Исходный код. Проект WpfStyles доступен в подкаталоге Chapter_27.

Логические деревья, визуальные деревья и стандартные шаблоны

Теперь, когда вы понимаете, что собой представляют стили и ресурсы, есть еще несколько тем, которые потребуется раскрыть, прежде чем приступить к изучению построения специальных элементов управления. В частности, необходимо выяснить разницу между логическим деревом, визуальным деревом и стандартным шаблоном. При вводе разметки XAML в Visual Studio или в редакторе вроде Kaхaml разметка является *логическим представлением* документа XAML. В случае написания кода C#, который добавляет в элемент управления StackPanel новые элементы, они вставляются в *логическое дерево*. По существу логическое представление отражает то, как содержимое будет позиционировано внутри разнообразных диспетчеров компоновки для главного элемента Window (или другого корневого элемента, такого как Page или NavigationWindow).

Однако за каждым логическим деревом стоит намного более сложное представление, которое называется *визуальным деревом* и внутренне применяется инфраструктурой WPF для корректной визуализации элементов на экране. Внутри любого визуального дерева будут находиться полные детали шаблонов и стилей, используемых для визуализации каждого объекта, включая все необходимые рисунки, фигуры, визуальные объекты и объекты анимации.

Полезно уяснить разницу между логическим и визуальным деревьями, потому что при построении специального шаблона элемента управления на самом деле производится замена всего или части стандартного визуального дерева элемента управления собственным вариантом. Следовательно, если нужно, чтобы элемент управления Button визуализировался в виде звездообразной фигуры, тогда можно определить новый шаблон такого рода и подключить его к визуальному дереву Button. Логически тип остается тем же типом Button, поддерживая все ожидаемые свойства, методы и события. Но визуально он выглядит совершенно по-другому. Один лишь упомянутый факт делает WPF исключительно полезным API-интерфейсом, поскольку другие инструментальные наборы для создания кнопки звездообразной формы потребовали бы построения совершенно нового класса. В инфраструктуре WPF понадобится просто определить новую разметку.

На заметку! Элементы управления WPF часто описывают как *лишенные внешности*. Это относится к тому факту, что внешний вид элемента управления WPF совершенно не зависит от его поведения и допускает настройку.

Программное инспектирование логического дерева

Хотя анализ логического дерева окна во время выполнения — не слишком распространенное действие при программировании с применением WPF, полезно упомянуть о том, что в пространстве имен System.Windows определен класс LogicalTreeHelper, который позволяет инспектировать структуру логического дерева во время выполнения. Для иллюстрации связи между логическими деревьями, визуальными деревьями и шаблонами элементов управления создадим новый проект приложения WPF по имени TreesAndTemplatesApp.

Заменяем элемент `Grid` приведенной ниже разметкой, которая содержит два элемента управления `Button` и крупный допускающий только чтение элемент `TextBox` с включенными линейками прокрутки. Создадим в IDE-среде обработчики событий `Click` для каждой кнопки. Вот результирующая разметка XAML:

```
<DockPanel LastChildFill="True">
  <Border Height="50" DockPanel.Dock="Top" BorderBrush="Blue">
    <StackPanel Orientation="Horizontal">
      <Button x:Name="btnShowLogicalTree" Content="Logical Tree of Window"
        Margin="4" BorderBrush="Blue" Height="40"
        Click="btnShowLogicalTree_Click"/>
      <Button x:Name="btnShowVisualTree" Content="Visual Tree of Window"
        BorderBrush="Blue" Height="40" Click="btnShowVisualTree_Click"/>
    </StackPanel>
  </Border>
  <TextBox x:Name="txtDisplayArea" Margin="10"
    Background="AliceBlue" IsReadOnly="True"
    BorderBrush="Red" VerticalScrollBarVisibility="Auto"
    HorizontalScrollBarVisibility="Auto" />
</DockPanel>
```

Внутри файла кода C# определим переменную-член `_dataToShow` типа `string`. В обработчике события `Click` объекта `btnShowLogicalTree` вызовем вспомогательную функцию, которая продолжит вызывать себя рекурсивно с целью заполнения строковой переменной логическим деревом `Window`. Для этого будет вызван статический метод `GetChildren()` объекта `LogicalTreeHelper`. Ниже показан необходимый код:

```
private string _dataToShow = string.Empty;

private void btnShowLogicalTree_Click(object sender, RoutedEventArgs e)
{
    _dataToShow = "";
    BuildLogicalTree(0, this);
    txtDisplayArea.Text = _dataToShow;
}

void BuildLogicalTree(int depth, object obj)
{
    // Добавить имя типа к переменной-члену _dataToShow.
    _dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";

    // Если элемент - не DependencyObject, тогда пропустить его.
    if (!(obj is DependencyObject))
        return;

    // Выполнить рекурсивный вызов для каждого логического дочернего элемента.
    foreach (var child in LogicalTreeHelper.GetChildren(
        (DependencyObject)obj))
    {
        BuildLogicalTree(depth + 5, child);
    }
}
```

После запуска приложения и щелчка на кнопке `Logical Tree of Window` (Логическое дерево окна) в текстовой области отобразится древовидное представление, которое выглядит как почти точная копия исходной разметки XAML (рис. 27.11).

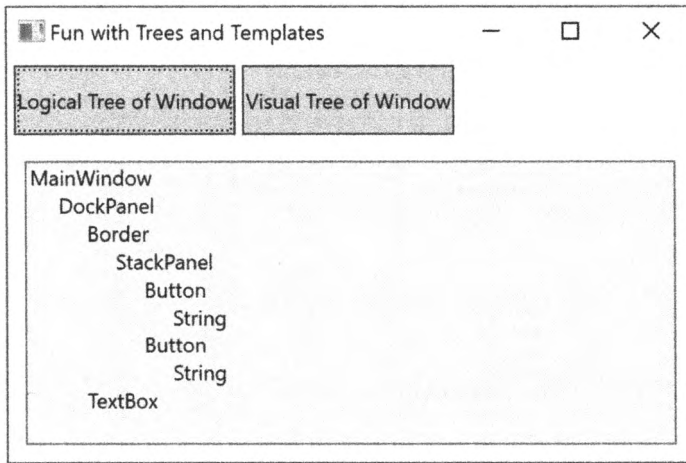


Рис. 27.11. Просмотр логического дерева во время выполнения

Программное инспектирование визуального дерева

Визуальное дерево объекта `Window` также можно инспектировать во время выполнения с использованием класса `VisualTreeHelper` из пространства имен `System.Windows.Media`. Далее приведена реализация обработчика события `Click` для второго элемента управления `Button` (`btnShowVisualTree`), которая выполняет похожую рекурсивную логику с целью построения текстового представления визуального дерева:

```

using System.Windows.Media;

private void btnShowVisualTree_Click(object sender, RoutedEventArgs e)
{
    _dataToShow = "";
    BuildVisualTree(0, this);
    txtDisplayArea.Text = _dataToShow;
}

void BuildVisualTree(int depth, DependencyObject obj)
{
    // Добавить имя типа к переменной-члену _dataToShow.
    _dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";

    // Выполнить рекурсивный вызов для каждого визуального дочернего элемента.
    for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
    {
        BuildVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
    }
}

```

На рис. 27.12 видно, что визуальное дерево открывает доступ к нескольким низкоуровневым агентам визуализации, таким как `ContentPresenter`, `AdornerDecorator`, `TextBoxLineDrawingVisual` и т.д.

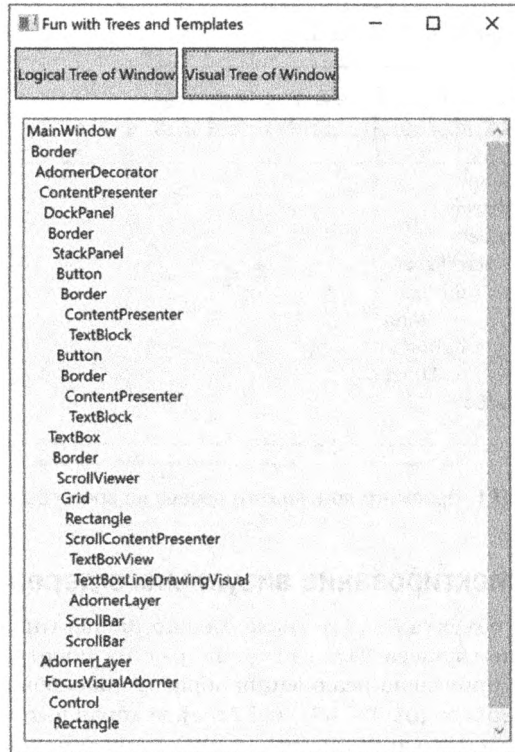


Рис. 27.12. Просмотр визуального дерева во время выполнения

Программное инспектирование стандартного шаблона элемента управления

Вспомните, что визуальное дерево применяется инфраструктурой WPF для выяснения, каким образом визуализировать элемент Window и все содержащиеся в нем элементы. Каждый элемент управления WPF хранит собственный набор команд визуализации внутри своего стандартного шаблона. С точки зрения программирования любой шаблон может быть представлен как экземпляр класса `ControlTemplate`. Кроме того, стандартный шаблон элемента управления можно получить через свойство `Template`:

```
// Получить стандартный шаблон элемента Button.
Button myBtn = new Button();
ControlTemplate template = myBtn.Template;
```

Подобным же образом можно создать в коде новый объект `ControlTemplate` и подключить его к свойству `Template` элемента управления:

```
// Подключить новый шаблон для использования в кнопке.
Button myBtn = new Button();
ControlTemplate customTemplate = new ControlTemplate();

// Предположим, что этот метод добавляет весь код для звездобразного шаблона.
MakeStarTemplate(customTemplate);
myBtn.Template = customTemplate;
```

Наряду с тем, что новый шаблон можно строить в коде, намного чаще это делается в разметке XAML. Тем не менее, прежде чем приступить к построению собственных шаблонов, давайте завершим текущий пример и добавим возможность просмотра стандартного шаблона для элемента управления WPF во время выполнения, что может оказаться по-настоящему полезным способом ознакомления с общей структурой шаблона. Добавим в разметку окна новую панель StackPanel с элементами управления; она стыкована с левой стороной главной панели DockPanel (находится прямо перед элементом <TextBox>) и определена следующим образом:

```
<Border DockPanel.Dock="Left" Margin="10" BorderBrush="DarkGreen"
    BorderThickness="4" Width="358">
    <StackPanel>
        <Label Content="Enter Full Name of WPF Control" Width="340"
            FontWeight="DemiBold" />
        <TextBox x:Name="txtFullName" Width="340" BorderBrush="Green"
            Background="BlanchedAlmond" Height="22"
            Text="System.Windows.Controls.Button" />
        <Button x:Name="btnTemplate" Content="See Template" BorderBrush="Green"
            Height="40" Width="100" Margin="5"
            Click="btnTemplate_Click" HorizontalAlignment="Left" />
        <Border BorderBrush="DarkGreen" BorderThickness="2" Height="260"
            Width="301" Margin="10" Background="LightGreen" >
            <StackPanel x:Name="stackTemplatePanel" />
        </Border>
    </StackPanel>
</Border>
```

Добавим пустой обработчик события btnTemplate_Click():

```
private void btnTemplate_Click(object sender, RoutedEventArgs e)
{
}
```

Текстовая область слева сверху позволяет вводить полностью заданное имя элемента управления WPF, расположенного в сборке PresentationFramework.dll. После того как библиотека загружена, экземпляр элемента управления динамически создается и отображается в большом квадрате слева внизу. Наконец, в текстовой области справа будет отображаться стандартный шаблон элемента управления. Добавим в класс C# новую переменную-член типа Control:

```
private Control _ctrlToExamine = null;
```

Ниже показан остальной код, который требует импортирования пространств имен System.Reflection, System.Xml и System.Windows.Markup:

```
private void btnTemplate_Click(object sender, RoutedEventArgs e)
{
    _dataToShow = "";
    ShowTemplate();
    txtDisplayArea.Text = _dataToShow;
}
private void ShowTemplate()
{
    // Удалить элемент, который в текущий момент находится
    // в области предварительного просмотра.
    if (_ctrlToExamine != null)
        stackTemplatePanel.Children.Remove(_ctrlToExamine);
```

```

try
{
    // Загрузить PresentationFramework и создать экземпляр
    // указанного элемента управления. Установить его размеры для
    // отображения, а затем добавить в пустой контейнер StackPanel.
    Assembly asm = Assembly.Load("PresentationFramework, Version=4.0.0.0," +
        "Culture=neutral, PublicKeyToken=31bf3856ad364e35");
    _ctrlToExamine = (Control)asm.CreateInstance(txtFullName.Text);
    _ctrlToExamine.Height = 200;
    _ctrlToExamine.Width = 200;
    _ctrlToExamine.Margin = new Thickness(5);
    stackTemplatePanel.Children.Add(_ctrlToExamine);
    // Определить настройки XML для предохранения отступов.
    var xmlSettings = new XmlWriterSettings{Indent = true};
    // Создать объект StringBuilder для хранения разметки XAML.
    var strBuilder = new StringBuilder();
    // Создать объект XmlWriter на основе имеющихся настроек.
    var xWriter = XmlWriter.Create(strBuilder, xmlSettings);
    // Сохранить разметку XAML в объекте XmlWriter на основе ControlTemplate.
    XmlWriter.Save(_ctrlToExamine.Template, xWriter);
    // Отобразить разметку XAML в текстовом поле.
    _dataToShow = strBuilder.ToString();
}
catch (Exception ex)
{
    _dataToShow = ex.Message;
}
}

```

Большая часть работы связана с отображением скомпилированного ресурса BAML на строку разметки XAML. На рис. 27.13 демонстрируется финальное приложение в действии на примере вывода стандартного шаблона для элемента управления System.Windows.Controls.DatePicker. Здесь отображается календарь, который доступен по щелчку на кнопке в правой части элемента управления.

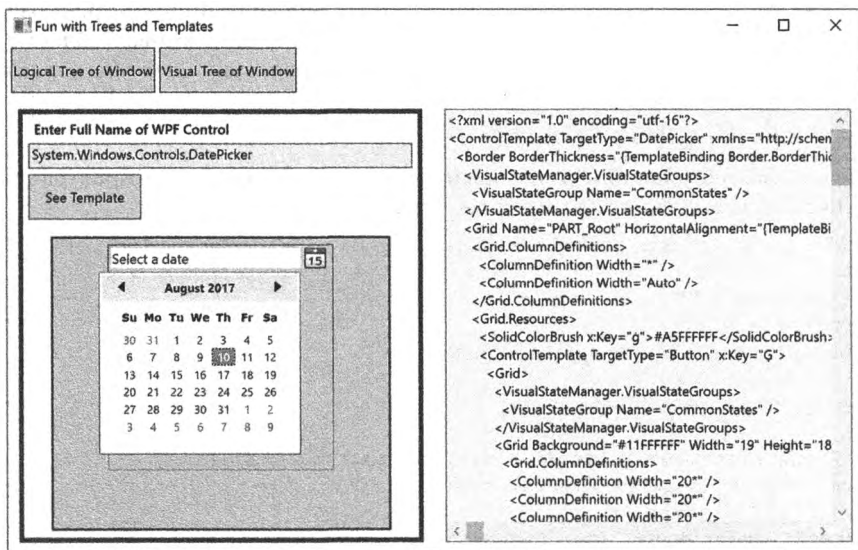


Рис. 27.13. Просмотр стандартного шаблона элемента управления во время выполнения

К настоящему моменту вы должны лучше понимать взаимосвязь между логическими деревьями, визуальными деревьями и стандартными шаблонами элементов управления. Остаток главы будет посвящен построению специальных шаблонов и пользовательских элементов управления.

Исходный код. Проект `TreesAndTemplatesApp` доступен в подкаталоге `Chapter_27`.

Построение шаблона элемента управления с помощью инфраструктуры триггеров

Специальный шаблон для элемента управления можно создавать с помощью только кода C#. Такой подход предусматривает добавление данных к объекту `ControlTemplate` и затем присваивание его свойству `Template` элемента управления. Однако большую часть времени внешний вид и поведение `ControlTemplate` будут определяться с использованием разметки XAML и фрагментов кода (мелких или крупных) для управления поведением во время выполнения.

В оставшемся материале главы вы узнаете, как строить специальные шаблоны с применением Visual Studio. Попутно вы ознакомитесь с инфраструктурой триггеров WPF и научитесь использовать анимацию для встраивания визуальных подсказок конечным пользователям. Применение при построении сложных шаблонов только IDE-среды Visual Studio может быть связано с довольно большим объемом клавиатурного набора и трудной работы. Конечно, шаблоны производственного уровня получают преимущество от использования продукта Microsoft Blend для Visual Studio, устанавливаемого вместе с Visual Studio. Тем не менее, поскольку текущее издание книги не включает описание Microsoft Blend, время засучить рукава и приступить к написанию некоторой разметки.

Для начала создадим новый проект приложения WPF по имени `ButtonTemplate`. Основной интерес в данном проекте представляют механизмы создания и применения шаблонов, так что заменим элемент `Grid` следующей разметкой:

```
<StackPanel Orientation="Horizontal">
  <Button x:Name="myButton" Width="100" Height="100" Click="myButton_Click"/>
</StackPanel>
```

В обработчике события `Click` мы просто отображаем окно сообщения (посредством вызова `MessageBox.Show()`) с подтверждением щелчка на элементе управления. При построении специальных шаблонов помните, что *поведение* элемента управления неизменно, но его *внешний вид* может варьироваться.

В настоящее время этот элемент `Button` визуализируется с использованием стандартного шаблона, который представляет собой ресурс BAML внутри заданной сборки WPF, как было проиллюстрировано в предыдущем примере. Определение собственного шаблона по существу сводится к замене стандартного визуального дерева своим вариантом. Для начала модифицируем определение элемента `Button`, указав новый шаблон с применением синтаксиса "свойство-элемент". Шаблон придаст элементу управления округлый вид.

```
<Button x:Name="myButton" Width="100" Height="100"
  Click="myButton_Click">
  <Button.Template>
    <ControlTemplate>
      <Grid x:Name="controlLayout">
        <Ellipse x:Name="buttonSurface" Fill = "LightBlue"/>
```

```

        <Label x:Name="buttonCaption" VerticalAlignment = "Center"
            HorizontalAlignment = "Center"
            FontWeight = "Bold" FontSize = "20" Content = "OK!"/>
    </Grid>
</ControlTemplate>
</Button.Template>
</Button>

```

Здесь определен шаблон, который состоит из именованного элемента Grid, содержащего именованные элементы Ellipse и Label. Поскольку в Grid не определены строки и столбцы, каждый дочерний элемент укладывается поверх предыдущего элемента управления, позволяя центрировать содержимое. Если теперь запустить приложение, то можно заметить, что событие Click будет инициироваться *только* в ситуации, когда курсор мыши находится внутри границ элемента Ellipse (т.е. не на углах, окружающих эллипс). Это замечательная возможность архитектуры шаблонов WPF, т.к. нет нужды повторно вычислять попадание курсора, проверять граничные условия или предпринимать другие низкоуровневые действия. Таким образом, если шаблон использует объект Polygon для отображения какой-то необычной геометрии, тогда можно иметь уверенность в том, что детали проверки попадания курсора будут соответствовать форме элемента управления, а не более крупного ограничивающего прямоугольника.

Шаблоны как ресурсы

В текущий момент наш шаблон внедрен в специфический элемент управления Button, что ограничивает возможности его многократного применения. В идеале шаблон круглой кнопки следовало бы поместить в словарь ресурсов, чтобы его можно было использовать в разных проектах, или как минимум перенести в контейнер ресурсов приложения для многократного применения внутри проекта. Давайте переместим локальный ресурс Button на уровень приложения, вырезав определение шаблона из разметки Button и вставив его в дескриптор Application.Resources внутри файла App.xaml. Добавим атрибуты Key и TargetType:

```

<Application.Resources>
    <ControlTemplate x:Key="RoundButtonTemplate" TargetType="{x:Type Button}">
        <Grid x:Name="controlLayout">
            <Ellipse x:Name="buttonSurface" Fill = "LightBlue"/>
            <Label x:Name="buttonCaption" VerticalAlignment = "Center"
                HorizontalAlignment = "Center" FontWeight = "Bold"
                FontSize = "20" Content = "OK!"/>
        </Grid>
    </ControlTemplate>
</Application.Resources>

```

Модифицируем разметку для Button, как показано далее:

```

<Button x:Name="myButton" Width="100" Height="100" Click="myButton_Click"
    Template="{StaticResource RoundButtonTemplate}">
</Button>

```

Из-за того, что этот ресурс доступен всему приложению, можно определять любое количество круглых кнопок, просто применяя имеющийся шаблон. В целях тестирования создадим два дополнительных элемента управления Button, которые используют данный шаблон (обрабатывать событие Click для них не нужно):

```

<StackPanel>
    <Button x:Name="myButton" Width="100" Height="100"
        Click="myButton_Click"
        Template="{StaticResource RoundButtonTemplate}"></Button>

```

```

<Button x:Name="myButton2" Width="100" Height="100"
    Template="{StaticResource RoundButtonTemplate}"></Button>
<Button x:Name="myButton3" Width="100" Height="100"
    Template="{StaticResource RoundButtonTemplate}"></Button>
</StackPanel>

```

Встраивание визуальных подсказок с использованием триггеров

При определении специального шаблона также удаляются все визуальные подсказки стандартного шаблона. Например, стандартный шаблон кнопки содержит разметку, которая задает внешний вид элемента управления при возникновении определенных событий пользовательского интерфейса, таких как получение фокуса, щелчок кнопкой мыши, включение (или отключение) и т.д. Пользователи довольно хорошо приучены к визуальным подсказкам подобного рода, т.к. они придают элементу управления некоторую осязаемую реакцию. Тем не менее, в шаблоне `RoundButtonTemplate` разметка такого типа не определена и потому внешний вид элемента управления остается идентичным независимо от действий мыши. В идеальном случае элемент должен выглядеть немного по-другому, когда на нем совершается щелчок (возможно, за счет изменения цвета или отбрасывания тени), чтобы уведомить пользователя об изменении визуального состояния.

Задачу можно решить с применением триггеров, как вы только что узнали. Для простых операций триггеры работают просто великолепно. Существуют дополнительные способы достижения цели, которые выходят за рамки настоящей книги, но больше информации доступно по адресу <https://docs.microsoft.com/en-us/dotnet/framework/wpf/controls/customizing-the-appearance-of-an-existing-control>.

В качестве примера обновим шаблон `RoundButtonTemplate` разметкой, которая добавляет два триггера. Первый триггер будет изменять цвет фона на синий, а цвет переднего плана на желтый, когда курсор находится на поверхности элемента управления. Второй триггер уменьшит размеры элемента `Grid` (а также его дочерних элементов) при нажатии кнопки мыши, когда курсор расположен в пределах элемента.

```

<ControlTemplate x:Key="RoundButtonTemplate" TargetType="Button" >
<Grid x:Name="controlLayout">
    <Ellipse x:Name="buttonSurface" Fill="LightBlue" />
    <Label x:Name="buttonCaption" Content="OK!" FontSize="20" FontWeight="Bold"
        HorizontalAlignment="Center" VerticalAlignment="Center" />
</Grid>
<ControlTemplate.Triggers>
    <Trigger Property = "IsMouseOver" Value = "True">
        <Setter TargetName = "buttonSurface" Property = "Fill" Value = "Blue"/>
        <Setter TargetName = "buttonCaption" Property = "Foreground" Value = "Yellow"/>
    </Trigger>
    <Trigger Property = "IsPressed" Value="True">
        <Setter TargetName="controlLayout" Property="RenderTransformOrigin"
            Value="0.5,0.5"/>
        <Setter TargetName="controlLayout" Property="RenderTransform">
            <Setter.Value>
                <ScaleTransform ScaleX="0.8" ScaleY="0.8"/>
            </Setter.Value>
        </Setter>
    </Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>

```


Роль расширения разметки {TemplateBinding}

Проблема с шаблоном элемента управления в том, что каждая кнопка выглядит и содержит тот же самый текст. Следующее обновление разметки не оказывает никакого влияния:

```
<Button x:Name="myButton" Width="100" Height="100"
        Background="Red" Content="Howdy!" Click="myButton_Click"
        Template="{StaticResource RoundButtonTemplate}" />
<Button x:Name="myButton2" Width="100" Height="100"
        Background="LightGreen" Content="Cancel!"
        Template="{StaticResource RoundButtonTemplate}" />
<Button x:Name="myButton3" Width="100" Height="100"
        Background="Yellow" Content="Format"
        Template="{StaticResource RoundButtonTemplate}" />
```

Причина в том, что стандартные свойства элемента управления (такие как `Background` и `Content`) переопределяются в шаблоне. Чтобы они стали доступными, их потребуется отобразить на связанные свойства в шаблоне. Решить такие проблемы можно за счет использования расширения разметки {TemplateBinding} при построении шаблона. Оно позволяет захватывать настройки свойств, которые определены элементом управления, применяющим шаблон, и использовать их при установке значений в самом шаблоне.

Ниже приведена переделанная версия шаблона `RoundButtonTemplate`, в которой расширение разметки {TemplateBinding} применяется для отображения свойства `Background` элемента `Button` на свойство `Fill` элемента `Ellipse`; здесь также обеспечивается действительная передача значения `Content` элемента `Button` свойству `Content` элемента `Label`:

```
<Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}"/>
<Label x:Name="buttonCaption" Content="{TemplateBinding Content}"
        FontSize="20" FontWeight="Bold" HorizontalAlignment="Center"
        VerticalAlignment="Center" />
```

После такого обновления появляется возможность создания кнопок с разными цветами и текстом. Результат обновления разметки XAML представлен на рис. 27.14.

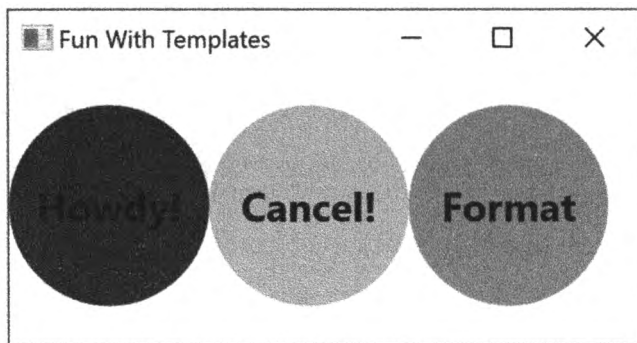


Рис. 27.14. Привязки шаблона позволяют передавать значения внутренним элементам управления

Роль класса ContentPresenter

При проектировании шаблона для отображения текстового значения элемента управления использовался элемент Label. Подобно Button он поддерживает свойство Content. Следовательно, если применяется расширение разметки {TemplateBinding}, тогда можно определять элемент Button со сложным содержимым, а не только с простой строкой.

Но что, если необходимо передать сложное содержимое члену шаблона, который не имеет свойства Content? Когда в шаблоне требуется определить обобщенную область отображения содержимого, то вместо элемента управления специфического типа (Label или TextBox) можно использовать класс ContentPresenter. Хотя в рассматриваемом примере в этом нет нужды, ниже показана простая разметка, иллюстрирующая способ построения специального шаблона, который применяет класс ContentPresenter для отображения значения свойства Content элемента управления, использующего шаблон:

```
<!-- Этот шаблон кнопки отобразит то, что установлено
      в свойстве Content размещающей кнопки -->
<ControlTemplate x:Key="NewRoundButtonTemplate" TargetType="Button">
  <Grid>
    <Ellipse Fill="{TemplateBinding Background}" />
    <ContentPresenter HorizontalAlignment="Center"
                      VerticalAlignment="Center" />
  </Grid>
</ControlTemplate>
```

Встраивание шаблонов в стили

В данный момент наш шаблон просто определяет базовый внешний вид и поведение элемента управления Button. Тем не менее, за процесс установки базовых свойств элемента управления (содержимого, размера шрифта, веса шрифта и т.д.) отвечает сам элемент Button:

```
<!-- Сейчас базовые значения свойств должен устанавливать
      сам элемент Button, а не шаблон -->
<Button x:Name="myButton" Foreground="Black" FontSize="20"
        FontWeight="Bold"
        Template="{StaticResource RoundButtonTemplate}"
        Click="myButton_Click"/>
```

При желании значения базовых свойств можно устанавливать в шаблоне. В сущности, таким способом фактически создаются стандартный внешний вид и поведение. Как вам уже должно быть понятно, это работа стилей WPF. Когда строится стиль (для учета настроек базовых свойств), можно определить шаблон *внутри* стиля! Ниже показан измененный ресурс приложения внутри файла App.xaml, которому назначен ключ RoundButtonStyle:

```
<!-- Стиль, содержащий шаблон -->
<Style x:Key="RoundButtonStyle" TargetType="Button">
  <Setter Property="Foreground" Value="Black"/>
  <Setter Property="FontSize" Value="14"/>
  <Setter Property="FontWeight" Value="Bold"/>
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="100"/>
  <!-- Далее следует сам шаблон -->
  <Setter Property="Template">
```

```

<Setter.Value>
  <ControlTemplate TargetType="Button">
    <!-- Шаблон элемента управления из предыдущего примера -->
  </ControlTemplate>
</Setter.Value>
</Setter>
</Style>

```

После такого обновления кнопочные элементы управления можно создавать с установкой свойства `Style` следующим образом:

```

<Button x:Name="myButton" Background="Red" Content="Howdy!"
  Click="myButton_Click" Style="{StaticResource RoundButtonStyle}"/>

```

Несмотря на то что внешний вид и поведение кнопки остаются такими же, преимущество внедрения шаблонов внутрь стилей связано с тем, что появляется возможность предоставить готовый набор значений для общих свойств. На этом обзор применения Visual Studio и инфраструктуры триггеров при построении специальных шаблонов для элемента управления завершен. Хотя об инфраструктуре WPF можно еще много чего сказать, теперь у вас имеется хороший фундамент для дальнейшего самостоятельного изучения.

Исходный код. Проект `ButtonTemplate` доступен в подкаталоге `Chapter_27`.

Резюме

Первой в главе рассматривалась система управления ресурсами WPF. Мы начали с исследования работы с двоичными ресурсами и роли объектных ресурсов. Вы узнали, что объектные ресурсы представляют собой именованные фрагменты разметки XAML, которые могут быть сохранены в разнообразных местах с целью многократного использования содержимого.

Затем был описан API-интерфейс анимации WPF. В приведенных примерах анимация создавалась с помощью кода C#, а также посредством разметки XAML. Для управления выполнением анимации, определенной в разметке, применяются элементы `Storyboard` и триггеры. Далее был продемонстрирован механизм стилей WPF, который интенсивно использует графику, объектные ресурсы и анимацию.

После этого выяснялось отношение между логическим и визуальным деревьями. В своей основе логическое дерево является однозначным соответствием разметке, которая создана для описания корневого элемента WPF. Позади логического дерева находится гораздо более глубокое визуальное дерево, содержащее детальные инструкции визуализации.

Кроме того, была изучена роль стандартного шаблона. Не забывайте, что при построении специальных шаблонов вы по существу заменяете все визуальное дерево элемента управления (или часть дерева) собственной реализацией.

ГЛАВА 28

Уведомления, проверка достоверности, команды и MVVM

В настоящей главе исследование программной модели WPF завершается рассмотрением возможностей, которые поддерживаются паттерном “модель-представление-модель представления” (Model View ViewModel — MVVM). Вы также узнаете о системе уведомлений WPF и ее реализации паттерна “Наблюдатель” (Observer) через наблюдаемые модели и коллекции. Обеспечение автоматического отображения пользовательским интерфейсом текущего состояния данных значительно улучшает его восприятие конечными пользователями и сокращает объем ручного кодирования, требуемого для получения того же результата с помощью более старых технологий (вроде Windows Forms).

Во время разработки на основе паттерна “Наблюдатель” вы ознакомитесь с механизмами добавления проверки достоверности в свои приложения. Проверка достоверности — жизненно важная часть любого приложения, которая позволяет не только сообщать пользователю о том, что что-то пошло не так, но и указывать, в чем именно заключается проблема. Вы также научитесь встраивать проверку достоверности в разметку представления для информирования пользователя о возникающих ошибках.

Затем мы более глубоко погрузимся в систему команд WPF и создадим специальные команды для инкапсуляции программной логики почти так, как поступали в главе 25 со встроенными командами. С созданием специальных команд связано несколько преимуществ, включая (но не ограничиваясь) возможность многократного использования кода, инкапсуляцию логики и разделение обязанностей.

Наконец, мы продемонстрируем весь рассмотренный материал на примере приложения MVVM.

Введение в паттерн MVVM

Прежде чем приступить к детальному исследованию уведомлений, проверки достоверности и команд в WPF, было бы неплохо пролить свет на конечную цель настоящей главы, которой является паттерн “модель-представление-модель представления” (MVVM). Будучи производным от паттерна проектирования “Модель представления” (Presentation Model) Мартина Фаулера, паттерн MVVM задействует обсуждаемые в главе возможности, специфичные для XAML, чтобы сделать процесс разработки приложений WPF более быстрым и ясным. Само название паттерна отражает его основные компоненты: модель (Model), представление (View) и модель представления (ViewModel).

Модель

Модель — это объектное представление имеющихся данных. В паттерне MVVM модели концептуально совпадают с моделями внутри нашего уровня доступа к данным (Data Access Layer — DAL). Иногда они являются теми же физическими классами, но поступать так вовсе не обязательно. По мере чтения главы вы узнаете, каким образом решать, применять ли модели DAL или же создавать новые модели.

Модели обычно используют в своих интересах встроенную (либо специальную) проверку достоверности через аннотации данных и интерфейс `INotifyDataErrorInfo` и сконфигурированы как наблюдаемые классы для связывания с системой уведомлений WPF. Все упомянутые темы рассматриваются позже в главе.

Представление

Представление — это пользовательский интерфейс приложения, который спроектирован так, чтобы быть чрезвычайно легковесным. Вспомните о стенде меню в ресторане для автомобилистов. На стенде отображаются позиции меню и цены, а также имеется механизм взаимодействия клиента с внутренними системами. Однако в стенд не внедрены какие-либо интеллектуальные возможности, разве что он может быть снабжен специальной логикой пользовательского интерфейса, такой как включение освещения в темное время суток.

Представления MVVM должны разрабатываться с учетом аналогичных целей. Любые интеллектуальные возможности необходимо встраивать в какие-то другие места приложения. Иметь прямое отношение к манипулированию пользовательским интерфейсом может только код в файле отделенного кода (например, в `MainWindow.xaml.cs`). Он не должен быть основан на бизнес-правилах или на чем-то еще, что нуждается в предохранении для будущего применения. Хотя это не является главной целью MVVM, хорошо разработанные приложения MVVM обычно имеют совсем небольшой объем отделенного кода.

Модель представления

В WPF и других технологиях XAML модель представления служит двум целям.

- Модель представления предлагает единственное местоположение для всех данных, необходимых представлению. Это вовсе не означает, что модель представления отвечает за получение действительных данных; взамен она является просто транспортным механизмом для перемещения данных из хранилища в представление. Обычно между представлениями и моделями представлений имеется отношение “один к одному”, но существуют архитектурные отличия, которые в каждом конкретном случае могут варьироваться.
- Вторая цель модели представления касается ее действия в качестве контроллера для представления. Почти как стенд меню модель представления принимает указание от пользователя и передает их соответствующему коду для выполнения подходящих действий. Довольно часто такой код имеет форму специальных команд.

Анемичные модели или анемичные модели представлений

На заре развития WPF, когда разработчики все еще были в поиске лучшей реализации паттерна MVVM, велись бурные (а временами и жаркие) дискуссии о том, где реализовывать элементы, подобные проверке достоверности и паттерну “Наблюдатель”. Один лагерь (сторонников анемичной (иногда называемой бескровной) модели) аргументировал, что все элементы должны находиться в моделях представлений, поскольку добавле-

ние таких возможностей к модели нарушает принцип разделения обязанностей. Другой лагерь (сторонников анемичной модели представления) утверждал, что все элементы должны находиться в моделях, т.к. тогда сокращается дублирование кода.

Естественно, реалистичным ответом будет “когда как”. Реализация классами моделей интерфейсов `INotifyPropertyChanged`, `IDataErrorInfo` и `INotifyDataErrorInfo` гарантирует, что соответствующий код близок к своей цели (как вы увидите далее в главе) и реализован только однократно для каждой модели. Другими словами, есть ситуации, когда сами классы моделей представлений необходимо разрабатывать как наблюдаемые. По большому счету вы должны самостоятельно выяснить, что имеет больший смысл для приложения, не приводя к чрезмерному усложнению кода и не принося в жертву преимущества MVVM.

На заметку! Для WPF доступны многочисленные инфраструктуры MVVM, такие как `MVVMLite`, `Calburn.Micro` и `Prism` (хотя `Prism` — нечто намного большее, чем просто инфраструктура MVVM). В настоящей главе обсуждается паттерн MVVM и функциональные средства WPF, которые поддерживают его реализацию. Исследование других инфраструктур и выбор среди них наиболее подходящей для нужд приложения остается за вами как разработчиком.

Система уведомлений привязки WPF

Значительным недостатком системы привязки Windows Forms является отсутствие уведомлений. Если находящиеся внутри представления данные модифицируются в коде, то пользовательский интерфейс также должен обновляться программно, чтобы оставаться в синхронном состоянии с ними. Итогом будет большое количество вызовов метода `Refresh()` на элементах управления, обычно превышающее абсолютно необходимое для обеспечения безопасности. Наряду с тем, что наличие слишком многих обращений к `Refresh()` обычно не приводит к серьезной проблеме с производительностью, недостаточное их число может отрицательно повлиять на пользовательский интерфейс.

Система привязки, встроенная в приложения на основе XAML, устраняет указанную проблему за счет того, что позволяет привязывать объекты данных и коллекции к системе уведомлений, разрабатывая их как наблюдаемые. Всякий раз, когда изменяется значение свойства в наблюдаемой модели либо происходит изменение в наблюдаемой коллекции (например, добавление, удаление или переупорядочение элементов), инициируется событие (`NotifyPropertyChanged` либо `NotifyCollectionChanged`). Инфраструктура привязки автоматически прослушивает такие события и в случае их появления обновляет привязанные элементы управления. Более того, разработчики имеют контроль над тем, для каких свойств выдаются уведомления. Выглядит безупречно, не так ли? На самом деле все не настолько безупречно. Настройка наблюдаемых моделей вручную требует написания довольно большого объема кода. К счастью, как вы вскоре увидите, существует инфраструктура с открытым кодом, которая значительно упрощает работу.

Наблюдаемые модели и коллекции

В этом разделе мы построим приложение, в котором используются наблюдаемые модели и коллекции. Для начала создадим новый проект приложения WPF по имени `WpfNotifications`. В приложении будет применяться форма “главная-подробности”, которая позволит пользователю выбирать объект автомобиля в элементе управления `ComboBox` и просматривать детальную информацию о нем в расположенных ниже элементах управления `TextBox`. Поместим в файл `MainWindow.xaml` следующую разметку:

```

<Grid IsSharedSizeScope="True" Margin="5,0,5,5">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <Grid Grid.Row="0">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" SharedSizeGroup="CarLabels"/>
      <ColumnDefinition Width="*/>
    </Grid.ColumnDefinitions>
    <Label Grid.Column="0" Content="Vehicle"/>
    <ComboBox Name="cboCars" Grid.Column="1" DisplayMemberPath="PetName" />
  </Grid>
  <Grid Grid.Row="1" Name="DetailsGrid">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" SharedSizeGroup="CarLabels"/>
      <ColumnDefinition Width="*/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Label Grid.Column="0" Grid.Row="0" Content="Make"/>
    <TextBox Grid.Column="1" Grid.Row="0" />
    <Label Grid.Column="0" Grid.Row="1" Content="Color"/>
    <TextBox Grid.Column="1" Grid.Row="1" />
    <Label Grid.Column="0" Grid.Row="2" Content="Pet Name"/>
    <TextBox Grid.Column="1" Grid.Row="2" />
    <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="3"
      HorizontalAlignment="Right" Orientation="Horizontal" Margin="0,5,0,5">
      <Button x:Name="btnAddCar" Content="Add Car" Margin="5,0,5,0"
        Padding="4, 2" />
      <Button x:Name="btnChangeColor" Content="Change Color" Margin="5,0,5,0"
        Padding="4, 2" />
    </StackPanel>
  </Grid>
</Grid>

```

Окно должно напоминать показанное на рис. 28.1.

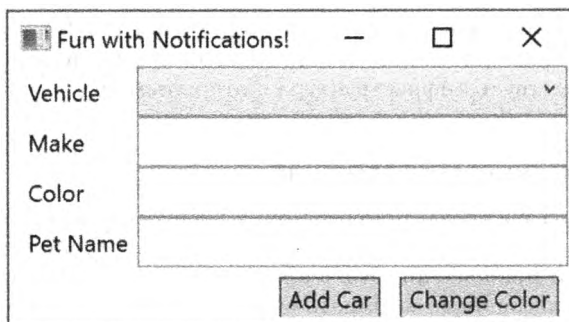


Рис. 28.1. Окно для отображения складской информации

Свойство `IsSharedSizeScope` элемента управления `Grid` заставляет дочерние сетки разделять размеры. Элемент `ColumnDefinitions`, помеченный как `SharedSizeGroup`, автоматически получит ту же самую ширину без каких-либо потребностей в программировании. В рассматриваемом примере, если размер метки `Pet Name` (Дружественное имя) изменяется из-за более длинного значения, тогда соответствующим образом корректируется и размер столбца `Vehicle` (Автомобиль), который находится в другом элементе управления `Grid`, сохраняя аккуратный внешний вид окна.

Щелкнем правой кнопкой мыши на имени проекта в окне `Solution Explorer`, выберем в контекстном меню пункт `Add⇒New Folder` (Добавить⇒Новая папка) и назовем новую папку именем `Models`. Создадим в новой папке класс под названием `Inventory`. Первоначально код класса выглядит так:

```
public class Inventory
{
    public int CarId { get; set; }
    public string Make { get; set; }
    public string Color { get; set; }
    public string PetName { get; set; }
}
```

Добавление привязок и данных

Следующий шаг заключается в создании операторов привязки для элементов управления. Вспомните, что конструкции привязки данных вращаются вокруг контекста данных, который может быть установлен в самом элементе управления или в родительском элементе управления. Здесь мы собираемся установить контекст в элементе `DetailsGrid`, так что каждый содержащийся внутри него элемент управления унаследует результирующий контекст данных.

Установим свойство `DataContext` в свойство `SelectedItem` элемента `ComboBox`. Модифицируем определение элемента `Grid`, содержащего элементы управления с информацией об автомобиле, следующим образом:

```
<Grid Grid.Row="1" Name="DetailsGrid"
    DataContext="{Binding ElementName=cboCars, Path=SelectedItem}">
```

Текстовые поля в элементе `DetailsGrid` будут отображать индивидуальные характеристики выбранного автомобиля. Добавим подходящие атрибуты `Text` и привязки к элементам управления `TextBox`:

```
<TextBox Grid.Column="1" Grid.Row="0" Text="{Binding Path=Make}" />
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=Color}" />
<TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Path=PetName}" />
```

Наконец, поместим нужные данные в элемент управления `ComboBox`. В файле `MainWindow.xaml.cs` создадим новый список записей `Inventory` и присвоим его свойству `ItemsSource` элемента `ComboBox`. Кроме того, добавим оператор `using` для пространства имен `WpfNotifications.Models`.

```
using WpfNotifications.Models;
// Для краткости код не показан
public partial class MainWindow : Window
{
    readonly IList<Inventory> _cars = new List<Inventory>();
    public MainWindow()
    {
        InitializeComponent();
```



```

        _cars.Add(new Inventory
        { CarId = 1, Color = "Blue", Make = "Chevy", PetName = "Kit" });
        _cars.Add(new Inventory
        { CarId = 2, Color = "Red", Make = "Ford", PetName = "Red Rider" });
        cboCars.ItemsSource = _cars;
    }
}

```

Запустим приложение. Как видно, в поле со списком Vehicle для выбора доступны два варианта автомобилей. Выбор одного из них приводит к автоматическому заполнению текстовых полей сведениями об автомобиле. Изменим цвет одного из автомобилей, выберем другой автомобиль и затем возвратимся к автомобилю, запись о котором редактировалась. Обнаружится, что новый цвет по-прежнему связан с автомобилем. Здесь нет ничего примечательного, просто демонстрируется мощь привязки данных XAML.

Изменение данных об автомобиле в коде

Несмотря на то что предыдущий пример работает ожидаемым образом, когда данные изменяются программно, пользовательский интерфейс не отразит изменения до тех пор, пока в приложении не будет предусмотрен код для обновления данных. Чтобы проиллюстрировать сказанное, добавим обработчик события Click для кнопки btnChangeColor:

```

<Button x:Name="btnChangeColor" Content="Change Color" Margin="5,0,5,0"
        Padding="4, 2" Click="BtnChangeColor_OnClick"/>

```

Внутри обработчика события BtnChangeColor_OnClick() с помощью свойства SelectedItem элемента управления ComboBox отыщем выбранную запись в списке автомобилей и изменим ее цвет на Pink:

```

private void BtnChangeColor_Click(object sender, RoutedEventArgs e)
{
    _cars.First(x => x.CarId ==
        ((Inventory) cboCars.SelectedItem)?.CarId).Color = "Pink";
}

```

Запустим приложение, выберем автомобиль и щелкнем на кнопке Change Color (Изменить цвет). Никаких видимых изменений не произойдет. Выберем другой автомобиль и затем снова первоначальный. Теперь можно заметить обновленное значение. Для пользователя такое поведение не особенно подходит.

Теперь добавим обработчик события Click для кнопки btnAddCar:

```

<Button x:Name="btnAddCar" Content="Add Car" Margin="5,0,5,0" Padding="4, 2"
        Click="BtnAddCar_OnClick" />

```

В обработчике события BtnAddCar_OnClick() добавим новую запись в список Inventory:

```

private void BtnAddCar_Click(object sender, RoutedEventArgs e)
{
    var maxCount = _cars?.Max(x => x.CarId) ?? 0;
    _cars?.Add(new Inventory
    { CarId=++maxCount, Color="Yellow", Make="VW", PetName="Birdie" });
}

```

Запустим приложение, щелкнем на кнопке Add Car (Добавить автомобиль) и посмотрим содержимое элемента управления ComboBox. Хотя известно, что в списке имеется три автомобиля, в элементе ComboBox отображаются только два! Чтобы устранить обе проблемы, мы преобразуем класс Inventory в наблюдаемую модель и будем использовать наблюдаемую коллекцию для хранения всех экземпляров Inventory.

Наблюдаемые модели

Проблема с тем, что изменение значения свойства модели не отображается в пользовательском интерфейсе, решается за счет реализации классом модели `Inventory` интерфейса `INotifyPropertyChanged`. Интерфейс `INotifyPropertyChanged` содержит единственное событие `PropertyChangedEvent`. Механизм привязки XAML прослушивает это событие для каждого привязанного свойства в классах, реализующих интерфейс `INotifyPropertyChanged`. Вот как определен интерфейс `INotifyPropertyChanged`:

```
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

Добавим в файл `Inventory.cs` следующие операторы `using`:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
```

Затем обеспечим реализацию классом `Inventory` интерфейса `INotifyPropertyChanged`:

```
public class Inventory : INotifyPropertyChanged
{
    // Для краткости код не показан
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Событие `PropertyChanged` принимает объектную ссылку и новый экземпляр класса `PropertyChangedEventArgs`:

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Model"));
```

Первый параметр представляет собой объект, который инициирует событие. Конструктор класса `PropertyChangedEventArgs` принимает строку, указывающую свойство, которое было изменено и нуждается в обновлении. Когда событие инициировано, механизм привязки ищет элементы управления, привязанные к именованному свойству данного объекта. В случае передачи конструктору `PropertyChangedEventArgs` значения `String.Empty` обновляются все привязанные свойства объекта.

Вы сами управляете тем, какие свойства вовлечены в процесс автоматического обновления. Автоматически обновляться будут только те свойства, которые генерируют событие `PropertyChanged` внутри блока `set`. Обычно в перечень входят все свойства классов моделей, но в зависимости от требований приложения некоторые свойства можно опускать. Вместо инициирования события `PropertyChanged` непосредственно в блоке `set` для каждого задействованного свойства распространенный подход предусматривает написание вспомогательного метода (как правило, называемого `OnPropertyChanged()`), который генерирует событие от имени свойств обычно в базовом классе для моделей. Добавим в класс `Inventory` следующий метод:

```
protected void OnPropertyChanged([CallerMemberName] string propertyName = "")
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

В версиях, предшествующих .NET 4.5, вспомогательному методу нужно было передавать строковое имя свойства. Когда имя свойства в классе изменялось, приходилось помнить о необходимости корректировки строки, переданной вспомогательному методу, иначе обновление не работало. Начиная с версии .NET 4.5, появилась возможность при-

менять атрибут [CallerMemberName], который присваивает помеченному им параметру (в данном случае propertyName) имя метода, вызывающего вспомогательный метод.

Обновим каждое автоматическое свойство класса Inventory, чтобы оно имело полноценные блоки get и set, а также поддерживающее поле. В случае если значение изменилось, вызовем вспомогательный метод OnPropertyChanged(). Вот модифицированное свойство CarId:

```
private int _carId;
public int CarId
{
    get => _carId;
    set
    {
        if (value == _carId) return;
        _carId = value;
        OnPropertyChanged();
    }
}
```

Проделаем то же самое со всеми остальными свойствами в классе и снова запустим приложение. Выберем автомобиль и щелкнем на кнопке Change Color. Изменение немедленно отобразится в пользовательском интерфейсе. Первая проблема решена!

Использование операции nameof

В версии C# 6 появилась операция nameof, которая возвращает строковое имя переданного ей элемента. Ее можно применять в вызовах метода OnPropertyChanged() внутри блоков set, например:

```
private string _color;
public string Color
{
    get { return _color; }
    set
    {
        if (value == _color) return;
        _color = value;
        OnPropertyChanged(nameof(Color));
    }
}
```

Обратите внимание на то, что в случае использования операции nameof удалять атрибут [CallerMemberName] из метода OnPropertyChanged() не обязательно (хотя он становится излишним). В конце концов, выбор между применением операции nameof или атрибута CallerMemberName зависит от личных предпочтений.

Наблюдаемые коллекции

Следующей проблемой, которую необходимо решить, является обновление пользовательского интерфейса при изменении содержимого коллекции, что достигается путем реализации интерфейса INotifyCollectionChanged. Подобно INotifyPropertyChanged данный интерфейс открывает доступ к единственному событию CollectionChanged. В отличие от INotifyPropertyChanged реализация интерфейса INotifyCollectionChanged вручную предполагает больший объем действий, чем просто вызов метода в блоке set свойства. Понадобится создать реализацию полного списка объектов и генерировать событие CollectionChanged каждый раз, когда он изменяется.

Использование класса *ObservableCollection<T>*

К счастью, существует намного более легкий способ, чем создание собственных классов коллекций. Класс *ObservableCollection<T>* реализует интерфейсы *INotifyCollectionChanged*, *INotifyPropertyChanged* и *Collection<T>* и входит в состав .NET Framework. Никакой дополнительной работы выполнять не придется. Чтобы продемонстрировать его применение, добавим оператор *using* для пространства имен *System.Collections.ObjectModel* и модифицируем закрытое поле *_cars* следующим образом:

```
private readonly IList<Inventory> _cars = new ObservableCollection<Inventory>();
```

Снова запустим приложение и щелкнем на кнопке *Add Car*. Новые записи будут должным образом появляться.

Реализация флага изменения

Еще одним преимуществом наблюдаемых моделей является способность отслеживать изменения состояния. В то время как некоторые инструменты объектно-реляционного отображения (ORM) вроде *Entity Framework* обеспечивают рудиментарное отслеживание состояния, благодаря наблюдаемым моделям отслеживание флагов изменения (когда изменено одно или более значений объекта) становится тривиальным. Добавим в класс *Inventory* свойство типа *bool* по имени *IsChanged*. Внутри его блока *set* вызовем метод *OnPropertyChanged()*, как поступали с другими свойствами класса *Inventory*.

```
private bool _isChanged;
public bool IsChanged {
    get => _isChanged;
    set
    {
        if (value == _isChanged) return;
        _isChanged = value;
        OnPropertyChanged();
    }
}
```

Свойство *IsChanged* необходимо устанавливать в *true* внутри метода *OnPropertyChanged()*. Важно не устанавливать свойство *IsChanged* в *true* в случае изменения его самого, иначе сгенерируется исключение переполнения стека! Модифицируем метод *OnPropertyChanged()* следующим образом (здесь используется описанная ранее операция *nameof*):

```
protected virtual
    void OnPropertyChanged([CallerMemberName] string propertyName = "")
{
    if (propertyName != nameof(IsChanged))
    {
        IsChanged = true;
    }
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

Откроем файл *MainWindow.xaml* и добавим в *DetailsGrid* дополнительный элемент *RowDefinition*. Поместим в конец элемента *Grid* показанную ниже разметку, которая содержит элементы управления *Label* и *CheckBox*, привязанные к свойству *IsChanged*:

```
<Label Grid.Column="0" Grid.Row="4" Content="Is Changed"/>
<CheckBox Grid.Column="1" Grid.Row="4" VerticalAlignment="Center"
    Margin="10,0,0,0" IsEnabled="False" IsChecked="{Binding Path=IsChanged}"/>
```

Если запустить приложение прямо сейчас, то окажется, что каждая отдельная запись отображается как измененная, хотя пока ничего не изменялось! Дело в том, что во время создания объекта устанавливаются значения свойств, а установка любых значений приводит к вызову метода `OnPropertyChanged()`, который и устанавливает свойство `IsChanged` объекта. Чтобы устранить проблему, установим свойство `IsChanged` в `false` последним в коде инициализации объекта. Откроем файл `MainWindow.xaml.cs` и модифицируем код, создающий список:

```
_cars.Add(
    new Inventory {CarId = 1, Color = "Blue", Make = "Chevy", PetName = "Kit",
        IsChanged = false});
_cars.Add(
    new Inventory {CarId = 2, Color = "Red", Make = "Ford",
        PetName = "Red Rider", IsChanged = false});
```

Снова запустим приложение, выберем автомобиль и щелкнем на кнопке `Change Color`. Флажок `Is Changed` (Изменено) становится отмеченным наряду с изменением цвета.

Обновление источника через взаимодействие с пользовательским интерфейсом

Во время выполнения приложения можно заметить, что при вводе в текстовых полях флажок `Is Changed` не становится отмеченным до тех пор, пока фокус не покинет элемент управления, в котором производился ввод. Причина кроется в свойстве `UpdateSourceTrigger` привязок элементов `TextBox`. Свойство `UpdateSourceTrigger` определяет, какое событие (изменение значения, переход фокуса и т.д.) является основанием для обновления пользовательским интерфейсом лежащих в основе данных. Перечисление `UpdateSourceTrigger` принимает значения, описанные в табл. 28.1.

Таблица 28.1. Значения перечисления `UpdateSourceTrigger`

Значение	Описание
Default	Устанавливает стандартное значение, принятое для элемента управления (например, <code>LostFocus</code> для <code>TextBox</code>)
Explicit	Обновляет объект источника только при вызове метода <code>UpdateSource()</code>
LostFocus	Обновляет, когда элемент управления теряет фокус. Является стандартным для <code>TextBox</code>
PropertyChanged	Обновляет при изменении свойства. Является стандартным для <code>CheckBox</code>

Стандартным событием обновления для элементов управления `TextBox` является `LostFocus`. Изменим его на `PropertyChanged`, модифицировав привязку для элемента `TextBox`, который отвечает за ввод цвета, следующим образом:

```
<TextBox Grid.Column="1" Grid.Row="1"
    Text="{Binding Path=Color, UpdateSourceTrigger=PropertyChanged}" />
```

Если теперь запустить приложение и начать ввод в текстовом поле `Color` (Цвет), то флажок `Is Changed` немедленно отметится. Может возникнуть вопрос о том, почему для элементов управления `TextBox` в качестве стандартного выбрано значение `LostFocus`. Дело в том, что проверка достоверности (рассматриваемая вскоре) для модели запускается в сочетании с `UpdateSourceTrigger`. В случае `TextBox` это может потенциально вызывать ошибки, которые будут постоянно возникать до тех пор, пока пользователь не введет корректное значение. Например, если правила проверки достоверности не разре-

шают вводить в элементе `TextBox` менее пяти символов, тогда сообщение об ошибке будет отображаться при каждом нажатии клавиши, пока пользователь не введет пять или более символов. В таких случаях с обновлением источника лучше подождать до момента, когда пользователь переместит фокус из элемента `TextBox` (завершив изменение текста).

Использование *PropertyChanged.Fody* для реализации наблюдаемых моделей

Легко представить, что в случае сложных моделей реализация интерфейса `INotifyPropertyChanged` может стать довольно утомительной. К счастью, умные головы нашли выход из затруднительного положения. Существует проект с открытым кодом под названием `PropertyChanged.Fody`, который решает данную проблему. Проект `PropertyChanged.Fody` является расширением проекта `Fody` (<https://github.com/Fody/Fody/>) — инструмента с открытым кодом, предназначенного для прошивки сборок .NET. Прошивка (weaving) представляет собой процесс манипулирования кодом IL, сгенерированным во время процесса построения. Проект `PropertyChanged.Fody` добавляет весь связующий код для `INotifyPropertyChanged`, и если есть свойство по имени `IsChanged`, то оно будет обновляться, когда изменяется другое свойство, в точности как это происходило в примере, рассмотренном ранее в главе.

На заметку! Дополнительные сведения о проекте `PropertyChanged` доступны по адресу <https://github.com/Fody/PropertyChanged>.

Чтобы установить необходимые пакеты, щелчком правой кнопкой мыши на имени проекта, выберем в контекстном меню пункт `Manage NuGet Packages` (Управление пакетами NuGet), в открывшемся окне `NuGet Package Manager` (Диспетчер пакетов NuGet) отыщем пакет `propertychanged.fody` и установим его. Затем переименуем класс `Inventory` на `InventoryManual` и создадим новый класс `Inventory` с таким кодом:

```
public class Inventory : INotifyPropertyChanged
{
    public bool IsChanged { get; set; }
    public int CarId { get; set; }
    public string Make { get; set; }
    public string Color { get; set; }
    public string PetName { get; set; }
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Когда проект скомпилируется, любой класс, реализующий интерфейс `INotifyPropertyChanged`, будет иметь весь связующий код (включая обновление блоков `set`, обработку события `IsChanged` и добавление метода `OnPropertyChanged()`). Запустив проект снова, можно заметить, что все обновления и функциональность работают ожидаемым образом!

Обновление пакета *Fody* может нарушить работу пакета *PropertyChanged.Fody*

Пакет `PropertyChanged.Fody` имеет зависимость от пакета `Fody`. В манифесте пакета `PropertyChanged.Fody` указана версия 2.0.0 пакета `Fody`, но текущей версией `Fody` является 2.1.2. Обновление пакета `Fody` до версии приведет к нарушению работы пакета `PropertyChanged.Fody`, во всяком случае, так было на момент написания главы. Причина в том, что обновление `Fody` переписывает файл `FodyWeavers.xml` и удаляет любые добавленные пакеты. К счастью, исправить проблему легко.

Откроем файл `FodyWeavers.xml` и поместим в него показанное ниже содержимое:

```
<?xml version="1.0" encoding="utf-8" ?>
<Weavers>
  <PropertyChanged />
</Weavers>
```

В результате пакет PropertyChanged.Fody вернется в проект.

Итоговые сведения об уведомлениях и наблюдаемых моделях

Применение интерфейсов `INotifyPropertyChanged` в моделях и классов `ObservableCollection` для списков улучшает пользовательский интерфейс приложения за счет поддержания его в синхронизированном состоянии с данными. В то время, как ни один из интерфейсов не является сложным, они требуют обновлений кода. К счастью, в инфраструктуре предусмотрен класс `ObservableCollection`, поддерживающий все необходимое для создания наблюдаемых коллекций. Также удачей следует считать обновление проекта Fody с целью автоматического добавления функциональности `INotifyPropertyChanged`. При наличии под рукой упомянутых двух инструментов нет никаких причин отказываться от реализации наблюдаемых моделей в своих приложениях WPF.

Исходный код. Проект `WpfNotifications` доступен в подкаталоге `Chapter_28`.

Проверка достоверности WPF

Теперь, когда интерфейс `INotifyPropertyChanged` реализован и задействован класс `ObservableCollection`, самое время заняться добавлением проверки достоверности в приложение. Приложениям необходимо проверять пользовательский ввод и обеспечивать обратную связь с пользователем, если введенные им данные оказываются некорректными. В настоящем разделе будут раскрыты наиболее распространенные механизмы проверки достоверности для современных приложений WPF, но это лишь часть возможностей, встроенных в инфраструктуру WPF.

На заметку! За полным описанием всех методов проверки достоверности в WPF обращайтесь к книге *WPF: Windows Presentation Foundation в .NET 4.5 с примерами на C# 5.0 для профессионалов* (ИД “Вильямс”).

Проверка достоверности происходит, когда привязка данных пытается обновить источник данных. В дополнение к встроенным проверкам, таким как исключения в блоках `set` для свойств, можно создавать специальные правила проверки достоверности. Если *любое* правило проверки достоверности (встроенное или специальное) нарушается, то в игру вступает класс `Validation`, который обсуждается позже в главе.

На заметку! В каждом разделе главы можно продолжить работу с проектом из предыдущего раздела или создать копию проекта, специально предназначенную для нового раздела. В каждом разделе главы мы создавали отдельное приложение, начиная с точки, где заканчивался предыдущий раздел. Такой подход позволяет видеть продвижение через материал главы в разных проектах. Обратите внимание, что специфичные для проектов пространства имен в каждом разделе будут ссылаться на используемый новый проект, поэтому удостоверьтесь в добавлении корректных пространств имен, если вы прорабатываете только какой-то один проект.

Модификация примера для демонстрации проверки достоверности

Код примера из предыдущего раздела был скопирован в новый проект по имени `WpfValidations`. Если вы работаете с тем же самым проектом, созданным в предыдущем разделе, то в последующих примерах кода просто уделяйте внимание изменениям пространств имен. Модифицируем файл `MainWindow.xaml`, добавив в `DetailsGrid` дополнительный элемент `RowDefinition`, в котором определим элементы `Label` и `TextBox` для свойства `CarId`, как показано далее (значения всех остальных свойств `Grid.Row` потребуется увеличить на 1):

```
<Label Grid.Column="0" Grid.Row="0" Content="Id"/>
<TextBox Grid.Column="1" Grid.Row="0" Text="{Binding Path=CarId}" />
```

Если запустить приложение и выбрать запись, то текстовое поле `Id` автоматически заполнится значением первичного ключа (как и ожидалось). Далее мы займемся исследованием процесса проверки достоверности в WPF.

Класс Validation

Прежде чем добавлять проверку достоверности в проект, важно понять назначение класса `Validation`. Он входит в состав инфраструктуры проверки достоверности и предоставляет методы и присоединяемые свойства, которые могут применяться для отображения результатов проверки. При обработке ошибок проверки обычно используются три основных свойства класса `Validation`, кратко описанные в табл. 28.2. Они будут применяться далее в разделе.

Таблица 28.2. Основные члены класса Validation

Член	Описание
<code>HasError</code>	Присоединяемое свойство, которое указывает, что правило проверки достоверности где-то в процессе было нарушено
<code>Errors</code>	Коллекция всех активных объектов <code>ValidationError</code>
<code>ErrorTemplate</code>	Шаблон элемента управления, который становится видимым и декорирует связанный элемент, когда свойство <code>HasError</code> установлено в <code>true</code>

Варианты проверки достоверности

Как упоминалось ранее, технологии XAML поддерживают несколько механизмов для встраивания логики проверки достоверности внутрь приложения. В последующих разделах рассматриваются три самых распространенных варианта проверки.

Уведомление по исключениям

Хотя исключения не должны использоваться для обеспечения выполнения бизнес-логики, они могут (и будут) возникать, а потому требуют обработки надлежащим образом. Если исключения не обработаны в коде, тогда пользователь должен получить визуальную обратную связь об имеющейся проблеме. В отличие от `Windows Forms` в инфраструктуре WPF исключения привязки (по умолчанию) не распространяются до пользователя как собственно исключения. Тем не менее, они указываются визуально с применением декоратора (визуального уровня, который находится над элементами управления).

Запустим приложение, выберем запись в элементе `ComboBox` и очистим значение в текстовом поле `Id`. Поскольку свойство `CarId` определено как имеющее тип `int` (не тип `int`,

допускающий null), требуется числовое значение. После покидания поля Id по нажатию клавиши <Tab> механизм привязки отправляет свойству CarId пустую строку, но из-за того, что пустая строка не может быть преобразована в значение int, внутри блока set генерируется исключение. В нормальных обстоятельствах необработанное исключение привело бы к отображению окна сообщения пользователю, но в данном случае ничего подобного не происходит. Взглянув на порцию Debug (отладка) окна Output (Вывод), можно заметить следующие строки:

```
System.Windows.Data Error: 7 : ConvertBack cannot convert value '' (type 'String').
BindingExpression:Path=CarId; DataItem='Inventory' (HashCode=24111608);
target element is
```

```
'TextBox' (Name=''); target property is 'Text' (type 'String')
```

```
FormatException:'System.
```

```
FormatException: Input string was not in a correct format.
```

Ошибка System.Windows.Data: 7 : ConvertBack не может преобразовать значение '' (типа String).

BindingExpression:Path=CarId; DataItem='Inventory' (HashCode=24111608); целевой элемент -

TextBox (Name=''); целевое свойство - Text (типа String)

FormatException:'System.

FormatException: Входная строка не имела корректный формат.

Визуально исключение представляется с помощью тонкого прямоугольника красного цвета вокруг элемента управления (рис. 28.2).

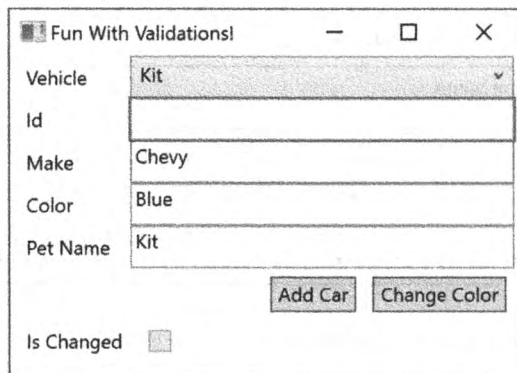


Рис. 28.2. Шаблон отображения ошибки по умолчанию

Прямоугольник красного цвета — это свойство `ErrorTemplate` объекта `Validation`, которое действует в качестве декоратора для связанного элемента управления. Несмотря на то что стандартный внешний вид говорит о наличии ошибки, нет никакого указания на то, что именно пошло не так. Хорошая новость в том, что шаблон отображения ошибки в свойстве `ErrorTemplate` является полностью настраиваемым, как вы увидите позже в главе.

Интерфейс `IDataErrorInfo`

Интерфейс `IDataErrorInfo` предоставляет механизм для добавления специальной проверки достоверности в классы моделей. Данный интерфейс добавляется прямо в классы моделей (или моделей представлений), а код проверки помещается внутрь классов моделей (предпочтительно в частичные классы). Такой подход централизует

код проверки достоверности в проекте, что совершенно не похоже на инфраструктуру Windows Forms, где проверка обычно делалась в самом пользовательском интерфейсе.

Показанный далее интерфейс `IDataErrorInfo` содержит два свойства: индексатор и строковое свойство по имени `Error`. Следует отметить, что механизм привязки WPF не задействует свойство `Error`.

```
public interface IDataErrorInfo
{
    string this[string columnName] { get; }
    string Error { get; }
}
```

Вскоре мы добавим частичный класс `Inventory`, но сначала необходимо модифицировать класс `Inventory` и пометить его как частичный. Добавим в папку `Models` еще один файл по имени `InventoryPartial.cs`. Переименуем его в `Inventory`, пометим как `partial` и реализуем интерфейс `IDataErrorInfo`. Затем реализуем члены интерфейса `IDataErrorInfo`. Вот начальный код:

```
public partial class Inventory : IDataErrorInfo
{
    private string _error;
    public string this[string columnName]
    {
        get { return string.Empty; }
    }
    public string Error => _error;
}
```

Чтобы привязанный элемент управления мог работать с интерфейсом `IDataErrorInfo`, в выражение привязки потребуется добавить `ValidatesOnDataErrors`. Модифицируем выражение привязки для текстового поля `Make` следующим образом (и аналогично обновим остальные конструкции привязки):

```
<TextBox Grid.Column="1" Grid.Row="1"
        Text="{Binding Path=Make, ValidatesOnDataErrors=True}" />
```

После внесения изменений в конструкции привязки индексатор вызывается на модели каждый раз, когда возникает событие `PropertyChanged`. В качестве параметра `columnName` индексатора используется имя свойства из события. Если индексатор возвращает `string.Empty`, то инфраструктура предполагает, что все проверки достоверности прошли успешно и какие-либо ошибки отсутствуют. Если индексатор возвращает значение, отличающееся от `string.Empty`, тогда в свойстве для данного объекта присутствует ошибка, из-за чего каждый элемент управления, привязанный к этому свойству специфического экземпляра класса, считается содержащим ошибку. Свойство `HasError` объекта `Validation` устанавливается в `true` и активизируется декоратор `ErrorTemplate` для элементов управления, на которые повлияла ошибка.

Добавим простую логику проверки достоверности к индексатору в файле `InventoryPartial.cs`. Правила проверки элементарны:

- если `Make` равно `ModelT`, то установить сообщение об ошибке в `"Too Old"` (слишком старая модель);
- если `Make` равно `Chevy` и `Color` равно `Pink`, то установить сообщение об ошибке в `($"{Make}'s don't come in {Color}"` (модель в таком цвете не поставляется).

Начнем с добавления оператора `switch` для каждого свойства. Во избежание применения “магических” строк в операторах `case` мы снова будем использовать операцию `nameof`. В случае сквозного прохода через оператор `switch` возвращается `string.Empty`.

Далее добавим правила проверки достоверности. В подходящих операторах `case` реализуем проверку значения свойства на основе приведенных выше правил. В операторе `case` для свойства `Make` первым делом проверим, равно ли значение `ModelT`. Если это так, тогда возвращается сообщение об ошибке. В случае успешного прохождения проверки в следующей строке кода вызывается вспомогательный метод, который возвращает сообщение об ошибке, если нарушено второе правило, или `string.Empty`, если нет. В операторе `case` для свойства `Color` просто вызовем тот же вспомогательный метод. Ниже показан код:

```
public string this[string columnName]
{
    get
    {
        switch (columnName)
        {
            case nameof(CarId):
                break;
            case nameof(Make):
                if (Make == "ModelT")
                {
                    return "Too Old";
                }
                return CheckMakeAndColor();
            case nameof(Color):
                return CheckMakeAndColor();
            case nameof(PetName):
                break;
        }
        return string.Empty;
    }
}

internal string CheckMakeAndColor()
{
    if (Make == "Chevy" && Color == "Pink")
    {
        return $"{Make}'s don't come in {Color}";
    }
    return string.Empty;
}
```

Запустим приложение, выберем автомобиль `Red Rider (Ford)` и изменим значение в поле `Make` (Производитель) на `ModelT`. После того, как фокус покинет поле, появится декоратор ошибки красного цвета. Выберем в поле со списком автомобилей `Kit (Chevy)` и щелкнем на кнопке `Change Color`, чтобы изменить его цвет на `Pink`. Вокруг поля `Color` незамедлительно появится декоратор ошибки красного цвета, но возле поля `Make` он будет отсутствовать. Изменим значение в поле `Make` на `Ford` и переместим фокус из этого поля: декоратор ошибки красного цвета не появляется!

Причина в том, что индекатор выполняется, только когда для свойства сгенерировано событие `PropertyChanged`. Как обсуждалось в разделе "Система уведомлений привязки WPF" ранее в главе, событие `PropertyChanged` инициируется при изменении исходного значения свойства объекта, что происходит либо через код (вроде обработчика события `Click` для кнопки `Change Color`), либо через взаимодействие с пользователем (синхронизируется с помощью `UpdateSourceTrigger`). При изменении цвета свойство `Make` не изменяется, а потому событие `PropertyChanged` для него не генерируется.

Поскольку событие не генерируется, индексатор не вызывается и проверка достоверности для свойства `Make` не выполняется.

Решить проблему можно двумя путями. Первый предусматривает изменение объекта `PropertyChangedEventArgs`, которое обеспечит обновление всех привязанных свойств, за счет передачи его конструктору значения `string.Empty` вместо имени поля. Как упоминалось ранее, это заставит механизм привязки обновить *каждое* свойство в данном экземпляре. Добавим метод `OnPropertyChanged()` со следующим кодом:

```
protected virtual
    void OnPropertyChanged([CallerMemberName] string propertyName = "")
{
    if (propertyName != nameof(IsChanged))
    {
        IsChanged = true;
    }
    // PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(string.Empty));
}
```

На заметку! Пакет `PropertyChanged.Fody` автоматически добавляет метод `OnPropertyChanged()`, если не обнаруживает его в классе, помеченном посредством `INotifyPropertyChanged`. Поскольку необходимо изменить стандартную реализацию, обновляя все поля, мы добавляем метод самостоятельно, и он будет задействован пакетом `PropertyChanged.Fody`.

Теперь при прогоне того же самого теста текстовые поля `Make` и `Color` декорируются с помощью шаблона отображения ошибки, когда одно из них обновляется. Так почему бы ни генерировать событие всегда в такой манере? В значительной степени причиной является производительность. Вполне возможно, что обновление каждого свойства объекта приведет к снижению производительности. Разумеется, без тестирования об этом утверждать нельзя, и конкретные ситуации могут (и вероятно будут) варьироваться.

Другое решение предполагает генерацию события `PropertyChanged` для зависимого поля (полей), когда одно из полей изменяется. Недостаток такого приема в том, что вы (или другие разработчики, сопровождающие ваше приложение) должны знать о взаимосвязи между свойствами `Make` и `Color` через код проверки достоверности.

Интерфейс `INotifyDataErrorInfo`

Интерфейс `INotifyDataErrorInfo`, появившийся в версии .NET 4.5, построен на основе интерфейса `IDataErrorInfo` и предлагает дополнительные возможности для проверки достоверности. Конечно, возросшая мощь сопровождается дополнительной работой! По разительному контрасту с предшествующими приемами проверки достоверности, которые вы видели до сих пор, свойство привязки `ValidatesOnNotifyDataErrors` имеет стандартное значение `true`, поэтому добавлять его к операторам привязки не обязательно.

Интерфейс `INotifyDataErrorInfo` чрезвычайно мал, но для обеспечения своей эффективности требует написания порядочного объема связующего кода, как вскоре будет показано. Ниже приведено определение интерфейса `INotifyDataErrorInfo`:

```
public interface INotifyDataErrorInfo
{
    bool HasErrors { get; }
    event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;
    IEnumerable GetErrors(string propertyName);
}
```

Свойство `HasErrors` используется механизмом привязки для выяснения, есть ли *какие-нибудь* ошибки в любых свойствах экземпляра. Если метод `GetErrors()` вызывается со значением `null` или пустой строкой в параметре `propertyName`, то он возвращает все ошибки, существующие в экземпляре. Если методу передан параметр `propertyName`, тогда возвращаются только ошибки, относящиеся к конкретному свойству. Событие `ErrorsChanged` (подобно событиям `PropertyChanged` и `CollectionChanged`) уведомляет механизм привязки о необходимости обновления пользовательского интерфейса для текущего списка ошибок.

Реализация поддерживающего кода

При реализации `INotifyDataErrorInfo` большая часть кода обычно помещается в базовый класс модели, поэтому она пишется только один раз.

Начнем с замены `IDataErrorInfo` интерфейсом `INotifyDataErrorInfo` в файле класса `InventoryPartial.cs` (код для `IDataErrorInfo` в классе можно оставить; мешать он не будет).

После добавления реализации членов интерфейса добавим закрытую переменную типа `Dictionary<string, List<string>>`, которая будет хранить сведения о любых ошибках, сгруппированные по именам свойств. Понадобится также добавить оператор `using` для пространства имен `System.Collections.Generic`. Вот как выглядит код:

```
using System.Collections.Generic;
private readonly Dictionary<string, List<string>> _errors =
    new Dictionary<string, List<string>>();
```

Свойство `HasErrors` должно возвращать `true`, если в словаре присутствуют *любые* ошибки, что легко достигается следующим образом:

```
public bool HasErrors => _errors.Count != 0;
```

Создадим вспомогательный метод для инициирования события `ErrorsChanged` (подобно инициированию события `PropertyChanged`):

```
private void OnErrorsChanged(string propertyName)
{
    ErrorsChanged?.Invoke(this, new DataErrorsChangedEventArgs(propertyName));
}
```

Как упоминалось ранее, метод `GetErrors()` должен возвращать любые ошибки в словаре, когда в параметре передается пустая строка или `null`. Если передается допустимое значение `propertyName`, то возвращаются ошибки, обнаруженные для указанного свойства. Если параметр не соответствует какому-либо свойству (или ошибки для свойства отсутствуют), тогда метод возвратит `null`.

```
public IEnumerable GetErrors(string propertyName)
{
    if (string.IsNullOrEmpty(propertyName))
    {
        return _errors.Values;
    }
    return _errors.ContainsKey(propertyName) ? _errors[propertyName] : null;
}
```

Финальный набор вспомогательных методов будет добавлять одну или большее число ошибок для свойства либо очищать все ошибки для свойства или всех свойств. Не следует забывать о вызове вспомогательного метода `OnErrorsChanged()` каждый раз, когда словарь изменяется.

```

private void AddError(string propertyName, string error)
{
    AddErrors(propertyName, new List<string> { error });
}
private void AddErrors(string propertyName, IList<string> errors)
{
    var changed = false;
    if (!_errors.ContainsKey(propertyName))
    {
        _errors.Add(propertyName, new List<string>());
        changed = true;
    }
    foreach (var err in errors)
    {
        if (_errors[propertyName].Contains(err)) continue;
        _errors[propertyName].Add(err);
        changed = true;
    }
    if (changed)
    {
        OnErrorsChanged(propertyName);
    }
}
protected void ClearErrors(string propertyName = "")
{
    if (String.IsNullOrEmpty(propertyName))
    {
        _errors.Clear();
    }
    else
    {
        _errors.Remove(propertyName);
    }
    OnErrorsChanged(propertyName);
}

```

Возникает вопрос: когда приведенный выше код активизируется? Механизм привязки прослушивает событие `ErrorsChanged` и обновляет пользовательский интерфейс, если в коллекции ошибок для выражения привязки возникает изменение. Но код проверки по-прежнему нуждается в триггере для запуска. Доступны два механизма, которые обсуждаются далее.

Использование интерфейса `INotifyDataErrorInfo` для проверки достоверности

Одним из мест выполнения проверки на предмет ошибок являются блоки `set` для свойств, как демонстрируется в показанном ниже примере, упрощенном до единственной проверки на равенство свойства `Make` значению `ModelT`:

```

public string Make
{
    get { return _make; }
    set
    {
        if (value == _make) return;
        _make = value;
        if (Make == "ModelT")
        {

```

```

        AddError(nameof(Make), "Too Old");
    }
    else
    {
        ClearErrors(nameof(Make));
    }
    OnPropertyChanged(nameof(Make));
    // OnPropertyChanged(nameof(Color));
}
}

```

С таким подходом связано несколько проблем. Во-первых, он не работает с внедрением поддержки `INotifyPropertyChanged` посредством `PropertyChanged.Fody`. Во-вторых, логику проверки достоверности приходится сочетать с блоками `set` для свойств, что делает код труднее в чтении и сопровождении.

Комбинирование `IDataErrorInfo` с `INotifyDataErrorInfo` для проверки достоверности

В предыдущем разделе было показано, что реализацию интерфейса `IDataErrorInfo` можно добавить к частичному классу, т.е. обновлять блоки `set` не понадобится. Кроме того, индекса́тор автоматически вызывается при возникновении события `PropertyChanged` в свойстве. Комбинирование `IDataErrorInfo` и `INotifyDataErrorInfo` предоставляет дополнительные возможности для проверки достоверности из `INotifyDataErrorInfo`, а также отделение от блоков `set`, обеспечиваемое `IDataErrorInfo`.

Добавим реализацию интерфейса `IDataErrorInfo` в класс `Inventory`, код которого находится в файле `InventoryPartial.cs`:

```
public partial class Inventory : IDataErrorInfo, INotifyDataErrorInfo
```

Цель применения `IDataErrorInfo` не в том, чтобы запускать проверку достоверности, а в том, чтобы гарантировать вызов кода проверки, который задействует `INotifyDataErrorInfo`, каждый раз, когда для объекта генерируется событие `PropertyChanged`. Поскольку интерфейс `IDataErrorInfo` не используется для проверки достоверности, необходимо всегда возвращать `string.Empty`. Модифицируем индекса́тор и вспомогательный метод `CheckMakeAndColor()` следующим образом:

```

public string this[string columnName]
{
    get
    {
        bool hasError = false;
        switch (columnName)
        {
            case nameof(CarId):
                break;
            case nameof(Make):
                hasError = CheckMakeAndColor();
                if (Make == "ModelT")
                {
                    AddError(nameof(Make), "Too Old");
                    hasError = true;
                }
            if (!hasError)
            {
                // Логика не безупречна, а просто иллюстрирует паттерн
                ClearErrors(nameof(Make));
                ClearErrors(nameof(Color));
            }
            break;
        }
    }
}

```

```

    case nameof(Color):
        hasError = CheckMakeAndColor();
        if (!hasError)
        {
            ClearErrors(nameof(Make));
            ClearErrors(nameof(Color));
        }
        break;
    case nameof(PetName):
        break;
}
return string.Empty;
}
}

internal bool CheckMakeAndColor()
{
    if (Make == "Chevy" && Color == "Pink")
    {
        AddError(nameof(Make), $"{Make}'s don't come in {Color}");
        AddError(nameof(Color), $"{Make}'s don't come in {Color}");
        return true;
    }
    return false;
}

```

Запустим приложение, выберем автомобиль Chevy и изменим цвет на Pink. В дополнение к декораторам красного цвета вокруг текстовых полей Make и Model будет также отображаться декоратор в виде красного прямоугольника, охватывающего целиком всю сетку, в которой находятся поля с детальной информацией из таблицы Inventory (рис. 28.3).

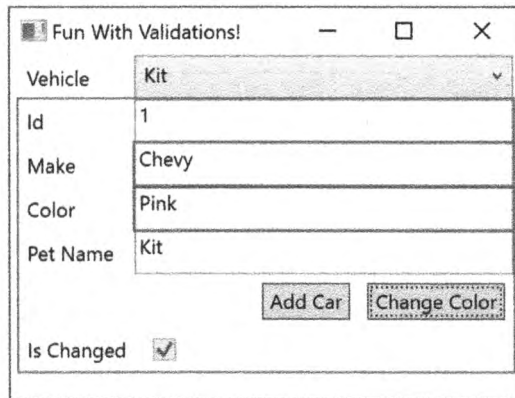


Рис. 28.3. Обновленный декоратор ошибки

Это еще одно преимущество применения интерфейса `INotifyDataErrorInfo`. В дополнение к элементам управления, которые содержат ошибки, элемент управления, определяющий контекст данных, также декорируется шаблоном отображения ошибки.

Отображение всех ошибок

Свойство `Errors` класса `Validation` возвращает все ошибки проверки достоверности для конкретного объекта в форме объектов `ValidationError`. Каждый объект `ValidationError` имеет свойство `ErrorContent`, которое содержит список сообщений

об ошибках для свойства. Это означает, что сообщения об ошибках, которые нужно отобразить, находятся в списке внутри списка. Чтобы вывести их надлежащим образом, понадобится создать элемент `ListBox`, содержащий еще один элемент `ListBox`. Звучит слегка запутанно, но вскоре все прояснится.

Первым делом добавим одну строку в `DetailsGrid` и увеличим значение свойства `Height` элемента `Window` до 300. В последнюю строку поместим элемент управления `ListBox` и привяжем его свойство `ItemsSource` к `DetailsGrid`, используя `Validation.Errors` для `Path`:

```
<ListBox Grid.Row="6" Grid.Column="0" Grid.ColumnSpan="2"
  ItemsSource="{Binding ElementName=DetailsGrid, Path=(Validation.Errors)}">
</ListBox>
```

Добавим к `ListBox` элемент `DataTemplate`, а в него — элемент управления `ListBox`, привязанный к свойству `ErrorContent`. Контекстом данных для каждого элемента `ListBoxItem` в этом случае является объект `ValidationError`, так что устанавливать контекст данных не придется, а только путь. Установим путь привязки в `ErrorContent`:

```
<ListBox.ItemTemplate>
  <DataTemplate>
    <ListBox ItemsSource="{Binding Path=ErrorContent}"/>
  </DataTemplate>
</ListBox.ItemTemplate>
```

Запустим приложение, выберем автомобиль `Chevy` и установим цвет в `Pink`. В окне отобразятся ошибки (рис. 28.4).

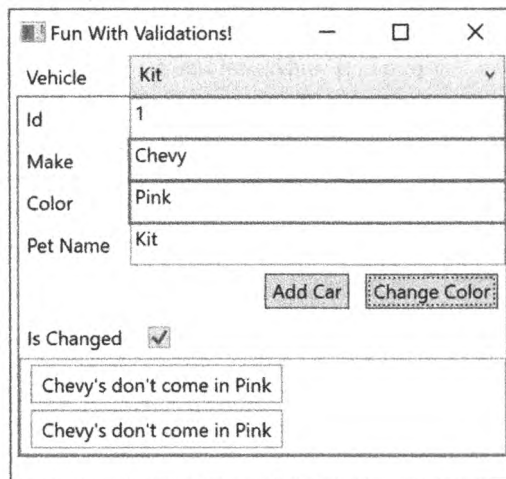


Рис. 28.4. Отображение коллекции ошибок

Мы лишь слегка коснулись поверхности того, что можно делать при проверке достоверности и отображении сообщений об ошибках, но представленных сведений должно быть вполне достаточно для выработки вами способа разработки информативных пользовательских интерфейсов, которые улучшают восприятие.

Перемещение поддерживающего кода в базовый класс

Вероятно, вы заметили, что в настоящий момент в классе `InventoryPartial` присутствует много кода. Поскольку в рассматриваемом примере есть только один класс модели, проблемы не возникают. Но по мере появления новых моделей в реальном при-

ложении добавлять весь связующий код в каждый частичный класс для моделей нежелательно. Гораздо эффективнее поместить поддерживающий код в базовый класс, что мы и сделаем.

Создадим в папке Models новый файл класса по имени EntityBase.cs. Добавим в него операторы using для пространств имен System.Collections и System.ComponentModel. Пометим класс как открытый и обеспечим реализацию им интерфейса INotifyDataErrorInfo.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
namespace Validations.Models
{
    public class EntityBase : INotifyDataErrorInfo
    }
}
```

Переместим в новый базовый класс весь код, относящийся к INotifyDataErrorInfo, из файла InventoryPartial.cs. Любые закрытые методы понадобится сделать защищенными. Удалим реализацию интерфейса INotifyDataErrorInfo из класса InventoryPartial и добавим EntityBase в качестве базового класса:

```
public partial class Inventory : EntityBase, IDataErrorInfo
{
    // Для краткости код не показан
}
```

Теперь любые создаваемые классы моделей будут наследовать весь связующий код INotifyDataErrorInfo.

Использование аннотаций данных в WPF

Для проверки достоверности в пользовательских интерфейсах инфраструктура WPF способна также задействовать аннотации данных. Давайте добавим несколько аннотаций данных к модели Inventory.

Добавление аннотаций данных к модели

Добавим ссылку на сборку System.ComponentModel.DataAnnotations.dll, откроем файл Inventory.cs и поместим в него оператор using для пространства имен System.ComponentModel.DataAnnotations. Добавим к свойствам CarId, Make и Color атрибут [Required], а к свойствам Make, Color, and PetName — атрибут [StringLength(50)]. Атрибут Required определяет правило проверки достоверности, которое регламентирует, что значение свойства не должно быть null (надо сказать, оно избыточно для свойства CarId, т.к. свойство не относится к типу int, допускающему null). Атрибут StringLength(50) определяет правило проверки достоверности, которое ограничивает длину значения свойства 50 символами.

Контроль ошибок проверки достоверности на основе аннотаций данных

В отличие от инфраструктуры ASP.NET, которая способна автоматически контролировать наличие ошибок проверки достоверности на основе аннотаций данных, в WPF это приходится делать программно. Двумя основными классами, отвечающими за проверку достоверности на основе аннотаций данных, являются ValidationContext и Validator. Класс ValidationContext предоставляет контекст для контроля за наличием ошибок проверки достоверности. Класс Validator позволяет проверять, есть ли в объекте ошибки, связанные с аннотациями данных, в ValidationContext.

Откроем файл EntityBase.cs и добавим в него следующие операторы using:

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
```

Далее создадим новый метод по имени GetErrorsFromAnnotations(). Это обобщенный метод, который принимает в качестве параметров строковое имя свойства и значение типа T, а возвращает строковый массив. Он должен быть помечен как protected. Вот его сигнатура:

```
protected string[] GetErrorsFromAnnotations<T>(string propertyName, T value)
```

Внутри метода GetErrorsFromAnnotations() создадим переменную типа List<ValidationResult>, которая будет хранить результаты выполненных проверок достоверности, и объект ValidationContext с областью действия, ограниченной именем переданного методу свойства. Затем вызовем метод Validate.TryValidateProperty(), который возвращает значение bool. Если все проверки (на основе аннотаций данных) прошли успешно, тогда метод возвращает true. В противном случае он возвратит false и наполнит List<ValidationResult> информацией о возникших ошибках. Полный код выглядит следующим образом:

```
protected string[] GetErrorsFromAnnotations<T>(string propertyName, T value)
{
    var results = new List<ValidationResult>();
    var vc = new ValidationContext(this, null, null) { MemberName = propertyName };
    var isValid = Validator.TryValidateProperty(value, vc, results);
    return (isValid) ? null : Array.ConvertAll(results.ToArray(), o => o.ErrorMessage);
}
```

Теперь можно модифицировать метод индексатора в файле InventoryPartial.cs, чтобы проверять наличие любых ошибок, основанных на аннотациях данных. Обнаруженные ошибки должны добавляться в коллекцию ошибок, поддерживаемую интерфейсом INotifyDataErrorInfo. Ниже показан обновленный код индексатора:

```
bool hasError = false;
switch (columnName)
{
    case nameof(CarId):
        AddErrors(nameof(CarId), GetErrorsFromAnnotations(nameof(CarId), CarId));
        break;
    case nameof(Make):
        hasError = CheckMakeAndColor();
        if (Make == "ModelT")
        {
            AddError(nameof(Make), "Too Old");
            hasError = true;
        }
        if (!hasError)
        {
            // Логика не безупречна, а просто иллюстрирует паттерн
            ClearErrors(nameof(Make));
            ClearErrors(nameof(Color));
        }
        AddErrors(nameof(Make), GetErrorsFromAnnotations(nameof(Make), Make));
        break;
    case nameof(Color):
        hasError = CheckMakeAndColor();
        if (!hasError)
```

```

    {
        ClearErrors(nameof(Make));
        ClearErrors(nameof(Color));
    }
    AddErrors(nameof(Color), GetErrorsFromAnnotations(nameof(Color), Color));
    break;
case nameof(PetName):
    AddErrors(nameof(PetName), GetErrorsFromAnnotations(nameof(PetName), PetName));
    break;
}
return string.Empty;

```

Понадобится также модифицировать метод `AddErrors()`, чтобы пустой список не добавлялся в словарь `_errors`:

```

protected void AddErrors(string propertyName, IList<string> errors)
{
    if (errors?.Count == 0)
    {
        return;
    }
    // Для краткости код не показан
}

```

Запустим приложение, выберем один из автомобилей и введем в поле `Color` текст, содержащий более 50 символов. После превышения порога в 50 символов аннотация данных `StringLength` создает ошибку проверки достоверности, которая сообщается пользователю (рис. 28.5).

Рис. 28.5. Проверка достоверности на основе аннотаций данных

Настройка свойства `ErrorTemplate`

Финальной темой является создание стиля, который будет применяться, когда элемент управления содержит ошибку, а также обновление `ErrorTemplate` для отображения более осмысленного сообщения об ошибке. Как объяснялось в главе 27, элементы управления допускают настройку посредством стилей и шаблонов элементов управления.

Начнем с добавления в раздел `Window.Resources` файла `MainWindow.xaml` нового стиля с целевым типом `TextBox`. Добавим к стилю триггер, который устанавливает свойства, когда свойство `Validation.HasError` имеет значение `true`. Свойствами и устанавливаемыми значениями являются `Background (Pink)`, `Foreground (Black)` и `ToolTip (ErrorContent)`. В элементах `Setter` для свойств `Background` и `Foreground` нет ничего нового, но синтаксис установки свойства `ToolTip` требует пояснения. Привязка (`Binding`) указывает на элемент управления, к которому применяется данный стиль, в этом случае `TextBox`. Путь (`Path`) представляет собой первое значение `ErrorContent` в коллекции `Validation.Errors`. Разметка выглядит следующим образом:

```
<Window.Resources>
  <Style TargetType="{x:Type TextBox}">
    <Style.Triggers>
      <Trigger Property="Validation.HasError" Value="true">
        <Setter Property="Background" Value="Pink" />
        <Setter Property="Foreground" Value="Black" />
        <Setter Property="ToolTip"
          Value="{Binding RelativeSource={RelativeSource Self},
            Path=(Validation.Errors)[0].ErrorContent}"/>
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

Запустим приложение и создадим условие для ошибки. Результат будет подобен показанному на рис. 28.6, укомплектованный всплывающей подсказкой с сообщением об ошибке.

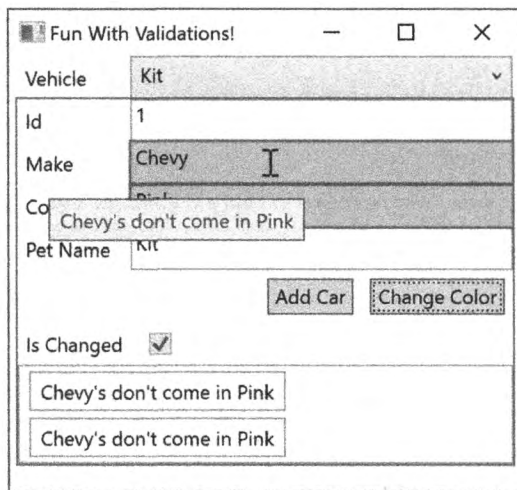


Рис. 28.6. Отображение специального шаблона `ErrorTemplate`

Определенный выше стиль изменяет внешний вид любого элемента управления `TextBox`, который содержит ошибку. Далее мы создадим специальный шаблон элемента управления с целью обновления свойства `ErrorTemplate` класса `Validation`, чтобы отобразить восклицательный знак красного цвета и установить всплывающие подсказки для восклицательного знака. Шаблон `ErrorTemplate` является *декоратором*, который располагается поверх элемента управления. Хотя только что созданный стиль

обновляет сам элемент управления, шаблон `ErrorTemplate` будет размещаться поверх элемента управления.

Поместим элемент `Setter` непосредственно после закрывающего дескриптора `Style.Triggers` внутри созданного стиля. Мы создадим шаблон элемента управления, состоящий из элемента `TextBlock` (для отображения восклицательного знака) и элемента `BorderBrush`, который окружает `TextBox`, содержащий сообщение об ошибке (или несколько сообщений). В языке XAML предусмотрен специальный дескриптор для элемента управления, декорированного с помощью `ErrorTemplate`, под названием `AdornedElementPlaceholder`. Добавляя имя такого элемента управления, можно получить доступ к ошибкам, которые ассоциированы с элементом управления. В рассматриваемом примере нам необходим доступ к свойству `Validation.Errors`, чтобы получить `ErrorContent` (как делалось в `Style.Trigger`). Вот полная разметка для элемента `Setter`:

```
<Setter Property="Validation.ErrorTemplate">
  <Setter.Value>
    <ControlTemplate>
      <DockPanel LastChildFill="True">
        <TextBlock Foreground="Red" FontSize="20" Text="!"
          Tooltip="{Binding ElementName=controlWithError,
            Path=AdornedElement.(Validation.Errors)[0].ErrorContent}"/>
        <Border BorderBrush="Red" BorderThickness="1">
          <AdornedElementPlaceholder Name="controlWithError" />
        </Border>
      </DockPanel>
    </ControlTemplate>
  </Setter.Value>
</Setter>
```

Запустим приложение и создадим условие для возникновения ошибки. Результат будет подобен представленному на рис. 28.7.

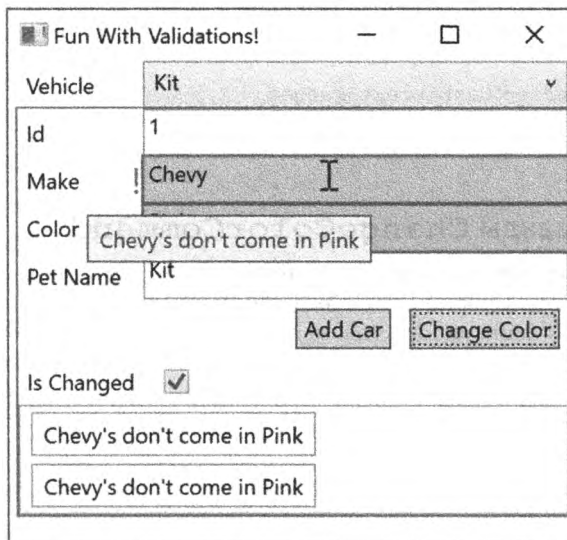


Рис. 28.7. Отображение обновленного специального шаблона `ErrorTemplate`

Итоговые сведения о проверке достоверности

На этом исследование методов проверки достоверности в WPF завершено. Разумеется, с их помощью можно делать намного большее. За дополнительными сведениями обращайтесь в документацию по WPF.

Исходный код. Проект `WpfValidations` доступен в подкаталоге `Chapter_28`.

Создание специальных команд

Как и в разделе, посвященном проверке достоверности, вы можете продолжить работу с тем же проектом или создать новый проект и скопировать в него весь код из предыдущего проекта. Мы создадим новый проект по имени `WpfCommands`. Если вы работаете с проектом из предыдущего раздела, тогда обращайте внимание на пространства имен в примерах кода и корректируйте их по мере необходимости.

В главе 25 объяснялось, что команды являются неотъемлемой частью WPF. Команды могут привязываться к элементам управления WPF (таким как `Button` и `MenuItem`) для обработки пользовательских событий, подобных щелчку. Вместо создания обработчика события напрямую и помещения его кода в файл отделенного кода при возникновении события выполняется метод `Execute()` команды. Метод `CanExecute()` используется для включения или отключения элемента управления на основе специального кода. В дополнение к встроенным командам, которые применялись в главе 25, можно создавать собственные команды, реализуя интерфейс `ICommand`. Когда вместо обработчиков событий используются команды, появляются преимущества инкапсуляции кода приложения, а также автоматического включения и отключения элементов управления с помощью бизнес-логики.

Реализация интерфейса `ICommand`

Как было показано в главе 25, интерфейс `ICommand` определен следующим образом:

```
public interface ICommand
{
    event EventHandler CanExecuteChanged;
    bool CanExecute(object parameter);
    void Execute(object parameter);
}
```

Добавление команды `ChangeColorCommand`

Мы будем заменять обработчики событий для элементов управления `Button` командами, начиная с кнопки `Change Color`. Создадим в проекте новую папку по имени `Cmds`. Добавим в нее новый файл класса `ChangeColorCommand.cs`. Сделаем класс открытым и реализующим интерфейс `ICommand`. Добавим приведенные ниже операторы `using` (первый может варьироваться в зависимости от того, создавался ли новый проект для данного примера):

```
using WpfCommands.Models;
using System.Windows.Input;
```

Код класса должен выглядеть примерно так:

```
public class ChangeColorCommand : ICommand
{
    public bool CanExecute(object parameter)
```

```

{
    throw new NotImplementedException();
}
public void Execute(object parameter)
{
    throw new NotImplementedException();
}
}
public event EventHandler CanExecuteChanged;
}

```

Если метод `CanExecute()` возвращает `true`, то привязанные элементы управления будут включенными, а если `false`, тогда они будут отключенными. Если элемент управления включен (`CanExecute()` возвращает `true`) и на нем совершается щелчок, то запустится метод `Execute()`. Параметры, передаваемые обоим методам, поступают из пользовательского интерфейса и основаны на свойстве `CommandParameter`, устанавливаемом в конструкциях привязки. Событие `CanExecuteChanged` предусмотрено в системе привязки и уведомлений для информирования пользовательского интерфейса о том, что результат, возвращаемый методом `CanExecute()`, изменился (почти как событие `PropertyChanged`).

В текущем примере кнопка `Change Color` должна работать, только если параметр отличается от `null` и принадлежит типу `Inventory`. Модифицируем метод `CanExecute()` следующим образом:

```
public bool CanExecute(object parameter) => (parameter as Inventory) != null;
```

Значение параметра для метода `Execute()` — то же, что и для метода `CanExecute()`. Поскольку метод `Execute()` может выполняться лишь в случае, если `object` имеет тип `Inventory`, аргумент потребуются привести к типу `Inventory` и затем обновить значение цвета:

```

public void Execute(object parameter)
{
    ((Inventory)parameter).Color="Pink";
}

```

Присоединение команды к `CommandManager`

Финальное обновление класса команды связано с присоединением команды к диспетчеру команд (`CommandManager`). Метод `CanExecute()` запускается при загрузке окна в первый раз и затем в ситуации, когда диспетчер команд инструктирует его о необходимости перезапуска. Каждый класс команды обязан присоединиться к диспетчеру команд, для чего мы модифицируем код, относящийся к событию `CanExecuteChanged`:

```

public event EventHandler CanExecuteChanged
{
    add => CommandManager.RequerySuggested += value;
    remove => CommandManager.RequerySuggested -= value;
}

```

Изменение файла `MainWindow.xaml.cs`

Следующее изменение связано с созданием экземпляра класса `ChangeColorCommand`, к которому может иметь доступ элемент управления `Button`. В настоящий момент мы сделаем это в файле отделенного кода для `MainWindow` (позже в главе код будет перемещен в модель представления). Откроем файл `MainWindow.xaml.cs` и удалим обработчик события `Click` для кнопки `Change Color`. Поместим в начало файла следующие операторы `using` (пространство имен может варьироваться в зависимости от того, работаете вы с предыдущим проектом или начали новый):


```
using WpfCommands.Cmds;
using System.Windows.Input;
```

Добавим открытое свойство по имени `ChangeColorCmd` типа `ICommand` с поддерживающим полем. В теле выражения для свойства возвратим значение поддерживающего поля (создавая экземпляр `ChangeColorCommand`, если поддерживающее поле равно `null`):

```
private ICommand _changeColorCommand = null;
public ICommand ChangeColorCmd =>
    _changeColorCommand ?? (_changeColorCommand = new ChangeColorCommand());
```

Изменение файла `MainWindow.xaml`

Как было показано в главе 25, реагирующие на щелчки элементы управления WPF (вроде `Button`) имеют свойство `Command`, которое позволяет назначать элементу управления объект команды. Для начала присоединим объект команды, созданный в файле отделенного кода, к кнопке `btnChangeColor`. Поскольку свойство для команды находится в классе `MainWindow`, с помощью синтаксиса привязки `RelativeSource` получается окно, содержащее нужную кнопку:

```
Command="{Binding Path=ChangeColorCmd,
    RelativeSource={RelativeSource Mode=FindAncestor,
    AncestorType={x:Type Window}}}"
```

Кнопка также нуждается в передаче объекта `Inventory` в качестве параметра для методов `CanExecute()` и `Execute()`. Такой объект назначается через свойство `CommandParameter`. Установим его в свойство `SelectedItem` элемента `ComboBox` по имени `cboCars`:

```
CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"
```

Вот завершенная разметка для кнопки:

```
<Button x:Name="btnChangeColor" Content="Change Color" Margin="5,0,5,0"
    Padding="4, 2" Command="{Binding Path=ChangeColorCmd,
    RelativeSource={RelativeSource Mode=FindAncestor, AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"/>
```

Тестирование приложения

Запустим приложение. Кнопка `Change Color` не будет включенной (рис. 28.8), т.к. автомобиль еще не выбран.

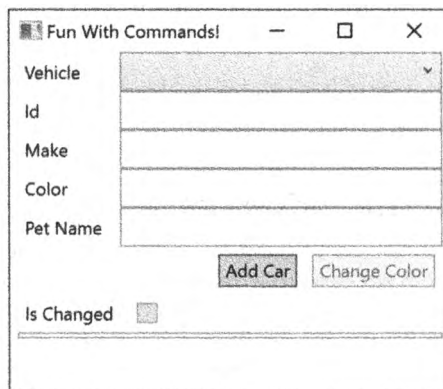


Рис. 28.8. Окно, где ничего не выбрано

Теперь выберем автомобиль; кнопка Change Color становится включенной и щелчок на ней изменит цвет, как ожидалось!

Создание класса *CommandBase*

Если распространить такой паттерн на AddCarCommand.cs, то появится код, повторяющийся внутри классов. Это хороший знак о том, что может помочь базовый класс. Создадим внутри папки Cmds новый файл класса по имени CommandBase.cs и добавим оператор using для пространства имен System.Windows.Input. Сделаем класс CommandBase открытым и реализующим интерфейс ICommand. Превратим класс и методы Execute() и CanExecute() в абстрактные. Наконец, добавим обновление в событие CanExecuteChanged из класса ChangeColorCommand. Ниже показана полная реализация:

```
using System;
using System.Windows.Input;

namespace WpfCommands.Cmds
{
    public abstract class CommandBase : ICommand
    {
        public abstract bool CanExecute(object parameter);
        public abstract void Execute(object parameter);
        public event EventHandler CanExecuteChanged
        {
            add => CommandManager.RequerySuggested += value;
            remove => CommandManager.RequerySuggested -= value;
        }
    }
}
```

Добавление команды *AddCarCommand*

Добавим в папку Cmds новый файл класса по имени AddCarCommand.cs. Сделаем класс открытым и укажем CommandBase в качестве базового класса. Поместим в начало файла следующие операторы using:

```
using System.Collections.ObjectModel;
using System.Linq;
using WpfCommands.Models;
```

Ожидается, что параметр должен относиться к типу ObservableCollection<Inventory>, а потому предусмотрим в методе CanExecute() соответствующую проверку. Если параметр имеет тип ObservableCollection<Inventory>, тогда метод Execute() должен добавить дополнительный объект автомобиля подобно обработчику события Click.

```
public class AddCarCommand : CommandBase
{
    public override bool CanExecute(object parameter)
    {
        return parameter != null && parameter is ObservableCollection<Inventory>;
    }
    public override void Execute(object parameter)
    {
        if (parameter is ObservableCollection<Inventory> cars)
        {
            var maxCount = cars?.Max(x => x.CarId) ?? 0;
            cars?.Add(new Inventory { CarId = ++maxCount, Color = "Yellow", Make = "VW",
```

```

        PetName = "Birdie" });
    }
}

```

Изменение файла *MainWindow.xaml.cs*

Добавим открытое свойство типа ICommand по имени AddCarCmd с поддерживающим полем. В теле выражения для свойства возвратим значение поддерживающего поля (создавая экземпляр AddCarCommand, если поддерживающее поле равно null):

```

private ICommand _addCarCommand = null;
public ICommand AddCarCmd => _addCarCommand ??
    (_addCarCommand = new AddCarCommand());

```

Изменение файла *MainWindow.xaml*

Модифицируем разметку XAML, чтобы удалить атрибут Click и добавить атрибуты Command и CommandParameter. Объект AddCarCommand будет получать список автомобилей из поля со списком cboCars. Ниже показана полная разметка XAML для кнопки:

```

<Button x:Name="btnAddCar" Content="Add Car" Margin="5,0,5,0" Padding="4, 2"
    Command="{Binding Path=AddCarCmd,
        RelativeSource={RelativeSource Mode=FindAncestor,
            AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars, Path=ItemsSource}"/>

```

В результате появляется возможность добавления автомобилей и обновления их цветов (пока с весьма ограниченной функциональностью) с помощью многократно используемого кода, содержащегося в автономных классах.

Изменение класса *ChangeColorCommand*

Финальным шагом будет обновление класса ChangeColorCommand, чтобы он стал унаследованным от CommandBase. Заменим интерфейс ICommand классом CommandBase, добавим к обоим методам ключевое слово override и удалим код события CanExecuteChanged. Все оказалось действительно так просто! Вот как выглядит новый код:

```

public class ChangeColorCommand : CommandBase
{
    public override bool CanExecute(object parameter) =>
        (parameter as Inventory) != null;
    public override void Execute(object parameter)
    {
        ((Inventory)parameter).Color = "Pink";
    }
}

```

Объекты RelayCommand

Еще одной реализацией паттерна "Команда" (Command) в WPF является RelayCommand. Вместо создания нового класса, представляющего каждую команду, данный паттерн применяет делегаты для реализации интерфейса ICommand. Реализация легковесна в том, что каждая команда не имеет собственного класса. Объекты RelayCommand обычно используются, когда нет необходимости в многократном применении реализации команды.

Создание базового класса *RelayCommand*

Как правило, объекты *RelayCommand* реализуются в двух классах. Базовый класс *RelayCommand* используется при отсутствии каких-либо параметров для методов *CanExecute()* и *Execute()*, а класс *RelayCommand<T>* применяется, когда требуется параметр. Начнем с базового класса *RelayCommand*, который задействует класс *CommandBase*. Добавим в папку *Cmds* новый файл класса по имени *RelayCommand.cs*. Сделаем его открытым и укажем *CommandBase* в качестве базового класса. Добавим две переменные уровня класса для хранения делегатов *Execute()* и *CanExecute()*:

```
private readonly Action _execute;
private readonly Func<bool> _canExecute;
```

Создадим три конструктора. Первый — стандартный конструктор (необходимый для производного класса *RelayCommand<T>*), второй — конструктор, который принимает параметр *Action*, и третий — конструктор, принимающий параметры *Action* и *Func*:

```
public RelayCommand() {}
public RelayCommand(Action execute) : this(execute, null) { }
public RelayCommand(Action execute, Func<bool> canExecute)
{
    _execute = execute ?? throw new ArgumentNullException(nameof(execute));
    _canExecute = canExecute;
}
```

Наконец, реализуем переопределенные версии *CanExecute()* и *Execute()*. Метод *CanExecute()* возвращает *true*, если параметр *Func* равен *null*; если же параметр *Func* не *null*, то он выполняется и возвращается *true*. Метод *Execute()* выполняет параметр *Action*.

```
public override bool CanExecute(object parameter)
    => _canExecute == null || _canExecute();
public override void Execute(object parameter) { _execute(); }
```

Создание класса *RelayCommand<T>*

Добавим в папку *Cmds* новый файл класса по имени *RelayCommandT.cs*. Класс *RelayCommandT* является почти полной копией базового класса, исключая тот факт, что все делегаты принимают параметр. Сделаем класс открытым и обобщенным, а также унаследованным от базового класса *RelayCommand*:

```
public class RelayCommand<T> : RelayCommand
```

Добавим две переменные уровня класса для хранения делегатов *Execute()* и *CanExecute()*:

```
private readonly Action<T> _execute;
private readonly Func<T, bool> _canExecute;
```

Создадим два конструктора. Первый из них принимает параметр *Action<T>*, а второй — параметры *Action<T>* и *Func<T, bool>*:

```
public RelayCommand(Action<T> execute) : this(execute, null) { }
public RelayCommand(Action<T> execute, Func<T, bool> canExecute)
{
    _execute = execute ?? throw new ArgumentNullException(nameof(execute));
    _canExecute = canExecute;
}
```

Наконец, реализуем переопределенные версии *CanExecute()* и *Execute()*.

Метод `CanExecute()` возвращает `true`, если `Func` равно `null`, а иначе выполняет `Func` и возвращает `true`. Метод `Execute()` выполняет параметр `Action`.

```
public override bool CanExecute(object parameter)
    => _canExecute == null || _canExecute((T)parameter);
public override void Execute(object parameter) { _execute((T)parameter); }
```

Изменение файла *MainWindow.xaml.cs*

Когда используются объекты `RelayCommand`, при конструировании новой команды должны указываться все методы для делегатов. Сказанное вовсе не означает, что код нуждается в помещении внутрь файла отделенного кода (как показано здесь); он просто должен быть доступным из файла отделенного кода. Код может находиться в другом классе (или даже в другой сборке), что дает преимущества инкапсуляции, связанные с созданием специального класса команды.

Добавим новую закрытую переменную типа `RelayCommand<Inventory>` и открытое свойство по имени `DeleteCarCmd`:

```
private RelayCommand<Inventory> _deleteCarCommand = null;
public RelayCommand<Inventory> DeleteCarCmd => _deleteCarCommand ??
    (_deleteCarCommand = new RelayCommand<Inventory>(DeleteCar, CanDeleteCar));
```

Также потребуется создать методы `DeleteCar()` и `CanDeleteCar()`:

```
private bool CanDeleteCar(Inventory car) => car != null;
private void DeleteCar(Inventory car)
{
    _cars.Remove(car);
}
```

Обратите внимание на строгую типизацию в методах — одно из преимуществ применения `RelayCommand<T>`.

Добавление и реализация кнопки удаления записи об автомобиле

Последний шаг заключается в добавлении кнопки `Delete Car` (Удалить автомобиль) и установке привязок `Command` и `CommandParameter`:

```
<Button x:Name="btnDeleteCar" Content="Delete Car"
    Margin="5,0,5,0" Padding="4, 2" Command="{Binding Path=DeleteCarCmd,
    RelativeSource={RelativeSource Mode=FindAncestor,
    AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"/>
```

Теперь, запустив приложение, можно удостовериться, что кнопка `Delete Car` включается, только если в раскрывающемся списке выбран автомобиль, и щелчок на ней действительно приводит к удалению записи об автомобиле из складской информации.

Итоговые сведения о командах

На этом краткий экскурс в команды WPF завершен. За счет перемещения кода обработки событий из файла отделенного кода в индивидуальные классы команд появляются преимущества инкапсуляции, многократного использования и улучшенной возможности сопровождения. Если настолько большое разделение обязанностей не требуется, тогда можно применять легковесную реализацию `RelayCommand`. Цель в том, чтобы улучшить возможность сопровождения и качество кода, так что выбирайте метод, который лучше подходит для вашей ситуации.

Перенос кода и данных в модель представления

Как и в разделе “Проверка достоверности WPF”, вы можете продолжить работу с тем же самым проектом или создать новый и скопировать в него весь код. Мы создадим новый проект по имени `WpfViewModel`. Если вы работаете с проектом из предыдущего раздела, тогда обращайте внимание на пространства имен в примерах кода и корректируйте их по мере необходимости.

Создадим в проекте новую папку под названием `ViewModels` и поместим в нее новый файл класса `MainWindowViewModel.cs`. Добавим операторы `using` для следующих пространств имен:

```
using System.Collections.ObjectModel;
using System.Windows.Input;
using WpfViewModel.Cmds;
using WpfViewModel.Models;
```

На заметку! Популярное соглашение предусматривает именование моделей представлений в соответствие с окном, которое их поддерживает. Обычно мы следуем такому соглашению и будем его соблюдать в настоящей главе. Тем не менее, как и любой паттерн или соглашение, это не норма, и на данный счет вы найдете широкий спектр мнений.

Перенос кода `MainWindow.xaml.cs`

В модель представления будет перемещен почти весь код из файла отделенного кода. В конце останется только несколько строк, включая вызов метода `InitializeComponent()` и код установки контекста данных для окна в модель представления.

Создадим открытое свойство типа `IList<Inventory>` по имени `Cars`:

```
public IList<Inventory> Cars { get; } = new ObservableCollection<Inventory>();
```

Создадим стандартный конструктор и перенесем в него весь код создания складских записей из файла `MainWindow.xaml.cs`, обновив имя списковой переменной. Можно также удалить переменную `_cars` из `MainWindow.xaml.cs`. Ниже показан конструктор модели представления:

```
public MainWindowViewModel()
{
    Cars.Add(
        new Inventory { CarId = 1, Color = "Blue", Make = "Chevy", PetName = "Kit",
                       IsChanged = false });
    Cars.Add(
        new Inventory { CarId = 2, Color = "Red", Make = "Ford",
                       PetName = "Red Rider",
                       IsChanged = false });
}
```

Далее переместим весь код, относящийся к командам, из файла отделенного кода окна в модель представления, поменяв ссылку на переменную `_cars` ссылкой на `Cars`. Вот измененный код:

```
// Для краткости остальной код не показан
private void DeleteCar(Inventory car)
{
    Cars.Remove(car);
}
```

Обновление кода и разметки MainWindow

Из файла `MainWindow.xaml.cs` кода была удалена большая часть кода. Удалим строку, которая устанавливает `ItemsSource` для поля со списком, оставив только вызов `InitializeComponent()`. Код должен выглядеть примерно так:

```
public MainWindow()
{
    InitializeComponent();
}
```

Добавим в начало файла следующий оператор `using`:

```
using WpfViewModel.ViewModels;
```

Создадим строго типизированное свойство для хранения экземпляра модели представления:

```
public MainWindowViewModel ViewModel { get; set; } =
    new MainWindowViewModel();
```

Добавим свойство `DataContext` к объявлению окна в разметке XAML:

```
DataContext="{Binding ViewModel, RelativeSource={RelativeSource Self}}"
```

Обновление разметки элементов управления

Теперь, когда свойство `DataContext` для Window установлено в модель представления, потребуется обновить привязки элементов управления в разметке XAML. Начиная с поля со списком, модифицируем разметку за счет добавления свойства `ItemsSource`:

```
<ComboBox Name="cboCars" Grid.Column="1" DisplayMemberPath="PetName"
    ItemsSource="{Binding Cars}" />
```

Прием работает, т.к. контекстом данных для Window является `MainWindowViewModel`, а `Cars` — открытое свойство модели представления. Вспомните, что конструкции привязки обходят дерево элементов до тех пор, пока не найдут контекст данных. Далее понадобится обновить привязки для элементов управления `Button`. Задача проста: поскольку привязки уже установлены на уровне окна, нужно лишь модифицировать конструкции привязки, чтобы они начинались со свойства `DataContext`:

```
<Button x:Name="btnAddCar" Content="Add Car" Margin="5,0,5,0" Padding="4, 2"
    Command="{Binding Path=DataContext.AddCarCmd,
        RelativeSource={RelativeSource Mode=FindAncestor,
            AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars, Path=ItemsSource}" />
<Button x:Name="btnDeleteCar" Content="Delete Car"
    Margin="5,0,5,0" Padding="4, 2"
    Command="{Binding Path=DataContext.DeleteCarCmd,
        RelativeSource={RelativeSource Mode=FindAncestor,
            AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}" />
<Button x:Name="btnChangeColor" Content="Change Color"
    Margin="5,0,5,0" Padding="4, 2"
    Command="{Binding Path=DataContext.ChangeColorCmd,
        RelativeSource={RelativeSource Mode=FindAncestor,
            AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}" />
```

Итоговые сведения о моделях представлений

Верите или нет, но мы только что закончили первое WPF-приложение MVVM. Вы можете подумать: “Это не настоящее приложение. Как насчет данных? Данные в примере жестко закодированы”. И вы будете совершенно правы. Это не настоящее приложение, а лишь демонстрация. Однако в ней легко оценить всю прелесть паттерна MVVM. Представлению ничего не известно о том, откуда поступают данные; оно просто привязывается к свойству модели представления. Реализации модели представления можно менять, скажем, использовать версию с жестко закодированными данными во время тестирования и версию, обращающуюся к базе данных, в производственной среде.

Можно было бы обсудить еще немало вопросов, в том числе разнообразные инфраструктуры с открытым кодом, паттерн “Локатор модели представления” (View Model Locator) и множество разных мнений на предмет того, как лучше реализовывать паттерн MVVM. В том и заключается достоинство паттернов проектирования программного обеспечения — обычно существует много правильных способов их реализации, и вам необходимо лишь отыскать стиль, который наилучшим образом подходит к имеющимся требованиям.

Изменение сборки AutoLotDAL для MVVM

В этом разделе мы собираемся подключить к приложению WPF уровень доступа к данным из главы 22. Начнем с создания нового проекта приложения WPF по имени `WpfMVVM`. Скопируем проекты `AutoLotDAL` и `AutoLotDAL.Models` из главы 22 (а не просто сошлемся на них) и вставим указанные проекты рядом с приложением WPF. Здесь будут вноситься изменения не во все модели внутри библиотеки доступа к данным, а только в класс модели `Inventory`. Внесение остальных изменений при желании расширить пример возлагается на читателя.

Нам нужно внести несколько изменений специально для WPF. Добавим в проект `WpfMVVM` ссылки на `AutoLotDAL` и `AutoLotDAL.Models`, а в проект `AutoLotDAL` — ссылку на `AutoLotDAL.Models` (иногда после копирования проектов ссылки на проекты теряются).

Обновление класса `EntityBase`

Первым делом понадобится обновить все модели кодом проверки достоверности, который создавался для поддержки `IDataErrorInfo` и `INotifyDataErrorInfo`. Переместим весь код из класса `WpfMvvm.EntityBase` в класс `AutoLotDAL.Base.EntityBase`. Кроме того, добавим интерфейсы `INotifyDataErrorInfo` и `INotifyPropertyChanged`. Добавим событие `PropertyChanged` (в текущий момент оно находится в классе `Inventory` внутри проекта приложения WPF). Наконец, добавим свойство типа `bool` по имени `IsChanged`, пометив его как `NotMapped`, чтобы инфраструктура EF не пыталась сохранять это свойство в базу данных.

```
public class EntityBase : INotifyDataErrorInfo, INotifyPropertyChanged
{
    [Key]
    public int Id { get; set; }
    [Timestamp]
    public byte[] Timestamp { get; set; }
    [NotMapped]
    public bool IsChanged { get; set; }
    public event PropertyChangedEventHandler PropertyChanged;
    // Для краткости остальной код не показан
}
```


Обновление частичного класса Inventory

Скопируем код из файла InventoryPartial.cs проекта WPF и вставим его в файл InventoryPartial.cs проекта AutoLotDAL.Models, сохранив незатронутым любой код, который уже присутствовал в целевом файле. Добавим к частичному классу интерфейс IDataErrorInfo и класс EntityBase.

```
public partial class Inventory : EntityBase, IDataErrorInfo
```

Добавление PropertyChanged.Fody в проект Models

Пакет PropertyChanged.Fody должен устанавливаться в любом проекте, которому необходимо внедрение кода NotifyPropertyChanged в классы, что в случае WPF касается всех моделей, в частности проекта AutoLotDAL.Models. С помощью диспетчера пакетов NuGet установим упомянутый пакет в проект AutoLotDAL.Models.

Вдобавок установим инфраструктуру Entity Framework в проект Models. Модифицировать файл App.config не придется, т.к. данные строки подключения будут находиться в проекте приложения WPF.

Добавление Entity Framework и строки подключения в проект приложения WPF

Установим инфраструктуру Entity Framework в проект приложения WPF и обновим файл App.config, добавив строку подключения (приведите ее в соответствие со своей средой разработки):

```
<connectionStrings>
  <add name="AutoLotConnection"
        connectionString="data source=(LocalDb)\MSSQLLocalDB;
        initial catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;
        App=EntityFramework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Обновление разметки XAML для MainWindow

В примере кода WPF в качестве первичного ключа для класса Inventory используется CarId, но в проектах AutoLotDAL первичный ключ для всех таблиц определен как Id. Откроем файл MainWindow.xaml и изменим привязку с CarID на Id:

```
<TextBox Grid.Column="1" Grid.Row="0" Text="{Binding Path=Id}" />
```

Обновление модели представления

Наступило время отбросить жестко закодированные данные и работать с уровнем доступа к данным в приложении. Откроем файл MainWindowViewModel.cs и добавим в его начало следующие операторы using:

```
using AutoLotDAL.Models;
using AutoLotDAL.Repos;
```

Модифицируем свойство Cars, как показано ниже:

```
public IList<Inventory> Cars { get; }
```

Удалим код из конструктора и добавим обращение к InventoryRepo:

```
public MainWindowViewModel()
{
    Cars = new ObservableCollection<Inventory>(new InventoryRepo().GetAll());
}
```

Обновление команды AddCarCommand

Последнее изменение, которое нужно внести в приложение WPF, связано с заменой ссылок на CarId ссылками на Id в файле AddCarCommand.cs. Обновленный код должен иметь следующий вид:

```
public override void Execute(object parameter)
{
    if (parameter is ObservableCollection<Inventory> cars)
    {
        var maxCount = cars?.Max(x => x.Id) ?? 0;
        cars?.Add(new Inventory { Id = ++maxCount, Color = "Yellow", Make = "VW",
            PetName = "Birdie" });
    }
}
```

Использование события ObjectMaterialized с инфраструктурой Entity Framework

Если запустить проект сейчас, то можно будет заметить, что каждая загруженная запись об автомобиле отображается как измененная. Причина в том, что инфраструктура EF при материализации объектов из хранилища данных обращается к блокам set для свойств. Вспомните из главы 22, что каждый раз, когда EF заканчивает воссоздание объекта из базы данных, но перед возвращением объекта вызывающему коду, инициируется событие ObjectMaterialized. Оно и будет применяться для решения проблемы.

Откроем файл AutoLotEntities.cs и внутри обработчика события OnObjectMaterialized проверим, имеет ли свойство Entity объекта ObjectMaterialized EventArgs тип EntityBase. Если имеет, то установим свойство IsChanged в false. Ниже приведен код:

```
private void OnObjectMaterialized(object sender, ObjectMaterializedEventArgs e)
{
    var model = (e.Entity as EntityBase);
    if (model != null)
    {
        model.IsChanged = false;
    }
}
```

Привяжем событие ObjectMaterialized к его обработчику в конструкторе:

```
public AutoLotEntities(): base("name=AutoLotConnection")
{
    var context = (this as IObjectContextAdapter).ObjectContext;
    context.ObjectMaterialized += OnObjectMaterialized;
}
```

Запустим приложение. После выбора записи в поле со списком можно заметить, что флажок Is Changed больше не отмечается при загрузке окна в первый раз. Если внести изменение в какое-нибудь поле, тогда флажок Is Changed становится отмеченным.

Резюме

В главе рассматривались аспекты WPF, относящиеся к поддержке паттерна MVVM. Сначала было показано, каким образом связывать классы моделей и коллекции с по-

мощью системы уведомлений в диспетчере привязки. Демонстрировалась реализация интерфейса `INotifyPropertyChanged` и применение классов наблюдаемых коллекций для обеспечения синхронизации пользовательского интерфейса и связанных с ним данных. Вы узнали, как использовать пакет `PropertyChanged.Fody`, чтобы выполнять работу по реализации `INotifyPropertyChanged` автоматически.

Вы научились добавлять код проверки достоверности к модели с применением интерфейсов `IDataErrorInfo` и `INotifyDataErrorInfo`, а также проверять наличие ошибок, основанных на аннотациях данных. Было показано, как отображать обнаруженные ошибки проверки достоверности в пользовательском интерфейсе, чтобы пользователь знал о проблеме и мог ее устранить. Вдобавок был создан стиль и специальный шаблон элементов управления для визуализации ошибок более эффективным способом.

Наконец, вы узнали, каким образом собрать все компоненты вместе за счет добавления модели представления, а также очистить разметку и отделенный код пользовательского интерфейса, чтобы усилить разделение обязанностей. В качестве примера была обновлена сборка `AutoLotDAL` для организации в ней проверки достоверности и уведомлений с обеспечением очистки объектов в обработчике события `ObjectMaterialized` по мере их материализации.

ЧАСТЬ VIII

ASP.NET

В этой части

Глава 29. Введение в ASP.NET MVC

Глава 30. Введение в ASP.NET Web API

Введение в ASP.NET MVC

В настоящей главе предлагается введение в ASP.NET MVC — инфраструктуру .NET для создания веб-приложений. Инфраструктура MVC произошла из сообщества пользователей (в частности движения ALT.NET), желающих заполучить инфраструктуру, которая бы более тесно взаимодействовала с HTTP, обеспечивала высокую тестируемость и поддерживала концепцию разделения обязанностей.

На заметку! В предыдущем издании книги подробно раскрывалась инфраструктура ASP.NET Web Forms. В текущем издании главы по Web Forms превращены в приложения и доступны в электронном виде на веб-сайте издательства.

Мы начнем главу с краткого объяснения паттерна MVC и затем перейдем непосредственно к исследованию процесса создания проекта MVC. После получения хорошего представления о паттерне MVC и генерируемых шаблонах мы построим страницы взаимодействия со складской информацией для приложения CarLotMVC.

Введение в паттерн MVC

Паттерн “модель-представление-контроллер” (Model-View-Controller — MVC) появился в 1970-х годах, будучи первоначально созданным для использования в Smalltalk. Относительно недавно его популярность возросла, в результате чего стали доступными реализации в различных языках, в том числе Java (Spring Framework), Ruby (Ruby on Rails) и многие клиентские инфраструктуры JavaScript, такие как Angular и EmberJS. Для разработчиков .NET паттерн MVC реализован в виде инфраструктуры под названием ASP.NET MVC, впервые вышедшей в 2007 году.

Модель

Модель — это данные в приложении. Данные обычно представляются с помощью простых старых объектов CLR (plain old CLR object — POCO), как те, что применялись в библиотеке доступа к данным, созданной в предшествующих двух главах. Модели представлений состоят из одной или большего числа моделей и приспособлены специально для представления, которое их использует. Воспринимайте модели и модели представлений как таблицы базы данных и представления базы данных. Они близко напоминают концепции из главы 28, где рассматривался паттерн MVVM в WPF.

С академической точки зрения модели должны быть в высшей степени чистыми и не содержать правила проверки достоверности или любые другие бизнес-правила. С практической точки зрения тот факт, содержит модель логику проверки достоверности или другие бизнес-правила, целиком зависит от применяемых языка и инфраструктур, а также специфических потребностей приложения. Например, в инфраструктуре EF Core

присутствует много аннотаций данных, которые имеют двойное назначение: механизм для формирования таблиц базы данных и средство для проверки достоверности в Core MVC. Примеры, приводимые в книге, сконцентрированы на сокращении дублированного кода, что приводит к размещению аннотаций данных и проверок достоверности там, где в них есть наибольший смысл.

Представление

Представление — это пользовательский интерфейс приложения. Представление принимает команды и визуализирует результаты команд для пользователя. Представление обязано быть как можно более легковесным и не выполнять какую-то фактическую работу; взамен оно должно передавать всю работу контроллеру. Сказанное уже наверняка знакомо вам по материалам главы 28.

Контроллер

Контроллер является своего рода мозговым центром функционирования. Обязанностей у контроллеров две; во-первых, они принимают от пользователя команды/запросы (называемые *действиями*) и упорядочивают их подходящим образом (как для хранилища), а во-вторых, они отправляют любые изменения представлению. Контроллеры (равно как модели и представления) должны быть легковесными и задействовать другие компоненты для поддержки разделения обязанностей. Все выглядит просто, не так ли? Прежде чем переходить к построению приложения MVC, обратимся к старому вопросу.

Почему появилась инфраструктура MVC?

Ко времени выпуска ASP.NET MVC в 2007 году инфраструктура ASP.NET Web Forms находилась в производственной эксплуатации уже шесть лет. Функционировали тысячи сайтов, построенных с применением Web Forms, и их число ежедневно увеличивалось. Так почему в Microsoft решили создать новую инфраструктуру с нуля? Перед тем, как ответить на этот вопрос, уместно провести краткий экскурс в прошлое.

Когда появилась инфраструктура ASP.NET Web Forms, разработка веб-приложений не была настолько продуктивной, как в наши дни. Парадигма отсутствия состояния оказалась трудной в восприятии, особенно для разработчиков интеллектуальных клиентов (занимающихся созданием настольных приложений с помощью Visual Basic 6, MFC и PowerBuilder). Чтобы заполнить пробел в знаниях и облегчить разработчикам процесс построения веб-сайтов, в Web Forms поддерживалось много концепций настольных приложений, таких как состояние (через состояние представления) и готовые элементы управления.

План сработал. Инфраструктура Web Forms в целом была воспринята хорошо, и многие разработчики сделали шаг в сторону разработки веб-приложений. В ногу с Web Forms и .NET шагала экосистема элементов управления Web Forms (и многих других элементов управления .NET) от преуспевающих независимых поставщиков. Словом, обстоятельства складывались удачно.

В то же самое время многие разработчики более глубоко изучили и освоились с концепцией отсутствия состояния, присущей программированию веб-приложений, протоколом HTTP, HTML и JavaScript. Такие разработчики все меньше и меньше нуждались в связующих технологиях и хотели иметь все больший и больший контроль над визуализируемыми представлениями.

С каждой новой версией Web Forms в инфраструктуру добавлялись дополнительные средства и возможности, делая приложения все более тяжеловесными. Увеличение

сложности разрабатываемых веб-сайтов означало разрастание элементов, таких как состояние представления, которое грозило выходом из-под контроля. Еще хуже то, что из-за ряда ранних решений, принятых при создании Web Forms (скажем, *местоположение* состояния представления внутри визуализируемой страницы), начали возникать проблемы вроде ухудшения показателей производительности. Это стало причиной заметного отказа от .NET в пользу других языков, подобных Ruby (Ruby on Rails).

Но в Microsoft не могли (и разумно не стали) удалять из ASP.NET связующие технологии и другой код, чтобы не рисковать потерей работоспособности миллионов строк кода. Что-то требовалось предпринять, и модификация Web Forms не была реалистичным вариантом, хотя в версии ASP.NET Web Forms 4.5 была проведена значительная работа для решения (или, по крайней мере, смягчения) массы проблем. В Microsoft столкнулись с необходимостью принятия ряда трудных решений: как сохранить высокую продуктивность для существующих разработчиков веб-приложений (а также поддержать экосистему элементов управления, выросшую благодаря Web Forms) и одновременно предоставить платформу разработчикам, желающим быть ближе к внутренним механизмам веб-сети.

Появление ASP.NET MVC

Благодаря всем упомянутым выше причинам появилась новая инфраструктура ASP.NET MVC. Она была создана для того, чтобы служить альтернативой ASP.NET Web Forms. Между ASP.NET Web Forms и ASP.NET MVC существуют заметные отличия, такие как устранение файлов отделенного кода для представлений, поддержки элементов управления серверной стороны и состояния представления. В дополнение к поддержке возможности тестирования, внедрения зависимостей, разделения обязанностей и многих других концепций чистого кода был создан механизм представлений Razor (начиная с версии 3 инфраструктуры ASP.NET MVC). Все проведенные улучшения пришли с одной оговоркой: использование ASP.NET MVC требовало более глубоких знаний языков HTML и JavaScript, а также того, как фактически работает протокол HTTP.

Соглашения по конфигурации

Одним из принципов ASP.NET MVC является соглашение по конфигурации (*convention over configuration*; или соглашение над конфигурацией, если делать акцент на преимуществе соглашения перед конфигурацией). Другими словами, для проектов MVC приняты специфические соглашения (такие как соглашения об именовании и структура папок), которые сокращают объем конфигурационных данных, требующихся для приложения. Хотя подобное сокращение объема ручной (или шаблонной) конфигурации обычно желательно, оно также означает необходимость знать соглашения. По мере чтения главы вы увидите многие соглашения в действии.

Шаблон приложения ASP.NET MVC

Достаточно теории. Наступило время для написания кода. Среда Visual Studio поставится с довольно полным шаблоном проекта, предназначенного для построения приложений ASP.NET MVC, и вы освоите его при создании примера приложения CarLotMVC.

Мастер создания проекта

Начнем с запуска Visual Studio и затем выберем пункт меню *File*⇒*New Project* (Файл⇒Создать проект). В древовидном представлении слева выберем узел *Web* внутри узла *Visual C#*, в панели по центру укажем *ASP.NET Web Application* (Веб-приложение ASP.NET) и введем в поле *Name* (Имя) имя *CarLotMVC* (рис. 29.1).

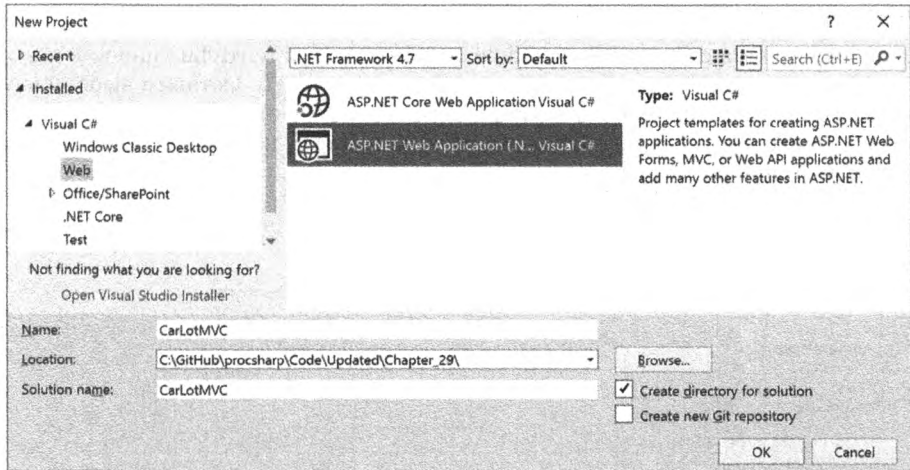


Рис. 29.1. Создание нового проекта веб-приложения ASP.NET

На появившемся далее экране мастера выберем шаблон MVC. Обратите внимание, что в области Add folders and core references for (Добавить папки и основные ссылки для) флажок MVC отмечен, а остальные флажки — нет. При желании создать гибридное приложение, которое поддерживает MVC и Web Forms, можно также отметить флажок Web Forms. В данном примере просто оставим все так, как показано на рис. 29.2. Здесь присутствует еще и флажок Add unit tests (Добавить модульные тесты). Если его отметить, то будет создан еще один проект, который предоставит базовую инфраструктуру для модульного тестирования разрабатываемого приложения ASP.NET. Из-за ограничений по объему модульное тестирование ASP.NET MVC в книге не рассматривается. Пока не будем щелкать на кнопке OK, т.к. нужно исследовать механизмы аутентификации, доступные для проекта.

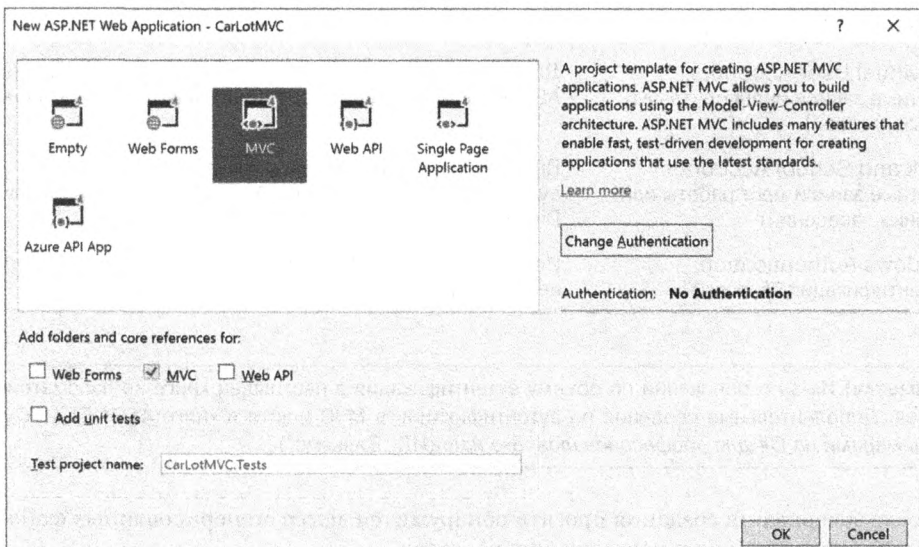


Рис. 29.2. Выбор шаблона MVC

Варианты аутентификации

Щелкнем на кнопке Change Authentication (Изменить аутентификацию), в результате чего откроется диалоговое окно, приведенное на рис. 29.3. Оставим выбранным переключатель No Authentication (Аутентификация отсутствует), который выбирается по умолчанию, щелкнем на кнопке ОК и затем на кнопке ОК на экране Select a Template мастера создания нового проекта.

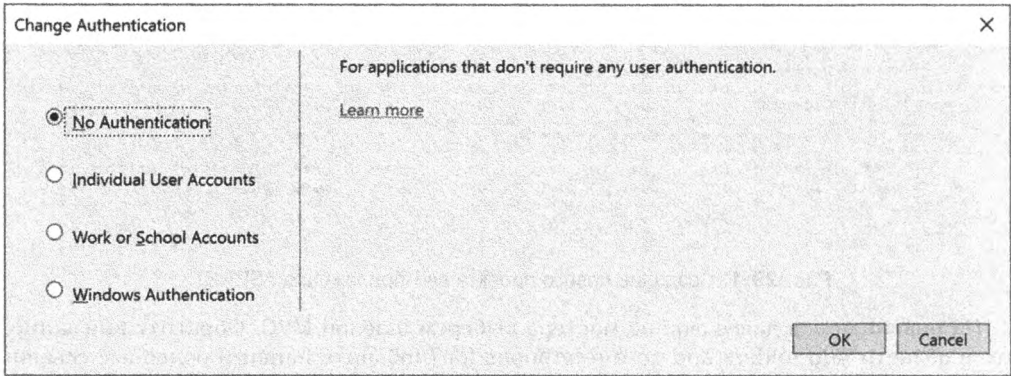


Рис. 29.3. Варианты аутентификации для проекта приложения ASP.NET MVC

В табл. 29.1 кратко описаны четыре варианта аутентификации, доступные приложениям MVC.

Таблица 29.1. Варианты аутентификации

Вариант	Описание
No Authentication (Аутентификация отсутствует)	Отсутствует механизм для входа, сущностные классы для членства или база данных членства
Individual User Accounts (Учетные записи индивидуальных пользователей)	Для аутентификации пользователей применяется система ASP.NET Identity (ранее известная как ASP.NET Membership)
Work and School Accounts (Учетные записи мест работы или учебных заведений)	Предназначен для приложений, которые производят аутентификацию с помощью Active Directory, Azure Active Directory или Office 365
Windows Authentication (Аутентификация Windows)	Использует аутентификацию Windows. Предназначен для веб-сайтов в корпоративной сети

На заметку! Из-за ограничений по объему аутентификация в настоящей книге не рассматривается. Дополнительные сведения по аутентификации в MVC ищите в книге *ASP.NET MVC 5 с примерами на C# для профессионалов*, 5-е изд. (ИД "Вильямс").

После завершения создания проекта обнаружится масса сгенерированных файлов и папок. Мы исследуем их в последующих разделах.

Окно обзора проекта

Окно обзора проекта (Overview) является заменой файла `Project_Readme.html` из предшествующих версий шаблонов ASP.NET MVC в Visual Studio. Когда проект загружается в первый раз (после создания), окно обзора проекта предоставляет ссылки на дополнительную документацию и инструктаж, связывание с подключенными службами и опубликование (рис. 29.4).

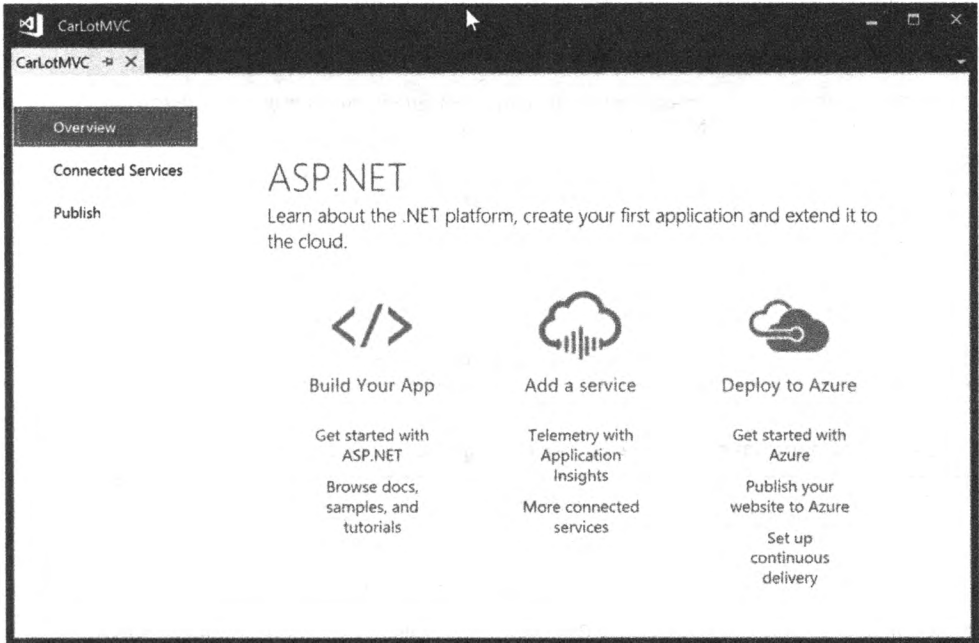


Рис. 29.4. Окно обзора проекта

Чтобы открыть окно обзора проекта снова, понадобится щелкнуть правой кнопкой мыши на имени проекта в окне Solution Explorer и выбрать в контекстном меню пункт Overview (Обзор).

Файлы в корневой папке проекта

Хотя многие файлы проектов MVC должны быть помещены в специфические местоположения, есть несколько файлов, которые находятся в корневой папке проекта. В табл. 29.2 перечислены файлы из корневой папки стандартного сайта MVC с указанием, разворачиваются они с приложением или нет.

Файл `Global.asax.cs`

Вмешательство в конвейер обработки ASP.NET производится внутри файла `Global.asax.cs`. В стандартном шаблоне проекта используется только обработчик события `Application_Start`, но существует намного больше событий, которые при необходимости можно перехватывать. Самые распространенные события приведены в табл. 29.3.

Таблица 29.2. Файлы в корневой папке проекта

Файл	Описание	Развертывается?
favicon.ico	Значок, который браузер отображает в адресной строке рядом с именем страницы. Отсутствие этого файла может привести к проблемам с производительностью, т.к. браузер будет непрерывно его искать	Да
Global.asax/ Global.asax.cs	Точка входа в приложение (подобно ASP.NET Web Forms)	Да
packages.config	Конфигурационная информация для пакетов NuGet, применяемых в проекте	Нет
ApplicationInsights. config	Используется для конфигурирования внутренних сведений о приложении (в книге не рассматривается)	Да
Startup.cs	Запускаемый класс для OWIN (Open Web Interface for .NET — открытый веб-интерфейс для .NET), применяемый системой ASP.NET Identity	Да (скомпилированный)
Web.config	Конфигурационный файл проекта	Да

Таблица 29.3. Распространенные события из файла Global.asax.cs

Событие	Описание
Application_Start	Возникает при первом обращении к приложению
Application_End	Возникает, когда приложение завершается
Application_Error	Возникает при появлении необработанной ошибки
Session_Start	Возникает при первом запросе нового сеанса
Session_End	Возникает, когда сеанс завершается (в том числе по причине тайм-аута)
Application_BeginRequest	Возникает, когда выполнен запрос к серверу
Application_EndRequest	Возникает как последнее событие в конвейерной цепочке выполнения HTTP, когда ASP.NET реагирует на запрос

Папка Models

В папке Models находятся классы моделей. В крупных приложениях классы моделей должны быть организованы в библиотеки доступа к данным. Папка Models чаще всего применяется для хранения моделей, связанных с представлениями, таких как классы моделей, которые генерируются Visual Studio для системы ASP.NET Identity.

Папка Controllers

В папке Controllers располагаются контроллеры, используемые внутри приложения. Контроллеры подробно рассматриваются позже в главе.

Папка Views

В папке Views хранятся представления MVC. Существует соглашение относительно структуры папок, содержащихся внутри папки Views. Каждый контроллер получает собственную папку внутри папки Views. Контроллер ищет свои представления в папке, имеющей такое же имя, как у класса контроллера (без слова Controller). Например, папка Views/Home содержит все представления для класса контроллера HomeController.

Внутри самой папки Views находятся файлы Web.config и _ViewStart.cshtml. Файл Web.config является специфичным для представлений в данной иерархии папок, определяет базовый тип страницы (например, System.Web.Mvc.WebViewPage) и в проектах, основанных на Razor, добавляет все ссылки на сборки и операторы using для механизма Razor.

Файл _ViewStart.cshtml выполняется перед визуализацией пользователю любых представлений. В текущий момент внутри него указывается стандартная страница компоновки для применения в ситуации, когда представлению явно не назначена страница компоновки. Страница компоновки аналогична мастер-странице в Web Forms и рассматривается позже в главе.

Папка Shared

Внутри папки Views есть специальная папка по имени Shared, которая доступна всем представлениям. Она содержит два файла: _Layout.cshtml и Error.cshtml.

В файле _Layout.cshtml находится стандартная компоновка, указанная в _ViewStart.cshtml, а в файле Error.cshtml — стандартный шаблон отображения ошибки для приложения.

На заметку! Почему имя файла _ViewStart.html (и файла _Layout.cshtml) начинается с символа подчеркивания? Механизм представлений Razor первоначально был создан для платформы WebMatrix, которая позволяла визуализировать любой файл, имя которого не начинается с подчеркивания, поэтому все основные файлы (такие как компоновка и конфигурация) имеют имена, начинающиеся с символа подчеркивания. Вы также увидите, что такое соглашение об именовании используется для частичных представлений. Тем не менее, инфраструктура MVC не следит за соблюдением данного соглашения, поскольку в ней отсутствует проблема, присущая WebMatrix, но по традиции символ подчеркивания продолжает применяться.

Папки ASP.NET

Существуют также папки, зарезервированные для ASP.NET. Примером может служить папка ASP.NET по имени App_Data, которая включена в стандартный шаблон проекта MVC. Она предназначена для хранения любых файлов данных, необходимых сайту. Также предусмотрены папки для хранения кода, ресурсов и тем.

Папки ASP.NET можно добавить, щелкнув правой кнопкой мыши на имени проекта, выбрав в контекстном меню пункт Add⇒Add ASP.NET Folder (Добавить⇒Добавить папку ASP.NET) и затем указав нужную папку в открывшемся диалоговом окне.

Папки ASP.NET не могут просматриваться из веб-сайта, даже если включена навигация по папкам. Доступные папки ASP.NET кратко описаны в табл. 29.4.

Таблица 29.4. Доступные папки ASP.NET

Папка	Описание
App_Code	Содержит файлы кода, которые будут динамически компилироваться
App_GlobalResources	Хранит файлы ресурсов, доступные всему приложению. Обычно используется для локализации
App_LocalResources	Содержит ресурсы, доступные специфичной странице. Обычно применяется для локализации
App_Data	Содержит файлы данных, используемые приложением
App_Browsers	Хранит файлы возможностей браузеров
App_Themes	Хранит темы для сайта

Папка App_Start

В ранних версиях MVC весь код конфигурации сайта (вроде маршрутизации и безопасности) содержался в классе `Global.asax.cs`. С ростом объема конфигурации разработчики инфраструктуры MVC разумно разнесли код по отдельным классам, чтобы более четко соблюдать принцип единственной обязанности. Вследствие такой переделки появилась папка `App_Start`, в которой находятся классы, перечисленные в табл. 29.5. Любой код в папке `App_Start` автоматически компилируется в результирующий сайт.

Таблица 29.5. Файлы в папке App_Start

Файл	Описание
BundleConfig.cs	Создает пакеты файлов JavaScript и CSS. В этом классе могут (и должны) создаваться дополнительные пакеты
FilterConfig.cs	Регистрирует фильтры действий (например, для аутентификации или авторизации) на глобальном уровне
IdentityConfig.cs	Содержит классы поддержки для ASP.NET Identity
RouteConfig.cs	Класс, в котором настраивается таблица маршрутизации
Startup.Auth.cs	Точка входа для конфигурации ASP.NET Identity

Файл BundleConfig.cs

Класс `BundleConfig` устанавливает настройки объединения в пакеты и минификации файлов CSS и JavaScript. По умолчанию в случае применения `ScriptBundle` все включенные файлы объединяются в пакет и подвергаются минификации (объединение в пакеты и минификация более подробно рассматриваются в следующем разделе) для версии выпуска, но такие действия не предпринимаются для отладочной версии. Управлять процессом можно посредством файла `Web.config` или в самом классе `ScriptBundle`. Чтобы отключить объединение в пакеты и минификацию, необходимо поместить показанную ниже разметку в раздел `system.web` файла `Web.config` верхнего уровня (если она еще не существует):

```
<system.web>
  <compilation debug="true" targetFramework="4.7" />
</system.web>
```

Или же нужно установить свойство `BundleTable.EnableOptimizations` в `false` внутри метода `RegisterBundles()` класса `BundleConfig`:

```
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new ScriptBundle("~/bundles/jquery").
        Include("~/Scripts/jquery-{version}.js"));
    // Для краткости остальной код не показан.
    BundleTable.EnableOptimizations = false;
}
```

Объединение в пакеты

Объединение в пакеты — это процесс комбинирования множества файлов в один. Причин объединения несколько; главная из них — увеличение скорости работы сайта. В браузерах имеется ограничение на количество параллельно загружаемых файлов из отдельно взятого сервера. Если ваш сайт содержит много мелких файлов, то пользовательский интерфейс может замедлить реагирование. Одно из решений предусматривает разнесение файлов по сетям доставки содержимого (content delivery network — CDN). Другое решение заключается в объединении файлов в пакет. Естественно, предпринимать такие действия следует сдержанно, поскольку наличие одного гигантского файла, возможно, будет ничуть не лучше огромного количества мелких файлов.

Минификация

Подобно объединению в пакеты *минификация* направлена на ускорение загрузки веб-страниц. В целях читабельности в файлах CSS и JavaScript применяются значения имен переменных и функций, комментарии и разнообразное форматирование (во всяком случае, так должно быть). Проблема в том, что при передаче по сети учитывается каждый мелкий фрагмент, особенно когда речь идет о мобильных клиентах.

Минификация представляет собой процесс замены длинных имен короткими (иногда даже односимвольными) именами, а также внесение других изменений в форматирование с целью уменьшения размера файла. Самые современные инфраструктуры поставляются с двумя версиями своих файлов CSS и JavaScript. Инфраструктура Bootstrap в данном отношении ничем не отличается, предлагая файл `bootstrap.css` для использования во время разработки приложения и файл `bootstrap.min.css` для производственной версии приложения.

Файл *FilterConfig.cs*

Фильтры — это специальные классы, которые предоставляют механизм для перехвата действий и запросов. Они могут применяться на уровне действия, контроллера или глобальном уровне. В MVC существуют четыре типа фильтров, которые кратко описаны в табл. 29.6. Из-за ограничений по объему фильтры в настоящей главе не рассматриваются.

Файл *IdentityConfig.cs*

Файлы `Identity.config.cs` и `Startup.Auth.cs` используются для поддержки системы ASP.NET Identity, которая настолько обширна, что в данной главе не рассматривается. На самом деле безопасности и удостоверениям можно было бы посвятить отдельную книгу.

Файл *RouteConfig.cs*

В ранних версиях ASP.NET Web Forms определение URL сайта осуществлялось на основе физической структуры папок проекта. Изменить это можно было с помощью модулей (`HttpModule`) и обработчиков (`HttpHandler`), но результат оказывался далеким от

идеала. Инфраструктура MVC с самого начала поддерживала маршрутизацию, которая позволяет придавать URL форму, лучше подходящую для пользователей. Мы рассмотрим маршрутизацию позже в главе.

Таблица 29.6. Фильтры в ASP.NET MVC

Тип	Описание
Фильтры авторизации	Реализуют интерфейс <code>IAuthorizationFilter</code> и запускаются перед любыми другими фильтрами. Двумя примерами служат атрибуты <code>[Authorize]</code> и <code>[AllowAnonymous]</code> . Скажем, класс <code>AccountController</code> снабжен атрибутом <code>[Authorize]</code> , чтобы требовать пользователя, аутентифицированного через ASP.NET Identity, а действие <code>Login</code> помечено атрибутом <code>[AllowAnonymous]</code> , чтобы разрешить доступ любому пользователю
Фильтры действий	Реализуют интерфейс <code>IActionFilter</code> и позволяют перехватывать выполнение действия с помощью методов <code>OnActionExecuting()</code> и <code>OnActionExecuted()</code>
Фильтры результатов	Реализуют интерфейс <code>IResultFilter</code> и позволяют перехватывать результат действия посредством методов <code>OnResultExecuting()</code> и <code>OnResultExecuted()</code>
Фильтры исключений	Реализуют интерфейс <code>IExceptionHandler</code> и выполняются при появлении необработанного исключения внутри конвейера выполнения ASP.NET. По умолчанию фильтр <code>HandleError</code> конфигурируется на глобальном уровне. Он отображает страницу представления ошибок <code>Error.cshtml</code> , расположенную в папке <code>Shared/Error</code>

Папка Content

Папка `Content` предназначена для хранения файлов CSS, изображений и другого содержимого сайта, которое будет визуализироваться напрямую. Стандартный шаблон создает специальный файл CSS по имени `Site.css`, содержащий начальные стили CSS для сайта.

Инфраструктура Bootstrap

Инфраструктура ASP.NET MVC поставляется вместе с Bootstrap — популярной инфраструктурой HTML, CSS и JavaScript с открытым кодом, которая в наши дни используется для разработки быстрореагирующих веб-сайтов, проектируемых по принципу *Mobile First* (Сначала мобильный).

В Microsoft начали включать инфраструктуру Bootstrap в версию MVC 4 и продолжили поставлять ее в версии MVC 5, так что в стандартном шаблоне проекта для MVC 5 инфраструктура Bootstrap применяется для стилизации шаблонных страниц.

На заметку! Для подробного раскрытия Bootstrap в книге не хватит места, но на веб-сайте <http://getbootstrap.ru/> доступна документация и демонстрационные примеры.

Папка Fonts

В состав Bootstrap входит набор шрифтов `GlyphIcons` `Halflings`, который будет использоваться позже в главе для улучшения пользовательского интерфейса приложения. Версия Bootstrap, с которой имеет дело шаблон проекта MVC, требует размещения шрифтов в папке `Fonts`.

Папка Scripts

В папке Scripts находятся файлы JavaScript. В табл. 29.7 перечислены файлы, входящие в состав стандартного проекта, и приведены их краткие описания.

Таблица 29.7. Файлы JavaScript в шаблоне проекта ASP.NET MVC

Файл JavaScript	Описание
bootstrap.js bootstrap.min.js	Это файлы JavaScript для Bootstrap. Файл .min представляет собой минифицированную версию
jquery-1.x.intellisense.js jquery-1.x.js jquery-1.x.min.js jquery-1.x.min.map	jQuery является инфраструктурой JavaScript для разработчиков веб-приложений. В дополнение к возможностям манипулирования моделью DOM от библиотеки jQuery зависят многие инфраструктуры, в том числе подключаемый модуль проверки достоверности, который применяется в шаблоне проекта MVC
jquery.validate-vsdoc.js jquery.validate.js jquery.validate.min.js	Подключаемый модуль jQuery Validate значительно упрощает проверку достоверности на стороне клиента. Файл -vsdoc предназначен для средства IntelliSense в Visual Studio, а файл .min — это минифицированная версия
jquery.validate.unobtrusive.js jquery.validate.unobtrusive.min.js	Подключаемый модуль Unobtrusive jQuery Validation работает с jQuery Validation, используя атрибуты HTML5 для выполнения проверки достоверности на стороне клиента
modernizr-2.x.js	Подключаемый модуль Modernizr содержит последовательность быстрых тестов для определения возможностей браузеров. Он работает напрямую с браузером, а не полагается на файлы возможностей браузеров, которые вполне могут оказаться устаревшими
respond.js respond.min.js	Respond.js — это экспериментальный подключаемый модуль jQuery для построения веб-сайтов с быстро реагирующим содержимым

Обновление пакетов NuGet проекта

Как видите, существует масса файлов и пакетов, которые входят в состав основного шаблона проекта MVC, и многие из них являются инфраструктурами с открытым кодом. Проекты с открытым кодом обновляются намного чаще, чем в Microsoft могут (или должны) выпускать обновления шаблонов Visual Studio. При создании нового проекта часто обнаруживается, что пакеты уже устарели.

К счастью, их легко обновить с помощью графического пользовательского интерфейса NuGet, что мы и будем делать повсеместно в книге. Щелкнем правой кнопкой мыши на имени проекта и выберем в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet). В открывшемся окне NuGet Package Manager (Диспетчер пакетов NuGet) перейдем на вкладку Updates (Обновления), отметим флажок Select all packages (Выбрать все пакеты) и щелкнем на кнопке Update (Обновить).

Обновление настроек проекта

По умолчанию приложения MVC конфигурируются на запуск страницы, выбранной в текущий момент внутри окна Solution Explorer. Обычно лучше всегда запускать стандартный маршрут, что требует обновления свойств проекта.

Откроем окно свойств проекта, в левой части окна перейдем на вкладку Web (Веб) и изменим настройку Start Action (Действие запуска) на Specific Page (Специфическая страница), как показано на рис. 29.5.

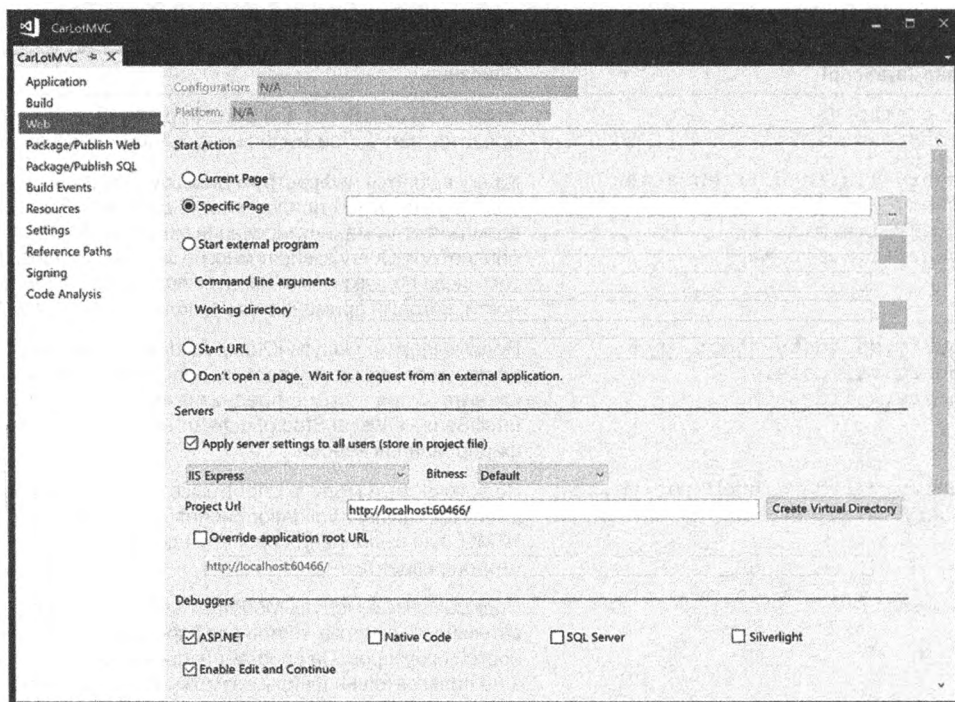


Рис. 29.5. Обновление настройки Start Action

Маршрутизация

Маршрутизация — это способ, которым MVC сопоставляет запросы URL с контроллерами и действиями в приложении, заменяющий первоначально принятый в Web Forms механизм сопоставления URL с файловой структурой. Запустим проект CarLotMVC и просмотрим URL. Он состоит из имени хоста и номера порта, например, `http://localhost:60466` (на вашей машине номер порта почти наверняка будет другим).

Теперь щелкнем на ссылке Contact (Контакт); в результате URL изменится на `http://localhost:60466/Home/Contact`. Заглянув в решение, вы не обнаружите там путь к папке Home/Contact. Маршрутизация MVC использует шаблоны URL для выяснения контроллера и метода действия, подлежащего выполнению.

Шаблоны URL

Записи маршрутизации состоят из шаблонов URL, включающих в себя переменные-заполнители и литералы, которые помещены в упорядоченную коллекцию, известную как *таблица маршрутов*. Каждая запись в ней определяет отличающийся шаблон URL, предназначенный для сопоставления. Заполнители могут быть специальными переменными или браться из заранее определенного списка.

Откроем файл `RouteConfig.cs` (из папки `App_Start`) и просмотрим его содержимое:

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index",
                           id = UrlParameter.Optional }
        );
    }
}

```

Первая строка кода указывает механизму маршрутизации на необходимость игнорирования запросов, имеющих расширение `.axd`, которое обозначает обработчик HTTP (`HttpHandler`). Метод `IgnoreRoute()` передает запрос веб-серверу, в данном случае IIS. Шаблон `{*pathInfo}` поддерживает переменное количество параметров, охватывая любой URL, который включает обработчик HTTP.

Метод `MapRoute()` добавляет запись в таблицу маршрутов. В вызове указывается имя, шаблон URL и любые стандартные значения для переменных в шаблоне URL. В предыдущем примере кода заранее определенные заполнители `{controller}` и `{action}` ссылаются на контроллер и метод действия. Заполнитель `{id}` является специальным и транслируется в параметр (по имени `id`) для метода действия.

Запрошенный URL проверяется на соответствие с таблицей маршрутов. При наличии совпадения надлежащий код выполняется. Примером URL, который мог бы обслуживаться таким маршрутом, является `Inventory/Delete/5`. В результате вызывается метод действия `Delete()` класса контроллера `InventoryController` с передачей значения 5 в параметре `id`.

В параметре `defaults` указано, каким образом заполнять пустые фрагменты в URL, которые содержат не все определенные компоненты. С учетом предыдущего кода, если в URL ничего не задано (например, `http://localhost:60466`), тогда механизм маршрутизации вызовет метод действия `Index()` класса `HomeController` без параметра `id`. Параметру `defaults` присуща поступательность, т.е. он допускает исключение справа налево. Однако пропускать части маршрута не разрешено. Ввод URL вида `http://localhost:60466/Delete/5` не пройдет сопоставление с шаблоном `{controller}/{action}/{id}`.

Важно отметить, что процесс сопоставления является последовательным и упорядоченным. Он проверяет URL на соответствие с записями коллекции в порядке их добавления. Процесс останавливается, когда найдено первое совпадение; совершенно не имеет значения, что дальше в таблице маршрутов может обнаружиться лучшее совпадение. При добавлении записей в таблицу маршрутов следует помнить об указанном факторе.

Обратите внимание, что маршрут не содержит адрес сервера (или домена). Веб-сервер IIS автоматически добавляет корректную информацию перед определенными параметрами маршрута. Например, если сайт функционирует по адресу `http://skimedic.com`, то корректный URL для маршрута мог бы выглядеть как `http://skimedic.com/Inventory/Delete/5`.

Создание маршрутов для страниц **Contact** и **About**

Одним из преимуществ маршрутизации является возможность придания URL формы, удобной для пользователей, что позволяет создавать URL, которые легко запоминать и искать с помощью поисковых механизмов.

Например, вместо `http://skimedic.com/Home/Contact` и `http://skimedic.com/Home/About` было бы лучше также иметь возможность достигать их с помощью `http://skimedic.com/Contact` и `http://skimedic.com/About` (конечно, без утраты более длинных отображений). Посредством маршрутизации добиться этого несложно.

Откроем файл `RouteConfig.cs` и поместим следующую строку кода после вызова `IgnoreRoutes()`, но *перед* установкой стандартного маршрута:

```
routes.MapRoute("Contact", "Contact",
    new { controller = "Home", action = "Contact" });
```

Такая строка добавляет в таблицу маршрутов новую запись по имени `Contact`, которая содержит только одно литеральное значение `Contact`. Она отображается на `Home/Contact`, но не со стандартными, а с жестко закодированными значениями. Чтобы протестировать ее, запустим приложение и щелкнем на ссылке `Contact`. В результате URL изменится на `http://localhost:60466/Contact` (номер порта у вас может отличаться) — именно то, что и требовалось, т.е. URL, легкий для запоминания пользователями.

А сейчас введем URL вида `http://localhost:60466/Home/Contact/Foo`. Он по-прежнему работает! Дело в том, что данный URL не прошел сопоставления с первой записью в таблице маршрутов и попал во вторую запись маршрута, которая дала совпадение. Изменим URL на `http://localhost:60466/Home/Contact/Foo/Bar`. На этот раз сопоставление завершилось неудачей, т.к. данный URL не соответствует ни одному из маршрутов. Устраним проблему добавлением `{*pathinfo}` к шаблону, что позволит указывать любое количество дополнительных параметров URL. Модифицируем запись маршрута `Contact`, как показано ниже:

```
routes.MapRoute("Contact", "Contact/{*pathinfo}",
    new { controller = "Home", action = "Contact" });
```

Теперь ввод URL вида `http://localhost:60466/Home/Contact/Foo/Bar` приведет к отображению той же самой страницы `Contact`. Миссия выполнена. Такой URL легко запоминается пользователями, и даже если они допишут к нему дополнительные ненужные данные, то все равно будут попадать на искомую страницу.

В завершение примера добавим следующую строку непосредственно после создания записи маршрута `Contact`, чтобы создать маршрут для страницы `About`:

```
routes.MapRoute("About", "About/{*pathinfo}",
    new { controller = "Home", action = "About" });
```

На заметку! В версии MVC 5 также поддерживается маршрутизация с помощью атрибутов, которая будет рассматриваться в главе 30.

Перенаправление пользователей с применением маршрутизации

Еще одно преимущество маршрутизации заключается в том, что больше не придется жестко кодировать URL для других страниц сайта. Записи маршрутов используются в двунаправленной манере, т.е. не только для сопоставления с входящими запросами, но также при построении URL для сайта. Например, откроем файл `_Layout.cshtml` из папки `Views/Shared`. Обратите внимание на приведенную далее строку:

```
@Html.ActionLink("Contact", "Contact", "Home")
```

Вспомогательный метод HTML по имени `ActionLink()` создает гиперссылку HTML с отображаемым текстом `Contact` для действия `Contact` в контроллере `Home`. Как и в отношении входящих запросов, механизм маршрутизации начинает просмотр сверху и двигается вниз до тех пор, пока не найдет совпадение. Показанная выше строка кода

соответствует добавленному ранее маршруту `Contact` и применяется для создания следующей ссылки:

```
<a href="/Contact">Contact</a>
```

Если бы маршрут `Contact` не добавлялся, то механизм маршрутизации создал бы такую ссылку:

```
<a href="/Home/Contact">Contact</a>
```

На заметку! В этом разделе введено несколько новых элементов, которые пока еще не раскрылись, в том числе синтаксис `@`, объект `Html` и файл `_Layout.cshtml`. Довольно скоро все они будут рассматриваться более подробно. Важно понимать, что таблица маршрутов используется не только для разбора входящих запросов и их передачи подходящему ресурсу на обработку, но также и для создания URL, основанных на указанных ресурсах.

Добавление библиотеки AutoLotDAL

Приложения нуждаются в данных, и `CarLotMVC` — не исключение. Начнем с копирования проектов `AutoLotDAL` и `AutoLotDAL.Models` из главы 22 в папку `CarLotMVC` (на том же уровне, что и файл решения `CarLotMVC`). Библиотека доступа к данным будет обновлена по сравнению с версией, построенной в главе 22, а потому нельзя просто добавить ссылки на готовые сборки `.dll`.

Добавим скопированные проекты в решение, щелкнув правой кнопкой мыши на решении `CarLotMVC`, выбрав в контекстном меню пункт `Add⇒Existing Project` (Добавить⇒Существующий проект) и в открывшемся диалоговом окне перейдя в нужную папку. Добавим в `CarLotMVC` ссылку на `AutoLotDAL` и `AutoLotDAL.Models`. Также проверим, что в `AutoLotDAL` имеется ссылка на `AutoLotDAL.Models`. Добавление проекта `AutoLotDAL.Models` первым должно сохранить ссылку.

Следующий шаг — добавление NuGet-пакета `Entity Framework (6.1.3)` в `CarLotMVC`. Затем мы добавим строку подключения в файл `Web.config` (в тот, что находится в корневой папке, а не тот, что внутри папки `Views`). Откроем файл `Web.config` и поместим в него такую разметку (ваши значения могут слегка отличаться):

```
<connectionStrings>
  <add name="AutoLotConnection" connectionString="data source=(localdb)\
    mssqllocaldb;initial catalog=AutoLot;
    integrated security=True;MultipleActive
    ResultSets=True;App=EntityFramework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Наконец, установим инициализатор данных при запуске приложения. Откроем файл `Global.asax.cs` и добавим в его начало следующие операторы `using`:

```
using System.Data.Entity;
using AutoLotDAL.EF;
```

Добавим в метод `Application_Start()` показанную ниже строку кода:

```
Database.SetInitializer(new MyDataInitializer());
```

В результате база данных будет удаляться и воссоздаваться каждый раз, когда приложение запускается. Такая функциональная возможность великолепно подходит для тестирования, но при развертывании производственного приложения вы должны дважды, а то и трижды проверить, удалили ли данную строку кода! Она также будет влиять на время запуска приложения, поэтому если вы заметили, что запуск требует довольно много времени, тогда прокомментируйте добавленную выше строку.

На заметку! Для применения библиотеки доступа к данным вам придется завершить изучение главы 22 по Entity Framework. По меньшей мере, понадобится получить код для проектов AutoLotDAL и AutoLotDAL.Models и выполнить миграции.

Контроллеры и действия

Как обсуждалось ранее, когда запрос поступает от браузера, он (обычно) отображается на метод действия из определенного класса контроллера. Хотя сказанное выглядит загадочно, на самом деле все довольно просто. Контроллер — это класс, который унаследован от одного из двух абстрактных классов, `Controller` и `AsyncController`. Обратите внимание, что вы также можете создать контроллер с нуля, реализовав интерфейс `IController`, но данная тема выходит за рамки настоящей книги. Метод действия — это открытый метод в классе контроллера.

Результаты действий

Действия обычно возвращают объект типа `ActionResult` (или типа `Task <ActionResult>` для асинхронных операций). Существует несколько производных от `ActionResult` типов, и некоторые часто используемые из них кратко описаны в табл. 29.8.

Таблица 29.8. Обычные классы, производные от `ActionResult`

Класс	Описание
<code>ViewResult</code> <code>PartialViewResult</code>	Возвращают в качестве веб-страницы представление (или частичное представление)
<code>RedirectResult</code> <code>RedirectToRouteResult</code>	Перенаправляют на другое действие
<code>JsonResult</code>	Возвращает клиенту сериализованный результат JSON
<code>FileResult</code>	Возвращает клиенту содержимое двоичного файла
<code>ContentResult</code>	Возвращает клиенту тип содержимого, определенный пользователем
<code>HttpStatusCodeResult</code>	Возвращает специфический код состояния HTTP

Добавление контроллера `Inventory`

Чтобы понять контроллеры и действия, лучше всего добавить новый контроллер с действиями с применением средств, встроенных в Visual Studio. Щелкнем правой кнопкой мыши на папке `Controllers` в проекте и выберем в контекстном меню пункт `Add ➞ Controller` (Добавить ➞ Контроллер), как демонстрируется на рис. 29.6.

В результате откроется диалоговое окно `Add Scaffold` (Добавление шаблона), показанное на рис. 29.7. Здесь доступно несколько вариантов, среди которых нужно выбрать `MVC 5 Controller with views, using Entity Framework` (Контроллер MVC 5 с представлениями, использующий Entity Framework).

Откроется диалоговое окно `Add Controller` (Добавление контроллера), которое позволяет указывать типы для контроллера и методов действий (рис. 29.8). Первым делом необходимо указать класс модели, где будет определен тип контроллера и методы действий. В раскрывающемся списке `Model class` (Класс модели) выберем `Inventory`.

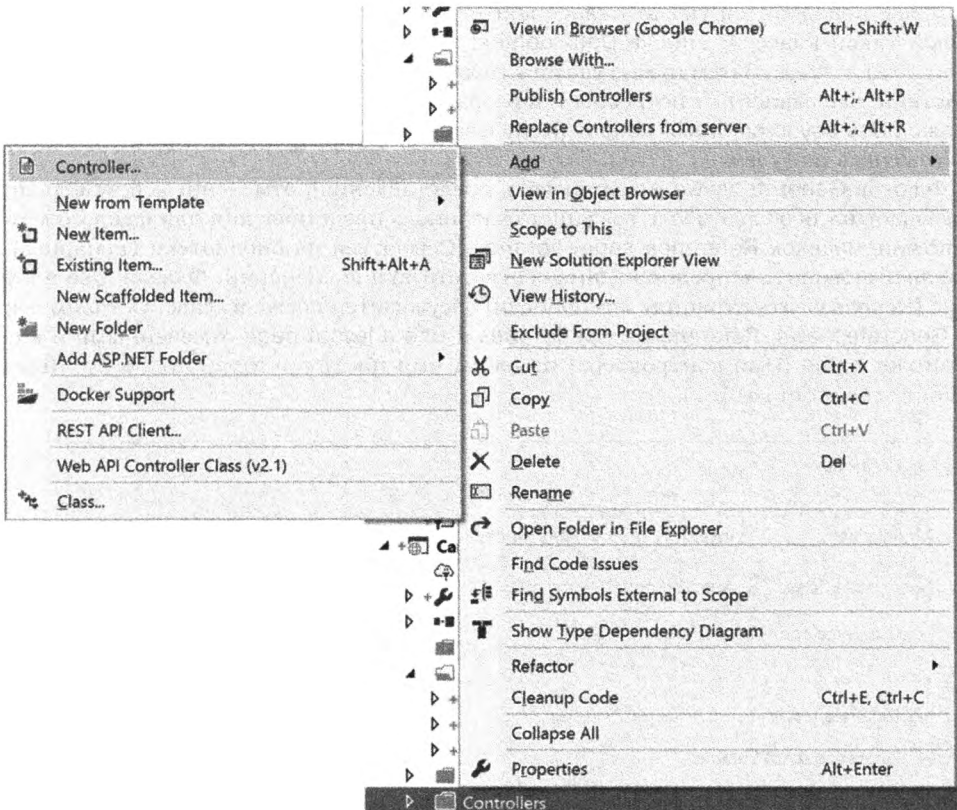


Рис. 29.6. Добавление нового контроллера

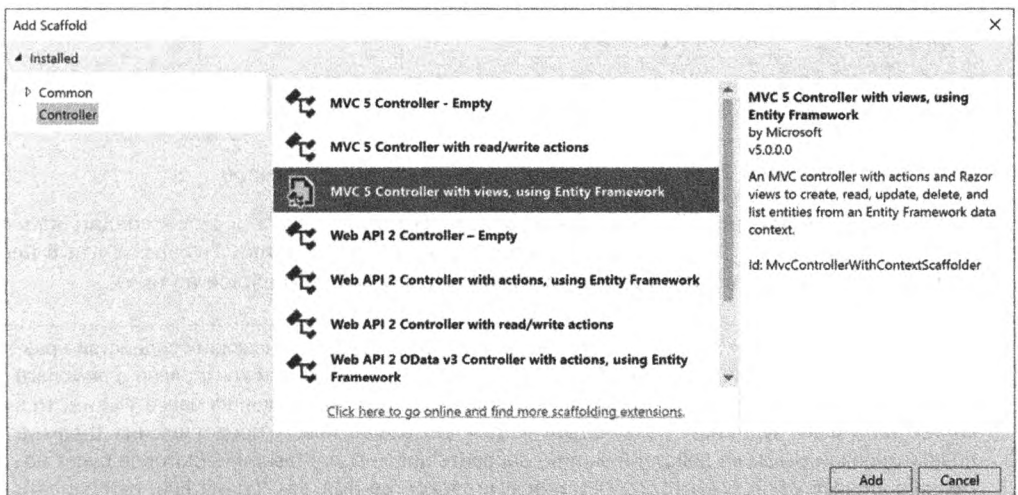


Рис. 29.7. Диалоговое окно Add Scaffold

Далее понадобится указать класс контекста. Если он не выбран, то будет создан новый такой класс. В списке Data context class (Класс контекста данных) выберем `AutoLotEntities`. Затем нужно указать, должны ли применяться асинхронные методы действий, что зависит от потребностей разрабатываемого проекта. В текущем примере флажок `Use async controller actions` (Использовать асинхронные действия контроллера) мы отмечать не будем.

Флажок `Generate views` (Генерировать представления), отмеченный по умолчанию, указывает на необходимость создания связанного представления для каждого метода действия. Флажок `Reference script libraries` (Ссылаться на библиотеки сценариев) заставляет включить в представления соответствующие сценарии. Флажок `Use a layout page` (Использовать страницу компоновки) обсуждается позже в главе. Оставим флажки `Generate views`, `Reference script libraries` и `Use a layout page` отмеченными и в поле `Controller name` (Имя контроллера) изменим имя на `InventoryController` (вместо `InventoriesController`).

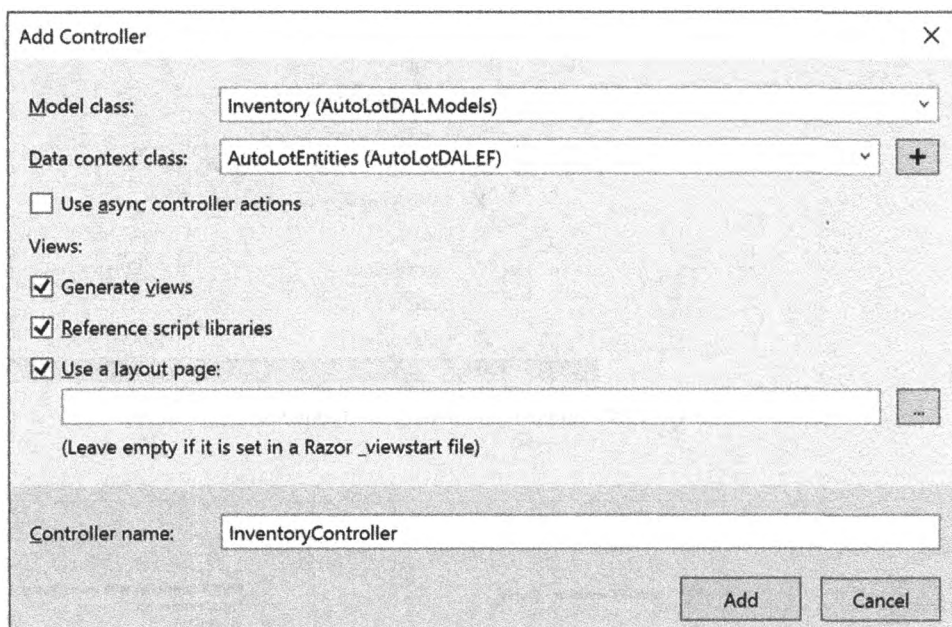


Рис. 29.8. Выбор модели, контекста и других аспектов

В результате было решено несколько задач. В папке `Controllers` создан класс `InventoryController`. Кроме того, в папке `Views` создана папка `Inventory`, а в нее добавлено пять представлений. Далее мы детально исследуем каждое из них.

На заметку! В среде Visual Studio доступно немало инструментов, оказывающих содействие разработке приложений MVC. Вы только что видели, как создавать новый контроллер с помощью диалогового окна `Add Controller`. Если щелкнуть правой кнопкой мыши на папке `Views`, то в контекстном меню будет присутствовать пункт для добавления нового представления. Щелкнув правой кнопкой мыши на действии, можно добавить новое представление (которое будет помещено в папку `Views/Controller` и получит такое же имя, как у действия) или перейти к подходящему представлению. Все возможности подобного рода опираются на соглашения, которые обсуждались ранее, так что если вы следуете правилам, то все будет в порядке.

Исследование и обновление InventoryController

Откроем файл `InventoryController.cs`. Обратите внимание, что по соглашению имя класса контроллера заканчивается словом `Controller`. Кроме того, класс является производным от абстрактного класса `Controller`. В нем имеется набор методов (действий), таких как `Index()`, `Edit()` и т.д. Мы исследуем их по очереди вместе с атрибутами, которыми они декорированы. Наконец, метод `Dispose()` переопределен, чтобы можно было освобождать любые дорогостоящие ресурсы (вроде экземпляра класса `Context` из EF), задействованные контроллером.

Использование хранилища InventoryRepo

В первой строке определения класса `InventoryController` создается новый экземпляр `AutoLotEntities`, как было указано при создании контроллера. Его понадобится заменить классом `InventoryRepo`. Добавим в начало файла следующие операторы `using`:

```
using AutoLotDAL.EF;
using AutoLotDAL.Models;
using AutoLotDAL.Repos;
using System.Data.Entity.Infrastructure;
```

Затем добавим в начало определения класса переменную экземпляра `InventoryRepo` и удалим переменную `AutoLotEntities`:

```
private readonly InventoryRepo _repo = new InventoryRepo();
```

В переопределенном методе `Dispose()` освободим `InventoryRepo`:

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        _repo.Dispose();
    }
    base.Dispose(disposing);
}
```

Действие Index

Действие `Index` получает все записи `Inventory` и возвращает данные представлению. Модифицируем вызов для работы с классом `InventoryRepo`:

```
// GET: Inventory
public ActionResult Index()
{
    return View(_repo.GetAll());
}
```

В коде вызывается перегруженный метод `View()` базового класса `Controller`, который возвращает новый объект `ViewResult`. Когда имя представления не указано (как в данном случае), то по соглашению представление именуется согласно методу действия и помещается в папку с именем контроллера, т.е. `Views/Inventory/Index.cshtml`. Имя представления можно изменить, передавая методу `View()` желаемое имя. Например, чтобы назначить представлению имя `Foo.cshtml`, вот как необходимо вызвать метод `View()`:

```
return View("Foo", _repo.GetAll());
```


Действие *Details*

Метод действия `Details()` возвращает все подробности для одной записи `Inventory`. При этом URL в формате `http://mysite.com/Inventory/Details/5` будет отображен на метод действия `Details()` класса `InventoryController` с параметром по имени `id`, которому передается значение 5. Обновим метод для применения в нем класса `InventoryRepo` вместо `AutoLotEntities`:

```
// GET: Inventory/Details/5
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Inventory inventory = _repo.GetOne(id);
    if (inventory == null)
    {
        return HttpNotFound();
    }
    return View(inventory);
}
```

В таком просто выглядящем методе есть пара интересных моментов. Вспомните из обсуждения маршрута, что параметр `id` является необязательным, поэтому URL вида `/Inventory/Details` будет корректно отображаться на данный метод. Однако если методу не передано значение `id`, то получить запись `Inventory` невозможно, так что метод возвращает код состояния HTTP 400 (`HttpStatusCode.BadRequest`). Запустив приложение и введя `Inventory/Details` (без части `id` в URL), вы получите экран ошибки. Подобным же образом, если запись `Inventory` найти не удалось, то метод действия возвращает код состояния HTTP 404.

В случае если с форматом URL все в порядке и запись `Inventory` найдена, тогда клиенту возвращается страница `Views/Inventory/Details.cshtml`.

Несколько слов о предотвращении дублированных отправок

Распространенная проблема веб-сайтов связана с тем, что пользователи щелкают на кнопке отправки внутри формы два раза, потенциально дублируя любое изменение, которое поддерживается страницей. Инфраструктурой MVC взят на вооружение паттерн, который значительно смягчает указанную проблему, но перед его обсуждением важно понять два метода HTTP, используемые в веб-приложениях MVC.

Сравнение `HttpPost` и `HttpGet`

В то время как инфраструктура ASP.NET Web Forms по большому счету игнорирует разницу между `HttpGet` и `HttpPost`, в MVC эти методы HTTP применяются соответственно их назначению. Протокол HTTP определяет вызов `HttpGet` как запрос данных из сервера, а вызов `HttpPost` — как отправку данных специфическому ресурсу на обработку.

В инфраструктуре MVC любое действие без атрибута HTTP (вроде `HttpPost`) будет выполняться как операция `HttpGet`. Рассмотренные ранее методы действий `Index()` и `Details()` применяют HTTP-метод POST. Чтобы указать операцию `HttpPost` (действие, при котором данные будут отправлены и потенциально обновлены), метод действия потребуется декорировать атрибутом `[HttpPost]`.

На заметку! Инфраструктура MVC не задействует HTTP-методы DELETE и PUT и придерживается только GET и POST, находясь в соответствии с функциональностью браузеров. В инфраструктуре ASP.NET Web API (глава 30) используются все четыре метода.

Паттерн “Отправка-перенаправление-получение”

Чтобы предотвратить дублирование отправок формы, в MVC применяется паттерн “Отправка-перенаправление-получение” (Post-Redirect-Get — PRG), который появился раньше MVC и широко использовался разработчиками веб-приложений. Он предусматривает перенаправление каждого успешного действия `HttpPost` на действие `HttpGet`, так что если пользователь щелкнет на кнопке отправки еще раз, то лишь обновится страница, выданная действием `HttpGet`, а повторная отправка не произойдет.

Вы увидите описанный паттерн в действиях `Create`, `Add` и `Delete`.

Действие `Create`

Действие `Create` реализует паттерн PRG с применением двух методов `Create()`: один не принимает параметров, а другой принимает в качестве параметра объект `Inventory`.

Версия `HttpGet`

Метод `Create()` без параметров обрабатывает запрос `HttpGet` и просто представляет страницу, которая позволяет пользователю указать информацию, необходимую для создания записи.

```
// GET: Inventory/Create
public ActionResult Create()
{
    return View();
}
```

Версия `HttpPost`

Перегруженная версия метода `Create()`, которая принимает в качестве параметра объект `Inventory`, снабжена двумя атрибутами уровня метода, `[HttpPost]` и `[ValidateAntiForgeryToken]`, а также одним атрибутом уровня параметра, `[Bind]`:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "Make,Color,PetName")]
    Inventory inventory)
```

Данная версия выполняется, когда пользователь щелкнул на кнопке отправки формы `Create` (при условии, что все проверки достоверности на стороне клиента прошли успешно). Все упомянутые элементы рассматриваются в последующих разделах.

Привязка модели

Привязка модели берет все пары “имя-значение” из HTML-формы (включая форму, строку запроса и т.д.) и пытается воссоздать объект указанного типа, используя рефлексию, за счет сопоставления имен свойств с именами в парах “имя-значение” и затем присваивает значения. Если одно или большее число значений присвоить не удастся (скажем, из-за проблем с преобразованием типа данных или ошибок проверки достоверности), тогда механизм привязки моделей укажет на ошибку, устанавливая свойство `ModelState.IsValid` в `false`. Если всем совпавшим свойствам успешно присвоены значения, то он устанавливает `ModelState.IsValid` в `true`.

В дополнение к свойству `IsValid` свойство `ModelState` имеет тип `ModelStateDictionary` и содержит информацию об ошибках для каждого проблемного свойства, а также информацию об ошибках уровня модели. При желании можно добавить специфическое сообщение об ошибке для свойства:

```
ModelState.AddModelError("Name", "Name is required");
```

Чтобы добавить сообщение об ошибке для целой модели, вместо имени свойства должно использоваться значение `string.Empty`:

```
ModelState.AddModelError(string.Empty, $"Unable to create record: {ex.Message}");
```

Существует явная и неявная привязка моделей. Для явной привязки модели необходимо вызвать метод `TryUpdateModel()`, передав ему экземпляр типа. Если привязка модели не удалась, то вызов `TryUpdateModel()` возвращает `false`; подобно неявной привязке моделей он также добавляет сведения о любых ошибках привязки в коллекцию `ModelState`. Например, метод `Create()` можно было бы реализовать следующим образом:

```
public ActionResult Create()
{
    var inv = new Inventory();
    if (TryUpdateModel(inv))
    {
        // Сохранить данные.
    }
}
```

В случае неявной привязки модели желаемый тип применяется в качестве параметра для метода. Механизм привязки моделей выполняет с параметром ту же самую операцию, которая делалась посредством метода `TryUpdateModel()` в предыдущем примере:

```
public ActionResult Create(Inventory inventory)
{
    if (ModelState.IsValid)
    {
        // Сохранить данные.
    }
}
```

В представлениях `Create`, `Delete` и `Edit`, если модель находится в недопустимом состоянии (`ModelState.IsValid` равно `false`), то пользователь возвращается к тому же самому представлению для повторения попытки с восстановлением всех ранее введенных данных. Когда состояние модели допустимо (`ModelState.IsValid` равно `true`), метод должен сохранить запись в базе данных и отправить пользователя на действие `Index` (метод `HttpGet`).

Маркеры противодействия подделке

Одним из средств защиты от взлома является атрибут `AntiForgeryToken`, который добавляет специальное значение формы. При поступлении запроса `HttpPost` инфраструктура MVC проверяет такое значение формы на стороне сервера (при условии, что в методе действия присутствует атрибут `[ValidateAntiForgeryToken]`). Хотя это не единственная мера защиты (тема безопасности веб-приложений выходит за рамки настоящей книги), каждая форма должна добавлять значение `AntiForgeryToken` и каждое действие `HttpPost` должно проверять его.

Атрибут [Bind]

Атрибут [Bind] в методах HttpPost позволяет помещать свойства в “белый” или “черный” список, чтобы предотвращать атаки чрезмерной отправкой данных (over-posting) пользователем. Когда поля находятся в “белом” списке, только им будут присвоены значения во время привязки модели. Помещение в “черный” список исключает свойства из процесса привязки модели. В шаблонном методе Create() все поля записи Inventory находятся в “белом” списке. Поскольку в действии Create нужно отправлять только свойства Make, Color и PetName, удалим поля Id и Timestamp из списка Include:

```
public async Task<ActionResult> Create([Bind(Include =
    "Make,Color,PetName")] Inventory inventory)
```

Обновление метода для использования InventoryRepo

Модифицируем метод Create() для использования InventoryRepo и добавим ошибку уровня модели ModelState, если при сохранении записи возникла проблема:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "Make,Color,PetName")]
    Inventory inventory)
{
    if (!ModelState.IsValid) return View(inventory);
    try
    {
        _repo.Add(inventory);
    }
    catch (Exception ex)
    {
        ModelState.AddModelError(string.Empty,
            $"{@"Unable to create record: {ex.Message}"}");
        // Не удается создать запись.
        return View(inventory);
    }
    return RedirectToAction("Index");
}
```

Действие Edit

Подобно методу действия Create() процесс редактирования применяет паттерн PRG, реализованный с использованием двух версий метода действия Edit().

ВерсияHttpGet

Первая версия метода Edit() принимает параметр id, получает запись Inventory и возвращает подходящее представление. За исключением возвращения другого представления она идентична методу Details(). Изменим код для работы с хранилищем Inventory вместо AutoLotEntities.

```
// GET: Inventory/Edit/5
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
}
```

```

Inventory inventory = _repo.GetOne(id);
if (inventory == null)
{
    return HttpNotFound();
}
return View(inventory);
}

```

Версия HttpPost

Вторая версия метода `Edit()` выполняется, когда пользователь щелкнул на кнопке отправки формы редактирования (при условии успешного прохождения всех проверок достоверности на стороне клиента). Если состояние модели не является допустимым, то метод еще раз возвращает представление `Edit`, отправляя текущие значения для объекта `Inventory`. Если состояние модели допустимо, тогда объект `Inventory` передается хранилищу для попытки его записи.

В дополнение к общей обработке ошибок (вроде применяемой в методе `Create()`) понадобится также добавить проверку на предмет исключения `DbUpdateConcurrencyException`, которое будет сгенерировано, если другой пользователь обновил запись после того, как текущий пользователь загрузил ее на веб-страницу. Если все прошло успешно, то метод действия возвращает результат вызова `RedirectToAction()`, обеспечивая перенаправление пользователя на метод действия `Index()` класса `InventoryController`.

```

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include = "Id,Make,Color,PetName,TimeStamp")]
                        Inventory inventory)
{
    if (!ModelState.IsValid) return View(inventory);
    try
    {
        _repo.Save(inventory);
    }
    catch (DbUpdateConcurrencyException ex)
    {
        ModelState.AddModelError(string.Empty,
            $"{@"Unable to save the record. Another user has updated it."}
            {ex.Message}");
        // Не удастся сохранить запись. Другой пользователь обновил ее.
        return View(inventory);
    }
    catch (Exception ex)
    {
        ModelState.AddModelError(string.Empty, $"{@"Unable to save the record."}
            {ex.Message}"); // Не удастся сохранить запись.
        return View(inventory);
    }
    return RedirectToAction("Index");
}

```

На заметку! За дополнительными сведениями об исключении `DbUpdateConcurrencyException` обращайтесь в главу 22.

Действие Delete

Метод действия `Delete()` также реализует паттерн PRG с помощью двух версий.

Версия `HttpGet`

Первая версия метода `Delete()` принимает параметр `id` и идентична версиям `HttpGet` методов `Details()` и `Edit()`. Изменим код для использования хранилища `Inventory` вместо `AutoLotEntities`:

```
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Inventory inventory = _repo.GetOne(id);
    if (inventory == null)
    {
        return HttpNotFound();
    }
    return View(inventory);
}
```

Версия `HttpPost`

Вторая версия `Delete()` выполняется, когда пользователь щелкнул на кнопке отправки формы удаления. Автоматически сгенерированная версия этого метода принимает только `id` в качестве параметра, т.е. она имеет ту же самую сигнатуру, что и версия `HttpGet` метода. Поскольку не разрешено иметь два метода с одним и тем же именем и сигнатурой, генерируемый метод именуется как `DeleteConfirmed()` и снабжается атрибутом `[ActionName("Delete")]`. Модифицируем сигнатуру метода следующим образом (можно также удалить атрибут `ActionName`, раз уж метод переименован на `Delete()`):

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete([Bind(Include="Id, Timestamp")] Inventory car)
```

Библиотека `AutoLotDAL` проверяет наличие конфликтов параллелизма и для удаления записи помимо свойства `Id` требует указания свойства `Timestamp`. Именно потому параметр `int id` изменяется на `Inventory inventory`. Атрибут `[Bind]` со строкой `"Id, Timestamp"` в `Include` обеспечит помещение в экземпляр `Inventory` только указанных значений и игнорирование остальных свойств.

Наконец, обновим метод для взаимодействия с хранилищем `Inventory` и добавим обработку ошибок (как делалось в версии `HttpPost` метода `Edit()`). Вот финальная версия кода:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete([Bind(Include="Id, Timestamp")] Inventory inventory)
{
    try
    {
        _repo.Delete(inventory);
    }
    catch (DbUpdateConcurrencyException ex)
    {
    }
```

```

        ModelState.AddModelError(string.Empty, $"Unable to delete record.
        Another user updated the record. {ex.Message}");
        // Не удастся удалить запись. Другой пользователь обновил ее.
    }
    catch (Exception ex)
    {
        ModelState.AddModelError(string.Empty, $"Unable to delete record:
        {ex.Message}"); // Не удастся удалить запись.
    }
    return RedirectToAction("Index");
}

```

Если запустить проект прямо сейчас и попробовать удалить запись `Inventory`, то выяснится, что удаление не выполняется, т.к. представление не отправляет свойство `Timestamp`, а только `Id`. Проблема будет устранена в последующих разделах.

Итоговые сведения о контроллерах и действиях

Мы привели довольно большой объем информации, но лишь слегка коснулись поверхности того, что можно делать в контроллерах и методах действий MVC. Подводя итоги, контроллеры — это просто классы C#, которые должны следовать соглашению об именовании вида `<Имя>Controller`. Действия — это открытые методы в классе контроллера, которые возвращают объект типа `ActionResult`. Методы действий могут быть декорированы атрибутом, который указывает, является метод `HttpPost` или `HttpGet` (по умолчанию), а все методы `HttpPost` должны применять маркер `ValidateAntiForgeryToken`.

Механизм представлений Razor

Прежде чем раскрывать детали представлений, важно освоить механизм представлений Razor и научиться добавлять в представления код серверной стороны, используя Razor. Механизм представлений Razor был создан как усовершенствование механизма представлений Web Forms и применяет Razor в качестве основного языка. Представления Razor не расширяют `System.Web.Page` (никакой директивы `Page` не предусмотрено), из-за чего появляется возможность тестирования.

Razor — это синтаксис разметки шаблонов, который на стороне сервера интерпретируется в код C# (или VB.NET). Как вы вскоре увидите, использование Razor в представлениях с HTML и CSS приводит к получению более чистой и простой для чтения разметки. Механизм представлений Razor поддерживает все то, что поддерживает механизм представлений Web Forms, а потому после перехода на Razor вы ничего не теряете.

Кроме того, Razor поддерживает создание методов для инкапсуляции кода в форме вспомогательных методов HTML, функций Razor и делегатов Razor. Они могут создаваться в представлении для локализованного применения или для общего использования путем их помещения в папку `App_Code` либо создания с указанием ключевого слова `static`.

На заметку! Хотя механизм представлений Web Forms все еще поддерживается в ASP.NET MVC 5, в версии ASP.NET Core его поддержка отсутствует. Если до сих пор вы сопротивлялись переходу на Razor, то самое время изменить свое мнение.

Синтаксис Razor

Первое отличие между механизмами представлений Web Forms и Razor заключается в том, что код вводится с символом `@`. Интеллектуальные возможности Razor устраняют потребность в добавлении закрывающих символов `@`, тогда как механизм представлений Web Forms требует открывающего и закрывающего дескрипторов (`<% %>`).

Операторы, блоки кода и разметка

Однострочные операторы снабжаются префиксом в виде символа @ без какого-либо закрывающего символа, например:

```
@ShowInventory(item)
```

Блоки операторов открываются символом @ и помещаются в фигурные скобки, как показано ниже:

```
@foreach (var item in Model)
{
}
// Или:
@{
    ShowInventory(item);
}
```

Блоки кода могут содержать смесь разметки и кода. Строки, которые начинаются с дескриптора разметки, интерпретируются как разметка HTML, а строки, начинающиеся с кода, трактуются как код:

```
@foreach (var item in Model)
{
    int x = 0;
    <tr></tr>
}
```

Символ @ перед именем переменной эквивалентен вызову `Response.Write()` и по умолчанию все значения в разметке HTML закодированы. Если вы хотите выводить незакодированные данные (т.е. потенциально небезопасные), тогда должны применять синтаксис `@Html.Raw(username)`. Код Razor и разметка могут использоваться вместе:

```
<h1>Hello, @username</h1>
```

Дескрипторы @: и <text> обозначают текст, который должен быть визуализирован как часть разметки:

```
@:Straight Text
<div>Value:@i</div>
<text>
Lines without HTML tag
</text>
```

Дескрипторы @* и *@ указывают многострочный комментарий:

```
@* Комментарий, который мог быть разнесен по нескольким строкам *@
```

Механизм представлений достаточно интеллектуален в различении ситуаций, когда символ @ должен трактоваться как просто знак, а когда он обозначает оператор. Великолепным примером могут служить адреса электронной почты. Если механизм Razor встречает адрес электронной почты, то он не интерпретирует его как код. Если код *выглядит* как адрес электронной почты, то указание круглых скобок после @ информирует Razor о необходимости трактовки строки как кода. Вот пример:

```
someone@foo.com @* адрес электронной почты *@
someone@(foo).com @* foo интерпретируется как код Razor
и переменная foo заменяется своим значением *@
```

Чтобы отменить знак @, его нужно продублировать, как показано в следующем коде, который помещает @foo в представление:

```
@@foo
```


Встроенные вспомогательные методы HTML

Доступно множество встроенных вспомогательных методов HTML, самые распространенные из которых кратко описаны в табл. 29.9. В оставшихся разделах главы вы увидите многие такие методы в действии.

Таблица 29.9. Распространенные вспомогательные методы HTML

Вспомогательные методы HTML	Описание
ActionLink	Создает дескриптор якоря
TextBox[For]	Каждый из них создает одноименный элемент управления HTML-формы для свойства модели. Версии For являются строго типизированными
TextArea[For]	
CheckBox[For]	
RadioButton[For]	
DropDownList[For]	
ListBox[For]	
Hidden[For]	
Password[For]	Визуализирует свойство как допускающее только чтение на основе шаблона. Если специальный шаблон не обнаружен, то используется стандартный шаблон (основанный на типе данных). Версия For является строго типизированной
Label[For]	
Display[For]	Визуализирует свойство как элементы управления формы на основе шаблона. Если специальный шаблон не обнаружен, то применяется стандартный шаблон (основанный на типе данных). Версия For является строго типизированной
Editor[For]	
DisplayForModel	Визуализируют разметку отображения или редактирования для целой модели. Используют специальный шаблон, когда он существует; иначе применяют стандартный шаблон
EditotForModel	
DisplayName[For]	Получает отображаемое имя, если оно назначено; в противном случае отображает имя свойства. Версия For является строго типизированной
ValidationSummary	Создает сводку по проверке достоверности или сообщение, когда возникает ошибка при проверке достоверности клиентской или серверной стороны
ValidationMessageFor	
@Html.BeginForm()	Создает дескриптор HTML-формы, содержащий все, что указано в скобках
@Html.AntiForgeryToken()	Работает в сочетании с атрибутом ValidateAntiForgeryToken методов действий

Шаблонные вспомогательные методы полагаются на стандартные шаблоны, если только в корректном местоположении не были созданы специальные шаблоны. Вскоре вы узнаете, как создавать специальные шаблоны.

Шаблонные вспомогательные методы HTML для редактирования

В то время как вспомогательные методы для отображения всего лишь выводят данные, вспомогательные методы для редактирования делают чуть больше, чем просто создание соответствующего элемента управления. Например, вспомогательный метод HTML по имени EditorFor() создает поле ввода на основе типа данных свойства, указанного в лямбда-выражении. Скажем, следующая строка:

```
@Html.EditorFor(model =>
    model.Make, new { htmlAttributes = new { @class = "form-control" } })
```

создает такую разметку:

```
<input name="Make" class="form-control text-box single-line"
    id="Make" type="text" value=""
    data-val-length-max="50"
    data-val-length="The field Make must be a string with a maximum length of 50."
    data-val="true">
```

Прежде чем двигаться дальше, давайте исследуем результирующую разметку. Значения атрибутов `name` и `id` элемента HTML поступают из свойства `name`, значение `type` — из типа данных свойства, а значение `class` — из комбинации вспомогательного метода HTML и дополнительных атрибутов HTML, добавленных через вспомогательный метод.

Атрибут `value` элемента управления устанавливается в значение свойства. В рассматриваемом случае значение получает пустую строку, поскольку это новый экземпляр `Inventory`. Код проверки достоверности генерируется аннотациями данных и будет более подробно обсуждаться далее в главе.

Корректировка разметки, визуализируемой вспомогательными методами HTML

Сами по себе вспомогательные методы HTML будут визуализировать простую разметку. Если вы хотите добавить стилизацию или другие атрибуты, тогда должны использовать одну из перегруженных версий. Звучит просто, но синтаксис, мягко говоря, несколько громоздок.

Вам придется создать новый анонимный объект с парами “имя-значение” разметки HTML, которую требуется визуализировать вместе с базовым выводом. Если атрибут HTML представляет собой зарезервированное слово в C#, то он должен быть отменен с помощью символа `@`. Прием выглядит ошибочным, т.к. символ `@` применяется для пометки элемента как кода Razor!

Например, при желании установить класс CSS для элемента метки использовался бы следующий код:

```
@Html.LabelFor(model => model.Color,
    htmlAttributes: new { @class = "control-label col-md-2" })
```

В результате разметка для метки будет обновлена, чтобы включать атрибут класса CSS.

Специальные вспомогательные методы HTML

Вспомогательные методы HTML в Razor визуализируют разметку. Существует множество встроенных вспомогательных методов, которые вы будете интенсивно применять, такие как `@Html.ActionLink()`, который создает дескриптор якоря. В дополнение к встроенным вспомогательным методам вы можете также строить специальные вспомогательные методы HTML с целью сокращения (или вообще устранения) повторяющегося кода.

Скажем, может понадобиться вспомогательный метод, который выводит детали записи `Inventory`. Откроем представление `Index.cshtml` в папке `Views\Inventory` и поместим в начало файла показанный ниже код вспомогательного метода HTML (после строки `@model`):

```
@using AutoLotDAL.Models
@helper ShowInventory(Inventory item)
{
    @item.Make<text></text>@item.Color<text></text>@item.PetName<text>
    </text>
}
```

Внутри @foreach добавим вызов ShowInventory():

```
@foreach (var item in Model)
{
    @ShowInventory(item)
    <!-- Для краткости остальной код не показан -->
}
```

Запустим приложение и перейдем на страницу Inventory/Index; появятся детали каждой записи в виде единственной строки. В реальном вспомогательном методе HTML было бы предусмотрено форматирование и согласование разметки с внешним видом всего сайта. Поскольку это просто демонстрация создания вспомогательного метода HTML, которая для сайта не требуется, закомментируем строку:

```
@*@ShowInventory(item)*@
```

Функции Razor

Функции Razor не возвращают разметку, а взамен служат для инкапсуляции кода с целью многократного использования. Чтобы увидеть их в действии, поместим приведенную далее функцию SortCars() после вспомогательного метода HTML внутри страницы представления Index.cshtml. Функция SortCars() принимает список элементов Inventory и сортирует их по PetName:

```
@functions
{
    public IList<Inventory> SortCars(IList<Inventory> cars)
    {
        var list = from s in cars orderby s.PetName select s;
        return list.ToList();
    }
}
```

Модифицируем оператор @foreach для вызова функции SortCars(). Переменная Model представляет реализацию IEnumerable<Inventory>, так что придется добавить еще и вызов метода ToList():

```
@foreach (var item in SortCars(Model.ToList()))
{
    <!-- Для краткости остальной код не показан. -->
}
```

Делегаты Razor

Делегаты Razor работают точно так же, как делегаты C#. Для примера определим следующий код делегата сразу после функции SortCars() в файле представления Index.cshtml. Этот делегат делает символы полужирными.

```
@{
    Func<dynamic, object> b = @<strong>@item</strong>;
}
```

Чтобы посмотреть на него в действии, добавим непосредственно после блока кода, определяющего делегат, такую строку:

```
This will be bold: @b("Foo")
```

Разумеется, пример тривиален, но в случае более сложного повторяющегося кода можно получить выигрыш от его помещения внутрь делегата. По существу здесь применимы все доводы “за” и “против”, связанные с делегатами C#. После запуска приложения и перехода на страницу `Inventory/Index` можно заметить, что слово `Foo` выделено полужирным. Теперь прокомментируем обращение к делегату, т.к. в оставшихся примерах в нем нет необходимости.

Представления MVC

Представления в инфраструктуре MVC изображают пользовательский интерфейс на сайтах MVC. Представления MVC задуманы быть очень легковесными, передавая обработку серверной стороны контроллерам, а обработку клиентской стороны — сценариям JavaScript. Представления компоновки похожи на мастер-страницы в Web Forms; представления визуализируются внутри компоновки и только частичные представления визуализируют себя сами (т.е. без компоновки).

Компоновки

Перейдем в папку `Views/Shared` и откроем файл `_Layout.cshtml`. Он является полноценным файлом HTML, укомплектованным дескрипторами `<head>` и `<body>`, а также смесью разметки HTML и вспомогательных методов HTML из Razor. Подобно мастер-страницам Web Forms страница `_Layout.cshtml` — это основа того, что будет отображаться пользователю при визуализации представлений (использующих страницу `_Layout.cshtml`).

При работе с компоновками важно помнить о существовании двух элементов: тело и области. В тело будет вставляться код представления, когда представление и компоновка визуализируются и позиционируются в компоновке с помощью следующей строки кода Razor:

```
@RenderBody()
```

Области — это разделы страницы компоновки, которые допускают заполнение во время выполнения. Они могут быть обязательными или необязательными и определяются внутри страницы компоновки посредством функции `RenderSection()` из Razor. В первом аргументе указывается имя области, а во втором — считается ли область обязательной. Приведенная ниже строка кода в `_Layout.cshtml` создает область по имени `scripts`, необязательную для представления:

```
@RenderSection("scripts", required: false)
```

Если нужно создать новую обязательную область по имени `Header`, то должен применяться такой код:

```
@RenderSection("Header", required: true)
```

Для визуализации области в представлении используется блок Razor под названием `@section`. Например, следующие блоки кода внутри страницы `Edit.cshtml`, находящейся в папке `Views/Inventory`, добавляют к визуализируемой странице пакет проверки достоверности `jQuery`:

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Выбор страницы компоновки

Представления MVC могут основываться на главной компоновке, чтобы придавать сайту унифицированный внешний вид и поведение. Представления могут явно объявлять представление компоновки, в котором они будут визуализироваться, или могут применять стандартное представление компоновки.

Стандартное представление компоновки объявлено в файле `_ViewStart.cshtml`. Файл `_ViewStart.cshtml` содержит один блок кода Razor, который устанавливает для компоновки специфический файл. Откроем данный файл и посмотрим его содержимое:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Установленная здесь компоновка затем будет назначаться любому представлению, находящемуся на уровне файла `_ViewStart.cshtml` или ниже, если оно само не указывает какую-то компоновку (за исключением частичных представлений, рассматриваемых позже).

Использование специфической страницы компоновки

В дополнение к применению стандартной страницы компоновки можно указывать, что представления должны использовать специфическую страницу, добавляя строку `Layout` в начало представления. Пусть имеется компоновка по имени `_LayoutNew.cshtml`. Тогда вот как можно было бы модифицировать представление:

```
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_LayoutNew.cshtml";
}
```

Обновление пунктов меню в компоновке

Представления компоновки обычно содержат средство навигации по сайту и разрабатываемому нами сайту оно также не помешает. Отыщем в файле `_Layout.cshtml` строку, содержащую `@Html.ActionLink("Home", "Index", "Home")`. Нам необходимо добавить меню для новых страниц `Inventory`, поэтому поместим после найденной строки следующий код:

```
<li>@Html.ActionLink("Inventory", "Index", "Inventory")</li>
```

Частичные представления

Частичные представления являются такими же, как обычные представления, исключая тот факт, что они не применяют представление компоновки в качестве своей основы и полезны для инкапсуляции пользовательского интерфейса. Отличие между частичным представлением и полным представлением связано с тем, как они визуализируются. Полное представление (возвращаемое из контроллера с помощью метода `View()`) будет использовать страницу компоновки, указанную либо в качестве стандартной в файле `_ViewStart.cshtml`, либо заданную через оператор `Layout` в коде Razor. Представление, визуализируемое с применением метода `PartialView()` (или вспомогательного метода HTML по имени `Partial()`), не использует стандартную компоновку, но все-таки задействует компоновку, если она указана посредством оператора `Layout` в коде Razor.

Чтобы продемонстрировать сказанное, откроем файл `InventoryController.cs` и изменим метод действия `Index()` для возвращения частичного представления вместо полного:

```
public ActionResult Index()
{
    return PartialView(_repo.GetAll());
}
```

Запустим приложение и щелкнем на ссылке `Inventory` в навигационном меню. Отобразятся те же данные, что и ранее, но без какой-либо компоновки.

Изменим метод действия `Index()`, чтобы он снова вызывал метод `View()`, а не `PartialView()`.

Вдобавок к визуализации представления из метода действия с помощью `PartialView()` частичное представление можно помещать внутрь другого представления, используя вспомогательный метод `HTML`. Вот как можно было бы поместить частично представление по имени `MyPartial.cshtml` в другое представление:

```
@Html.Partial("MyPartial")
```

Отправка данных представлениям

При обсуждении класса `InventoryController` вы узнали, что методы действий способны возвращать данные представлению. Это делалось передачей объекта (или списка объектов) в метод `View()`. Данные, переданные в метод `View()`, становились моделью для представления.

Строго типизированные представления и модели представлений

Представления в MVC обычно строго типизированы в отношении используемых ими данных. Ожидаемый тип данных определяется с помощью оператора `@model` в представлении. Например, представление `Index.cshtml` применяет в качестве своего источника данных тип `IEnumerable<Inventory>`:

```
@model IEnumerable<AutoLotDAL.Models.Inventory>
```

Для доступа к таким данным в оставшейся части представления используется ключевое слово `Model`. Обратите внимание, что в ключевом слове `Model` буква *M* записывается в верхнем регистре, а в строке `@model` применяется буква *m* в нижнем регистре. При ссылке на данные, содержащиеся в представлении, используется `Model` (с заглавной буквой *M*), как в следующем коде, который проходит по всем записям `Inventory`:

```
@foreach (var item in Model)
{
    // Делать здесь что-то интересное.
}
```

Объекты `ViewBag`, `ViewData` и `TempData`

Объекты `ViewBag`, `ViewData` и `TempData` — это механизмы для отправки небольших объемов данных представлению. Примером может служить строка в начале каждого представления `Inventory`, устанавливающая свойство `ViewBag.Title`; вот как она выглядит в представлении `Index.cshtml`:

```
@{
    ViewBag.Title = "Index";
}
```

Свойство `ViewBag.Title` применяется для отправки заголовка представления компоновке с целью использования в следующей строке внутри `_Layout.cshtml`:

```
<title>@ViewBag.Title - My ASP.NET Application</title>
```

Детали всех трех механизмов кратко описаны в табл. 29.10.

Таблица 29.10. Способы передачи данных представлению

Объект транспортировки данных	Описание
TempData	Недолговечный объект, который работает только во время текущего запроса и запроса, следующего за ним
ViewData	Словарь, позволяющий хранить данные в парах "имя-значение", например, ViewData["Title"] = "Foo"
ViewBag	Динамическая оболочка для словаря ViewData, например, ViewBag.Title = "Foo"

Теперь, когда вы понимаете, каким образом данные отправляются представлениям, добавим несколько аннотаций данных, чтобы очистить модели для MVC.

Аннотации данных Display

Многие аннотации данных, применяемые инфраструктурой Entity Framework, также используются механизмом представлений MVC для визуализации разметки и проверки достоверности. Существуют дополнительные аннотации данных, которые игнорируются EF, но применяются для MVC, такие как аннотации данных Display.

Аннотации данных Display сообщают MVC имя, подлежащее использованию вместо имени свойства, когда в представлении применяется вспомогательный метод HTML под названием DisplayName() или DisplayNameFor(). Синтаксис прост, как иллюстрируется в показанном ниже примере, где отображаемое имя изменяется с PetName (имя свойства) на более читабельное Pet Name:

```
[Display(Name="Pet Name")]
```

Хотя в главе рассматриваются только аннотации данных Display, есть много других аннотаций, которые могут использоваться в той же самой манере. Больше сведений доступно в документации .NET Framework.

Специальные файлы метаданных

В то время как дополнительные аннотации данных можно добавлять к самим классам моделей, зачастую полезно помещать их в отдельный класс, чтобы предохранить целостность классов моделей. Добавим в проект AutoLotDAL.Models новую папку по имени MetaData, а в нее — новый файл класса под названием InventoryMetaData.cs. Сделаем класс открытым и добавим в начало файла класса следующий оператор using:

```
using System.ComponentModel.DataAnnotations;
```

Добавим в класс свойство типа string по имени PetName и применим к нему атрибут [Display(Name="Pet Name")]. Код класса должен выглядеть так:

```
public class InventoryMetaData
{
    [Display(Name="Pet Name")]
    public string PetName;
}
```

Откроем файл класса InventoryPartial.cs и модифицируем операторы using, как показано ниже:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using AutoLotDAL.Models.MetaData;
```

Добавим к классу атрибут `[MetadataType]`:

```
[MetadataType(typeof(InventoryMetaData))]
public partial class Inventory
{
    public override string ToString() =>
        $"{this.PetName ?? "***No Name***"} is a {this.Color} {this.Make} with ID {this.Id}.";
}
```

Файлы метаданных не являются файлами полных определений классов; они используются только для загрузки атрибутов в другие файлы. Следовательно, добавлять к такому свойству синтаксис `get/set` не придется и фактически это делаться не должно. Может возникнуть вопрос: как инфраструктура узнает, что указанный класс предоставляет атрибуты для класса `Inventory`? В настоящий момент никак. К классу `Inventory` необходимо добавить атрибут уровня класса, чтобы инфраструктуре стало известно, что класс `Inventory` содержит дополнительные атрибуты для нее.

Запустив приложение и щелкнув на ссылке `Inventory`, легко удостовериться в том, что метка `PetName` отображается как `Pet Name` без внесения изменений в код представления.

Шаблоны Razor

В табл. 29.9 присутствовало несколько вспомогательных методов HTML, которые полагаются на шаблоны, либо специальные, либо стандартные. Механизм Razor поддерживает два типа специальных шаблонов — шаблоны для отображения и шаблоны для редактирования. Шаблоны создаются как представления Razor и должны размещаться в местах на основе соглашения.

Шаблоны для отображения обязаны находиться в папке по имени `DisplayTemplates`, которая расположена либо внутри папки `Views\Shared`, либо внутри папки представления, специфичного для контроллера, такой как `Views\Inventory`. К шаблонам применимы те же самые правила, касающиеся представлений; любые шаблоны из папки `Views\Shared` доступны повсюду в приложении, а шаблоны из папки, специфичной для контроллера, доступны только внутри представлений, относящихся к данному контроллеру. К шаблонам для редактирования применяются все те же правила за исключением того, что папка, где они хранятся, должна иметь имя `EditorTemplates`.

К шаблонам можно обращаться либо явно, либо неявно. Для явного обращения к шаблону нужно сослаться на него по имени. Например, вот как обратиться к шаблону отображения под названием `InventoryList.cshtml`:

```
@Html.DisplayFor(modelItem=>item, "InventoryList")
```

Для неявного обращения к шаблону ему необходимо назначить такое же имя, как у типа данных, который он должен визуализировать. Затем шаблон будет инициализироваться автоматически, когда тип данных свойства совпадает с именем шаблона. Например, если шаблон именован как `Inventory.cshtml`, то он будет активизирован для экземпляра `Inventory`:

```
@Html.DisplayFor(modelItem=>item)
```

Создание специального шаблона для отображения

Чтобы построить специальный шаблон для отображения, начнем с создания внутри `View\Inventory` новой папки по имени `DisplayTemplates`. Добавим в нее новое представление, щелкнув правой кнопкой мыши на имени папки, выбрав в контекстном

меню пункт Add⇒View (Добавить⇒Представление) и заполнив открывшееся диалоговое окно Add View (Добавление представления), как показано на рис. 29.9.

Рис. 29.9. Создание шаблона для отображения

Итоговый шаблон будет заменять код, который используется при отображении каждой строки в представлении `Inventory/Index`. Удалим весь сгенерированный код и вместо него добавим следующий код (свойство `Timestamp` здесь отсутствует):

```
@model AutoLotDAL.Models.Inventory

<tr>
  <td>@Html.DisplayFor(model => model.Make)</td>
  <td>@Html.DisplayFor(model => model.Color)</td>
  <td>@Html.DisplayFor(model => model.PetName) </td>
  <td>
    @Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |
    @Html.ActionLink("Details", "Details", new { id = Model.Id }) |
    @Html.ActionLink("Delete", "Delete", new { id = Model.Id })
  </td>
</tr>
```

Теперь модифицируем оператор `foreach` в представлении `Index`:

```
@foreach (var item in SortCars(Model.ToList()))
{
  @Html.DisplayFor(modelItem=>item, "InventoryList")
}
```

Наконец, необходимо убрать заголовок с отметкой времени со страницы `Index`. Откроем файл `Index.cshtml` и удалим следующий код:

```
<th>
  @Html.DisplayNameFor(model => model.Timestamp)
</th>
```

Запустив приложение и перейдя на представление `Inventory/Index`, можно заметить, что оно визуализирует то же, что и ранее, минус свойство `Timestamp` (которое

лишь запутывало пользователя). Любые другие списки записей Inventory должны отображаться с применением данного шаблона, что обеспечит согласованный внешний вид приложения с гораздо меньшим объемом клавиатурного набора.

Создание специального шаблона для редактирования

Создадим внутри Views\Inventory папку по имени EditorTemplates. Добавим в нее файл представления под названием Inventory.cshtml (как делалось при создании шаблона для отображения) и удалим сгенерированный код. Добавим следующую модель:

```
@model AutoLotDAL.Models.Inventory
```

Откроем файл Edit.cshtml, скопируем весь код, находящийся внутри элемента <div class="form-horizontal"> (включая сам элемент <div>) и вставим его в новый шаблон редактирования. Удалим элемент <div> для свойства Timestamp и добавим для этого свойства новое скрытое поле. Ниже приведен окончательный код:

```
@model AutoLotDAL.Models.Inventory

<div class="form-horizontal">
  <h4>Inventory</h4>
  <hr />
  @Html.ValidationSummary(true, "", new { @class = "text-danger" })
  <div class="form-group">
    @Html.LabelFor(model => model.Make,
      htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
      @Html.EditorFor(model => model.Make,
        new { htmlAttributes = new { @class = "form-control" } })
      @Html.ValidationMessageFor(model => model.Make, "",
        new { @class = "text-danger" })
    </div>
  </div>
  <div class="form-group">
    @Html.LabelFor(model => model.Color,
      htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
      @Html.EditorFor(model => model.Color,
        new { htmlAttributes = new { @class = "form-control" } })
      @Html.ValidationMessageFor(model => model.Color, "",
        new { @class = "text-danger" })
    </div>
  </div>
  <div class="form-group">
    @Html.LabelFor(model => model.PetName,
      htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
      @Html.EditorFor(model => model.PetName,
        new { htmlAttributes = new { @class = "form-control" } })
      @Html.ValidationMessageFor(model => model.PetName, "",
        new { @class = "text-danger" })
    </div>
  </div>
</div>
```

Финальный шаг связан с очисткой представления Edit.cshtml. Модифицируем его код следующим образом:

```

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    @Html.HiddenFor(model => model.Id)
    @Html.HiddenFor(model => model.Timestamp)
    @Html.EditorForModel()
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Save" class="btn btn-default" />
        </div>
    </div>
}

```

В итоге мы значительно очистили представление `Edit.cshtml` и инкапсулировали разметку для редактирования записи `Inventory`. Приведем содержимое файла представления `Create.cshtml` к такому виду:

```

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    @Html.EditorForModel()
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Create" class="btn btn-default" />
        </div>
    </div>
}

```

Один и тот же шаблон работает в двух местах, существенно сокращая объем разметки на обеих страницах.

Работа с формами

Если только вы не строите сайт, предназначенный целиком для маркетинговых целей, то получение данных от пользователя через формы будет жизненно важным компонентом разработки веб-приложений. С учетом сказанного необходимо ближе ознакомиться со вспомогательными методами HTML по имени `BeginForm()` и `AntiForgeryToken()`.

Вспомогательный метод HTML по имени `BeginForm()`

Вспомогательный метод HTML по имени `BeginForm()` создает дескриптор `<form>` в выводе HTML. По умолчанию свойство `action` формы получает текущий URL, а свойство `method` — `post` (оба свойства допускают настройку за счет применения разных перегруженных версий метода `BeginForm()`). Блок `using` в коде `Razor` будет инкапсулировать все, что указано между фигурными скобками, внутри сгенерированных открывающего и закрывающего дескрипторов HTML. Например, поместим в представление следующий блок кода `Razor`:

```

@using (Html.BeginForm())
{
    <input name="foo" id="foo" type="text"/>
}

```

Если URL для запроса `HttpGet` имеет вид `Inventory/Create`, тогда вспомогательный метод `Html.BeginForm()` создаст такую разметку:

```

<form action="/Inventory/Create" method="post">
    <input name="foo" id="foo" type="text"/>
</form>

```

Вспомогательный метод HTML по имени AntiForgeryToken()

Вспомните, что атрибут [ValidateAntiForgeryToken] должен добавляться ко всем методам действий HttpPost (и по умолчанию он добавляется, когда используются встроенные шаблонные представления). Вспомогательный метод HTML по имени AntiForgeryToken() создает значение клиентской стороны, которое контролируется проверкой достоверности серверной стороны. Если значения не совпадают, то запрос отклоняется.

Обновление представления Delete

Необходимо также изменить шаблонное представление Delete. Первым делом удалим поле Timestamp, которое имеет мало смысла для пользователя (и потенциально запутывает). Кроме того, представление Delete не создает каких-либо складских записей; при ссылке на запись, подлежащую удалению, оно полагается целиком на параметры URL.

Модифицируем код формы, как показано ниже, чтобы передать обязательные значения, которые сделают возможным удаление складской записи:

```
@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
    @Html.HiddenFor(x => x.Id)
    @Html.HiddenFor(x => x.Timestamp)
    @Html.ValidationSummary(true, "", new { @class = "text-danger" })
    <div class="form-actions no-color">
        <input type="submit" value="Delete" class="btn btn-default" /> |
        @Html.ActionLink("Back to List", "Index")
    </div>
}
```

Проверка достоверности

Приложения MVC имеют два уровня проверки достоверности: на серверной стороне и на клиентской стороне. Проверка достоверности серверной стороны осуществлялась ранее в главе, когда производилось добавление ошибок в ModelState как дополнение к ошибкам, которые возникали в процессе привязки модели. Проверка достоверности клиентской стороны реализуется с помощью JavaScript, как вскоре будет показано.

Отображение ошибок

Ошибки ModelState отображаются в пользовательском интерфейсе посредством вспомогательных методов HTML с названиями ValidationMessageFor() и ValidationSummary(). Метод ValidationSummary() будет показывать ошибки ModelState, не связанные со свойствами, а также ошибки, которые относятся к свойствам (при условии, что свойство ValidationSummary.ExcludePropertyErrors установлено в false). Обычно ошибки, связанные со свойствами, отображаются рядом со свойствами, а с применением метода ValidationSummary() выводятся только ошибки, не имеющие отношения к свойствам. Например, следующая строка кода (в представлениях Create, Edit и Delete) обеспечит отображение всех ошибок модели (но не ошибок, касающихся свойств) с использованием шрифта красного цвета:

```
@Html.ValidationSummary(true, "", new { @class = "text-danger" })
```

Для отображения ошибок, относящихся к свойствам, применяется вызов вспомогательного метода `ValidationMessageFor()` рядом со свойствами на странице представления:

```
@Html.ValidationMessageFor(model => model.Make, "", new { @class = "text-danger" })
```

Вот результирующая разметка:

```
<span class="field-validation-valid text-danger"
      data-valmsg-replace="true" data-valmsg-for="Make"></span>
```

Чтобы посмотреть, как отображаются сообщения об ошибках, запустим проект, перейдем на страницу `Inventory>Create New` (Склад<gtСоздать новую запись) и введем в поле `Make` (Производитель) какую-нибудь строку, содержащую более 50 символов. После покидания элемента управления по нажатию клавиши `<Tab>` процесс проверки достоверности на стороне клиента сгенерирует ошибку, которая затем отобразится вспомогательным методом HTML по имени `ValidationMessageFor()`.

Обновим вызов `ValidationSummary()`, чтобы указать на необходимость отображения всех ошибок, в том числе и тех, что связаны со свойствами:

```
@Html.ValidationSummary(false, "", new { @class = "text-danger" })
```

Снова запустим проект и воспроизведем ту же самую ошибку. Теперь она отсутствует в сводке по проверке достоверности. Причина в том, что метод `ValidationSummary()` и проверка клиентской стороны, к сожалению, не уживаются друг с другом. Чтобы увидеть в действии проверку достоверности серверной стороны, проверка достоверности клиентской стороны должна быть отключена.

Отключение проверки достоверности клиентской стороны

К счастью, отключить проверку достоверности клиентской стороны для представлений MVC очень легко. Изменим код `@section` в представлении `Create.cshtml`, следующим образом:

```
@section Scripts {
    @*@Scripts.Render("~/bundles/jqueryval")*@
}
```

Запустим проект, воспроизведем ту же самую ошибку и удостоверимся в том, что она больше не отображается после покидания элемента управления `Make` по нажатию клавиши `<Tab>`. Щелкнем на кнопке `Create` (Создать); появится сводка и сообщения об ошибках уровня полей.

Прежде чем переходить к следующему разделу, понадобится снова включить проверку достоверности клиентской стороны.

Проверка достоверности клиентской стороны

Проверка достоверности клиентской стороны обрабатывается посредством библиотек jQuery, проверки достоверности jQuery и ненавязчивой проверки достоверности jQuery. Инфраструктура MVC работает с jQuery, анализируя атрибуты модели и затем добавляя подходящие проверки достоверности к визуализируемой разметке HTML, когда используются вспомогательные методы HTML для редактирования. Например, вспомогательный метод HTML по имени `EditorFor()` сгенерирует следующую разметку для свойства `Make`:

```
<input name="Make" class="form-control text-box single-line input-validation-error"
      id="Make" aria-invalid="true" aria-describedby="Make-error" type="text"
      value="" data-val-length-max="50" data-val-length="The field Make must be a
      string with a maximum length of 50." data-val="true">
```

Атрибуты данных также поддерживают специальные сообщения об ошибках. Откроем файл класса `InventoryMetaData.cs` из проекта `AutoLotDAL.Models` и добавим в него такой код:

```
[StringLength(50,
    ErrorMessage = "Please enter a value less than 50 characters long.")]
public string Make;
```

Запустив приложение и повторив тест, можно увидеть более дружелюбное сообщение об ошибке.

На заметку! Хотя в примере было показано, что можно иметь один атрибут, определенный дважды для свойства, обычно поступать так не рекомендуется. Дублирование программного кода любого вида считается признаком недоброкачественного кода; более приемлемый подход предусматривает добавление сообщения прямо к атрибуту класса `Inventory`.

Итоговые сведения об инфраструктуре ASP.NET MVC

Часто возникает вопрос: Web Forms или MVC? При работе с полной инфраструктурой .NET Framework ответить на него не так-то просто. Если разработчикам более комфортно иметь дело с операциями перетаскивания при создании пользовательского интерфейса или у них вызывает трудности лишенная состояния природа HTTP, то вероятно лучше выбрать Web Forms. Если же разработчики предпочитают располагать полным контролем над пользовательским интерфейсом (что также означает меньший объем “магии”, выполняемой автоматически) и строить приложения, лишенные состояния, которые задействуют методы HTTP (такие как `HttpGet` и `HttpPost`), тогда лучшим вариантом будет MVC. Конечно, при поиске решения придется учитывать намного больше факторов. Мы упомянули лишь несколько из них.

При построении приложений с помощью полной инфраструктуры .NET Framework вам не придется делать выбор между инфраструктурами MVC и Web Forms, поскольку они обе основаны на `System.Web` и могут применяться вместе даже в рамках одного приложения. После того, как в Visual Studio 2013 была введена концепция “One ASP.NET” (“Одна инфраструктура ASP.NET”), смешивать две инфраструктуры в единственном проекте стало намного легче.

Важно отметить, что .NET Core и ASP.NET Core поддерживают только приложения в стиле MVC, и не следует особо рассчитывать на возможность переписывания инфраструктуры Web Forms для .NET Core. Такое положение дел может изменить ваше решение о том, какую инфраструктуру выбрать, когда приходится работать с полной версией .NET Framework.

Естественно, в главе мы лишь слегка коснулись поверхности ASP.NET MVC. Тема MVC слишком обширна, чтобы ее можно было раскрыть в одной главе. Исчерпывающее описание всего того, что инфраструктура MVC способна предложить, можно найти в книге *ASP.NET MVC 5 с примерами на C# для профессионалов, 5-е изд.* (ИД “Вильямс”).

Резюме

В настоящей главе были исследованы многие аспекты ASP.NET MVC. Для начала вы ознакомились с паттерном Model-View-Controller (MVC) и построили свой первый сайт MVC. Вы узнали о соглашениях по конфигурации для инфраструктуры MVC, обо всех шаблонных файлах, создаваемых как часть шаблона проекта, а также о папках и их предназначении. Были рассмотрены все классы из папки `App_Start` и показано, каким образом они помогают в создании приложений MVC. Вы также освоили объединение в пакеты и минификацию и научились при необходимости отключать их.

Затем обсуждалась маршрутизация и способы направления запросов контроллерам и действиям. Вы создали новые маршруты для страниц `About` и `Contact` и научились перенаправлять пользователей на другие ресурсы внутри сайта с использованием маршрутизации вместо жестко закодированных URL.

Вы создали контроллер для страниц `Inventory` и узнали, каким образом встроенное в Visual Studio средство формирования шаблонов создает базовые методы действий и представления. Вы ознакомились с запросами `HttpGet` и `HttpPost` и выяснили, как они взаимодействуют с механизмом маршрутизации для обеспечения еще большей степени контроля над тем, какой метод действия будет вызван. После этого вы модифицировали методы действий для применения библиотеки `AutoLotDAL`, а также обновили сигнатуры и код для удовлетворения бизнес-требованиям.

Вы узнали о механизме представлений `Razor`, его синтаксисе, вспомогательных методах, функциях и делегатах. Также были приведены дополнительные сведения о строго типизированных представлениях, частичных представлениях и компоновках. Вы научились отправлять данные представлению, используя объекты `ViewBag`, `ViewData` и `TempData`.

Наконец, вы изменили каждое шаблонное представление, должным образом обновили действия `InventoryController` и поэкспериментировали с проверкой достоверности клиентской и серверной сторон.

В следующей главе исследуется очень близкий родственник ASP.NET MVC — инфраструктура ASP.NET Web API.

ГЛАВА 30

Введение в ASP.NET Web API

В предыдущей главе рассматривалась инфраструктура ASP.NET MVC — как сам паттерн, так и его реализация .NET. В настоящей главе представляется ASP.NET Web API — инфраструктура служб, в значительной степени построенная на основе MVC, которая разделяет многие концепции, включая маршрутизацию, контроллеры и действия. Инфраструктура ASP.NET Web API позволяет использовать имеющиеся знания MVC для построения служб REST, не прибегая к конфигурированию и написанию связующего кода, как того требует инфраструктура WCF (см. главу 23). Мы создадим службу REST по имени CarLotWebAPI, которая открывает доступ ко всей функциональности создания, чтения, обновления и удаления (CRUD) для складских записей. Глава завершается обновлением приложения CarLotMVC с целью применения CarLotWebAPI в качестве источника данных вместо прямой работы с уровнем доступа к данным.

На заметку! В главе не будет повторяться содержимое из предыдущей главы. Взамен мы сосредоточим внимание на особенностях создания службы REST с помощью проектов ASP.NET Web API. Если вы не знакомы с ASP.NET MVC, тогда рекомендуется сначала прочитать главу 29.

Инфраструктура ASP.NET Web API

В главе 23 вы узнали, что Windows Communication Foundation (WCF) является полноценной инфраструктурой для создания основанных на .NET служб, которые способны взаимодействовать с широким диапазоном клиентов. Наряду с тем, что инфраструктура WCF отличается исключительной мощностью, когда требуются лишь простые службы на основе HTTP, то разработка службы с использованием WCF может оказаться сложнее, чем необходимо или желательно. И здесь в игру вступает ASP.NET Web API — еще одна инфраструктура для построения API-интерфейсов веб-сети в .NET, к которым возможен доступ из любого клиента, поддерживающего HTTP. С точки зрения разработки MVC инфраструктура Web API является логическим дополнением инструментального набора .NET. Она построена на основе MVC и задействует многие из тех же самых концепций наподобие моделей, контроллеров и маршрутизации. Инфраструктура Web API была впервые выпущена в составе Visual Studio 2012 как часть MVC 4 и обновлена до версии 2.2 в Visual Studio 2013 Update 1.

Создание проекта Web API

Начнем с добавления проекта Web API по имени CarLotWebAPI. Откроем диалоговое окно New Project (Новый проект), в левой части окна перейдем на вкладку Web (Веб) и выберем шаблон ASP.NET Web Application (.NET Framework) (Веб-приложение ASP.NET (.NET Framework)), как показано на рис. 30.1. Щелкнем правой кнопкой мыши на имени решения, выберем в контекстном меню пункт Add⇒New Project (Добавить⇒Новый проект) и укажем вариант ASP.NET Web Application (Веб-приложение ASP.NET), как показано на рис. 30.1.

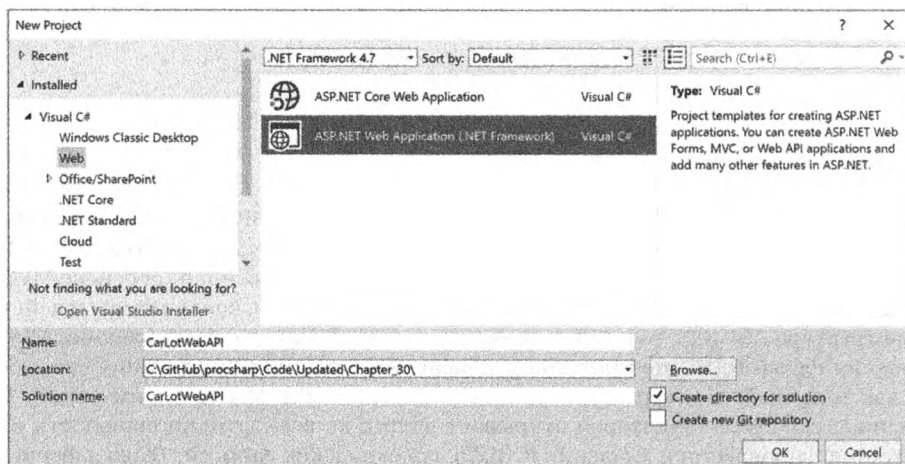


Рис. 30.1. Добавление нового проекта веб-приложения ASP.NET

Открывшееся диалоговое окно уже должно быть знакомым. Однако в данный момент мы выберем шаблон Empty (Пустой), а в области Add folders and core references for (Добавить папки и основные ссылки для) отметим флажок Web API (рис. 30.2).

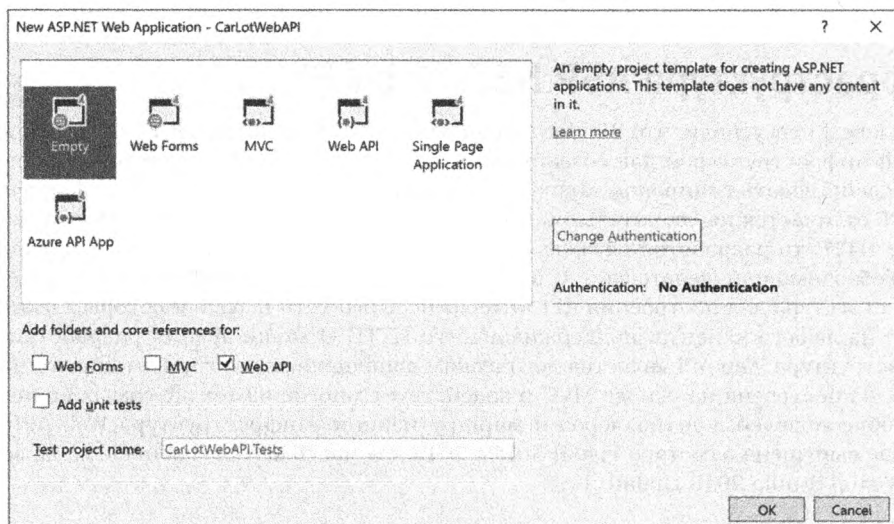


Рис. 30.2. Добавление шаблона проекта Empty с поддержкой Web API

В случае выбора шаблона Web API к проекту добавляется большой объем стереотипного кода (включая контроллеры и представления MVC), но нас интересует только базовый связующий код Web API.

Щелкнем на кнопке ОК для добавления проекта к решению.

Управление пакетами NuGet

Как и в ситуации с проектами Web Forms и MVC, многие включенные пакеты NuGet к моменту создания проекта устарели. Щелкнем правой кнопкой мыши на имени проекта в окне Solution Explorer, выберем в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet) и в поле со списком Filter (Фильтр) выберем Upgrade Available (Доступно обновление). Обновим все пакеты, которые можно обновить.

Вернем в поле Filter вариант Browse (Обзор) и установим пакеты AutoMapper и EntityFramework (6.1.3).

Добавление проектов AutoLotDAL и AutoLotDAL.Models

Скопируем проекты AutoLotDAL и AutoLotDAL.Models из главы 29 в папку CarLotWebAPI (на тот же уровень, где находится файл решения CarLotWebAPI). Добавим скопированные проекты в решение, щелкнув правой кнопкой мыши на решении CarLotMVC, выбрав в контекстном меню пункт Add⇒Existing Project (Добавить⇒Существующий проект) и в открывшемся диалоговом окне перейдя в нужную папку. Добавим в CarLotWebAPI ссылку на AutoLotDAL и AutoLotDAL.Models. Также проверим, что в AutoLotDAL имеется ссылка на AutoLotDAL.Models. Добавление проекта AutoLotDAL.Models первым должно сохранить ссылку.

Наконец, добавим строку подключения в файл Web.config (в зависимости от пути установки она может отличаться):

```
<connectionStrings>
  <add name="AutoLotConnection"
    connectionString="data source=(localdb)\mssqllocaldb;
    initial catalog=AutoLot;integrated security=True;
    MultipleActiveResultSets=True;App=EntityFramework"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

Создание класса InventoryController

Несмотря на наличие в Visual Studio поддержки шаблонных классов для проектов Web API (как и для проектов MVC), мы собираемся построить класс вручную. Добавим в папку Controllers файл класса по имени InventoryController.cs. Поместим в начало файла следующие операторы using:

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Threading.Tasks;
using System.Web.Http;
using System.Web.Http.Description;
using AutoLotDAL.Models;
using AutoLotDAL.Repos;
using AutoMapper;
```

Сделаем класс открытым и унаследованным от ApiController:

```
public class InventoryController : ApiController
{
}
```

Маршрутизация

Файл `WebApiConfig.cs` конфигурирует проект. Обращение к нему происходит из файла `Global.asax.cs`. Откроем файл `WebApiConfig.cs` из папки `App_Start`. В начале кода находится строка, которая включает маршрутизацию с помощью атрибутов (рассматривается далее). Следующий фрагмент кода добавляет в таблицу маршрутов стандартный маршрут в точности, как в ASP.NET MVC. Разница в том, что стандартный маршрут начинается со строкового литерала `api/`.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // Конфигурация и службы Web API
        // Маршруты Web API
        config.MapHttpAttributeRoutes();
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

Удалим (или закомментируем) код, который конфигурирует стандартный маршрут, поскольку в проекте будет применяться маршрутизация с помощью атрибутов.

Маршрутизация с помощью атрибутов

При маршрутизации с помощью атрибутов маршруты определяются с использованием атрибутов C# на классах контроллеров и их методах действий. Итогом может стать более точная маршрутизация, но одновременно есть вероятность увеличения объема конфигурации, потому что контроллеры и действия нуждаются в указании маршрутизирующей информации.

В целях демонстрации поместим приведенный ниже код над объявлением класса контроллера:

```
[Route("api/Inventory/{id?}")]
public class InventoryController : ApiController
```

Мы имеем то же самое определение маршрута, которое создавалось как стандартный маршрут, но с одним крупным отличием — переменная `{controller}` отсутствует. Когда применяется маршрутизация с помощью атрибутов, контроллер выводится. В версии Web API 2.2 маршрутизация с помощью атрибутов обычно резервируется для использования с методами действий, но нет ничего плохого в том, чтобы применять ее с контроллером. Фактически предыдущий код можно было бы модифицировать, изменив URL с `/Inventory` на `/Cars`:

```
[Route("api/Cars/{id?}")]
public class InventoryController : ApiController
```

Добавим два метода действий:

```
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}
```

```
// GET api/values/5
public string Get(int id)
{
    return id.ToString();
}
```

Вы могли заметить, что ни стандартный маршрут, ни атрибутный маршрут не определяют действие. Дело в том, что механизм маршрутизации вместо указания имени действия использует методы HTTP (GET, PUT, POST, DELETE). В версии Web API 2.2 имя метода действия транслируется прямо в метод. Поскольку в рассматриваемом примере оба метода начинаются с *Get*, они будут запрашиваться с применением *HttpGet*.

Хотя такая возможность удобна, в ASP.NET Core она объявлена устаревшей, так что лучше привыкнуть к указанию надлежащего метода. Модифицируем атрибуты методов и контроллера, как показано далее:

```
[RoutePrefix("api/Inventory")]
public class InventoryController : ApiController
{
    [HttpGet, Route("")]
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    // GET api/values/5
    [HttpGet, Route("{id}")]
    public string Get(int id)
    {
        return id.ToString();
    }
}
```

Здесь есть пара элементов, заслуживающих обсуждения. Во-первых, атрибут контроллера был изменен с *Route* на *RoutePrefix*. Если оставить *Route*, тогда атрибуты *Route* на методах действий переопределят атрибут контроллера. Во-вторых, атрибут метода *Get()* задает маршрут в ответ на HTTP-запрос GET, а потому метод действия можно именовать любым желаемым образом. Пустой атрибут *Route* на методе *Get()* требуется для завершения запроса, т.к. атрибут *RoutePrefix* обозначает только начало маршрута, а не весь маршрут.

Изменение страницы запуска

Приложения Web API являются автоматическими службами, что означает отсутствие пользовательского интерфейса. Таким образом, стандартные настройки проекта, которые обеспечивают запуск страницы, выбранной в текущий момент в окне *Solution Explorer*, не имеют смысла.

Приложение может быть сконфигурировано так, чтобы при запуске ничего не делать (переключатель *Don't open a page. Wait for a request from an external application* (Не открывать страницу. Ожидать запроса от внешнего приложения)) или запускать специфический маршрут. В таком случае выберем переключатель *Specific Page* (Специфическая страница) и в поле рядом с ним введем *api/Inventory* (рис. 30.3). В результате приложение будет запускать стандартный HTTP-запрос GET для контроллера *Inventory*.

Теперь после запуска проекта в браузере должны отображаться значения из метода *Get()*. В зависимости от браузера они будут иметь вид либо просто строки, либо более формального представления данных.

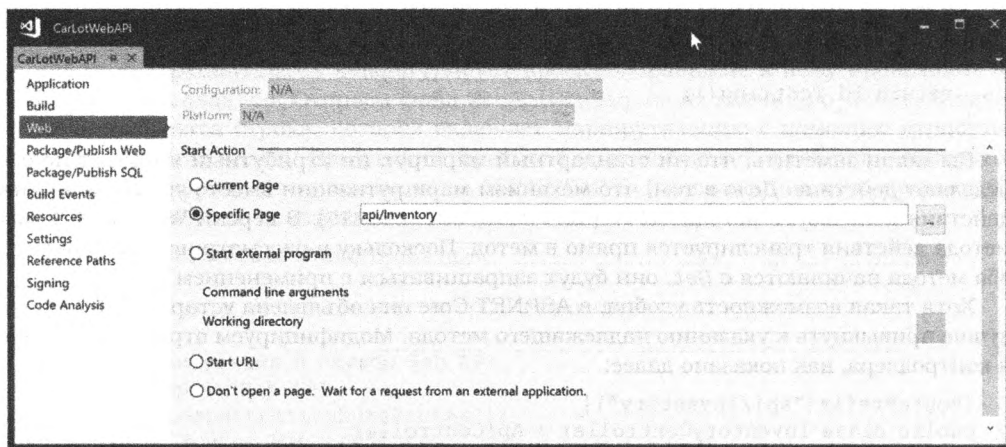


Рис. 30.3. Конфигурирование запуска для проекта CarLotWebAPI

Формат JSON

Формат JSON (JavaScript Object Notation — запись объектов JavaScript) является одним из способов передачи данных между службами, который практически занял место XML в качестве механизма транспортировки данных. Это простое текстовое представление в виде пар “ключ-значение” объектов и классов. Например, взгляните на следующее представление JSON элемента Inventory:

```
{"CarId":1,"Make":"VW","Color":"Black","PetName":"Zippy",
  "Timestamp":"AAAAAAAB9o=", "Orders":[]}
```

Каждый объект JSON начинается и заканчивается фигурными скобками, а имена свойств и строковые значения находятся в двойных кавычках. Объекты JSON также могут быть вложенными. Если бы свойство Make было не строкой, а объектом (со свойствами Builder и Year), тогда вот как могло бы выглядеть представление JSON:

```
{"CarId":1,"Make":{"Builder":"VW","Year":2015},"Color":"Black",
  "PetName":"Zippy","Timestamp":"AAAAAAAB9o=", "Orders":[]}
```

По свойству Orders видно, что списки обозначаются квадратными скобками ([]). Если служба отправила список объектов Inventory, то представление JSON может напоминать следующее:

```
[{"CarId":1,"Make":"VW","Color":"Black","PetName":"Zippy",
  "Timestamp":"AAAAAAAB9o=", "Orders":[]}, {"CarId":2,"Make":"Ford",
  "Color":"Rust","PetName":"Rusty","Timestamp":"AAAAAAAB9s=", "Orders":[]}]
```

На заметку! Шаблон проекта Web API включает бесплатную инфраструктуру с открытым кодом Json.NET от Newtonsoft. Она предназначена для создания представлений JSON из объектов и объектов из представлений JSON. Инфраструктура Json.NET будет использоваться позже в главе, а дополнительные сведения о ней (включая документацию и примеры) доступны по адресу www.newtonsoft.com/json.

Результаты действий Web API

Методы действий Web API могут возвращать один из четырех ответов:

- `void` — возвращает HTTP 204 (содержимое отсутствует);
- `HttpResponseMessage` — преобразует прямо в стандартный ответ HTTP;
- `IHttpActionResult` — создает ответ HTTP для возвращения;
- другой тип — записывает сериализованное возвращаемое значение вместе с HTTP 200 (успешно).

Вариант `void` самоочевиден. Ответ `HttpResponseMessage` требует построения возвращаемого сообщения, как демонстрируется в следующем примере:

```
public HttpResponseMessage Get()
{
    HttpResponseMessage response = Request.CreateResponse(HttpStatusCode.OK, "value");
    response.Content = new StringContent("hello", Encoding.Unicode);
    response.Headers.CacheControl = new CacheControlHeaderValue()
    {
        MaxAge = TimeSpan.FromMinutes(20)
    };
    return response;
}
```

Интерфейс `IHttpActionResult` появился в версии Web API 2 и по существу представляет собой фабрику `HttpResponseMessage`. Чаще всего возвращается одна из реализаций, определенных в пространстве имен `System.Web.Http.Results`, которая полностью документирована здесь: <https://msdn.microsoft.com/en-us/library/system.web.http.results.aspx>. Как вы увидите далее в главе, базовый класс `ApiController` имеет множество удобных вспомогательных методов. Для справочных целей в табл. 30.1 перечислены наиболее распространенные из них.

Таблица 30.1. Типовые классы, производные от `IHttpActionResult`

Результат действия	Описание
<code>OkResult</code>	Возвращает HTTP 200 (успешно)
<code>RedirectResult</code> <code>RedirectToRouteResult</code>	Перенаправляет на другое действие или маршрут
<code>JsonResult</code>	Возвращает клиенту сериализованный результат JSON
<code>BadRequestResult</code>	Возвращает пустой результат ошибочного запроса
<code>NotFoundResult</code>	Указывает, что запрошенный элемент не найден
<code>FormattedContentResult</code>	Возвращает клиенту форматированный тип содержимого, определенный пользователем
<code>StatusCodeResult</code>	Возвращает специфический код состояния HTTP

В отношении всех остальных типов ответов инфраструктура Web API применяет усредненный формater для сериализации возвращаемого значения, записывает значение в тело сообщения и возвращает HTTP 200 (успешно). Такой формат будет использоваться для возвращения клиенту записей `Inventory`.

Проблема сериализации, связанная с использованием Entity Framework

После добавления ссылки на другой проект (или сборку) вы способны передавать повсюду экземпляры классов .NET. Веб-сеть работает иначе в том, что все сериализируется в другой формат, подобный JSON.

Приемы сериализации рассматривались в главе 20, но хорошая новость связана с тем, что инфраструктура Web API берет на себя выполнение большей части трудной работы. Тем не менее, в случае использования моделей EF существует проблема: навигационные свойства. Процесс сериализации проходит по всей объектной модели, пытаясь сериализовать все, что встречается у него на пути. Это значит, что процесс сериализации затрагивает также и все зависимые объекты. Данные могут стать гораздо более крупными, чем ожидалось. Или же процесс сериализации может потерпеть неудачу, если включена ленивая загрузка.

При отправке данных в формате JSON (или XML) из службы REST важно посылать лишь те данные, которые нужны. В нашей ситуации речь идет только о самом классе `Inventory`, но не о связанных объектах. К счастью, доступна великолепная утилита с открытым кодом, которую можно применять для решения указанной проблемы.

Утилита AutoMapper

Пакет AutoMapper (установленный ранее в главе) является бесплатной утилитой с открытым кодом, которая предназначена для создания экземпляра одного типа из экземпляра другого типа. При установлении совпадающих свойств она использует рефлексии, но также может конфигурироваться для специальных случаев.

В разрабатываемом примере приложения нужно, чтобы выборке записи `Inventory` отбрасывались любые связанные объекты. Утилита AutoMapper способна создавать новый экземпляр того же самого типа, что мы и будем делать здесь. Добавим в класс `InventoryController` конструктор со следующим кодом:

```
public InventoryController()
{
    Mapper.Initialize(
        cfg =>
        {
            cfg.CreateMap<Inventory, Inventory>()
                .ForMember(x => x.Orders, opt => opt.Ignore());
        });
}
```

В приведенном коде создается отображение между типом `Inventory` и им же самим (или коллекцией типов `Inventory`) с игнорированием свойства `Orders`. Для преобразования экземпляра `Inventory` в новый экземпляр (или список `List<Inventory>` экземпляров) будет применяться следующий код (где `inventory` и `inventories` — исходные элементы):

```
var newInv = Mapper.Map<Inventory, Inventory>(inventory)
var newList = Mapper.Map<List<Inventory>, List<Inventory>>(inventories)
```

На заметку! Хотя в главе недостаточно места для глубоких исследований утилиты AutoMapper, отметим, что она является активной и широко используемой разработчиками приложений .NET. Вы должны обдумать добавление AutoMapper в свой арсенал инструментов. Дополнительные сведения, включая документацию и примеры, можно найти на домашней странице проекта по адресу <http://automapper.org>.

Получение складских данных

Теперь, когда основа на месте, наступило время заняться обновлением класса `InventoryController`, чтобы он выполнял реальную работу. Начнем с добавления переменной уровня класса для хранилища `Inventory`.

```
private readonly InventoryRepo _repo = new InventoryRepo();
```

Удалим (или закомментируем) два созданных ранее метода и заменим их таким кодом:

```
// GET: api/Inventory
[HttpGet, Route("")]
public IEnumerable<Inventory> GetInventory()
{
    var inventories = _repo.GetAll();
    return Mapper.Map<List<Inventory>, List<Inventory>>(inventories);
}

// GET: api/Inventory/5
[HttpGet, Route("{id}", Name = "DisplayRoute")]
[ResponseType(typeof(Inventory))]
public async Task<IHttpActionResult> GetInventory(int id)
{
    Inventory inventory = _repo.GetOne(id);
    if (inventory == null)
    {
        return NotFound();
    }
    return Ok(Mapper.Map<Inventory, Inventory>(inventory));
}
```

Первый метод получает все записи `Inventory` из базы данных и возвращает список в формате XML или JSON в зависимости от клиента. Браузеры Chrome и Firefox возвращают данные в формате XML, а браузеры Edge и Fiddler (рассматривается далее) — в формате JSON. Имя маршрута будет использоваться позже в контроллере.

Второй метод ищет специфическую складскую запись и возвращает код состояния HTTP 404 (не найдено), если найти ее не удалось. Если запись обнаружена, то метод возвращает ответ HTTP 200 (успешно) с записью внутри его содержимого.

Добавление метода `Dispose()`

Освободим экземпляр хранилища в методе `Dispose()` контроллера:

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        _repo.Dispose();
    }
    base.Dispose(disposing);
}
```

Использование Fiddler

Fiddler — бесплатный инструмент производства Progress Software Corporation, который предназначен для создания и мониторинга веб-запросов. Он полезен при тестировании приложений Web API, поддерживая все HTTP-методы, которые необходимо про-

верить (GET, POST, PUT и DELETE). Инструмент Fiddler доступен для загрузки по адресу www.telerik.com/fiddler.

После установки Fiddler откроем главное окно инструмента и перейдем на вкладку Composer (Формирователь). Появится место для ввода URI запроса и типа. В раскрывающемся списке выберем Get и добавим URI приложения (<http://localhost:60710/api/Inventory/>), как показано на рис. 30.4.

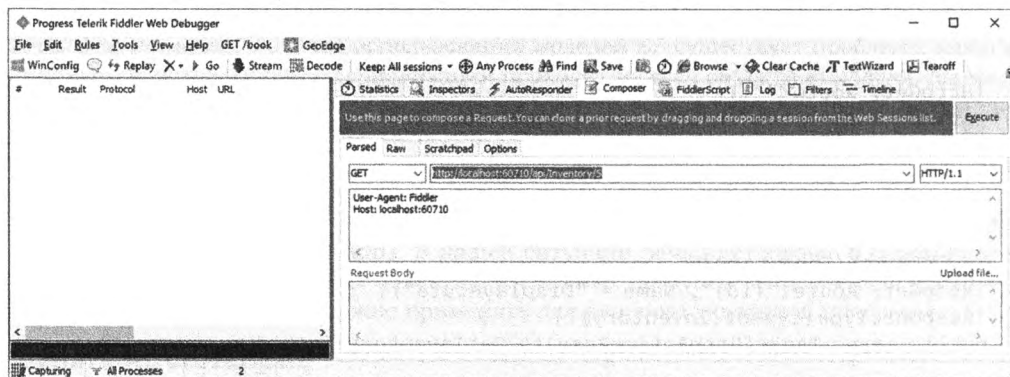


Рис. 30.4. Тестирование действия по получению всех складских записей с помощью Fiddler

При функционирующем приложении щелчком на кнопке Execute (Выполнить) и затем отыщем запрос в области слева. Двойной щелчок на нем приведет к отображению результатов обращения в области справа. Здесь можно перемещаться между вкладками JSON и XML, просматривать заголовки, а также изучать все аспекты запроса и ответа (рис. 30.5).

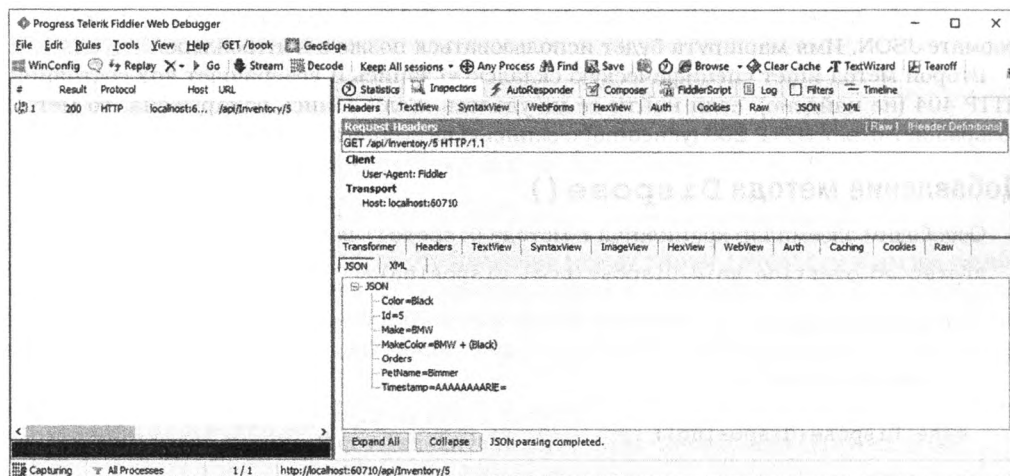


Рис. 30.5. Исследование ответа из приложения Web API

Инструмент Fiddler будет применяться далее в главе.

Обновление складской записи (HttpPut)

Обновление записи в HTTP достигается вызовом метода `HttpPut`. В текущем примере ему передается идентификатор записи, подлежащей обновлению, и экземпляр `Inventory` в формате JSON. Инфраструктура Web API использует привязку модели (подобно Web Forms), чтобы создать экземпляр класса `Inventory` со значениями, отправленными из клиента в теле сообщения.

Создадим метод `PutInventory()`, который принимает значение `id` и объект `Inventory`. Ниже представлен листинг кода с последующим обсуждением:

```
[HttpPut, Route("{id}")]
[ResponseType(typeof(void))]
public IHttpActionResult PutInventory(int id, Inventory inventory)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    if (id != inventory.Id)
    {
        return BadRequest();
    }
    try
    {
        _repo.Save(inventory);
    }
    catch (Exception ex)
    {
        // В производственном приложении здесь должны быть дополнительные действия
        throw;
    }
    return StatusCode(HttpStatusCode.NoContent);
}
```

Внутри метода действия сначала проверяется допустимость состояния модели (`ModelState`). Если состояние модели оказывается недопустимым, тогда возвращается `HttpBadRequest (400)`. Если же оно допустимо, то затем проверяется, совпадает ли значение параметра `id`, переданного в URL, со значением `Id` записи `Inventory` (из тела сообщения). Это помогает сократить (но не устранить) возможность взлома URL злоумышленниками. Далее предпринимается попытка сохранить запись, и если она завершается успешно, тогда возвращается код состояния HTTP 204 (содержимое отсутствует). В случае если возникло исключение, то оно просто возвращается клиенту. В приложении производственного уровня пришлось бы соответствующим образом обрабатывать все исключения.

Установка инициализатора данных

Прежде чем проверять изменения в базе данных, разумно предусмотреть инициализатор данных для воссоздания базы данных при каждом запуске приложения. Хотя такой прием увеличит время запуска приложения, он обеспечит согласованное поведение во время разработки приложения.

Откроем файл `Global.asax.cs` и поместим в его начало следующие операторы `using`:

```
using System.Data.Entity;
using AutoLotDAL.EF;
```

Добавим в метод `Application_Start()` приведенную ниже строку:

```
Database.SetInitializer(new MyDataInitializer());
```

Тестирование методов PUT

Откроем Fiddler и перейдем на вкладку Composer. Введем URI (например, `http://localhost:60710/api/Inventory/5`), выберем в качестве метода PUT и добавим в верхнюю область следующую строку:

```
Content-Type: application/json
```

В нижней области добавим код JSON для обновляемой записи. Учитывая включенную проверку параллелизма, для успешного прохождения теста понадобятся допустимые значения идентификатора и отметки времени.

На заметку! Получить код JSON легче всего с помощью Fiddler, выполнив запрос GET, дважды щелкнув на ответе и выбрав `TextView` в панели результатов. Затем результаты нужно скопировать в тело запроса PUT.

Пример запроса показан на рис. 30.6.

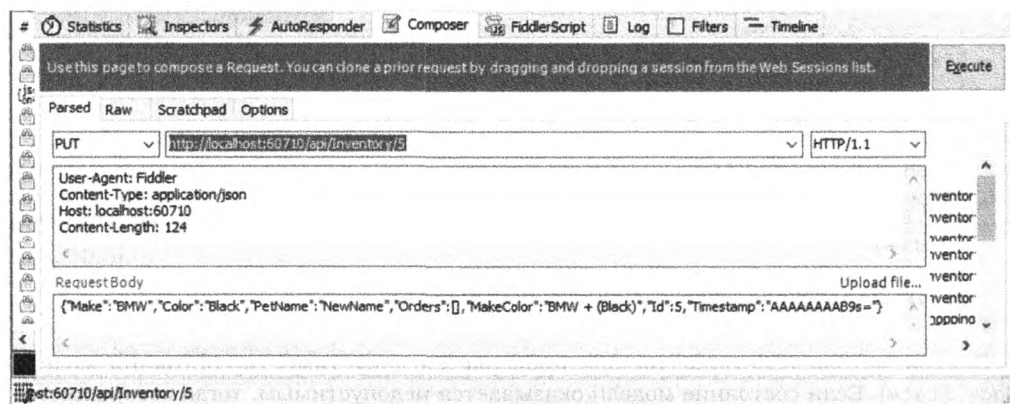


Рис. 30.6. Создание запроса PUT в Fiddler

После выполнения запроса можно посредством Fiddler запустить запрос GET для подтверждения факта принятия изменений.

Добавление складских записей (HttpPost)

Добавление записей в HTTP производится с использованием вызовов метода `HttpPost` и передачей ему добавляемого объекта в теле сообщения. Метод `PostInventory()` также применяет привязку модели для создания экземпляра класса `Inventory` из тела сообщения. Добавим следующий код:

```
// POST: api/Inventory
[HttpPost, Route("")]
[ResponseType(typeof(Inventory))]
public IHttpActionResult PostInventory(Inventory inventory)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
}
```

```

try
{
    _repo.Add(inventory);
}
catch (Exception ex)
{
    // В производственном приложении здесь должны быть дополнительные действия
    throw;
}
return CreatedAtRoute("DisplayRoute", new { id = inventory.Id }, inventory);
}

```

Действие `PostInventory()` устанавливается в `ResponseType` типа `Inventory`, т.к. оно возвращает добавленную запись `Inventory`, дополненную сгенерированными сервером значениями. Кроме того, данный метод действия использует привязку модели для получения значений из тела сообщения, проверяет `ModelState` и возвращает `HttpBadRequest (400)`, если во время привязки модели возникли проблемы. В случае успешной привязки модели метод пытается добавить новую запись. Если добавление прошло успешно, тогда возвращается `HttpCreated (201)` с новой записью `Inventory` в теле сообщения. Метод `CreatedAtRoute()` добавляет URI нового элемента в заголовок, так что при необходимости клиент сможет с ним работать.

Протестируем созданный метод тем же самым способом, каким тестировался метод `HttpPut`. Изменим тип запроса на `POST` и уберем свойства `Id` и `Timestamp` из тела сообщения JSON. При выполнении запроса `POST` посмотрим заголовки и тело сообщения ответа.

В ответе должны находиться следующие заголовки (или, по крайней мере, похожие):

```

HTTP/1.1 201 Created
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Location: http://localhost:60710/api/Inventory/10

```

Текстовое представление покажет полный код JSON созданной записи, включая свойства `Id` и `Timestamp`.

Удаление складских записей (`HttpDelete`)

Осталось создать метод действия `DeleteInventory()`. Добавим два параметра, `id` целочисленного типа и `inventory` типа `Inventory`; последний будет заполняться из тела сообщения. Вот код:

```

// DELETE: api/Inventory/5
[HttpDelete, Route("{id}")]
[ResponseType(typeof(void))]
public IHttpActionResult DeleteInventory(int id, Inventory inventory)
{
    if (id != inventory.Id)
    {
        return BadRequest();
    }
    try
    {
        _repo.Delete(inventory);
    }
}

```

```

catch (Exception ex)
{
    // В производственном приложении здесь должны быть дополнительные действия
    throw;
}
return Ok();
}

```

Как видите, метод имеет такую же структуру, как предшествующие методы.

Итоговые сведения о Web API

Инфраструктура ASP.NET Web API задействует порядочную часть инфраструктуры ASP.NET MVC, а также дает вам возможность применить свои знания в области построения веб-приложений и служб REST с использованием ASP.NET.

С помощью Web API можно решать гораздо больше задач, но объем книги ограничен. Настоящая глава указала вам правильное направление для движения, а дополнительные сведения доступны по адресу:

<https://docs.microsoft.com/ru-ru/aspnet/web-api/>.

Инфраструктуру ASP.NET Web API можно применять различными способами — как автономную службу, как серверную часть для интерфейсных частей JavaScript (вроде Angular или React) или как серверную часть для приложений MVC. Вы увидите это в действие далее.

Исходный код. Решение CarLotWebAPI доступно в подкаталоге Chapter_30.

Обновление проекта CarLotMVC для использования CarLotWebAPI

Внутри сайта ASP.NET MVC по имени CarLotMVC из главы 29 применялась библиотека AutoLotDAL для всех операций CRUD. В текущем разделе мы заменим ее только что построенным проектом CarLotWebAPI.

Добавление приложения CarLotMVC

Вы можете продолжить работу с тем же решением, которое было создано в начале главы. Для вашего удобства в загружаемом коде примеров предлагаются два решения: первое представляет собой как раз проект CarLotWebAPI, а второе объединяет все компоненты в приложение CarLotMVC.

Скопируем папку CarLotMVC (из Chapter_29) в папку CarLotWebAPI и добавим ее в решение. Установим проекты CarLotWebAPI и CarLotMVC как запускаемые в указанном порядке, для чего щелкнем правой кнопкой мыши на имени решения, выберем в контекстном меню пункт Set Startup Projects (Установить как запускаемые проекты) и приведем диалоговое окно к виду, показанному на рис. 30.7.

Удалим ссылки на проект AutoLotDAL из проекта CarLotMVC. Нам понадобится только доступ к моделям. Наконец, откроем файл Global.asax.cs из проекта MVC и удалим строку, в которой устанавливается DatabaseInitializer, а также строку `using AutoLotDAL.EF`.

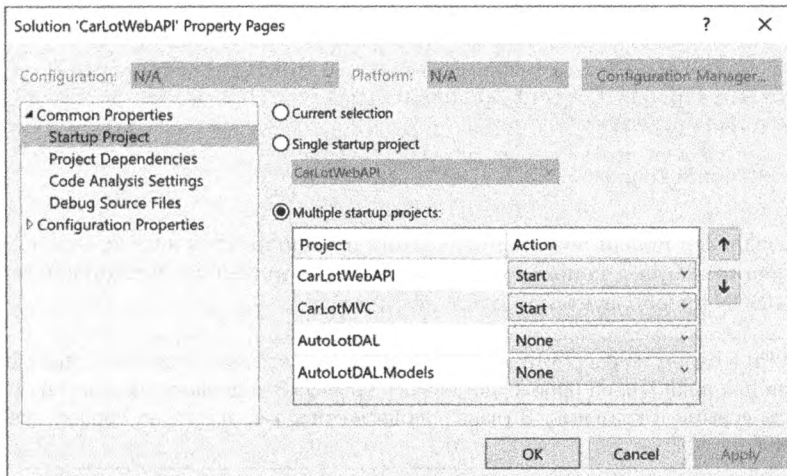


Рис. 30.7. Установка множества запускаемых проектов

Обновление класса InventoryController в проекте MVC

В классе InventoryController внутри проекта MVC используется класс InventoryRepo, и мы удалим все ссылки на него.

Удалим оператор using для AutoLotDAL.Repos и добавим приведенные ниже операторы using:

```
using System.Collections.Generic;
using System.Net.Http;
using System.Threading.Tasks;
using AutoLotDAL.Models;
using Newtonsoft.Json;
using System.Text;
```

Также добавим переменную уровня класса, которая будет хранить базовый URL для сайта Web API (в своем проекте укажите корректный номер):

```
private string _baseUrl = "http://localhost:60710/api/Inventory";
```

Каждый метод будет обновлен, чтобы обращаться к службе Web API с применением класса HttpClient. Методы Web API возвращают реализацию IActionResult, которая имеет два свойства, принимающие участие в процессе: IsSuccessStatusCode и Content.

Свойство IsSuccessStatusCode получает значение true, если вызов сработал. Оно устраняет необходимость в проверке всех возможных возвращаемых кодов, т.к. в зависимости от типа обращения многие из них считаются успешным завершением. Свойство Content предоставляет доступ к телу сообщения. Каждый метод потребуется сделать асинхронным (async) и возвращающим Task<ActionResult>.

Обновление метода действия Index ()

Обновим метод действия Index() следующим образом:

```
public async Task<ActionResult> Index()
{
    var client = new HttpClient();
    var response = await client.GetAsync(_baseUrl);
```

```

    if (response.IsSuccessStatusCode)
    {
        var items = JsonConvert.DeserializeObject<List<Inventory>>(
            await response.Content.ReadAsStringAsync());
        return View(items);
    }
    return HttpNotFound();
}

```

В коде создается новый объект `HttpClient` и выполняется запрос `GET`. Если запрос прошел успешно, тогда с использованием пакета `Json.NET` от `Newtonsoft` содержимое сообщения `JSON` преобразуется в `List<Inventory>`.

На заметку! Ни в одном из рассмотренных примеров не обеспечивается обработка ошибок, необходимая для приложения производственного уровня. Это сделано намеренно, чтобы примеры были ясными и краткими. В главе 7 объяснялось, как элегантно обрабатывать любые исключения.

Обновление метода действия `Details()`

Модифицируем код метод действия `Details()`, как показано ниже:

```

// GET: Inventory/Details/5
public async Task<ActionResult> Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    var client = new HttpClient();
    var response = await client.GetAsync($"({_baseUrl}/{id.Value}");
    if (response.IsSuccessStatusCode)
    {
        var inventory = JsonConvert.DeserializeObject<Inventory>(
            await response.Content.ReadAsStringAsync());
        return View(inventory);
    }
    return HttpNotFound();
}

```

Главным изменением здесь (как и в методе действия `Index()`) является получение записи с применением нового экземпляра `HttpClient`. Проверяется код состояния ответа, и если вызов прошел успешно, то с помощью `Json.NET` содержимое сообщения десериализуется в объект `Inventory`, после чего возвращается представление.

Обновление метода действия `Create()`

Первый метод действия `Create()` в обновлении не нуждается, поскольку он загружает представление без какого-либо взаимодействия с базой данных. Версия `HttpPost` требует модификации. Вот полный код метода действия:

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult>
    Create([Bind(Include = "Make,Color,PetName")] Inventory inventory)
{

```

```

if (!ModelState.IsValid) return View(inventory);
try
{
    var client = new HttpClient();
    string json = JsonConvert.SerializeObject(inventory);
    var response = await client.PostAsync(_baseUrl,
        new StringContent(json, Encoding.UTF8, "application/json"));
    if (response.IsSuccessStatusCode)
    {
        return RedirectToAction("Index");
    }
}
catch (Exception ex)
{
    ModelState.AddModelError(string.Empty, $"Unable to create record:
        {ex.Message}"); // Не удастся создать запись
}
return View(inventory);
}

```

Метод `PostAsync()` требует определенного форматирования тела сообщения. Это достигается преобразованием объекта `Inventory` в формат JSON и последующим созданием объекта `StringContent`, который затем передается методу `PostAsync()` вместе с URI для выполнения вызова. Если все проходит успешно, то пользователь перенаправляется на представление `Index`.

Обновление метода действия `Edit()`

Оба метода действия `Edit()` нуждаются в обновлении. Версия `HttpGet` изменяется аналогично изменению метода действия `Details()`:

```

public async Task<ActionResult> Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    var client = new HttpClient();
    var response = await client.GetAsync($"{_baseUrl}/{id.Value}");
    if (response.IsSuccessStatusCode)
    {
        var inventory = JsonConvert.DeserializeObject<Inventory>(
            await response.Content.ReadAsStringAsync());
        return View(inventory);
    }
    return new HttpNotFoundResult();
}

```

Версия `HttpPost` модифицируется почти так же, как метод действия `Create()`, но только в ней вместо вызова `PostAsync()` используется вызов `PutAsync()`:

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Edit([Bind(Include =
    "Id,Make,Color,PetName,Timestamp")] Inventory inventory)
{
    if (!ModelState.IsValid) return View(inventory);
}

```



```

var client = new HttpClient();
string json = JsonConvert.SerializeObject(inventory);
var response = await client.PutAsync($"{_baseUrl}/{inventory.Id}",
    new StringContent(json, Encoding.UTF8, "application/json"));
if (response.IsSuccessStatusCode)
{
    return RedirectToAction("Index");
}
return View(inventory);
}

```

На заметку! Вас может интересовать, почему метод действия в контроллере MVC помечен атрибутом `HttpPost`, но обращение к веб-службе производится с помощью запроса `PUT`. Важно понимать, что метод `HTTP`, применяемый для вызова действий MVC, не обязан совпадать с методом `HTTP`, который используется для обращения к методам действий `Web API`. Они являются разрозненными операциями.

Обновление метода действия `Delete()`

Существуют два метода действий `Delete()` и подобно версии `HttpGet` метода `Edit()` единственное изменение касается обращения к веб-службе для получения данных:

```

// GET: Inventory/Delete/5
public async Task<ActionResult> Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    var client = new HttpClient();
    var response = await client.GetAsync($"{_baseUrl}/{id.Value}");
    if (response.IsSuccessStatusCode)
    {
        var inventory = JsonConvert.DeserializeObject<Inventory>(
            await response.Content.ReadAsStringAsync());
        return View(inventory);
    }
    return new HttpNotFoundResult();
}

```

Версия `HttpPost` метода действия `Delete()` требует дополнительной работы. Как и можно было предположить, на самом деле имеется расширяющий метод `DeleteAsync()` класса `HttpClient`, но он не принимает каких-либо параметров для содержимого из тела сообщения. Применение этого метода приведет к отказу запроса на удаление, потому что должно передаваться значение отметки времени для проверки параллелизма. Взамен объект `HttpRequestMessage` придется создавать вручную. Ниже показан код:

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Delete([Bind(Include = "Id, Timestamp")]
    Inventory inventory)
{
    try
    {
        var client = new HttpClient();

```

```

HttpRequestMessage request =
    new HttpRequestMessage(HttpMethod.Delete, $"{_baseUrl}/{inventory.Id}")
    {
        Content =
            new StringContent(JsonConvert.SerializeObject(inventory),
                Encoding.UTF8, "application/json")
    };
var response = await client.SendAsync(request);
return RedirectToAction("Index");
}
catch (Exception ex)
{
    ModelState.AddModelError(string.Empty, $"Unable to create record:
        {ex.Message}");
}
return View(inventory);
}

```

Конструктор `HttpRequestMessage` принимает в первом параметре `HttpMethod`, а во втором — URL. Инфраструктура `Json.NET` сериализует объект `Inventory`, поэтому он может быть добавлен в экземпляр `StringContent`, что является частью инициализации объектов для сообщения. Если все прошло успешно, тогда пользователь перенаправляется обратно на представление `Index`.

Исходный код. Скомбинированное решение `CarLotWebAPI` и `CarLotMVC` доступно в подкаталоге `CarLotWebAPI` внутри `Chapter_30`.

Резюме

В главе были исследованы многие аспекты Web API. Демонстрировалось создание нового проекта Web API, добавление контроллера, использование маршрутизации с помощью атрибутов и тестирование посредством инструмента Fiddler. Кроме того, для устранения ряда проблем, связанных с сериализацией моделей Entity Framework, применялась утилита AutoMapper.

После завершения действий `InventoryController` в проекте `CarLotWebAPI` мы обновили `CarLotMVC` с целью обращения к URL службы `CarLotWebAPI`, используя `Json.NET` для сериализации и десериализации записей. Также было показано, как делать вызовы с применением дополнительных методов HTTP, используемых инфраструктурой Web API.

На этом рассмотрение инфраструктуры .NET Framework 4.7 завершено. В следующей части книги вы приступите к изучению .NET Core, Entity Framework Core и ASP.NET Core.

часть IX

.NET Core

В этой части

Глава 31. Философия .NET Core

Глава 32. Введение в Entity Framework Core

Глава 33. Введение в веб-приложения ASP.NET Core

Глава 34. Введение в приложения служб ASP.NET Core

ГЛАВА 31

Философия .NET Core

27 июня 2016 года в Microsoft анонсировали выход .NET Core версии 1.0 — кардинально новой платформы для всех разработчиков .NET. Новая платформа была выпущена одновременно для операционных систем (ОС) Windows, macOS и Linux. Она основывалась на языке C# и платформе .NET Framework, которую вы изучали на протяжении предшествующих тридцати глав. Начальный выпуск включал исполняющую среду .NET Core, ASP.NET Core и Entity Framework Core. С тех пор появились два дополнительных выпуска: 1.1 и 2.0 (текущая версия).

Вдобавок к возможностям межплатформенного развертывания было внесено еще одно значительное изменение. Платформа .NET Core и связанные инфраструктуры являются системами с открытым кодом. Они не только предлагают свободный доступ к своему исходному коду, который можно просматривать и проводить с ним эксперименты, но представляют собой полноценные проекты с открытым кодом. Запросы на изменение принимались и фактически даже ожидались. Вклад со стороны сообщества разработчиков был весьма существенным — в первоначальном выпуске .NET Core участвовало более 10 000 разработчиков. То, что было относительно небольшой командой разработчиков в Microsoft, строящей .NET и связанные инфраструктуры, превратилось в крупное сообщество по разработке, которое теперь способно предоставлять дополнительную функциональность, улучшения производительности и исправления дефектов. Такая инициатива открытого кода охватывает не только программное обеспечение, но и документацию тоже. В действительности все ссылки на документацию, которые приводились в предшествующих главах, указывали на новую платформу документации с открытым кодом, находящуюся по адресу <http://docs.microsoft.com>.

В настоящей главе предлагается введение в философию .NET Core. В последующих трех главах раскрываются Entity Framework Core (глава 32) и ASP.NET Core (главы 33 и 34).

От проекта Project K до .NET Core

Как отметил Рич Лэндер в блоге .NET Blog (<https://blogs.msdn.microsoft.com/dotnet/2016/06/27/announcing-net-core-1-0/>), в Microsoft получили значительное количество запросов относительно функционирования ASP.NET на платформах, отличающихся от Windows. Исходя из этих запросов и параллельных обсуждений с командой разработчиков для Windows, касающихся Windows Nano, родился проект Project K, ориентированный на дополнительные платформы.

Начальные усилия были сосредоточены на ASP.NET. Для запуска под управлением других ОС требовалось устранить зависимость от System.Web, что было нетривиальным и по существу привело бы к переписыванию инфраструктуры. Нужно было принять решение о том, что будет включено. При исследовании разных платформ ASP.NET (Web Forms, ASP.NET MVC и ASP.NET Web API) выяснилось, что наилучшим подходом

было создание одной платформы, которая могла бы использоваться для веб-приложений и служб REST. Платформы MVC и Web API в конечном итоге были объединены в одну инфраструктуру (вместо того, чтобы быть тесно связанными родственниками как в полной платформе .NET Framework). Новая платформа для разработки веб-приложений сначала называлась ASP.NET 5.

Новой веб-платформе понадобился уровень доступа к данным, и очевидным выбором была инфраструктура Entity Framework. По существу все также сводилось к переписыванию версии EF 6, и было выбрано начальное название Entity Framework 7.

Откровенно говоря, по нашему мнению это наихудшие названия, какие только можно было выбрать. Трудно сказать, сколько времени было потрачено нами на объяснение заказчикам и участникам конференций, что ASP.NET 5 не является следующей версией полной платформы .NET Framework, даже если 5 больше 4.6 (в тот момент текущая версия полной платформы .NET Framework), а EF 7 — не следующая версия EF, даже если 7 больше 6.

Хорошая новость в том, что недоразумения были устранены, а названия межплатформенных версий .NET получили добавку Core, как в .NET Core, ASP.NET Core и Entity Framework Core. Такое разграничение привело в порядок большую часть неразберихи (хоть и не всю), связанной с различными инфраструктурами и их местом в экосистеме разработки.

Будущее полной платформы .NET Framework

Что можно сказать о полной платформе .NET Framework и связанных инфраструктурах? Поддерживаются ли они? Придется ли переписывать все ранее созданные приложения до того, как поддержка прекратится? По обоим пунктам — безусловно, нет! Данная книга должна развеять любые сомнения относительно продолжения разработки C# и .NET. Если бы платформа .NET не обновлялась, то книга не добралась бы до 8-го издания, раскрывающего C# 7.1 и .NET 4.7.

Продолжают поступать дополнительные вопросы о будущем разнообразных инфраструктур, таких как Windows Forms, Web Forms, WPF, WFC и т.д. Надо сказать, каких-то потрясающих новостей, связанных с многочисленными инфраструктурами .NET, было не сильно много, но это вовсе не означает, что работа над ними была прекращена. Можно утверждать, что такие технологии “закончены”. Инфраструктуры Windows Forms и Web Forms все еще широко применяются в наши дни, и отсутствие описания упомянутых инфраструктур в основных главах книги совершенно не означает, что мы не верим в их жизнеспособность; просто нам нечего сообщать сверх того, что приводилось в предшествующих изданиях.

На заметку! По-прежнему остается вопрос, какой будет следующая старшая версия полной платформы .NET. Логичным номером следующей версии является 5, но, к сожалению, данная карта сыграна. Мы уверены, что со временем все прояснится, а настоящая книга поможет понять текущее положение дел.

Цели .NET Core

Запуск под управлением ОС, отличающихся от Windows, был достаточно высокой целью для новой платформы, но далеко не единственной. Примерно с 2002 года .NET существовала как производственная инфраструктура, и с той поры в нее были внесены многочисленные изменения. Разработчики стали умнее, компьютеры — быстрее, требования пользователей возросли, а мобильность превратилась в доминирующую силу (и это лишь несколько аспектов).

Ниже перечислены некоторые цели .NET Core.

- *Межплатформенная поддержка.* Платформа .NET Core способна функционировать в средах Windows, Linux и macOS. Приложения .NET Core могут строиться на разных платформах с помощью Visual Studio Code или Visual Studio для Mac. Кроме того, включение Xamarin добавляет iOS и Android к числу платформ, поддерживаемых для развертывания.
- *Производительность.* Производительность .NET Core постоянно приближается к вершине всех важных графиков производительности, и в каждый выпуск вносятся дополнительные усовершенствования.
- *Переносимые библиотеки классов,* пригодные к потреблению всеми исполняющими средами .NET. В .NET Core введен стандарт .NET Standard — формальная спецификация, устанавливающая согласованное поведение исполняющей среды .NET.
- *Переносимое или автономное развертывание.* Приложения .NET Core могут развертываться параллельно с инфраструктурой либо использовать установленную копию .NET Core на уровне машины.
- *Полная поддержка командной строки.* Платформа .NET Core сосредоточена на полной поддержке командной строки в качестве основной цели.
- *Открытый код.* Как упоминалось, платформа .NET Core и ее документация являются системами с открытым кодом, укомплектованными запросами на изменение от мирового сообщества разработчиков.
- *Возможность взаимодействия с полной платформой .NET Framework.* Выпуск 2.0 платформы .NET Core разрешает ссылаться на библиотеки .NET Framework.

Мы уверены, что разработчики из Microsoft просмотрели существующую кодовую базу полной платформы .NET Framework и с учетом заявленных целей осознали невозможность изменения полной платформы .NET Framework при сохранении работоспособности всех производственных приложений. Явно требовалось целиком переписать код платформы .NET, а не только ASP.NET.

Давайте теперь рассмотрим каждую цель подробнее.

Межплатформенная поддержка

Ранее говорилось, что приложения .NET Core не ограничиваются выполнением в средах ОС, основанных на Windows. Это открывает широкий спектр вариантов для развертывания (и разработки) приложений с применением .NET и C#. Первоначальный выпуск .NET Core поддерживал ОС Windows (само собой разумеется), macOS и несколько дистрибутивов Linux, а также iOS и Android посредством Xamarin. В каждый последующий выпуск включались новые дистрибутивы и расширенная поддержка версий Linux. Список поддерживаемых ОС доступен по адресу <https://github.com/dotnet/core/blob/master/roadmap.md#technology-roadmaps>.

Вас может интересовать, каким образом межплатформенная версия .NET может помочь. В конце концов, возможно, вы уже посвятили себя ОС Windows и разработке на машине Windows (или на машине Mac с использованием технологии виртуализации).

Разработка приложений .NET Core где угодно

Имея дело с .NET Core, строить мобильные и веб-приложения можно где угодно. Среда Visual Studio по-прежнему выполняется на машине Windows (конечно же), но теперь доступна версия Visual Studio для Mac, так что разрабатывать приложения .NET Core можно прямо на Mac. А благодаря основанной на файлах системе проектов (которая вскоре будет раскрыта), появляется возможность применения при разработке приложений .NET Core межплатформенных инструментов вроде Visual Studio Code.

Дополнительные варианты развертывания

Наряду с тем, что опытные разработчики .NET уже могли запускать свои приложения под управлением Windows, платформа .NET Core привносит дополнительные варианты развертывания. Поддерживать серверы Linux обычно дешевле, чем серверы Windows, особенно в облаке, так что наличие возможности развертывания веб-приложений и служб в Linux может вылиться в существенную экономию финансовых средств. Главным образом это благоприятно для стартапов и компаний, желающих сократить количество серверов Windows.

Контейнеризация

В дополнение к поддержке дистрибутивов Linux платформа .NET Core поддерживает контейнеризацию приложений .NET Core. Популярные поставщики контейнеров наподобие Docker значительно уменьшили сложность выпуска приложений в разных средах (например, от разработки до тестирования). Приложение и необходимые файлы времени выполнения (включая ОС) упаковываются в один контейнер, который затем копируется из одной среды в другую, не беспокоясь об установке. Благодаря добавлению в Windows и Azure поддержки Docker положение дел стало еще лучше.

Теперь вы можете вести разработку в контейнере Docker и по готовности приложения к интеграционному тестированию просто перемещать контейнер из одной среды в другую. Никакого процесса установки или сложного развертывания! Когда приложение готово для эксплуатации в производственной среде, вы всего лишь переносите контейнер на производственный сервер. Проблемы с выпуском производственных версий, когда приложения отказывались функционировать в целевой среде, хотя разработчики гордо заявляли о том, что они успешно работали на их машинах, остались в прошлом.

Производительность

Приложения ASP.NET Core постоянно приближаются к вершине всех эталонных тестов. Причина в том, что в рамках проектных целей производительность считается “полноправным гражданином”. За прошедшие пятнадцать лет полная платформа .NET Framework крайне разрослась, поэтому выжать максимальную производительность из трудного в изменении кода в лучшем случае нелегко.

Поскольку .NET Core в значительной степени является переписанной версией платформы и инфраструктур, у ее создателей была возможность заблаговременно обдумать вопросы оптимизации и помнить о них во время разработки, а не после того, как платформа готова. Здесь самое важное кроется в слове “заблаговременно”. Производительность теперь — неотъемлемая часть архитектурных и проектных решений.

Переносимые библиотеки классов, соответствующие стандарту .NET Standard

Стандарт .NET Standard представляет собой формальную спецификацию для API-интерфейсов .NET, которые доступны всем исполняющим средам .NET. Он предназначен для обеспечения большей согласованности в экосистеме .NET. Благодаря стандарту .NET Standard становятся возможными следующие основные сценарии.

- Определение унифицированного набора API-интерфейсов в библиотеке базовых классов для всех реализаций .NET независимо от рабочей нагрузки.
- Появление у разработчиков возможности производить переносимые библиотеки, пригодные к потреблению различными реализациями .NET.
- Сокращение (или устранение) условной компиляции разделяемых ресурсов.

Любая сборка, нацеленная на специфическую версию .NET Standard, будет доступна любой другой сборке, нацеленной на ту же самую версию .NET Standard. Разные реализации .NET нацелены на специфические версии .NET Standard. С дополнительными сведениями о стандарте .NET Standard, а также о совместимости между платформами, инфраструктурами и версиями можно ознакомиться по адресу <https://docs.microsoft.com/ru-ru/dotnet/standard/net-standard>.

Переносимые или автономные модели развертывания

В отличие от полной версии .NET Framework платформа .NET Core поддерживает подлинную установку бок о бок. Установка добавочных версий .NET Core не затрагивает существующие установленные версии или приложения. В итоге спектр моделей развертывания для приложений .NET Core расширяется.

При переносимой модели развертывания приложение конфигурируется согласно версии .NET Core, установленной на целевой машине, а в данные развертывания включаются только пакеты, специфичные для приложения. Такой подход обеспечивает небольшой размер установочного пакета, но требует наличия на машине установленной копии целевой версии инфраструктуры.

При автономной модели развертывания пакет содержит все файлы приложения, а также обязательные файлы CoreFX и CoreCLR для целевой платформы. Такой подход очевидным образом увеличивает размер установочного пакета, но изолирует приложение от любых проблем, связанных с машиной (таких как отсутствие нужной версии .NET Core).

Полная поддержка командной строки

Понимая, что при разработке приложений .NET Core далеко не все будут использовать Visual Studio, в команде приняли решение сконцентрироваться на поддержке командной строки, прежде чем снабжать инструментами Visual Studio. В каждом выпуске до .NET Core 2.0 инструментарий отставал от того, что можно было делать с помощью текстового редактора и интерфейса командной строки .NET Core.

Инструментарий совершил большой шаг в своем развитии с момента выхода .NET Core 1.0 (когда он даже не был RTM), но все еще отстает от инструментальных средств Visual Studio, доступных для разработки приложений .NET Framework.

Открытый код

Как уже упоминалось, компания Microsoft выпустила платформу .NET Core и связанные инфраструктуры как подлинные системы с открытым кодом. Над первоначальным выпуском трудилось свыше 10 000 участников, и их число продолжает расти.

Возможность взаимодействия с .NET Framework

С выходом Visual Studio 15.3 и .NET Core 2.0 появилась возможность ссылаться из библиотек .NET Standard на библиотеки .NET Framework. Такое обновление улучшает механизм для переноса кода существующих приложений на платформу .NET Core.

Разумеется, есть ряд ограничений. В сборках, на которые можно ссылаться, должны применяться только типы, поддерживаемые стандартом .NET Standard. Хотя у вас может получиться работать со сборкой, использующей API-интерфейсы, которые отсутствуют в .NET Standard, данный сценарий не является поддерживаемым и должен сопровождаться всесторонним тестированием.

Состав .NET Core

В общих терминах платформа образована из четырех основных частей:

- исполняющая среда .NET Core;
- набор библиотек инфраструктуры;
- инструменты SDK и хост приложений dotnet;
- компиляторы языков.

Исполняющая среда .NET Core (CoreCLR)

Это базовая библиотека для .NET Core. Она включает сборщик мусора, компилятор JIT, базовые типы .NET и множество низкоуровневых классов. Исполняющая среда предоставляет мост между библиотеками инфраструктуры .NET Core (CoreFX) и лежащими в основе ОС. В ее состав включены только типы, которые имеют строгую зависимость от внутренней работы исполняющей среды. Большая часть библиотеки классов реализована в виде независимых пакетов NuGet.

При проектировании CoreCLR разработчики пытались свести к минимуму объем реализованного кода, оставляя специфические реализации многих классов инфраструктуры на CoreFX. В результате получилась небольшая гибкая кодовая база, которую можно модифицировать и быстро развертывать для исправления дефектов или добавления функциональных средств. Сама по себе среда CoreCLR делает не особо много работы. Любой определенный в ней библиотечный код скомпилирован в сборку System.Private.CoreLib.dll, которая не предназначена для потребления за пределами CoreCLR или CoreFX.

В состав дополнительного инструментария, предлагаемого CoreCLR, входит ILDASM и ILASM (версии .NET Core программ, которые вы применяли ранее в книге), а также хост тестирования — небольшая оболочка для запуска DLL-библиотек IL из командной строки.

Библиотеки инфраструктуры (CoreFX)

Это набор фундаментальных библиотек, включающий классы для коллекций, файловых систем, консоли, разметки XML, асинхронной работы и многих других элементов. Библиотеки инфраструктуры построены поверх CoreCLR и предоставляют для других инфраструктур интерфейсные точки в исполняющую среду. Помимо специфических реализаций CoreCLR, содержащихся в CoreFX, остальные библиотеки имеют независимую от исполняющей среды и платформы природу. В CoreCLR находится mscorlib.dll — открытый фасад CoreCLR. Вместе CoreCLR и CoreFX образуют .NET Core.

Инструменты SDK и хост приложений dotnet

Включенные в SDK инструменты представляют собой интерфейс командной строки (command-line interface — CLI) платформы .NET, используемый для построения приложений и библиотек .NET Core. Хост приложений dotnet является универсальным драйвером для выполнения команд CLI, включая приложения .NET Core.

Команды CLI и приложения .NET Core запускаются с применением хоста приложений dotnet. Чтобы увидеть его в действии, необходимо открыть окно командной строки (оно не обязательно должно быть командной подсказкой для разработчиков из Visual Studio) и ввести следующую команду:

```
dotnet --version
```

Команда запускает хост приложений dotnet и отображает установленную в текущий момент и доступную через переменную среды PATH версию .NET Core SDK. С помощью CLI можно вводить много готовых команд, самые распространенные из которых перечислены в табл. 31.1.

Таблица 31.1. Распространенные команды CLI в .NET Core

Команда	Описание
--help	Выводит справочную информацию
--version	Выводит версию .NET Core SDK
--info	Выводит версию инструментов командной строки .NET Core и разделяемого хоста инфраструктуры
new	Инициализирует пример консольного приложения .NET Core
restore	Восстанавливает зависимости для заданного приложения. В версии 2.0 неявно выполняется с командами build и run
build	Строит приложение .NET Core
clean	Очищает вывод проекта
publish	Публикует переносимое или независимое приложение .NET
run	Запускает приложение из исходного кода
test	Прогоняет тесты, используя указанное средство выполнения тестов
pack	Создает пакет NuGet из исходного кода

В дополнение к встроенным командам CLI инфраструктуры могут добавлять собственные команды. Например, инфраструктура Entity Framework Core вводит серию команд для миграций и обновлений базы данных, которые будут применяться в следующей главе.

Компиляторы языков

Платформа .NET Compiler ("Roslyn") предоставляет компиляторы с открытым кодом языков C# и Visual Basic с развитыми API-интерфейсами для анализа кода. Несмотря на то что в выпуск .NET Core 2.0 была добавлена некоторая поддержка Visual Basic, язык C# по-прежнему считается основным и поддерживается во всех инфраструктурах .NET Core.

Жизненный цикл поддержки .NET Core

Платформа .NET Core поддерживается компанией Microsoft, хотя она и с открытым кодом. Каждый выпуск имеет определенный жизненный цикл и разделяется на две категории: долгосрочная поддержка (Long Term Support — LTS) и текущая (Current), ранее называемая ускоренной поддержкой (Fast Track Support). Каждый старший или младший выпуск, как правило, будет поддерживаться на протяжении трех лет, пока исправления сохраняются текущими (скажем, 1.0.1), с несколькими исключениями, которые обсуждаются ниже.

Выпуски LTS:

- представляются старшими номерами версий (например, 1.0, 2.0);
- поддерживаются в течение трех лет после общей доступности (general availability — GA) выпуска LTS;
- или поддерживаются в течение одного года после следующего выпуска LTS (какой бы он ни был коротким).

Выпуски Current:

- представляются младшими номерами версий (например, 1.1, 1.2);
- поддерживаются в рамках того же самого трехлетнего окна, как и родительский выпуск LTS;
- поддерживаются в течение трех месяцев после общей доступности последующего выпуска Current;
- или поддерживаются в течение одного года после общей доступности последующего выпуска LTS.

Если все выглядит сложным и запутанным, то не рассчитывайте найти у нас какие-то аргументы против этого. Лицензирование продуктов Microsoft никогда не было простым, и .NET Core ничем в данном плане не отличается. Отправьте свою команду юристов по адресу <https://www.microsoft.com/net/core/support> за дополнительной информацией и обеспечьте соблюдение всех условий.

На заметку! В пояснительной записке к выпуску .Net Core 2.0 упоминалось, что версия .NET Core 1.1 присоединилась к .NET Core 1.0 как продукт с поддержкой LTS, а не Current.

Установка .NET Core 2.0

Если вы еще не установили продукт Visual Studio 2017, тогда установите в соответствии с инструкциями из главы 2 (по крайней мере) такие рабочие нагрузки:

- .NET desktop development (Разработка настольных приложений .NET);
- ASP.NET and web development (Разработка приложений ASP.NET и веб-приложений);
- .NET Core cross-platform development (Разработка межплатформенных приложений .NET Core).

Для использования .NET Core 2.0 требуется располагать, по меньшей мере, версией 15.3 продукта Visual Studio 2017. Вдобавок на время написания настоящей главы комплект .NET Core 2.0 SDK должен был устанавливаться отдельно и загружаться по адресу <https://dot.net/core>.

После установки Visual Studio 15.3 и .NET Core 2.0 SDK нужно открыть окно командной строки и ввести следующую команду:

```
dotnet --version
```

В окне консоли должна отобразиться версия 2.0.0 (или выше, что зависит от того, когда вы читаете эту книгу). Затем понадобится ввести команду:

```
dotnet --info
```

В окне консоли должны быть выведены такие данные:

```
.NET Command Line Tools (2.0.0)
Product Information:
  Version:           2.0.0
  Commit SHA-1 hash: cdc1928c9

Runtime Environment:
  OS Name:            Windows
  OS Version:         10.0.16257
  OS Platform:        Windows
  RID:                win10-x64
  Base Path:          C:\Program Files\dotnet\sdk\2.0.0\

Microsoft .NET Core Shared Framework Host
  Version   :    2.0.0
  Build    :   e8b8861ac7faf042c87a5c2f9f2d04c98b69f28d
```

Сведения подтвердят, что на машине установлены подходящие версии для финальных глав книги.

Сравнение с полной платформой .NET Framework

Наряду с тем, что в .NET Core применяются язык C# и многие из тех же API-интерфейсов, которые присутствуют в полной платформе .NET Framework, существует несколько важных отличий:

- расширенная поддержка платформы;
- открытый код;
- сниженное количество поддерживаемых моделей приложений;
- меньшее число реализованных API-интерфейсов и подсистем.

Первые два отличия уже были раскрыты ранее в главе, а оставшиеся два рассматриваются в последующих разделах.

Сниженное количество поддерживаемых моделей приложений

Ранее упоминалось, что .NET Core не поддерживает все модели приложений .NET Framework, особенно те, которые построены прямо на основе технологий Windows. Кроме того, любое приложение .NET Core будет либо консольным приложением, либо библиотекой классов. Даже приложения ASP.NET Core являются просто консольными приложениями, которые создают веб-хост, как будет показано в главах 33 и 34.

Меньшее число реализованных API-интерфейсов и подсистем

Полная платформа .NET Framework совершенно статична и попытка изменить все API-интерфейсы одновременно была бы почти неразрешимой задачей. Следуя принципам гибкой разработки, создатели .NET Core определили, какой минимальный жизнеспособный продукт обеспечивал поддержку .NET Core 1.0, и выпустили именно то, чего достаточно для удовлетворения таких требований. Каждый последующий выпуск .NET Core включал многие дополнительные API-интерфейсы из полной платформы .NET Framework, причем выпуск 2.0 реализует свыше 32 000 API-интерфейсов.

Платформа .NET Core реализует только подмножество подсистем из полной платформы .NET Framework, преследуя цель предложить более простую реализацию и модель программирования. Например, наряду с тем, что рефлексия поддерживается, безопасность доступа кода (Code Access Security — CAS) — нет.

Резюме

В настоящей главе была заложена основа для последующих трех глав путем демонстрации сходных черт и отличий между полной платформой .NET Framework и .NET Core. Как будет показано в дальнейших главах, разработка приложений .NET Core похожа на разработку приложений .NET Framework. Приходится иметь дело с меньшим числом инфраструктур и API-интерфейсов, но в остальном это всего лишь код C#.

При проектировании .NET Core преследовались многие цели, и команда создателей (включая мировое сообщество с открытым кодом) за короткое время выполнила потрясающую работу, снабдив разработчиков приложений .NET замечательным набором инструментов, который можно использовать для межплатформенной разработки и развертывания.

На заметку! Последующие три главы, посвященные EF Core 2.0 (глава 32) и ASP.NET Core 2.0 (главы 33 и 34), ограничены по объему того, что можно было бы раскрыть. Более подробное изложение данных тем, а также особенностей применения инфраструктур JavaScript для построения пользовательских интерфейсов с помощью .NET Core можно найти в книге *Building Web Applications with Visual Studio 2017* (<https://www.amazon.com/Building-Applications-Visual-Studio-2017/dp/1484224779/>).

ГЛАВА 32

Введение в Entity Framework Core

Вместе с .NET Core и ASP.NET Core компания Microsoft выпустила инфраструктуру Entity Framework Core, которая представляла собой полностью переписанную версию EF, использующую все ценные свойства .NET Core. Однако в рамках времени, выделенного на выпуск версии .NET Core 1.0, разработчикам попросту не хватило времени, чтобы воссоздать все функциональные средства EF 6. На основе сравнения возможностей можно сказать, что первая версия оказалась слабой. На самом деле нехватка функциональности даже не позволяла считать ее продуктом, готовым к производственной эксплуатации.

С выходом версии EF Core 1.1 ситуация изменилась. В этом выпуске стал доступным больший объем возможностей из EF 6 и также были введены новые средства. Независимо от того, что продукт все еще был незавершенным, он предлагал разработчикам приложения ASP.NET достаточную функциональность, чтобы начать его применять в реальных проектах. В версии EF Core 2.0 появилось еще больше средств, улучшилась производительность и расширилась область использования, не ограничиваясь только ASP.NET. В то время как версия EF Core 1.1 не обладала достаточными возможностями для удовлетворения ваших потребностей, у версии EF Core 2.0 есть шанс.

На заметку! В настоящей главе предполагается наличие у вас опыта работы с Entity Framework 6. Если опыт отсутствует, тогда прежде чем продолжать просмотрите главу 22.

В первой части главы проводится сравнение функциональных средств EF 6 и EF Core 2. Остаток главы посвящен построению версии EF Core 2 библиотеки AutoLotDAL, которая будет применяться в следующих двух главах, посвященных ASP.NET Core 2.

Сравнение наборов функциональных возможностей

Есть те функциональные возможности EF 6, которые переноситься не будут, есть завершенные средства, а есть функциональность, все еще находящаяся в процессе разработки. В версии EF Core присутствуют также новые функциональные возможности и улучшения в плане производительности.

Как упоминалось ранее, EF Core представляет собой полностью новую кодовую базу для инфраструктуры. Это позволило команде разработчиков добавить средства, которые было слишком сложно или дорого реализовать в EF 6. Но вместе с тем команде разработчиков предстоит сделать еще очень многое.

На заметку! В каждый выпуск добавляются новые функциональные возможности, а команда разработчиков старается поддерживать документацию в актуальном состоянии. С результатами сравнения EF 6 и EF Core можно ознакомиться по адресу <https://docs.microsoft.com/ru-ru/ef/efcore-and-ef6/>. Кроме того, доступен план дальнейших действий: <https://docs.microsoft.com/ru-ru/ef/core/what-is-new/roadmap>.

Средства, которые не дублировались

К счастью, список будет коротким. К главным функциональным возможностям, не перенесенным в EF Core, относятся визуальный конструктор EF и средства построения моделей EDMX. Хотя визуальный конструктор удовлетворял потребность на ранней стадии освоения EF, современные разработчики отдают предпочтение парадигме Code First, отказавшись от использования генерируемых конструктором моделей. Объем работы по замене данного средства, применение которого пошло на спад, предотвратил бы выпуск значительного количества других функциональных возможностей. Таким образом, в EF Core поддерживается только парадигма Code First.

На заметку! Как обсуждалось в главе 22, парадигма Code First вовсе не означает невозможность работы с существующей базой данных. На самом деле она означает "ориентацию на код". Строить шаблоны всего кода EF можно по-прежнему из существующей базы данных или создавать базу данных на основе имеющегося кода.

Изменения по сравнению с EF 6

На первый взгляд многие функциональные средства EF Core и EF 6 выглядят одинаковыми. Тем не менее, в EF Core имеется ряд тонких (и не очень тонких) изменений. Инфраструктура EF находилась в производственной эксплуатации с 2008 года, и при переносе в EF Core команда разработчиков получила шанс переписать многие средства с целью их улучшения. Ниже приведен список отличий между EF Core и EF 6.

- Создание и конфигурирование `DbContext` основаны на внедрении зависимостей, а не на простом наследовании.
- Установка является модульной и ориентированной на поставщика.
- Поддерживается меньше аннотаций данных и существует большая опора на Fluent API.
- Внесены изменения в API-интерфейсы `DbSet<T>` и `Database`, такие как добавление методов `Update()` и `FromSql()`.
- Произведены значительные улучшения в плане производительности через более чистый код, а также улучшения запросов вроде пакетирования.

Мы исследуем каждое отличие во время построения версии EF Core сборки `AutoLotDAL`.

Новые функциональные средства в EF Core

В версии EF Core 2.0 есть несколько новых функциональных средств, которые отсутствовали в EF 6. Избранные средства такого рода кратко описаны в табл. 32.1.

Таблица 32.1. Некоторые новые функциональные средства в EF Core

Средство	Описание
Теневые свойства состояния	Свойства, которые не определены в модели, но могут отслеживаться и обновляться инфраструктурой EF Core. Они обычно используются для внешних ключей
Запасные ключи	Предоставляет цели первичных ключей для внешних ключей
Генерация ключей: клиент	Позволяет значениям ключей генерироваться на стороне клиента
Глобальные фильтры запросов	Делает возможными глобальные фильтры запросов, такие как мягкое удаление или идентификатор участника
Поддерживающие поля/ отображение полей	Инфраструктура EF способна выполнять запись в поля вместо свойств
Смешанная клиент-серверная оценка	Позволяет запросам запускать код клиентской стороны, смешанный с кодом серверной стороны
Энергичная загрузка с помощью Include() и ThenInclude()	Улучшенная энергичная загрузка с помощью ThenInclude(), призванная сделать операторы LINQ более чистыми
Низкоуровневые запросы SQL с помощью LINQ	Возвращает отслеживаемые сущности посредством FromSQL()
Пакетирование операторов	Сокращает многословность взаимодействий с базой данных в целях более высокой производительности
Отображение скалярных функций	Отображает скалярную функцию на функцию C# для применения в операторах запросов
Конструкция LIKE для SQL Server (специфичная для поставщика)	Добавляет возможности LIKE в операторы LINQ
Организация пула DbContext	В версии ASP.NET Core организует пул экземпляров DbContext в целях более высокой производительности
Явно скомпилированные запросы	Отображает запрос LINQ на функцию C# для предварительной компиляции и достижения более высокой производительности

Дополнительные сведения по всем перечисленным в табл. 32.1 средствам доступны в документации по адресу <https://docs.microsoft.com/ru-ru/ef/#pivot=efcore>.

Сценарии использования

Инфраструктура EF Core может применяться в любом приложении .NET, даже ориентированном на полную платформу .NET Framework. Хотя версия EF Core построена поверх .NET Core, работа не ограничена только приложениями .NET Core. Инфраструктура EF Core допускает использование с ASP.NET MVC 5, Web API 2.2 и даже Windows Presentation Foundation.

Почему может возникнуть желание применять EF Core с приложениями для полной платформы .NET Framework? Причин много, в числе которых наличие новых функциональных средств, описанных в табл. 32.1. Но главная причина (по нашему мнению) — аспект производительности. Мы использовали EF Core в производственных приложениях MVC 5 и отметили значительное повышение производительности по сравнению с EF 6.

Сказанное подводит нас к следующему моменту: инфраструктуры EF 6 и EF Core способны успешно сосуществовать в рамках одного приложения. Нам нужно поместить контекст EF 6 и контекст EF Core в разные сборки, но они могут разделять те же самые модели.

Создание AutoLotCoreDAL_Core2

Настало время построить версию .NET Core сборки AutoLotDAL.

Создание проектов и решения

Начнем с создания нового проекта библиотеки классов .NET Core по имени AutoLotDAL_Core2 (рис. 32.1). Удалим стандартный файл Class1.cs.

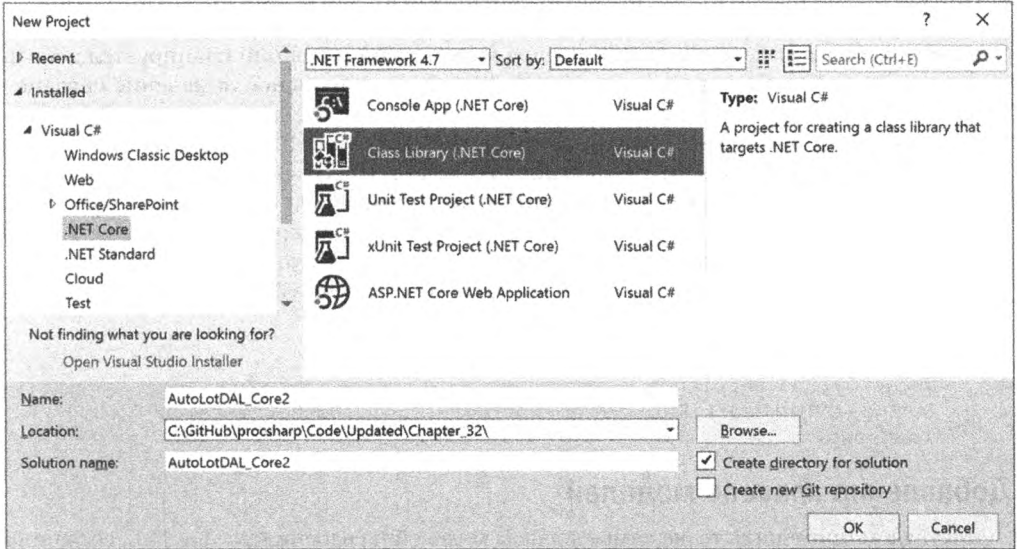


Рис. 32.1. Создание начального проекта и решения .NET Core

Добавим в решение еще один проект библиотеки классов .NET Core под названием AutoLotDAL_Core2.Models и удалим стандартный файл Class1.cs. Наконец, добавим в решение проект консольного приложения .NET Core по имени AutoLotDAL_Core2.TestDriver. Установим проект AutoLotDAL_Core2.TestDriver в качестве запускаемого.

Добавим в AutoLotDAL_Core2 ссылку на AutoLotDAL_Core2.Models, а в AutoLotDAL_Core2.TestDriver ссылку на AutoLotDAL_Core2 и AutoLotDAL_Core2.Models.

Добавление пакетов NuGet

Начиная с проекта AutoLotDAL_Core2.Models, щелкнем правой кнопкой мыши на имени проекта и выберем в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet). Щелкнем на кнопке Browse (Обзор) в левом верхнем углу и введем Microsoft.AspNetCore.Mvc.DataAnnotations. Установим данный пакет в проект. Этот пакет содержит версию .NET Core специфичных для MVC аннотаций данных, включая обновленную аннотацию MetadataType, которая теперь называется ModelMetadataType.

Далее щелкнем правой кнопкой мыши на имени проекта AutoLotDAL_Core2 и выберем в контекстном меню пункт Manage NuGet Packages. В отличие от EF 6 инфраструктура EF Core состоит из серии небольших пакетов. Существуют главные пакеты, которые должны устанавливаться при любом сценарии применения EF Core и с паке-

тами, специфичными для поставщика. В рассматриваемом приложении используется поставщик SQL Server, так что понадобится установить следующие пакеты:

- Microsoft.EntityFrameworkCore;
- Microsoft.EntityFrameworkCore.Design;
- Microsoft.EntityFrameworkCore.Relational;
- Microsoft.EntityFrameworkCore.SqlServer;
- Microsoft.EntityFrameworkCore.Tools (необязателен; дает возможность применять консоль диспетчера пакетов (Package Manager Console) для миграций EF).

Последний пакет необязателен, если для выполнения миграций планируется использовать интерфейс командной строки (CLI) .NET Core. Финальное изменение связано с добавлением в проект версии .NET Core пакета Microsoft.EntityFrameworkCore.Tools. К сожалению, добавлять его придется вручную. Среда Visual Studio облегчает задачу, разрешая напрямую редактировать файлы проектов. Щелчком правой кнопкой мыши на имени проекта и выберем в контекстном меню пункт Edit AutoLotDAL_Core2.csproj (Редактировать AutoLotDAL_Core2.csproj). Файл проекта откроется в редакторе Visual Studio. Добавим к содержимому файла проекта приведенную ниже разметку (в собственном элементе ItemGroup), что обеспечит запуск команд EF непосредственно в любой командной строке:

```
<ItemGroup>
  <DotNetCliToolReference
    Include="Microsoft.EntityFrameworkCore.Tools.DotNet"Version="2.0.0"/>
</ItemGroup>
```

Добавление классов моделей

Мы будем применять те же самые классы моделей из версии AutoLotDAL, созданной в главе 29. Если вы прорабатывали примеры в указанной главе, тогда можете скопировать классы моделей из имеющегося проекта (не забыв изменить пространства имен), либо извлечь их из папки Chapter_32 внутри загружаемого кода для книги. В случае использования версий из главы 29 потребуется внести ряд изменений для учета .NET Core. Классы описаны в последующих разделах.

Класс *EntityBase*

Создадим внутри проекта AutoLotDAL_Core2.Models папку по имени Base и добавим в нее файл под названием EntityBase.cs. Удалим из файла начальный код и поместим в него следующий код:

```
using System.ComponentModel.DataAnnotations;
namespace AutoLotDAL_Core2.Models.Base
{
    public class EntityBase
    {
        [Key]
        public int Id { get; set; }
        [Timestamp]
        public byte[] Timestamp { get; set; }
    }
}
```

Первичный ключ (Id) будет последовательностью в SQL Server, а столбец Timestamp будет применяться для проверки параллелизма. Они работают точно так же, как в EF 6.

Класс *CreditRisk*

Добавим в проект `AutoLotDAL_Core2.Models` новый файл по имени `CreditRisk.cs` и поместим в него такой код:

```
using System.ComponentModel.DataAnnotations;
using AutoLotDAL_Core2.Models.Base;
namespace AutoLotDAL_Core2.Models
{
    public partial class CreditRisk : EntityBase
    {
        [StringLength(50)]
        public string FirstName { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
    }
}
```

Обратите внимание на отсутствие аннотаций данных `Index`, которые содержались в коде из главы 29. В EF Core индексы на основе нескольких столбцов не могут быть сконфигурированы с помощью аннотаций данных; они должны конфигурироваться посредством Fluent API, как вскоре будет показано.

Класс *Customer*

Добавим новый файл класса под названием `Customer.cs` со следующим кодом:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using AutoLotDAL_Core2.Models.Base;
namespace AutoLotDAL_Core2.Models
{
    public partial class Customer : EntityBase
    {
        [StringLength(50)]
        public string FirstName { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        public List<Order> Orders { get; set; } = new List<Order>();
        [NotMapped]
        public string FullName => $"{FirstName} {LastName}";
    }
}
```

По сравнению с версией EF 6 есть два отличия. Первое — удалено ключевое слово `virtual`. Оно разрешало инфраструктуре EF 6 выполнять ленивую загрузку данных, но поскольку ленивая загрузка в EF Core пока еще не поддерживается, модификатор можно удалить.

Второе отличие — свойство `Orders` определено как `List<Order>`, а не комбинация `ICollection<Order>/HashSet<Order>`. Это одно из тонких усовершенствований инфраструктуры EF Core, заключающееся в том, что коллекции могут быть простыми типами `List<T>`.

Класс *Inventory*

Добавим новый файл класса по имени `Inventory.cs` и поместим в него приведенный ниже код:

```

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using AutoLotDAL_Core2.Models.Base;
namespace AutoLotDAL_Core2.Models
{
    [Table("Inventory")]
    public partial class Inventory : EntityBase
    {
        [StringLength(50)]
        public string Make { get; set; }
        [StringLength(50)]
        public string Color { get; set; }
        [StringLength(50)]
        public string PetName { get; set; }
        public List<Order> Orders { get; set; } = new List<Order>();
    }
}

```

И снова свойство `Orders` определено как `List<Order>` вместо комбинации `ICollection<Order>/HashSet<Order>`.

Класс *InventoryMetaData*

Добавим в проект `AutoLotDAL_Core2.Models` новую папку под названием `MetaData`. а нее файл по имени `InventoryMetaData.cs`. Заменяем первоначальный код следующим кодом:

```

using System.ComponentModel.DataAnnotations;
namespace AutoLotDAL_Core2.Models.MetaData
{
    public class InventoryMetaData
    {
        [Display(Name = "Pet Name")]
        public string PetName;
        [StringLength(50,
            ErrorMessage = "Please enter a value less than 50 characters long.")]
        public string Make;
    }
}

```

Класс *InventoryPartial*

Возвратимся в главную папку проекта, добавим новый файл под названием `InventoryPartial.cs` и поместим в него представленный далее код:

```

using System.ComponentModel.DataAnnotations.Schema;
using AutoLotDAL_Core2.Models.MetaData;
using Microsoft.AspNetCore.Mvc;
namespace AutoLotDAL_Core2.Models
{
    [ModelMetadataType(typeof(InventoryMetaData))]
    public partial class Inventory
    {
        public override string ToString()
        {
            // Поскольку столбец PetName может быть пустым,
            // определить стандартное имя **No Name**.

```

```

        return $"{this.PetName ?? "***No Name***"} is a {this.Color} {this.Make}
with ID {this.Id}.";
    }
    [NotMapped]
    public string MakeColor => $"{Make} + ({Color})";
}
}

```

Обратите внимание, что имя атрибута изменено на `ModelMetadataType` (вместо `MetadataType`), а пространство имен изменено на `Microsoft.AspNetCore.Mvc`, содержащее аннотацию.

Класс Order

Добавим новый файл по имени `Order.cs` со следующим кодом:

```

using System.ComponentModel.DataAnnotations.Schema;
using AutoLotDAL_Core2.Models.Base;

namespace AutoLotDAL_Core2.Models
{
    public partial class Order : EntityBase
    {
        public int CustomerId { get; set; }
        public int CarId { get; set; }
        [ForeignKey(nameof(CustomerId))]
        public Customer Customer { get; set; }
        [ForeignKey(nameof(CarId))]
        public Inventory Car { get; set; }
    }
}

```

Единственное изменение здесь — удаление модификатора `virtual` из двух навигационных свойств.

Создание класса AutoLotContext

Создадим в проекте `AutoLotDAL_Core2` новую папку под названием `EF` и добавим в нее файл класса по имени `AutoLotContext.cs`. Поместим в начало файла показанные ниже операторы `using`:

```

using System;
using AutoLotDAL_Core2.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;

```

Сделаем класс открытым и унаследованным от `DbContext`:

```

public class AutoLotContext : DbContext

```

Работа классов, производных от `DbContext`, в `EF Core` изменилась главным образом в плане поддержки внедрения зависимостей и конфигурации. Вместо стандартного конструктора, который передает базовому классу имя строки подключения, производные от `DbContext` классы в `EF Core` конфигурируются через экземпляры `DbContextOptions`, внедряемые в конструктор. Создадим новый конструктор:

```

    public AutoLotContext(DbContextOptions options) : base(options)
    {
    }
}

```

Экземпляр `DbContextOptions` создается классом `DbContextOptionsBuilder` и позволяет указывать поставщика базы данных, строку подключения и любые параметры, специфичные для поставщика. Факт его внедрения в класс делает возможным изменение поставщика во время выполнения. Хотя менять SQL Server на Oracle может оказаться непрактично (по слишком многим соображениям), такая возможность прекрасно подходит для перехода на новый поставщик `InMemory`. Указанный поставщик великолепно работает как инструмент тестирования для уровня доступа к данным, устраняя необходимость иметь физическую базу данных.

Вдобавок к поддержке внедрения зависимостей (dependency injection — DI) в базовый метод `OnConfiguring()` встроен резервный механизм. Данный метод открывает доступ к объекту `DbContextOptionsBuilder`, который может использоваться для подтверждения, что контекст был сконфигурирован, и если нет, то позволить его сконфигурировать. Добавим следующий код:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        var connectionString =
            @"server=(LocalDb)\MSSQLLocalDB;database=AutoLotCore2;
            integrated security=True; MultipleActiveResultSets=True;
            App=EntityFramework;";
        optionsBuilder.UseSqlServer(connectionString,
                                   options => options.EnableRetryOnFailure())
            .ConfigureWarnings(warnings=>
                warnings.Throw(RelationalEventId.QueryClientEvaluationWarning));
    }
}
```

Свойство `IsConfigured` устанавливается в `true`, если контекст был сконфигурирован в конструкторе, и в `false`, если нет. Сконфигурированные настройки объясняются в двух разделах далее в главе.

На заметку! В главе 22 внутри строки подключения вместо `server` и `database` применялись конструкции `data source` и `initial catalog`. Работать будет любой из вариантов, но рекомендуется использовать новый подход.

Устойчивость подключений

Активация устойчивости подключений заставляет EF автоматически повторять операции базы данных в случае возникновения кратковременных ошибок. Такие попытки повторения выполняются определенное количество раз с установленной задержкой между попытками. Данный прием способен улучшить восприятие приложения пользователями, сокращая отказы, когда происходят небольшие сбои в подключении (например).

Вы можете создать специальную стратегию выполнения, но для SQL Server в этом нет необходимости, т.к. команда разработчиков уже предоставила класс `SqlServerRetryingExecutionStrategy`. Кроме того, команда поставщика сделала возможным сокращение для добавления к подключению такой стратегии выполнения. За счет добавления следующего параметра `DbContext` при возникновении кратковременных ошибок SQL Server и/или SQL Azure операции базы данных будут повторяться до шести раз (количество попыток и задержка между ними допускают настройку):

```
options => options.EnableRetryOnFailure())
```

Объект `SqlServerRetryingExecutionStrategy` может быть сконфигурирован с целью установки максимального числа повторений, задержки между повторениями и обработки дополнительных ошибок помимо стандартных.

Если лимит повторений превышен, тогда инфраструктура EF сгенерирует исключение `RetryLimitExceededException`. Перехват указанной ошибки в хранилищах позволяет действовать в соответствии с требованиями приложений.

Смешанная клиент-серверная оценка

Вспомните из главы 22, что запросы EF 6 не выполняются до тех пор, пока не будет производиться проход по их результатам с применением `First()` или похожего метода LINQ. Также вспомните, что в операторах LINQ нельзя смешивать обработку серверной и клиентской сторон. Начиная с версии EF Core 1.0, ситуация изменилась. Смешанная клиент-серверная оценка является возможностью EF Core, которая позволяет операторам LINQ задействовать код серверной и клиентской сторон.

При смешанной оценке запросов, когда инфраструктура EF обнаруживает вызов клиентской стороны, она выполняет часть серверной стороны запроса, возвратив полученные в результате данные, и затем по существу будет использовать LINQ to Objects на клиентской стороне. Наряду с тем, что это может оказаться полезным, возможно возникновение проблем, если порядок операций LINQ некорректен.

Например, пусть имеется таблица, в которой есть поле, определенное как GUID, и нужно фильтровать записи на основе данного поля. Теперь предположим, что параметр для фильтра относится к типу `string`, а не GUID. Может показаться, что приведенный ниже запрос будет успешно работать:

```
return Table.FirstOrDefault(x => x.TheGuid.ToString() == g)
```

И он работает, просто не настолько хорошо. Механизм, который преобразует операторы LINQ в операторы SQL, не воспринимает вызов `TheGuid.ToString()` и не транслирует его в код SQL. Взамен он выполняет такой код на клиентской стороне. Это означает, что из таблицы извлекаются все записи, поля GUID преобразуются в строки (по одному за раз) и затем сравниваются со строкой `g`, пока не будет найдено первое совпадение.

Простое заблуждение вроде продемонстрированного способно нанести ущерб приложению, особенно если не проводилось тщательное тестирование кода доступа к данным. Чтобы обеспечить выполнение запроса на стороне сервера, понадобится изменить параметр `g` в GUID *перед* вызовом метода LINQ, после чего осуществлять поиск:

```
Guid g = Guid.Parse(guid); return Table.FirstOrDefault(x => x.Guid == g);
```

Отличие довольно тонкое и существует много других ситуаций, когда вы можете быть (неприятно) удивлены оценкой клиентской стороны. Мы рекомендуем держаться подальше от смешанного режима оценки, если только в нем нет крайней необходимости, и внимательно тестировать код доступа к данным.

К сожалению, смешанный режим оценки нельзя по-настоящему отключить. Лучшее, что можно сделать — генерировать исключение, когда происходит оценка в смешанном режиме. Вот финальная настройка в `DbContextOptionsBuilder`:

```
.ConfigureWarnings(warnings=>warnings.Throw(
    RelationalEventId.QueryClientEvaluationWarning));
```

При такой настройке, по крайней мере, можно перехватывать возникновение оценки в смешанном режиме и надлежащим образом обновлять свои запросы. Мы применяем данную настройку по умолчанию во всех разрабатываемых приложениях, а если обнаруживается случай, когда приложение выигрывает от смешанного режима оценки, то просто внедряем другой набор параметров, который не приводит к генерации ошибки при выполнении кода клиентской стороны.

Добавление конструктора без параметров

Позже в главе внутри проекта `AutoLotDAL_Core2.TestDriver` для создания нового экземпляра `AutoLotContext` будет использоваться конструктор без параметров. Когда экземпляр класса контекста создается с применением конструктора без параметров, выполняется резервный механизм `OnConfiguring()`, устанавливая подходящие значения в контексте.

Однако для того, чтобы использовалось новое средство организации пула `DbContext` из ASP.NET Core, в классе контекста может быть только один конструктор, который принимает экземпляр `DbContextOptions`. Для удовлетворения обоих требований создадим внутренний конструктор без параметров:

```
internal AutoLotContext()
{
}
```

Чтобы применить созданный конструктор и задействовать обработчик событий `OnConfiguring()`, проекту `AutoLotDAL_Core2.TestDriver` потребуется открыть доступ к внутренним элементам проекта `AutoLotDAL_Core2`. Щелкнем правой кнопкой мыши на имени проекта `AutoLotDAL_Core2` и выберем в контекстном меню пункт `Add New Item` (Добавить новый элемент). Укажем элемент `Assembly Information File` (Файл информации о сборке) и назовем его `AssemblyInfo.cs`. Поместим в файл `AssemblyInfo.cs` следующий код:

```
using System.Runtime.CompilerServices;
[assembly: InternalsVisibleTo("AutoLotDAL_Core2.TestDriver")]
```

На заметку! На момент написания главы среда Visual Studio при добавлении такого файла сообщала об ошибке. Просто проигнорируйте сообщение и продолжайте работу.

Добавление фабрики контекстов для этапа проектирования

Для запуска миграций EF в классе контекста должен быть открытый конструктор без параметров или реализация `IDesignTimeDbContextFactory<TContext>`. Поскольку наличие открытого конструктора без параметров препятствует использованию средства организации пула `DbContext`, мы собираемся добавить класс, который реализует интерфейс `IDesignDbTimeContextFactory<TContext>`. Данный интерфейс имеет один метод `CreateDbContext()`. Если механизм миграций EF находит экземпляр реализации интерфейса `IDesignDbTimeContextFactory<TContext>` в той же самой сборке, где применяется контекст, тогда для создания контекста он будет использовать метод `CreateDbContext()`, а не открытый конструктор.

Добавим в папку EF проекта `AutoLotDAL_Core2` новый файл класса по имени `AutoLotContextFactory.cs`. Удалим шаблонный код и поместим в файл такой код:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.EntityFrameworkCore.Diagnostics;

namespace AutoLotDAL_Core2.EF
{
    public class AutoLotContextFactory :
        IDesignTimeDbContextFactory<AutoLotContext>
    {
        public AutoLotContext CreateDbContext(string[] args)
        {
            var optionsBuilder = new DbContextOptionsBuilder<AutoLotContext>();
```

```

var connectionString =
    @"server=(LocalDb)\MSSQLLocalDB;database=AutoLotCore2;
    integrated security=True; MultipleActiveResultSets=True;
    App=EntityFramework;";
optionsBuilder.UseSqlServer(
    connectionString, options => options.EnableRetryOnFailure()
        .ConfigureWarnings(warnings => warnings.Throw(RelationalEventId.
            QueryClientEvaluationWarning));
return new AutoLotContext(optionsBuilder.Options);
}
}
}

```

Добавление *DbSet<T>*

Далее добавим таблицы, которыми необходимо управлять с помощью EF — почти как в EF 6. И снова отличие заключается в том, что свойства не сопровождаются модификатором `virtual`. Поместим в файл класса `AutoLotContext.cs` следующий код:

```

public DbSet<AutoLotDAL_Core2.Models.CreditRisk> CreditRisks { get; set; }
public DbSet<AutoLotDAL_Core2.Models.Customer> Customers { get; set; }
public DbSet<AutoLotDAL_Core2.Models.Inventory> Cars { get; set; }
public DbSet<AutoLotDAL_Core2.Models.Order> Orders { get; set; }

```

На заметку! Из-за дефекта средства формирования шаблонов ASP.NET Core в ситуации, когда контекст и модели находятся внутри разных сборок, типы в свойствах `DbSet<T>` должны снабжаться названиями пространств имен, как демонстрируется в предыдущем коде. Любые операторы `using` игнорируются. Мы полагаем, что дефект вскоре будет устранен, может быть даже ко времени выхода настоящей книги.

Использование *Fluent API* для завершения моделей

Интерфейс *Fluent API* — еще один способ создания моделей, а также таблиц и столбцов результирующей базы данных. Как упоминалось ранее, аннотации данных в EF Core поддерживаются в меньшей степени, чем в EF 6. Для создания индексов, включающих несколько столбцов, должен применяться *Fluent API*. Добавим переопределенную версию метода `OnModelCreating()`:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // Создать индекс, включающий несколько столбцов
    modelBuilder.Entity<CreditRisk>(entity =>
    {
        entity.HasIndex(e => new { e.FirstName, e.LastName }).IsUnique();
    });
    // Установить параметр каскадирования на отношении
    modelBuilder.Entity<Order>()
        .HasOne(e => e.Car)
        .WithMany(e => e.Orders)
        .OnDelete(DeleteBehavior.ClientSetNull);
}

```

Первый блок кода устанавливает в таблице `CreditRisk` уникальный индекс, состоящий из нескольких столбцов. Второй блок кода создает между таблицами `Inventory` и `Order` отношение “один ко многим” и устанавливает параметр каскадирования в отсутствие действия (стандартное поведение).

Добавление функции `GetTableName()`

Осталось создать последний метод, который будет возвращать имя таблицы SQL Server для `DbSet<T>`. Он необходим из-за того, что имена классов и таблиц не обязаны совпадать, а имя таблицы SQL Server потребуется в коде инициализации данных (мы напишем его позже в главе).

Добавим в файл класса `AutoLotContext.cs` следующий метод:

```
public string GetTableName(Type type)
{
    return Model.FindEntityType(type).SqlServer().TableName;
}
```

Итак, класс `AutoLotContext` завершен.

Создание базы данных с помощью миграций

В EF Core миграции изменились к лучшему. Миграции EF 6 создавали хеш определения базы данных и сохраняли его в таблице миграций внутри базы данных. Если два разработчика вносили изменения, то не было какого-то способа объединить такие изменения, поскольку хеши имели однонаправленную природу. В результате использование миграций при командной разработке оказывалось в лучшем случае затруднительным.

В EF Core хеш был полностью убран из процесса. В базе данных хранится лишь имя миграции и номер версии EF. Определение базы данных сохраняется в файле C# с именем `<ИмяКонтекста>ModelSnapshot.cs`. В таком случае разработчики могут применять стандартный инструмент для выявления отличий, чтобы разрешить любые конфликтующие изменения.

Миграции могут запускаться с использованием команд консоли диспетчера пакетов (Package Manager Console), например, `add-migration`, `update-database` и т.д., либо с применением нового интерфейса командной строки .NET Core (CLI). Преимущество команд .NET Core CLI в том, что их можно запускать в любой командной строке, а не только из консоли диспетчера пакетов.

Использование команд EF .NET CLI

До того, как применять команды EF .NET CLI, файл проекта должен быть сконфигурирован с корректным пакетом. Это делалось ранее в главе путем добавления в файл проекта `AutoLotDAL_Core2` следующего элемента:

```
<ItemGroup>
  <DotNetCliToolReference
    Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
</ItemGroup>
```

Текущим должен быть каталог, где находится файл проекта, в котором определены параметры команд .NET Core CLI. Откроем окно командной строки и перейдем в каталог, содержащий файл проекта `AutoLotDAL_Core2.csproj`. Команды можно также запускать из консоли диспетчера пакетов, т.к. она дублирует функции окна командной строки. Нужно просто перейти в корректный каталог посредством такой команды (предполагается, что консоль диспетчера пакетов была только что открыта):

```
cd .\AutoLotDAL_Core2
```

Следующая команда позволяет вывести полный список команд, доступных для EF через .NET CLI:

```
dotnet ef
```

Команда выведет изображение единорога, являющееся логотипом инфраструктуры EF, с помощью символов ASCII и за ним перечень доступных команд. Для получения дополнительной информации по любой команде ее необходимо запускать с параметром `-h` (или `--help`):

```
dotnet ef migrations -h
```

Чтобы добавить миграцию для создания нашей базы данных, сохраним все файлы, выполним построение проекта (в качестве меры безопасности) и запустим такую команду:

```
dotnet ef migrations add Initial
--context AutoLotDAL_Core2.EF.AutoLotContext -o
EF\Migrations
```

Миграция получит имя `Initial`, используя полностью заданный контекст `AutoLotContext`, а результирующие миграции будут помещены в каталог `EF\Migrations`. В итоге создаются файлы с именами, включающими отметки времени (подобно миграциям EF 6), которые применяются для обновления базы данных, и файл `AutoLotContextModelSnapshot.cs`, хранящий полную модель базы данных.

Для применения миграции служит следующая команда:

```
dotnet ef database update
```

Инициализация базы данных начальной информацией

В EF Core отсутствует возможность начального заполнения базы данных, используя миграции, т.е. базовые классы, которые удаляют и воссоздают базу данных. К счастью, такую функциональность легко реализовать вручную. Начнем с создания в проекте `AutoLotDAL_Core2` новой папки по имени `DataInitialization`. Добавим в нее файл класса под названием `MyDataInitializer.cs`.

Сделаем класс открытым и статическим и поместим в начало файла класса показанные ниже операторы `using`:

```
using System.Collections.Generic;
using AutoLotDAL_Core2.EF;
using AutoLotDAL_Core2.Models;
using Microsoft.EntityFrameworkCore;
```

Начальное заполнение базы данных

Начальное заполнение базы данных означает просто создание записей в коде и их добавление с применением стандартных методов EF. Добавим статический метод `InitializeData()` со следующим кодом:

```
public static void InitializeData(AutoLotContext context)
{
    var customers = new List<Customer>
    {
        new Customer {FirstName = "Dave", LastName = "Brenner"},
        new Customer {FirstName = "Matt", LastName = "Walton"},
        new Customer {FirstName = "Steve", LastName = "Hagen"},
        new Customer {FirstName = "Pat", LastName = "Walton"},
        new Customer {FirstName = "Bad", LastName = "Customer"},
    };
    customers.ForEach(x => context.Customers.Add(x));
    context.SaveChanges();
    var cars = new List<Inventory>
    {
```

```

    new Inventory {Make = "VW", Color = "Black", PetName = "Zippy"},
    new Inventory {Make = "Ford", Color = "Rust", PetName = "Rusty"},
    new Inventory {Make = "Saab", Color = "Black", PetName = "Mel"},
    new Inventory {Make = "Yugo", Color = "Yellow", PetName = "Clunker"},
    new Inventory {Make = "BMW", Color = "Black", PetName = "Bimmer"},
    new Inventory {Make = "BMW", Color = "Green", PetName = "Hank"},
    new Inventory {Make = "BMW", Color = "Pink", PetName = "Pinky"},
    new Inventory {Make = "Pinto", Color = "Black", PetName = "Pete"},
    new Inventory {Make = "Yugo", Color = "Brown", PetName = "Brownie"},
);
context.Cars.AddRange(cars);
context.SaveChanges();
var orders = new List<Order>
{
    new Order {Car = cars[0], Customer = customers[0]},
    new Order {Car = cars[1], Customer = customers[1]},
    new Order {Car = cars[2], Customer = customers[2]},
    new Order {Car = cars[3], Customer = customers[3]},
};
orders.ForEach(x => context.Orders.Add(x));
context.SaveChanges();
context.CreditRisks.Add(
    new CreditRisk
    {
        Id = customers[4].Id,
        FirstName = customers[4].FirstName,
        LastName = customers[4].LastName,
    });
context.Database.OpenConnection();
try
{
    var tableName = context.GetTableName(typeof(CreditRisk));
    // В версии 2.0 интерполяция строк .NET должна отделяться
    // от интерполяции SQL
    var rawSqlString = $"SET IDENTITY_INSERT dbo.{tableName} ON;";
    context.Database.ExecuteNonQuery(rawSqlString);
    context.SaveChanges();
    rawSqlString = $"SET IDENTITY_INSERT dbo.{tableName} OFF;";
    context.Database.ExecuteNonQuery(rawSqlString);
}
finally
{
    context.Database.CloseConnection();
}
}

```

Код близок к своей версии для EF 6 за исключением нескольких заметных изменений. Первое — в `DbSet<T>` отсутствует метод `AddOrUpdate()`, так что операции ограничиваются вызовами `Add()` или `AddRange()`. Второе изменение касается синтаксиса вокруг включения вставки идентичности. В приведенном ниже коде демонстрируется пример привлечения существующего подключения для выполнения множества команд вместе. Это также можно делать с помощью транзакций.

```

context.Database.OpenConnection();
try
{
    var tableName = context.GetTableName(typeof(CreditRisk));

```

```
// В версии 2.0 интерполяция строк .NET должна отделяться от интерполяции SQL
var rawSqlString = $"SET IDENTITY_INSERT dbo.{tableName} ON;";
context.Database.ExecuteSqlCommand(rawSqlString);
context.SaveChanges();
rawSqlString = $"SET IDENTITY_INSERT dbo.{tableName} OFF";
context.Database.ExecuteSqlCommand(rawSqlString);
}
finally
{
    context.Database.CloseConnection();
}
```

Финальным изменением является действие интерполяции строк в низкоуровневых командах SQL, что рассматривается далее.

Интерполяция строк в низкоуровневых запросах SQL

Нововведением EF Core 2.0 является сочетание интерполяции строк C# и параметризации низкоуровневых запросов SQL при использовании методов `FromSql()` и `ExecuteSqlCommand()`. В EF 6 и версиях EF Core, предшествующих 2.0, следующие операторы были эквивалентными (при условии, что `tableName` имеет значение `CreditRisk`):

```
context.Database.ExecuteSqlCommand($"SET IDENTITY_INSERT dbo.{tableName} ON;");
context.Database.ExecuteSqlCommand("SET IDENTITY_INSERT dbo.CreditRisk ON;");
```

В EF Core 2 первый оператор транслируется в параметризированный запрос вроде такого:

```
SET IDENTITY_INSERT dbo.@p0 ON
```

В большинстве случаев новый способ обработки интерполированных строк работает лучше. Тем не менее, в рассматриваемом случае это не так, и если параметризированный запрос не нужен, тогда интерполяцию строк C# придется вынести за пределы низкоуровневых команд SQL.

Создание и сброс базы данных

В дополнение к загрузке базы данных во время тестирования полезно сбрасывать базу данных в ее исходное состояние. В EF 6 такая работа выполнялась посредством базового класса `DropCreateDatabaseAlways<T>`. В EF Core сброс придется делать вручную.

Для удаления и воссоздания базы данных доступно несколько вспомогательных методов из свойства `Database` класса контекста, которые перечислены в табл. 32.2.

Таблица 32.2. Вспомогательные методы `Database`, предназначенные для удаления и воссоздания базы данных

Метод	Описание
<code>EnsureDeleted()</code>	Удаляет базу данных, если она существует; в противном случае не делает ничего
<code>EnsureCreated()</code>	Создает базу данных на основе кода в файле <code><ИмяКонтекста>ModelSnapshot.cs</code> , если он существует. Является взаимоисключающим по отношению к <code>Migrate()</code>
<code>Migrate()</code>	Создает базу данных и выполняет все миграции, если они существуют; применяет любые пропущенные миграции при их наличии. Является взаимоисключающим по отношению к <code>EnsureCreated()</code>

Если только в миграциях нет специфического кода (скажем, для создания хранимых процедур), то результат методов `EnsureCreated()` и `Migrate()` будет тем же, исключая наполнение таблицы `__EFMigrationsHistory`. Метод `EnsureCreated()` никогда не запускает миграции и потому не наполняет указанную таблицу. Таким образом, если попытаться запустить миграции после вызова `EnsureCreated()`, то они потерпят неудачу, поскольку объекты в миграциях уже будут существовать.

Чтобы удалить базу данных и затем создать ее, добавим в класс `MyDataInitializer` статический метод `RecreateDatabase()`:

```
public static void RecreateDatabase(AutoLotContext context)
{
    context.Database.EnsureDeleted();
    context.Database.Migrate();
}
```

Удаление и последующее воссоздание базы данных может требоваться не всегда, поэтому мы реализуем методы для удаления существующих данных и сброса идентичности:

```
public static void ClearData(AutoLotContext context)
{
    ExecuteDeleteSql(context, "Orders");
    ExecuteDeleteSql(context, "Customers");
    ExecuteDeleteSql(context, "Inventory");
    ExecuteDeleteSql(context, "CreditRisks");
    ResetIdentity(context);
}

private static void ExecuteDeleteSql(AutoLotContext context, string tableName)
{
    // В версии 2.0 интерполяция строк должна отделяться,
    // если параметры не передаются
    var rawSqlString = $"Delete from dbo.{tableName}";
    context.Database.ExecuteNonQuery(rawSqlString);
}

private static void ResetIdentity(AutoLotContext context)
{
    var tables = new[] { "Inventory", "Orders", "Customers", "CreditRisks" };
    foreach (var itm in tables)
    {
        // В версии 2.0 интерполяция строк должна отделяться,
        // если параметры не передаются
        var rawSqlString = $"DBCC CHECKIDENT (\"dbo.{itm}\", RESEED, -1)";
        context.Database.ExecuteNonQuery(rawSqlString);
    }
}
```

Код инициализации данных завершен.

Добавление хранилищ для многократного использования кода

Хранилища в версии EF Core 2.0 слегка отличаются от хранилищ в EF 6. Они изменены, чтобы извлечь преимущества от улучшений, внесенных в EF Core 2.0. Начнем с добавления в проект `AutoLotDAL_Core2` новой папки по имени `Repos`.

Добавление интерфейса IRepo

Добавим в папку Repos новый файл интерфейса под названием IRepo.cs. Обновим операторы using, как показано ниже:

```
using System;
using System.Collections.Generic;
using System.Linq.Expressions;
```

Вот полный интерфейс:

```
public interface IRepo<T>
{
    int Add(T entity);
    int Add(ICollection<T> entities);
    int Update(T entity);
    int Update(ICollection<T> entities);
    int Delete(int id, byte[] timeStamp);
    int Delete(T entity);
    T GetOne(int? id);
    List<T> GetSome(Expression<Func<T, bool>> where);
    List<T> GetAll();
    List<T> GetAll<TSortField>(Expression<Func<T, TSortField>> orderBy,
        bool ascending);
    List<T> ExecuteQuery(string sql);
    List<T> ExecuteQuery(string sql, object[] sqlParametersObjects);
}
```

В этой версии методы Add()/AddRange() и Update()/UpdateRange() были преобразованы соответственно в Add() и Update(), имеющие перегруженные версии. Существует новый метод GetSome(), который принимает выражение для конструкции where, и есть перегруженная версия метода GetAll(), принимающая выражение для выполнения сортировки.

Чтобы увидеть новые методы в действии, их понадобится вызвать следующим образом:

```
return GetSome(x=>x.Color == "Pink");
return GetAll(x=>x.PetName, true);
```

Добавление класса BaseRepo

Добавим в папку Repos еще один класс по имени BaseRepo, который будет реализовывать интерфейс IRepo и предлагать основную функциональность для хранилищ, специфичных к типам. Модифицируем операторы using:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using AutoLotDAL_Core2.EF;
using AutoLotDAL_Core2.Models.Base;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
```

Сделаем класс открытым и обобщенным, реализуем интерфейсы IRepo и IDisposable, а также ограничим обобщенные типы классами EntityBase:

```
public class BaseRepo<T> : IDisposable, IRepo<T> where T:EntityBase, new()
```


Добавим две переменные `private readonly`, предназначенные для хранения контекста и объекта `DbSet<T>`, и одно поле `protected` для открытия доступа к контексту любым производным классам:

```
private readonly DbSet<T> _table;
private readonly AutoLotContext _db;
protected AutoLotContext Context => _db;
```

Конструктор нуждается в обновлении. Добавим новый конструктор, который будет принимать экземпляр `AutoLotContext`, чтобы задействовать средство внедрения зависимостей в ASP.NET Core. Конструктор без параметров будет связан в цепочку с версией конструктора, принимающей экземпляр контекста. Метод `Dispose()` будет тем же, что и в версии EF 6. Поместим в класс следующий код:

```
public BaseRepo() : this(new AutoLotContext())
{
}
public BaseRepo(AutoLotContext context)
{
    _db = context;
    _table = _db.Set<T>();
}
public void Dispose()
{
    _db?.Dispose();
}
```

Метод `Add()` похож на свою версию для EF 6 кроме того, что здесь применяется метод `Add()` и его перегруженные версии. Но внутренняя функциональность осталась той же самой.

```
public int Add(T entity)
{
    _table.Add(entity);
    return SaveChanges();
}
public int Add(ICollection<T> entities)
{
    _table.AddRange(entities);
    return SaveChanges();
}
```

Методы обновления изменились, т.к. в EF Core появились методы `Update()` и `UpdateRange()`. Вместо изменения `EntityState` на `EntityState.Modified` достаточно просто вызвать `Update()/UpdateRange()` на `DbSet<T>`:

```
public int Update(T entity)
{
    _table.Update(entity);
    return SaveChanges();
}
public int Update(ICollection<T> entities)
{
    _table.UpdateRange(entities);
    return SaveChanges();
}
```

Методы `Delete()`, `GetOne()` и пустой метод `GetAll()` — те же, что и в версии EF 6:

```

public int Delete(int id, byte[] timeStamp)
{
    _db.Entry(new T()
        { Id = id, Timestamp = timeStamp }).State = EntityState.Deleted;
    return SaveChanges();
}
public int Delete(T entity)
{
    _db.Entry(entity).State = EntityState.Deleted;
    return SaveChanges();
}
public T GetOne(int? id) => _table.Find(id);
public virtual List<T> GetAll() => _table.ToList();

```

Для улучшения запросов в обновленном методе `GetAll()` и новом методе `GetSome()` используются выражения. Метод `GetAll()` изменяет порядок сортировки на основе значения параметра `ascending`.

```

public List<T> GetAll<TSortField>(Expression<Func<T, TSortField>>
    orderBy, bool ascending)
    => (ascending? _table.OrderBy(orderBy) :
        _table.OrderByDescending(orderBy)).ToList();
public List<T> GetSome(Expression<Func<T, bool>> where)
    => _table.Where(where).ToList();

```

Остальные методы рассматриваются в последующих разделах.

Извлечение записей с помощью *FromSql()*

В EF Core появился новый метод, применяемый для обработки низкоуровневых запросов SQL. В текущий момент метод `FromSql()` способен только наполнять свойство `DbSet<T>` класса контекста. Существует нерешенная задача, касающаяся разрешения наполнять произвольный класс, но с ней не связан временной график. Это не означает, что класс обязан отображаться на какую-то таблицу базы данных; он просто должен быть в контексте как `DbSet<T>`. Можно добавить в контекст свойство, которое не отображено на таблицу, и использовать его с `FromSql()`:

```

[NotMapped]
public DbSet<MyViewModel> ViewModels {get;set;}

```

Столбцы запроса обязаны соответствовать отображенным столбцам модели, должны возвращаться все столбцы, а возвращать связанные данные (из исходного запроса) нельзя. Одна из замечательных характеристик метода `FromSql()` заключается в том, что к его вызову можно добавлять запросы LINQ. Например, при желании вернуть все записи `Inventory` вместе со связанными записями `Order` и `Customer` можно поступить так:

```

return Context.Cars.FromSql("SELECT * FROM Inventory").Include(x => x.Orders)
    .ThenInclude(x => x.Customer).ToList();

```

Здесь также демонстрируется новый метод `ThenInclude()`, извлекающий связанные данные после `Include()`.

На заметку! Как было указано в главе 22, при выполнении низкоуровневых запросов SQL в отношении хранилища данных следует соблюдать предельную осторожность, особенно когда запросы принимают результаты пользовательского ввода. В противном случае приложение становится уязвимым к атакам внедрением в SQL. Тема безопасности в настоящей книге не рассматривается, но мы хотим акцентировать внимание на угрозах, связанных с выполнением низкоуровневых операторов SQL.

Реализация вспомогательного метода *SaveChanges()*

Последним методом контекста, который мы реализуем, является `SaveChanges()`. Исключения, возникающие в методе `SaveChanges()` контекста `DbContext`, просто генерируются повторно. В приложении производственного уровня любые исключения должны быть соответствующим образом обработаны.

```
internal int SaveChanges()
{
    try
    {
        return _db.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
        // Генерируется, когда возникла ошибка, связанная с параллелизмом.
        // Пока что просто сгенерировать исключение повторно.
        throw;
    }
    catch (RetryLimitExceededException ex)
    {
        // Генерируется, когда достигнуто максимальное количество попыток.
        // Дополнительные детали можно найти во внутреннем исключении (исключениях).
        // Пока что просто сгенерировать исключение повторно.
        throw;
    }
    catch (DbUpdateException ex)
    {
        // Генерируется, когда обновление базы данных потерпело неудачу.
        // Дополнительные детали и затронутые объекты можно
        // найти во внутреннем исключении (исключениях).
        // Пока что просто сгенерировать исключение повторно.
        throw;
    }
    catch (Exception ex)
    {
        // Возникло какое-то другое исключение, которое должно быть обработано.
        throw;
    }
}
```

На заметку! Создание нового экземпляра `DbContext` может оказаться затратным процессом с точки зрения производительности. В версиях ASP.NET Core 2 и EF Core 2 появилась поддержка пула контекстов, который будет обсуждаться в главе 33.

Создание *InventoryRepo*

Нам осталось построить класс `InventoryRepo`, отвечающий за выполнение специфической работы в таблице `Inventory`, которую не покрывает базовый класс хранилища. Также понадобится создать интерфейс `IInventoryRepo` для настройки внедрения зависимостей ASP.NET Core.

Добавим в папку `Repos` новый файл интерфейса по имени `IInventoryRepo.cs` со следующим кодом:

```
using System.Collections.Generic;
using AutoLotDAL_Core2.Models;
namespace AutoLotDAL_Core2.Repos
{
    public interface IInventoryRepo : IRepo<Inventory>
    {
        List<Inventory> Search(string searchString);
        List<Inventory> GetPinkCars();
        List<Inventory> GetRelatedData();
    }
}
```

Далее добавим в папку Repos новый файл класса под названием InventoryRepo.cs. Поместим в начало файла показанные ниже операторы using:

```
using System.Collections.Generic;
using System.Linq;
using AutoLotDAL_Core2.EF;
using AutoLotDAL_Core2.Models;
using Microsoft.EntityFrameworkCore;
using static Microsoft.EntityFrameworkCore.EF;
```

Сделаем класс открытым, унаследованным от BaseRepo<Inventory> и реализующим интерфейс IInventoryRepo:

```
public class InventoryRepo : BaseRepo<Inventory>, IInventoryRepo
```

Добавим конструктор, который принимает в качестве параметра объект AutoLotContext, и свяжем его с базовым конструктором. Он будет применяться инфраструктурой внедрения зависимостей ASP.NET Core для создания хранилища с использованием экземпляра AutoLotContext из пула.

```
public InventoryRepo(AutoLotContext context) : base(context)
{
}
}
```

Добавим переопределенную версию метода GetAll() и метод GetPinkCars(), которые уже обсуждались:

```
public override List<Inventory> GetAll() => GetAll(x=>x.PetName,true).ToList();
public List<Inventory> GetPinkCars() => GetSome(x => x.Color == "Pink");
```

Выполнение запроса SQL с операцией LIKE

В версии EF Core 2.0 введено новое свойство EF.Functions, которое поставщики баз данных могут применять для реализации специфических операций. Первая реализация, выпущенная для поставщика SQL Server, представляет собой реализацию .NET операции LIKE языка SQL.

Чтобы увидеть ее в действии, добавим следующий код (обратите внимание, что операцию % потребуется добавить самостоятельно):

```
public List<Inventory> Search(string searchString)
=> Context.Cars.Where(c => Functions.Like(c.PetName,
    $"{searchString}%")).ToList();
```

В результате с использованием функции Like создается такой запрос SQL (при условии, что searchString == "foo"):

```
SELECT * FROM Inventory WHERE PetName like '%foo%'
```

Использование методов *Include()* и *ThenInclude()* для связанных данных

Финальный метод, который мы добавим, иллюстрирует работу методов `FromSql()`, `Include()` и `ThenInclude()`:

```
public List<Inventory> GetRelatedData()
    => Context.Cars.FromSql("SELECT * FROM Inventory")
        .Include(x => x.Orders).ThenInclude(x => x.Customer).ToList();
```

Как упоминалось ранее, метод `ThenInclude()` получает связанные данные из текущей позиции в реляционном дереве.

Тестирование библиотеки *AutoLotDAL_Core2*

Прежде чем тестировать код доступа к данным посредством консольного приложения, необходимо решить, планируется ли применение в тестах полного внедрения зависимостей или же будет просто создаваться новый контекст с помощью конструктора без параметров и использоваться запасная конфигурация в методе `OnConfiguring()`. В текущем примере мы собираемся применять механизм запасной конфигурации.

Откроем файл `Program.cs` из проекта `AutoLotDAL_Core2_TestDriver`. Добавим в начало файла указанные ниже операторы `using`:

```
using System;
using System.Linq;
using AutoLotDAL_Core2.DataInitialization;
using AutoLotDAL_Core2.EF;
using AutoLotDAL_Core2.Models;
using AutoLotDAL_Core2.Repos;
using Microsoft.EntityFrameworkCore;
```

Модифицируем метод `Main()` следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with ADO.NET EF Core 2 *****\n");
    using (var context = new AutoLotContext())
    {
        MyDataInitializer.RecreateDatabase(context);
        MyDataInitializer.InitializeData(context);
        foreach (Inventory c in context.Cars)
        {
            Console.WriteLine(c);
        }
    }
    Console.WriteLine("***** Using a Repository *****\n");
    using (var repo = new InventoryRepo())
    {
        foreach (Inventory c in repo.GetAll())
        {
            Console.WriteLine(c);
        }
    }
    Console.ReadLine();
}
```

Код удаляет и воссоздает базу данных (используя миграции), наполняет ее данными и затем извлекает записи сначала с применением `AutoLotContext`, а после того с использованием `InventoryRepo`. При желании можно скопировать дополнительный код тестирования из версии, приведенной в главе 22, чтобы проверить остальную функциональность нового уровня доступа к данным.

Исходный код. Решение `AutoLotDAL_Core2` доступно в подкаталоге `Chapter_32`.

Резюме

В главе было предложено введение в новейшую версию Entity Framework — EF Core 2.0. Инфраструктура EF позволяет писать код для концептуальной модели, которая точно отображается на предметную область. Наряду с тем, что сущностям можно придавать любую желаемую форму, инфраструктура EF обеспечивает отображение измененных данных на корректные данные физических таблиц.

В начале главы обсуждались отличия EF Core от EF 6 и затем уровень доступа к данным EF 6 был преобразован в межплатформенную версию, использующую EF Core 2.0. Созданная новая версия послужит уровнем доступа к данным для проектов ASP.NET, которые будут строиться в главах 33 и 34.

ГЛАВА 33

Введение в веб-приложения ASP.NET Core

Настоящая глава является первой из двух глав, в которых дается введение в ASP.NET Core. Как упоминалось в главе 31, с выходом ASP.NET Core инфраструктура ASP.NET MVC и ASP.NET Web API были объединены в единую инфраструктуру. Тонкие (и не особо тонкие) отличия между MVC 5 и Web API исчезли. Так случилось, что их названия различались. Теперь это просто ASP.NET Core.

Инфраструктура ASP.NET Core построена поверх .NET Core, чтобы обеспечить возможности межплатформенной разработки и развертывания веб-приложений и служб REST с использованием языка C# (и, само собой разумеется, Razor, HTML, JavaScript и т.д.). Подобно EF Core инфраструктура ASP.NET Core представляет собой не просто перенесенную версию — она была создана заново. В результате у команды разработчиков появляется возможность привести в порядок свою кодовую базу, сосредоточив внимание на производительности и совершенствовании функциональности.

Сначала в главе строится решение ASP.NET Core для веб-приложения, затем рассматриваются функциональные средства ASP.NET Core и, наконец, в приложение CarLotMVC вносятся финальные штрихи.

На заметку! В главе предполагается наличие у вас практических знаний ASP.NET MVC 5. Если это не так, тогда прежде чем продолжить чтение ознакомьтесь с материалами главы 28.

Шаблон ASP.NET Core Web Application

Как и в случае ASP.NET MVC 5, среда Visual Studio поставляется с довольно полным шаблоном проекта, предназначенным для построения веб-приложений ASP.NET Core. Кроме того, существуют шаблоны для построения проектов, ориентированных на службы в стиле Web API, проектов, в которых применяется средство Razor Pages, и проектов, использующих Angular, React, а также React и Redux.

Мастер создания проекта

Чтобы создать решение для данной главы, мы начнем с запуска Visual Studio и выбора пункта меню File⇒New⇒Project (Файл⇒Создать⇒Проект). В боковой панели слева выберем элемент Web (Веб) внутри Visual C#, затем в центральной панели выберем шаблон ASP.NET Core Web Application (Веб-приложение ASP.NET Core), после чего в поле Name (Имя) изменим имя на AutoLotMVC_Core2 (рис. 33.1).

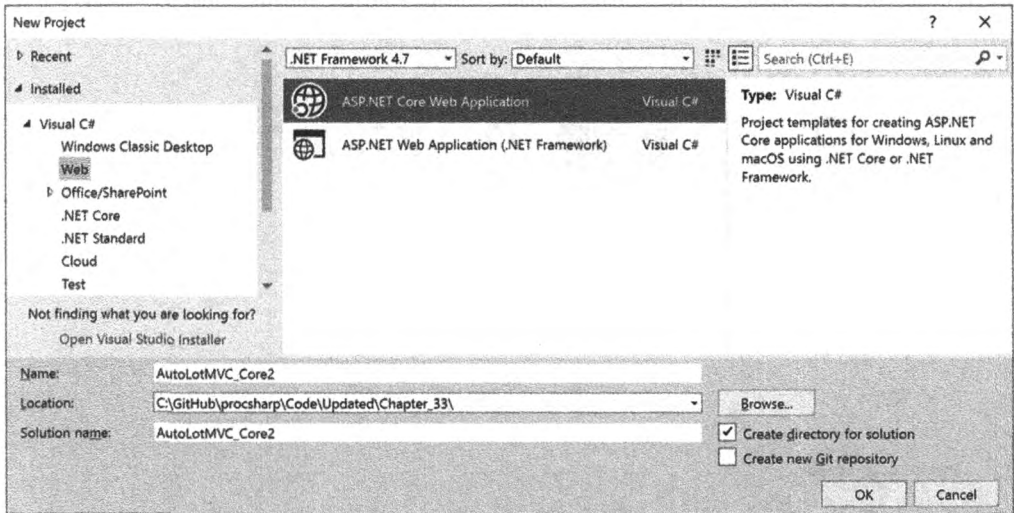


Рис. 33.1. Создание нового проекта веб-приложения ASP.NET Core

На следующем экране отображаются новые шаблоны, доступные вместе с ASP.NET Core 2.0 и Visual Studio 2017 версии 15.3. Они перечислены в табл. 33.1.

Таблица 33.1. Шаблоны ASP.NET Core 2.0

Шаблон	Описание
Empty (Пустой)	Пустой шаблон проекта для создания приложений ASP.NET Core
Web API (Web API)	Шаблон проекта для создания служб REST с примером контроллера. Может применяться для приложений MVC
Web Application (Веб-приложение)	Шаблон проекта для создания веб-приложений с использованием Razor Pages, укомплектованный примером содержимого
Web Application (Model-View-Controller) (Веб-приложение (модель-представление-контроллер))	Шаблон проекта для создания приложений ASP.NET Core с представлениями и контроллерами, содержащий их примеры. Также может применяться для построения служб REST
Angular (Angular)	Шаблон проекта для создания приложений ASP.NET Core с помощью Angular
React.js (React.js)	Шаблон проекта для создания приложений ASP.NET Core с помощью React.js
React.js and Redux (React.js и Redux)	Шаблон проекта для создания приложений ASP.NET Core с помощью React.js and Redux

Выберем шаблон Web Application (Model-View-Controller). Оставим флажок Enable Docker Support (Включить поддержку Docker) неотмеченным и установим аутентификацию в No Authentication (Аутентификация отсутствует), как показано на рис. 33.2.

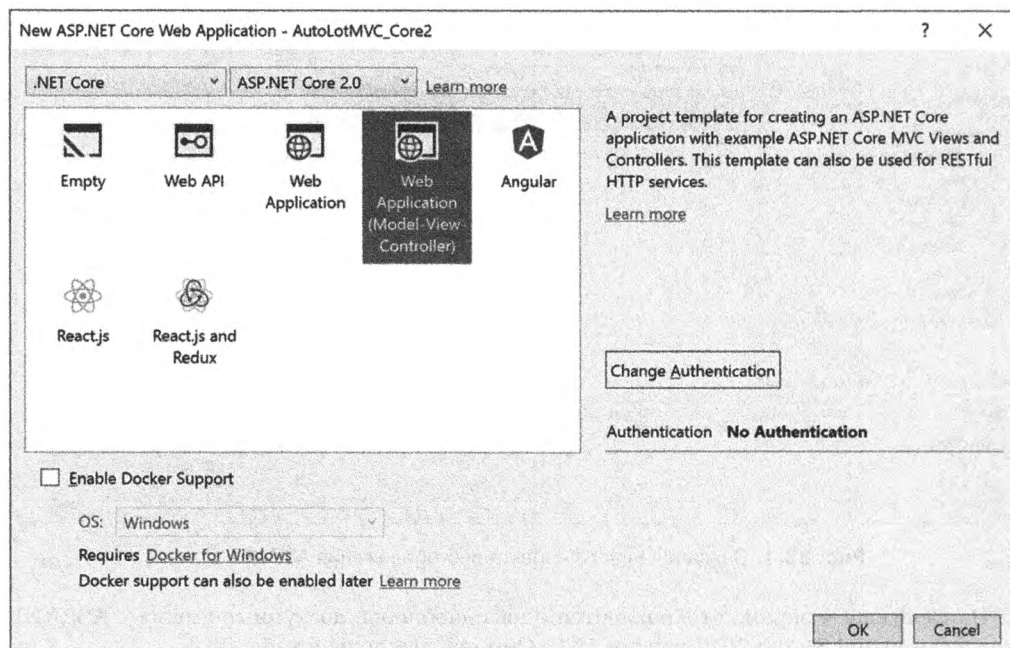


Рис. 33.2. Выбор шаблона Web Application (Model-View-Controller)

На заметку! Docker — это технология контейнеризации, рассмотрение которой выходит за рамки данной книги. Полная информация доступна по адресу <https://docs.microsoft.com/ru-ru/azure/vs-azure-tools-docker-hosting-web-apps-in-docker>.

Организация проекта ASP.NET Core

По сравнению с проектами приложений MVC 5 проекты ASP.NET Core структурированы иначе. Новая структура гораздо совершеннее и отделяет содержимое от кода. В папке `wwwroot` хранится содержимое клиентской стороны, такое как стили CSS, сценарии JavaScript, изображения и любое другое содержимое, не связанное с программированием.

Программное содержимое (наподобие моделей, представлений, контроллеров и т.д.) находится вне папки `wwwroot` и структурировано так, как можно было бы ожидать от приложения MVC.

Добавление библиотеки доступа к данным

Скопируем проекты `AutoLotDAL_Core2` и `AutoLotDAL_Core2.Models` из главы 32 в ту же папку, где содержится проект `AutoLotMVC_Core2`. Щелчком правой кнопкой мыши на решении `AutoLotMVC_Core2`, выберем в контекстном меню пункт `Add⇒Existing Project` (Добавить⇒Существующий проект) и добавим в решение два проекта доступа к данным. Если добавить в решение сначала проект `Models`, тогда ссылка из уровня доступа к данным на проект `Models` должна остаться невредимой. Добавим ссылки из проекта `AutoLotMVC_Core2` на проекты `AutoLotDAL_Core2` и `AutoLotDAL_Core2.Models`.

Обновление пакетов NuGet

Щелчком правой кнопкой мыши на проекте `AutoLotMVC_Core2` и выберем в контекстном меню пункт `Manage NuGet Packages` (Управление пакетами NuGet). В открывшемся окне `NuGet Package Manager` (Диспетчер пакетов NuGet) перейдем на вкладку `Installed` (Установленные); отобразятся только два пакета. Одним из них является `Microsoft.NETCore.App`, который поступает со всеми приложениями .NET Core.

Другой пакет, `Microsoft.AspNetCore.dll`, представляет собой метапакет, поставляемый с ASP.NET Core 2.0. Он содержит все необходимое для создания веб-приложения ASP.NET Core, значительно упрощая процесс выбора пакетов. Помимо включения всего того, что может понадобиться для веб-приложений ASP.NET, процесс развертывания удаляет пакеты, которые приложению не нужны.

Перейдем на вкладку `Updates` (Обновления) и обновим любые пакеты, которые требуют обновления.

Запуск приложений ASP.NET Core

Предшествующие версии веб-приложений ASP.NET запускались из IIS (или IIS Express во время разработки). Команда создателей ASP.NET Core приняла в качестве основной платформы выполнения продукт Kestrel — веб-сервер с открытым кодом, основанный на libuv. Продукт Kestrel обеспечивает единообразную практику разработки для всех платформ.

Добавление Kestrel и интерфейса командной строки (command-line interface — CLI) платформы .NET Core привело к появлению дополнительных вариантов запуска приложений ASP.NET Core. Теперь они могут запускаться (на этапе разработки) одним из четырех способов:

- из Visual Studio с использованием IIS Express;
- из Visual Studio с применением Kestrel;
- из консоли диспетчера пакетов через .NET CLI с использованием Kestrel;
- из окна командной строки через .NET CLI с применением Kestrel.

Файл `launchsettings.json` (расположенный внутри узла `Properties` (Свойства) в окне `Solution Explorer`) конфигурирует способ запуска приложения под управлением как Kestrel, так и IIS Express. Ниже для справки приведено содержимое файла `launchsettings.json` (номера портов могут отличаться):

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:54471/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  },
}
```

```

"AutoLotMVC_Core2": {
  "commandName": "Project",
  "launchBrowser": true,
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  },
  "applicationUrl": "http://localhost:54472/"
}
}
}

```

Узел `iisSettings` и узел `IIS Express` (под узлом `profiles`) соответствует пользовательскому интерфейсу для раздела `Debug` (Отладка) окна свойств проекта. Любые изменения, внесенные в этот пользовательский интерфейс, записываются в файл `launchSettings.json`, а любые изменения, внесенные в данный файл, обновляют пользовательский интерфейс. В разделе `iisSettings` определены настройки для запуска приложения с использованием `IIS Express` как веб-сервера. Среди самых важных настроек следует отметить `applicationUrl`, которая определяет порт, и блок `environmentVariables`, определяющий среду времени выполнения. Указанная настройка заменяет любую настройку среды на уровне машины.

Второй профиль (`AutoLotMVC_Core2`) определяет настройки для случая, когда приложение запускается с применением `Kestrel` в качестве веб-сервера. В профиле определяется настройка `applicationUrl` и порт, а также среда.

На заметку! В ранних версиях `.NET Core` и `ASP.NET Core` файл `launchSettings.json` использовался только при запуске приложения из `Visual Studio`. В текущем выпуске настройки применяются даже при запуске из `.NET CLI`.

Команда `run` в `Visual Studio` допускает выбор либо `IIS Express`, либо `Kestrel` (рис. 33.3).

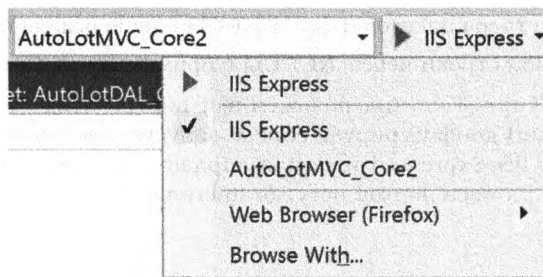


Рис. 33.3. Доступные профили отладки `Visual Studio`

Нажатие `<F5>` при выбранном профиле `IIS Express` приводит к открытию браузера со следующим URL (номер порта может отличаться):

```
http://localhost:54471/
```

Когда выбран профиль `Kestrel`, отладчик `Visual Studio` запускает браузер с таким URL (номер порта может отличаться):

```
http://localhost:54472/
```

Третий способ предусматривает запуск приложения из консоли диспетчера пакетов, используя `.NET CLI`. Для этого потребуется ввести в окне консоли диспетчера пакетов следующую команду:

```
dotnet run
```

На заметку! На время написания главы существовала проблема с применением .NET Core из консоли диспетчера пакетов — не было возможности завершить работу сервера Kestrel.

Последний способ предполагает открытие любого окна командной строки (которое не обязательно должно быть окном командной строки разработчика), переход в папку с файлом `AutoLotMVC_Core2.csproj` и ввод команды `dotnet run`. Чтобы закончить процесс, понадобится нажать комбинацию `<Ctrl+C>`.

Развертывание приложений ASP.NET Core

Предшествующие версии приложений ASP.NET можно было развертывать на серверах Windows только с использованием IIS. Инфраструктура ASP.NET Core допускает развертывание многими способами в средах многочисленных операционных систем (ОС), в том числе вне веб-сервера. Вот список высокоуровневых вариантов:

- на сервере Windows (включая Azure) с применением IIS;
- на сервере Windows (включая службы приложений Azure) вне IIS;
- на сервере Linux с использованием Apache;
- на сервере Linux с применением NGINX;
- на сервере Windows или Linux в контейнере Docker.

Безотносительно к цели развертывания основные шаги остаются неизменными.

1. Опубликовать приложение в папку на размещающем сервере.
2. Настроить диспетчер процессов, который запустит приложение при поступлении запросов (обеспечив его перезапуск в случае аварийного отказа или перезагрузки).
3. Настроить инвертированный прокси-сервер, который будет пересылать запросы приложению.

Все шаги подробно описаны в документации (<https://docs.microsoft.com/ru-ru/aspnet/core/host-and-deploy/index?tabs=aspnetcore2x&view=aspnetcore-2.1>).

Нововведения ASP.NET Core

Платформы ASP.NET MVC и ASP.NET Web API по-прежнему являются успешными и широко используются в наши дни. Так для чего понадобилась новая платформа? Первая причина уже обстоятельно обсуждалась: потребность запуска ASP.NET и Entity Framework в средах ОС, отличающихся от Windows, что было достигнуто устранением зависимости от `System.Web`.

Другая причина переписывания связана с тем, что кодовые базы для ASP.NET MVC 5 и ASP.NET Web API 2.2 устарели, по крайней мере, в эпоху Интернета. При наличии старой кодовой базы или в данном случае двух похожих кодовых баз, которые должны сохранять обратную совместимость, вводить новшества может быть затруднительно.

Решение повторно построить ASP.NET Core и иметь лишь одну инфраструктуру (вместо отдельных инфраструктур MVC и Web API) предоставило команде разработчиков возможность заняться переменами и совершенствованием набора функциональных средств, а также сосредоточиться на производительности в качестве первоочередной задачи. К тому же подобно EF Core, исходный код ASP.NET Core полностью открыт.

Ниже перечислены дополнительные улучшения и нововведения:

- унифицированный подход к построению веб-приложений и служб REST;
- встроенное внедрение зависимостей;

- система конфигурации, основанная на среде и готовая к взаимодействию с облачными технологиями;
- способность выполняться под управлением .NET Core или полной платформы .NET Framework;
- легковесный, высокопроизводительный и модульный конвейер запросов HTTP;
- интеграция современных инфраструктур клиентской стороны и рабочих потоков разработки;
- введение вспомогательных функций дескрипторов;
- введение компонентов представлений.

Все нововведения рассматриваются в последующих разделах.

Унифицированный подход к построению веб-приложений и служб

В отличие от MVC 5 и Web API 2.2 в ASP.NET Core есть только один базовый тип контроллера. Типы `Controller`, `ApiController` и `AsyncController` скомбинированы в один класс `Controller`. Действия контроллера возвращают `ActionResult` (или `Task<ActionResult>` для асинхронных операций) либо класс, который реализует `ActionResult`. Интерфейс `IActionResult` из Web API 2.2 совмещен с интерфейсом `IActionResult` из ASP.NET Core.

Встроенное внедрение зависимостей

Внедрение зависимостей (dependency injection — DI) представляет собой механизм для поддержки слабой связанности между объектами. Вместо создания зависимых объектов напрямую или передачи методам специфических реализаций классы и методы программируются так, чтобы получать интерфейсные типы. В результате методам и классам могут передаваться любые реализации интерфейсов, что разительно увеличивает гибкость приложения.

Поддержка DI — один из главных принципов, заложенных в переписанную версию ASP.NET Core. Не только класс `Startup` (рассматриваемый позже в главе) получает все службы конфигурации и промежуточного ПО через внедрение зависимостей, но ваши специальные классы могут (и должны) быть добавлены в контейнер службы, чтобы внедряться в другие части приложения.

Контейнер DI в ASP.NET Core обычно конфигурируется в методе `ConfigureServices()` класса `Startup` с применением экземпляра реализации `IServiceCollection`, который и сам внедряется. При конфигурировании элемента в контейнере DI доступны четыре варианта времени существования, кратко описанные в табл. 33.2.

Таблица 33.2. Варианты времени жизни для служб

Вариант времени существования	Предоставляемая функциональность
Временный (transient)	Создается каждый раз, когда в нем возникает необходимость
С заданной областью (scoped)	Создается один раз для каждого запроса. Рекомендуется для объектов <code>DbContext</code> инфраструктуры <code>Entity Framework</code>
Одноэлементный (singleton)	Создается один раз при первом запросе и затем многократно используется во время существования объекта. Рекомендуемый подход при реализации класса в соответствии с паттерном “Одиночка” (Singleton)
Связанный с экземпляром (instance)	Подобный одноэлементному, но создается при обращении к экземпляру в первом запросе

Дополнительные сведения о внедрении зависимостей и ASP.NET Core доступны в документации по адресу <https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.1>.

Система конфигурации, основанная на среде и готовая к взаимодействию с облачными технологиями

Фактически здесь охватываются два усовершенствования. Первое — это способность приложений ASP.NET Core задействовать настройки среды машины в конфигурационном процессе. Второе усовершенствование — значительно более простая система конфигурации.

Выяснение среды времени выполнения

Чтобы выяснить среду времени выполнения, приложения ASP.NET Core ищут переменную среды по имени `ASPNETCORE_ENVIRONMENT`. Значение указанной переменной среды обычно читается из файла настроек, из инструментов разработки (в случае запуска в режиме отладки) либо из переменных среды машины, выполняющей код. Имена могут быть любыми, но по соглашению разные среды определяются как `Development` (среда разработки), `Staging` (подготовительная среда) и `Production` (производственная среда).

После того, как среда установлена, разработчикам доступно много удобных методов для применения настройки в своих интересах. Например, следующий код включает регистрацию в журнале для среды разработки, но не для производственной или подготовительной среды:

```
if (env.IsDevelopment())
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();
}
```

Такое средство проверки среды используется в ASP.NET Core повсюду. В качестве простых примеров можно назвать выяснение, какие конфигурационные файлы необходимо загрузить, настройку параметров отладки, сообщения об ошибках и регистрации в журнале, а также загрузку файлов JavaScript и CSS, специфичных для среды. Вы увидите каждый прием в действии при построении веб-приложения `AutoLotMVC_Core2`.

За дополнительной информацией обращайтесь в документацию по адресу <https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/environments?view=aspnetcore-2.1>.

Конфигурация приложений

В предшествующих версиях ASP.NET для конфигурирования служб и приложений применялся файл `web.config`, и разработчики получали доступ к конфигурационным настройкам посредством класса `System.Configuration`. Разумеется, помещение в файл `web.config` всех конфигурационных настроек для сайта, а не только специфичных для приложения, делало его (потенциально) запутанной смесью.

В ASP.NET Core появилась значительно более простая система конфигурации. По умолчанию она основывается на простых файлах JSON, которые хранят конфигурационные настройки в виде пар “имя-значение”. Стандартный файл для конфигурации называется `appsettings.json`. Начальная версия файла `appsettings.json`, созданная шаблоном Visual Studio 2017, содержит просто конфигурационную информацию для регистрации в журнале:

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

Шаблон также создает файл `appsettings.Development.json`. Система конфигурации работает в сочетании с осведомленностью о среде времени выполнения, чтобы загружать дополнительные конфигурационные файлы на основе среды времени выполнения. Цель достигается инструктированием системы конфигурации о необходимости загрузки файла с именем `appsettings.{имя_среды}.json` после файла `appSettings.json`. В случае запуска приложения в среде разработки после файла начальных настроек загружается файл `appsettings.Development.json`. Если запуск происходит в подготовительной среде, тогда загружается файл `appsettings.Staging.json`. Важно отметить, что при загрузке более одного файла любые настройки, присутствующие в нескольких файлах, переопределяются настройками из последнего загруженного файла; они не являются аддитивными.

Строки подключения

Строки подключения обычно варьируются в зависимости от среды. Система конфигурации, основанная на среде, великолепно приспособлена для этого. Обновим файл `appsettings.Development.json`, чтобы добавить строку подключения (она должна совпадать со строкой подключения, которая использовалась в главе 32):

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "ConnectionStrings": {
    "AutoLot":
      "server=(LocalDb)\\MSSQLLocalDB;database=AutoLotCore2;
integrated security=True;MultipleActiveResultSets=True;App=EntityFramework;"
  }
}
```

Извлечение настроек

После построения конфигурации к настройкам можно обращаться с применением традиционного семейства методов `GetXXX()`, таких как `GetSection()`, `GetValue()` и т.д.:

```
Configuration.GetSection("Logging")
```

Также доступно сокращение для получения строк подключения:

```
Configuration.GetConnectionString("AutoLot")
```

Способность выполняться под управлением .NET Core или полной платформы .NET Framework

Как и EF Core, инфраструктуру ASP.NET Core можно использовать в любом приложении .NET, будь оно предназначено для полной платформы .NET Framework или для .NET Core, при условии развертывания на машине с Windows. В случае развертывания на машине с macOS или Linux есть только один вариант — применение ASP.NET Core в приложениях .NET Core.

Вас может интересовать, по какой причине может возникнуть желание использовать ASP.NET Core в приложении для полной платформы .NET Framework. Одна причина довольно проста: необходимость задействовать в своих интересах все функциональные возможности и улучшенную производительность, предлагаемые ASP.NET Core. Еще одна причина — доверие к имеющемуся коду служебного или среднего уровня (либо даже коду в сторонних библиотеках), который не был перенесен в .NET Core.

Однако в отличие от EF Core и EF 6 инфраструктуры ASP.NET Core и ASP.NET MVC не могут сосуществовать в рамках одного приложения. Это означает, что придется обдумать перенос существующих приложений MVC 5 или Web API 2.2 в ASP.NET Core. Кроме того, по правде говоря, нет какого-то пути “модернизации” от MVC 5 (или Web API) до ASP.NET Core. Фактически речь идет о чем-то большем, нежели о переносе приложения. Несмотря на то что много кода можно применить повторно, копируя и вставляя его в новое приложение, отсутствует некая “магическая кнопка”, нажав которую удалось бы внезапно получить приложение ASP.NET Core.

Легковесный и модульный конвейер запросов HTTP

Следуя основным принципам .NET Core, все в ASP.NET Core происходит по подписке. По умолчанию в приложение ничего не загружается. Как вы увидите позже в главе, даже поддержка статических файлов должна добавляться явно.

Такой подход позволяет приложениям быть насколько возможно легковесными, улучшая производительность и сводя к минимуму объем их кода.

Интеграция современных инфраструктур клиентской стороны

Вы заметили на рис. 33.2, что существуют готовые шаблоны для приложений ASP.NET Core, использующие Angular и React. Распространение инфраструктур клиентской стороны означает, что любое веб-приложение должно быть в состоянии применять инфраструктуру JavaScript по своему выбору, не замыкаясь на стеке Microsoft.

На заметку! Хотя в настоящей книге не раскрывается использование инфраструктур JavaScript клиентской стороны JavaScript вместе с ASP.NET Core, в книге *Building Web Applications with Visual Studio 2017* (<https://www.amazon.com/Building-Applications-Visual-Studio-2017/dp/1484224779/>) рассматривается применение Angular и React с инфраструктурой ASP.NET Core.

Вспомогательные функции дескрипторов

Вспомогательные функции дескрипторов — новое средство в ASP.NET Core, которое значительно улучшает практику разработки и читабельность представлений MVC. В отличие от вспомогательных методов HTML, которые вызываются как методы Razor, вспомогательные функции дескрипторов добавляются к элементам HTML.

Вспомогательные функции дескрипторов инкапсулируют код серверной стороны, который придает форму присоединенному элементу. Наряду с тем, что вспомогательные методы HTML по-прежнему поддерживаются, для элементов управления вводом в приложениях ASP.NET Core рекомендуется использовать вспомогательные функции дескрипторов. Если разработка ведется с помощью Visual Studio, то для вспомогательных функций дескрипторов имеется дополнительное преимущество в виде IntelliSense.

Чтобы увидеть разницу между вспомогательными методами HTML и вспомогательными функциями дескрипторов, исследуем приведенный ниже вспомогательный метод HTML, который создает метку для свойства FullName модели представления:

```
@Html.Label("FullName", "Full Name:", new { @class = "customer" })
```

В результате генерируется следующая разметка HTML:

```
<label class="customer" for="FullName">Full Name:</label>
```

Вы видели его (и многие другие вспомогательные методы HTML) в действии в главе 29. В то время как вспомогательные методы HTML знакомы разработчикам на языке C#, имевшим дело с ASP.NET MVC и Razor, синтаксис вспомогательных методов HTML непрост для понимания особенно теми, кто работал с HTML/CSS/JavaScript, но не с C#. Проблема связана с добавлением атрибутов HTML и данных маршрутизации (где необходимо), т.к. требуется создавать и передавать анонимные объекты, а поддержка IntelliSense отсутствует.

В качестве сравнения ниже показан фрагмент разметки HTML, которая делает то же самое, что и предыдущий вспомогательный метод HTML:

```
<label class="customer" asp-for="FullName">Full Name:</label>
```

Обратите внимание на аккуратность разметки. Дескриптор HTML создается нормальным образом с добавлением нового атрибута asp-for, который представляет собой вспомогательную функцию дескриптора. Он ссылается на то же свойство FullName модели, обнаруживается IntelliSense и подобно вспомогательному методу HTML поддерживает аннотацию данных DisplayName. Дополнительные атрибуты HTML не нужно передавать с анонимным объектом; они просто добавляются к HTML-дескриптору label.

Предыдущий фрагмент разметки HTML и вспомогательная функция дескриптора выпускают такой же вывод, как и представленный ранее пример вспомогательного метода HTML, но обладают большей читабельностью, что особенно важно для дизайнеров, которые могут не понимать синтаксис C# или вспомогательных методов HTML.

Существует много встроенных вспомогательных функций дескрипторов, которые предназначены для применения вместо соответствующих вспомогательных методов HTML. Тем не менее, не все вспомогательные методы HTML имеют связанные вспомогательные функции дескрипторов. В табл. 33.3 перечислены доступные вспомогательные функции дескрипторов вместе с соответствующими им вспомогательными методами HTML и атрибутами. Все они кроме вспомогательной функции дескриптора label, которая демонстрировалась выше, будут обсуждаться в последующих разделах.

В дополнение к длинному списку готовых вспомогательных функций дескрипторов можно также создавать специальные вспомогательные функции дескрипторов, которые будут обсуждаться после встроенных вспомогательных функций дескрипторов.

Таблица 33.3. Встроенные вспомогательные функции дескрипторов

Вспомогательная функция дескриптора	Вспомогательный метод HTML	Доступные атрибуты
Дескриптор form	Html.BeginForm() Html.BeginRouteForm() Html.AntiForgeryToken()	<p>asp-route: для именованных маршрутов (не может использоваться с атрибутами контроллеров или действий).</p> <p>asp-antiforgery: указывает, должен ли добавляться маркер противодействия подделке (по умолчанию true).</p> <p>asp-area: устанавливает имя области.</p> <p>asp-controller: устанавливает контроллер.</p> <p>asp-action: устанавливает действие.</p> <p>asp-route-<ИмяПараметра>: добавляет параметр к маршруту (например, asp-route-id="1").</p> <p>asp-all-route-data: словарь с дополнительными значениями для маршрута.</p> <p>Примечание: эта вспомогательная функция дескриптора автоматически добавляет маркер противодействия подделке, если только он явно не отключен с применением вспомогательной функции дескриптора asp-antiforgery</p>
Дескриптор a	Html.ActionLink()	<p>asp-route: для именованных маршрутов (не может использоваться с атрибутами контроллеров или действий).</p> <p>asp-area: устанавливает имя области.</p> <p>asp-controller: определяет контроллер.</p> <p>asp-action: определяет действие.</p> <p>asp-protocol: определяет протокол HTTP или HTTPS.</p> <p>asp-fragment: определяет фрагмент URL.</p> <p>asp-host: устанавливает имя хоста.</p> <p>asp-route-<ИмяПараметра>: добавляет параметр к маршруту (например, asp-route-id="1").</p> <p>asp-all-route-data: словарь с дополнительными значениями для маршрута</p>

Вспомогательная функция дескриптора	Вспомогательный метод HTML	Доступные атрибуты
Дескриптор input	Html.TextBox()/Html.TextBoxFor() Html.Editor()/Html.EditorFor()	asp-for: свойство модели. Можно перемещаться по модели (Customer.Address.AddressLine1) и применять выражения (asp-for="@localVariable"). Атрибуты id и name генерируются автоматически. Любые атрибуты data-val из HTML5 генерируются автоматически.
Дескриптор textarea	Html.TextAreaFor()	asp-for: свойство модели. Можно перемещаться по модели (Customer.Address.Description) и использовать выражения (asp-for="@localVariable"). Атрибуты id и name генерируются автоматически. Любые атрибуты data-val из HTML5 генерируются автоматически.
Дескриптор label	Html.LabelFor()	asp-for: свойство модели. Можно перемещаться по модели (Customer.Address.AddressLine1) и применять выражения (asp-for="@localVariable").
Дескриптор select	Html.DropDownListFor() Html.ListBoxFor()	asp-for: свойство модели. Можно перемещаться по модели (Customer.Address.AddressLine1) и использовать выражения (asp-for="@localVariable"). asp-items: указывает элементы option. Автоматически генерирует атрибут selected="selected". Атрибуты id и name генерируются автоматически. Любые атрибуты data-val из HTML5 генерируются автоматически.
Сообщение об ошибке при проверке достоверности (дескриптор span)	Html.ValidationMessageFor()	asp-validation-for: свойство модели. Можно перемещаться по модели (Customer.Address.AddressLine1) и применять выражения (asp-for="@localVariable"). Добавляет атрибут data-valmsg-for к элементу span.
Сводка по проверке достоверности (дескриптор div)	Html.ValidationSummaryFor()	asp-validation-summary: одно из значений All, ModelOnly и None. Добавляет атрибут data-valmsg-summary к элементу div.

Вспомогательная функция дескриптора	Вспомогательный метод HTML	Доступные атрибуты
Дескриптор link	–	<p>asp-append-version: добавляет версию к имени файла.</p> <p>asp-fallback-href: запасной файл для использования, если основной файл оказывается недоступным; обычно применяется с источниками CDN.</p> <p>asp-fallback-href-include: список, содержащий шаблоны имен файлов, которые должны быть включены в запасные.</p> <p>asp-fallback-href-exclude: список, содержащий шаблоны имен файлов, которые должны быть исключены из запасных.</p> <p>asp-fallback-test-*: свойства для использования в проверке обхода. Включены class, property и value.</p> <p>asp-href-include: список, содержащий шаблоны имен файлов, который нужно включить.</p> <p>asp-href-exclude: список, содержащий шаблоны имен файлов, который нужно исключить.</p>
Дескриптор script	–	<p>asp-append-version: добавляет версию к имени файла.</p> <p>asp-fallback-src: запасной файл для использования, если основной файл оказывается недоступным; обычно применяется с источниками CDN.</p> <p>asp-fallback-src-include: список, содержащий шаблоны имен файлов, которые должны быть включены в запасные.</p> <p>asp-fallback-src-exclude: список, содержащий шаблоны имен файлов, которые должны быть исключены из запасных.</p> <p>asp-fallback-test: сценарный метод для использования в проверке обхода.</p> <p>asp-src-include: список, содержащий шаблоны имен файлов, который нужно включить.</p> <p>asp-src-exclude: список, содержащий шаблоны имен файлов, который нужно исключить.</p>

Вспомогательная функция дескриптора	Вспомогательный метод HTML	Доступные атрибуты
Дескриптор image	-	asp-append-version: добавляет версию к имени файла.
Дескриптор environment	-	name: одно из значений Development, Staging и Production.

Вспомогательная функция дескриптора *form*

Вспомогательная функция дескриптора `form` заменяет вспомогательные методы HTML под названиями `Html.BeginForm()` и `Html.BeginRouteForm()`. Одно из преимуществ вспомогательной функции дескриптора `form` перед версией вспомогательного метода HTML связано с тем, что все вспомогательные функции дескриптора `form` по умолчанию включают маркер противодействия подделке.

Например, для создания формы, которая отправляет версию POST действия `Edit` из контроллера `InventoryController` с одним параметром (`id`), применяется приведенный ниже код:

```
<form method="post"
    asp-controller="Inventory"
    asp-action="Edit"
    asp-route-id="5">
    <!-- Для краткости разметка не показана -->
</form>
```

Подобно аналогу `HTML.BeginForm()` стандартные установки можно не указывать. Исходя из предположения, что страницей редактирования является `\Inventory\Edit\5` и запрос `HttpPost` должен проследовать по тому же самому маршруту, вспомогательную функцию дескриптора `form` можно было бы написать примерно так:

```
<form asp-action="Edit">
    <!-- Для краткости разметка не показана -->
</form>
```

Обратите внимание, что должна быть включена хотя бы одна вспомогательная функция дескриптора для ASP.NET Core, чтобы это считалось формой MVC. Вот как выглядит результирующая разметка HTML для представления:

```
<form action="/Inventory/Edit/5" method="post" novalidate="novalidate">
    <input name="__RequestVerificationToken" value="CfDJ8MS4lwJsL5BBhMUQM4d2aMQ2
LWeBJhUL4Iz1KM3LxaOr_NpB_QWvurqZybbB0tPXHVdME6M5bGOoR2aBkh1tRt6mTl6fnctt2CP_
kaHcVoiqcGNM4XcesnihpmEK0vyg8mFybqRMeU6ezvjOB6VIT50" type="hidden">
    <!-- Для краткости разметка не показана -->
</form>
```

Вспомогательная функция дескриптора *a*

Вспомогательная функция дескриптора `a` заменяет вспомогательный метод HTML по имени `Html.ActionLink()`. Например, чтобы создать ссылку на редактирование элемента, необходимо использовать следующий код:

```
<a asp-controller="Inventory" asp-action="Edit"
    asp-route-id="@Model.Id">Edit</a>
```

Указанная вспомогательная функция дескриптора создает URL из таблицы маршрутов с применением контроллера `InventoryController`, метода действия `Edit()` и текущего значения для `Id`. Текстом дескриптора а вполне ожидаемо будет `Edit`. Ниже показан результирующий URL:

```
<a href="/Inventory/Edit/5">Edit</a>
```

Вспомогательная функция дескриптора `input`

Вспомогательная функция дескриптора `input` является самой универсальной. Помимо автоматической генерации разметки HTML-атрибутов `id` и `name`, а также любых атрибутов `data-val` из HTML5, связанных с проверкой достоверности, вспомогательная функция дескриптора `input` строит соответствующую разметку HTML на основе типа данных целевого свойства. В табл. 33.4 перечислены атрибуты типа HTML, которые создаются на базе типа .NET свойства.

Таблица 33.4. Атрибуты типа HTML, генерируемые из типов .NET с использованием вспомогательной функции дескриптора `input`

Тип .NET	Генерируемый атрибут типа HTML
Bool	<code>type="checkbox"</code>
String	<code>type="text"</code>
DateTime	<code>type="datetime"</code>
Byte, Int, Single, Double	<code>type="number"</code>

Кроме того, вспомогательная функция дескриптора `input` добавит атрибуты типов HTML5, основываясь на аннотациях данных. В табл. 33.5 представлены некоторые распространенные аннотации данных и генерируемые атрибуты типов HTML5.

Таблица 33.5. Атрибуты типов HTML5, генерируемые из аннотаций данных .NET

Аннотация данных .NET	Генерируемый атрибут типа HTML5
<code>EmailAddress</code>	<code>type="email"</code>
<code>Url</code>	<code>type="url"</code>
<code>HiddenInput</code>	<code>type="hidden"</code>
<code>Phone</code>	<code>type="tel"</code>
<code>DataType(DataType.Password)</code>	<code>type="password"</code>
<code>DataType(DataType.Date)</code>	<code>type="date"</code>
<code>DataType(DataType.Time)</code>	<code>type="time"</code>

Например, чтобы отобразить текстовое поле для редактирования значения свойства `Make` автомобиля, понадобится ввести разметку `<input asp-for="Make" class="form-control" />`.

Предполагая, что значением поля `Make` является `BMW`, сгенерируется следующая разметка HTML:

```
<input class="form-control" data-val="true"
data-val-length="The field Make must be a string with a maximum length of 50."
data-val-length-max="50" id="Make" name="Make" value="BMW" type="text">
```

Атрибуты `id`, `name`, `value` и `data-val` добавляются автоматически.

Вспомогательная функция дескриптора *textarea*

Вспомогательная функция дескриптора `textarea` автоматически добавляет атрибуты `id` и `name`, а также любые дескрипторы HTML5, связанные с проверкой достоверности, которые определены для свойства. Например, приведенная далее строка создает дескриптор `textarea` для свойства `Description`:

```
<textarea asp-for="Description"></textarea>
```

Вот результирующая разметка:

```
<textarea name="Description" id="Description" data-val="true"
  data-val-maxlength-max="3800"
  data-val-maxlength="The field Description must be a string or array type
with a maximum length of '3800'.">
  This is the text that is shown in the text area control</textarea>
```

Вспомогательная функция дескриптора *select*

Вспомогательная функция дескриптора `select` строит дескрипторы `select` из свойства модели и коллекции. Как и с другими вспомогательными функциями дескрипторов `input`, в разметку добавляются атрибуты `id` и `name`, а также любые атрибуты `data-val` из HTML5. Если значение свойства модели совпадает с одним из значений в списке элементов `select`, тогда в разметку для данного элемента добавляется атрибут `selected`.

Например, возьмем модель, которая имеет свойство по имени `Country` и свойство типа `SelectList` по имени `Countries` со следующим определением:

```
public List<SelectListItem> Countries { get; } = new List<SelectListItem>
{
    new SelectListItem { Value = "MX", Text = "Mexico" },
    new SelectListItem { Value = "CA", Text = "Canada" },
    new SelectListItem { Value = "US", Text = "USA" },
};
```

Показанная ниже разметка визуализирует дескриптор `select` с соответствующими элементами:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Если значение свойства `Country` установлено в `CA`, то в представление будет выведена такая разметка:

```
<select id="Country" name="Country">
  <option value="MX">Mexico</option>
  <option selected="selected" value="CA">Canada</option>
  <option value="US">USA</option>
</select>
```

Мы лишь слегка коснулись поверхности того, на что способна вспомогательная функция дескриптора `select`. Больше сведений можно найти в документации по адресу <https://docs.microsoft.com/ru-ru/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.1#the-select-tag-helper>.

Вспомогательные функции дескрипторов для результатов проверки достоверности

Вспомогательные функции дескрипторов для сообщений об ошибках при проверке достоверности и сводке по проверке достоверности близко отражают вспомогательные методы HTML под названиями `Html.ValidationMessageFor()` и `Html.ValidationSummaryFor()`.

Первая применяется к HTML-дескриптору `span` для специфического свойства модели, а вторая — к дескриптору `div`, относящемуся ко всей модели. Вспомогательная функция дескриптора для сводки по проверке достоверности поддерживает варианты `All` (отображать все ошибки), `ModelOnly` (исключить ошибки, связанные со свойствами модели) и `None` (ничего не отображать).

Следующая разметка добавляет вспомогательную функцию дескриптора для сообщений об ошибках к свойству `Make` модели:

```
<span asp-validation-for="Make" class="text-danger"></span>
```

Если значение свойства `Make` окажется ошибочным, тогда вывод будет примерно таким:

```
<span class="text-danger field-validation-error" data-valmsg-for="Make" data-valmsgreplace="true"><span id="Make-error" class="">
  The field Make must be a string with a maximum length of 50.</span></span>
```

Проверка достоверности и привязка моделей работают точно так же, как в инфраструктурах ASP.NET MVC 5 и ASP.NET Web API 2.2.

Вспомогательные функции дескрипторов *link* и *script*

Вспомогательные функции дескрипторов `link` и `script` используются для улучшения восприятия пользователями интерфейса. Атрибут `asp-append-version` добавляет к `href` хеш файла, на который производится ссылка, так что кеш браузера становится недействительным, когда файл изменяется. Это вызывает повторную загрузку файла, даже если его имя не менялось. Ниже приведен пример атрибута `asp-append-version`:

```
<link rel="stylesheet" href="~/css/site.css" asp-append-version="true"/>
```

Вот сгенерированная разметка:

```
<link href="/css/site.css?v=v9cmzjNgxPHiyLIrNom5fw3tZj3TNT2QD7a0hBrSa4U"
  rel="stylesheet">
```

Атрибуты `asp-fallback-*` обычно применяются с источниками файлов из сети доставки содержимого (content delivery network — CDN). Например, следующая разметка будет пытаться загрузить файл `jquery-3.1.1.min.js` из сети Microsoft CDN. Если попытка терпит неудачу, тогда загрузится локальная версия `jquery.min.js`.

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.1.1.min.js"
  asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
  asp-fallback-test="window.jQuery">
```

Вспомогательная функция дескриптора *image*

Вспомогательная функция дескриптора `image` предоставляет атрибут `asp-append-version`, который работает аналогично атрибутам вспомогательных функций дескрипторов `link` и `script`.

Вспомогательная функция дескриптора *environment*

Вспомогательная функция дескриптора `environment` используется для условной загрузки файлов на основе текущей среды, внутри которой выполняется сайт. Показанная ниже разметка будет загружать неминифицированные файлы, если сайт запущен в среде разработки, и минифицированные версии, если сайт выполняется не в среде разработки:

```
<environment include="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</environment>
```



```
<environment exclude="Development">
  <link rel="stylesheet"
    href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
    asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
    asp-fallback-test-class="sr-only"
    asp-fallback-test-property="position"
    asp-fallbacktest-value="absolute" />
  <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

На заметку! До выхода версии ASP.NET Core 2.0 во вспомогательной функции дескриптора `environment` вместо атрибута `include` применялся атрибут `names`, где нужно было указывать специфические имена каждой среды. В версии ASP.NET Core 2.0 взамен используются атрибуты `include` и `exclude`.

Включение вспомогательных функций дескрипторов

Подобно любому другому коду вспомогательные функции дескрипторов должны быть сделаны видимыми для кода, в котором их желательно применять.

Файл `_ViewImports.html` содержит следующую строку:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Данная строка делает все вспомогательные функции дескрипторов в сборке `Microsoft.AspNetCore.Mvc.TagHelpers` (которая содержит все встроенные вспомогательные функции дескрипторов) доступными всем представлениям.

Специальные вспомогательные функции дескрипторов

Можно также создавать специальные вспомогательные функции дескрипторов, которые способны помочь в устранении повторяющегося кода, такого как построение ссылок `mailto:..`. Создадим в корне проекта `AutoLotMVC_Core2` новую папку по имени `TagHelpers`, а в ней новый файл класса `EmailTagHelper.cs`. Поместим в начало файла такой оператор `using`:

```
using Microsoft.AspNetCore.Razor.TagHelpers;
```

Унаследуем класс от `TagHelper`:

```
public class EmailTagHelper : TagHelper
{
}
```

Добавим два свойства для хранения `EmailName` и `EmailDomain`:

```
public string EmailName { get; set; }
public string EmailDomain { get; set; }
```

Любые открытые свойства специальной вспомогательной функции дескриптора доступны как ее атрибуты, представленные в "кебаб"-стиле нижнего регистра (`lower-kebab-case`). Для `EmailTagHelper` значения им передаются следующим образом:

```
<email email-name="blog" email-domain="skimedic.com"></email>
```

При обращении к специальной вспомогательной функции дескриптора вызывается метод `Process()`, принимающий два параметра — `TagHelperContext` и `TagHelperOutput`. Параметр `TagHelperContext` используется для получения любых других атрибутов дескриптора и словаря объектов, который применяется для взаимодействия с остальными вспомогательными функциями дескрипторов, нацеленными на дочерние элементы. Параметр `TagHelperOutput` используется для создания визуализированного вывода.

Добавим в класс `EmailTagHelper` метод `Process()`:

```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    output.TagName = "a"; // Заменяет <email> дескриптором <a>
    var address = EmailName + "@" + EmailDomain;
    output.Attributes.SetAttribute("href", "mailto:" + address);
    output.Content.SetContent(address);
}
```

Обеспечение видимости специальных вспомогательных функций дескрипторов

Подобно операторам `using` команда `@addTagHelper` применяется для того, чтобы сделать вспомогательные функции дескрипторов видимым для представлений, в которых они будут использоваться. Ее можно добавлять в каждое представление, где нужен доступ к вспомогательным функциям дескрипторов, или с помощью нового файла `_ViewImports.chtml` предоставить глобальный доступ любому представлению на уровне папки, где находится файл `_ViewImports.chtml` или ниже ее.

Откроем файл `_ViewImports.cshtml` в корне папки `Views` и добавим в него следующую строку, чтобы сделать класс `EmailTagHelper` видимым для представлений в приложении:

```
@addTagHelper *, AutoLotMVC_Core2
```

Звездочка (*) указывает на то, что должны быть доступными все вспомогательные функции дескрипторов в сборке `AutoLotMVC_Core2`. При необходимости можно также перечислить специфические вспомогательные функции дескрипторов.

Использование специальных вспомогательных функций дескрипторов

Откроем файл представления `Contact.cshtml` и найдем в нем строки, которые отображают адреса электронной почты:

```
<address>
    <strong>Support:</strong>
    <a href="mailto:Support@example.com">Support@example.com</a><br />
    <strong>Marketing:</strong>
    <a href="mailto:Marketing@example.com">Marketing@example.com</a>
</address>
```

Заменим их приведенным ниже кодом, чтобы задействовать новую специальную вспомогательную функцию дескриптора:

```
<address>
    <strong>Support:</strong>
    <email email-name="Support" email-domain="example.com" /><br />
    <strong>Marketing:</strong>
    <email email-name="Marketing" email-domain="example.com" />
</address>
```

Компоненты представлений

Компоненты представлений являются нововведением ASP.NET Core. Они сочетают в себе преимущества частичных представлений с дочерними действиями, предназначенных для визуализации частей пользовательского интерфейса. Подобно частичным представлениям компоненты представлений вызываются из другого представления, но в отличие от самих по себе частичных представлений они также имеют компонент серверной стороны. Такая комбинация делает компоненты представлений великолепно подходящими для функций вроде создания динамических меню, панелей входа, содер-

жимого боковых панелей и всего того, что нуждается в запуске кода серверной стороны, но не может квалифицироваться как отдельное представление. Позже в главе мы рассмотрим пример компонента представления.

Изменения механизма представлений Razor

Механизм представлений Razor в ASP.NET Core несколько отличается по сравнению с MVC 5. Хотя большинство его возможностей дублируются в ASP.NET Core, специальные вспомогательные методы HTML не поддерживаются.

Построение AutoLotMVC_Core2

Теперь, когда вы ознакомились с новыми средствами ASP.NET Core, наступило время изучить остаток инфраструктуры, построив приложение AutoLotMVC_Core2. В процессе работы будет детально исследоваться каждая порция инфраструктуры ASP.NET Core и при необходимости обновляться шаблонные файлы.

Файл Program.cs

Приложения ASP.NET Core — это просто консольные приложения, создающие веб-сервер в методе Main(). Веб-сервер строится с применением экземпляра класса WebHostBuilder. Откроем файл Program.cs и взглянем на его содержимое, которое в справочных целях показано далее:

```
public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }
    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}
```

Метод CreateDefaultBuilder() сжимает обычную настройку до одного вызова метода. Данный метод образован следующим кодом:

```
return new WebHostBuilder()
    .UseKestrel()
    .UseContentRoot(Directory.GetCurrentDirectory())
    .ConfigureAppConfiguration(
        (Action<WebHostBuilderContext, IConfigurationBuilder>)
            ((hostingContext, config) =>
            {
                IHostingEnvironment hostingEnvironment = hostingContext.HostingEnvironment;
                config.AddJsonFile("appsettings.json", true, true).AddJsonFile(
                    string.Format("appsettings.{0}.json",
                        (object) hostingEnvironment.EnvironmentName), true, true);
                if (hostingEnvironment.IsDevelopment())
                {
                    Assembly assembly =
                        Assembly.Load(new AssemblyName(hostingEnvironment.ApplicationName));
                    if (assembly != (Assembly) null)
                        config.AddUserSecrets(assembly, true);
                }
            })
    );
```

```

        config.AddEnvironmentVariables();
        if (args == null)
            return;
        config.AddCommandLine(args);
    })
    .ConfigureLogging((Action<WebHostBuilderContext, ILoggingBuilder>)
        ((hostingContext, logging) =>
        {
            logging.AddConfiguration((IConfiguration)
                hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
        }))
    .UseIISIntegration()
    .UseDefaultServiceProvider((Action<WebHostBuilderContext,
        ServiceProviderOptions>)
        ((context, options) =>
            options.ValidateScopes = context.HostingEnvironment.IsDevelopment()));

```

Предыдущий код иллюстрирует модульность инфраструктуры ASP.NET Core и природу предоставления ее служб по подписке. После создания нового экземпляра `WebHostBuilder` код включает только средства, которые нужны приложению. Затем включается веб-сервер и в качестве каталога проекта устанавливается корневой каталог для содержимого. Следующий блок кода конфигурирует приложение и демонстрирует, как использовать разные файлы JSON на основе текущей среды выполнения.

На заметку! Описанное положение дел отличается от предшествующих версий ASP.NET Core, где весь конфигурационный код обрабатывался в классе `Startup`, а в методе `Main()`. Код был перемещен в `Main()`, чтобы добиться большей чистоты класса `Startup` и приблизиться к полному соблюдению принципа единственной обязанности.

После конфигурирования приложения и регистрации в журнале добавляется интеграция с IIS (в дополнение к Kestrel). Наконец, конфигурируется стандартный поставщик служб для внедрения зависимостей.

Возвращаясь к методу `BuildWebHost()`, метод `UseStartup()` устанавливает класс конфигурации в `Startup.cs` и строит экземпляр `WebHostBuilder`, полностью сконфигурированный для приложения.

Внутри метода `Main()` на экземпляре `WebHostBuilder` вызывается метод `Run()` для активизации веб-хоста.

Добавление поддержки инициализации данных

Инициализация данных должна происходить в файле `Program.cs`. Именно здесь будет выполняться инициализатор данных `AutoLotDAL_Core2`. Добавим в начало файла `Program.cs` перечисленные далее операторы `using`:

```

using AutoLotDAL_Core2.DataInitialization;
using AutoLotDAL_Core2.EF;
using Microsoft.Extensions.DependencyInjection;

```

Модифицируем код метода `Main()`, как показано ниже:

```

var webHost = BuildWebHost(args);
using (var scope = webHost.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<AutoLotContext>();
}

```

```

    MyDataInitializer.RecreateDatabase(context);
    MyDataInitializer.InitializeData(context);
}
webHost.Run();

```

Код получает область действия контейнера DI и применяет ее для извлечения объекта `AutoLotContext` из контейнера. Затем он вызывает методы для удаления и восстановления базы данных и добавляет тестовые данные.

На заметку! Попытка запуска проекта прямо сейчас закончится неудачей, поскольку не был сконфигурирован контейнер DI. Мы сделаем это следующим.

Файл `Startup.cs`

Класс `Startup` устанавливает способ обработки приложениям запросов и ответов HTTP, конфигурирует любые необходимые службы и настраивает контейнер внедрения зависимостей. Класс может иметь любое имя при условии, что оно соответствует имени, указанному в строке `UseStartup<T>` внутри конфигурации `WebHostBuilder`, но по соглашению класс именуется как `Startup`.

Откроем файл `Startup.cs` и обновим операторы `using`:

```

using AutoLotDAL_Core2.EF;
using AutoLotDAL_Core2.Repos;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

```

Располагая надлежащими операторами `using`, самое время выяснить, какие службы доступны в классе `Startup`.

Доступные службы для класса `Startup`

Процесс запуска нуждается в доступе к инфраструктуре, а также к службам среды и значениям; все это внедряется в класс инфраструктурой. Классу `Startup` доступны пять служб для конфигурирования приложения, которые перечислены в табл. 33.6. В последующих разделах службы будут продемонстрированы в действии.

Таблица 33.6. Доступные службы для `Startup`

Служба	Предоставляемая функциональность
<code>IApplicationBuilder</code>	Определяет класс, который предоставляет механизм для конфигурирования конвейера запросов приложения
<code>IHostingEnvironment</code>	Предоставляет информацию о среде веб-хостинга, в которой выполняется приложение
<code>ILoggerFactory</code>	Используется для конфигурирования системы ведения журнала и создания экземпляров журнальных классов из зарегистрированных поставщиков журналов
<code>IServiceCollection</code>	Указывает контракт для коллекции дескрипторов служб. Является частью инфраструктуры внедрения зависимостей
<code>IConfiguration</code>	Представляет собой экземпляр конфигурации приложения, созданный в методе <code>Main()</code> класса <code>Program</code>

Конструктор принимает экземпляр реализации `IConfiguration` и необязательный экземпляр реализации `IHostingEnvironment`, хотя в типовых решениях применяется только параметр `IConfiguration`. Метод `Configure()` должен принимать экземпляр реализации `IApplicationBuilder`, но может также принимать экземпляры реализаций `IHostingEnvironment` и/или `ILoggerFactory`. Метод `ConfigureServices()` принимает экземпляр реализации `IServiceCollection`.

Шаблонный класс `Startup` с внедренными службами выглядит примерно так:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // Этот метод вызывается исполняющей средой.
    // Используйте его для добавления служб в контейнер.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    // Этот метод вызывается исполняющей средой.
    // Применяйте его для конфигурирования конвейера запросов HTTP.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseBrowserLink();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
        }
        app.UseStaticFiles();
        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

Все методы класса `Startup` будут обсуждаться в последующих разделах.

Конструктор

Конструктор принимает экземпляр реализации интерфейса `IConfiguration`, который был создан методом `WebHost.CreateDefaultBuilder()` в файле `Program.cs`, и присваивает его свойству `Configuration` для использования в других местах класса.

Метод `Configure()`

Метод `Configure()` применяется для настройки приложения с целью реагирования на запросы HTTP. Если среда установлена в `Development`, то включается расширенная

страница исключения для отладки, а также средство связи с браузером (Browser Link). Если среда — не Development, тогда при возникновении ошибки используется стандартный маршрут исключения:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseBrowserLink();
}
else
{
    app.UseExceptionHandler("/Home/Error");
}
```

Далее добавляется поддержка статических файлов. Мы имеем пример природы предоставления ее служб по подписке инфраструктуры ASP.NET Core. Явно добавляются лишь нужные средства.

```
app.UseStaticFiles();
```

Последние строки кода настраивают все службы, необходимые для выполнения приложения MVC, в том числе стандартный маршрут:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Добавление дополнительных маршрутов

Инфраструктура ASP.NET Core поддерживает традиционную таблицу маршрутов, а также маршрутизацию с помощью атрибутов. Модифицируем код `app.UseMvc()`, чтобы включить маршруты `Contact` и `About`, которые создавались для версии MVC 5:

```
app.UseMvc(routes =>
{
    routes.MapRoute("Contact", "Contact/{*pathInfo}", new { controller =
"Home", action = "Contact" });
    routes.MapRoute("About", "About/{*pathInfo}", new { controller = "Home",
action = "About" });
    routes.MapRoute(name: "default", template: "{controller=Home}/
{action=Index}/{id?}");
});
```

Метод *ConfigureServices()*

Метод `ConfigureServices()` применяется для конфигурирования любых служб, в которых нуждается приложение, и вставки их в контейнер внедрения зависимостей. Стандартной службой является MVC.

```
services.AddMvc();
```

Конфигурирование `AutoLotContext` и `InventoryRepo` для внедрения зависимостей

Инфраструктура EF Core конфигурируется также с использованием `IServiceCollection`. Конфигурирование класса `DbContext` из EF предусматривает установку требуемого объекта `DbContextOptions` и его добавление в контейнер DI. Конфигурация класса контекста `AutoLot` подробно рассматривалась в главе 32. Добавление объекта `DbContext` в контейнер DI дает возможность указывать в конструкторе класса только

параметр типа `AutoLotContext`, а контейнер DI позаботится о создании сконфигурированного экземпляра. В версиях ASP.NET Core 2.0 и EF Core 2.0 появилась возможность добавления пула экземпляров, что значительно улучшает показатели производительности. Поместим следующий код после вызова `services.AddMvc()`:

```
services.AddDbContextPool<AutoLotContext>(
    options => options.UseSqlServer(Configuration.GetConnectionString("AutoLot"),
        o => o.EnableRetryOnFailure())
    .ConfigureWarnings(warnings => warnings.Throw(RelationalEventId.
        QueryClientEvaluationWarning)));
```

Метод `The AddDbContextPool` создает пул классов контекста внутри инфраструктуры DI. Поскольку создание экземпляра `DbContext` может оказаться затратной операцией, организация пула из нескольких экземпляров способно существенно повысить производительность приложения. Несмотря на то что в предшествующих версиях ASP.NET Core и EF Core можно было предпринимать похожие действия, преимущество применения пула объектов контекста заключается в сбросе `DbChangeTracker` для каждого экземпляра после его возвращения обратно в пул. Такой подход предотвращает влияние одного пользователя на другого.

Вставим новую строку кода после конфигурации `AutoLotContext`, чтобы добавить `InventoryRepo` в контейнер DI:

```
services.AddScoped<IInventoryRepo, InventoryRepo>();
```

Теперь для любых классов, которые имеют в своих конструкторах параметр типа `AutoLotContext` или `IInventoryRepo`, будет создаваться подходящий экземпляр `InventoryRepo`.

Управление пакетами с помощью Bower

Инструмент Bower (<https://bower.io/>) представляет собой мощный диспетчер библиотек клиентской стороны для веб-сайтов, содержащих тысячи пакетов. Его библиотека гораздо обширнее, чем NuGet, и в целом управляется веб-сообществом. Инструмент Bower — не замена NuGet, а должен рассматриваться как дополнение. Среда Visual Studio 2017 и инфраструктура ASP.NET Core используют Bower для управления пакетами клиентской стороны.

Файл `.bowerrc` — это конфигурационный файл для Bower. В окне Solution Explorer он находится ниже файла `bower.json`. Данный файл применяется для установки относительного пути к корневому местоположению, где должны быть установлены пакеты; по умолчанию принимается папка `lib` внутри `wwwroot`:

```
{
  "directory": "wwwroot/lib"
}
```

Пакеты Bower, включенные в приложение ASP.NET Core, управляются через файл `bower.json`, который допускается редактировать напрямую для добавления/удаления пакетов либо использовать графический пользовательский интерфейс Visual Studio 2017. Диспетчер пакетов Bower (Bower Package Manager) работает подобно диспетчеру пакетов NuGet и доступен посредством щелчка правой кнопкой мыши на имени проекта в окне Solution Explorer и выбора в контекстном меню пункта `Manage Bower Packages` (Управлять пакетами Bower).

В файле `bower.json` под узлом `dependencies` перечислены все пакеты. Шаблон Visual Studio 2017 устанавливает четыре пакета:


```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.7",
    "jquery": "2.2.0",
    "jquery-validation": "1.14.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

Выполнение Bower

Среда Visual Studio 2017 отслеживает файл `bower.json` на предмет изменений, и когда изменение обнаруживается, пакеты добавляются или удаляются (на основе перечисленных пакетов) из отдельных подпапок внутри `wwwroot\lib`. Можно обеспечить принудительное обновление пакетов средой Visual Studio, щелкнув правой кнопкой мыши на файле `bower.json` и выбрав в контекстном меню пункт `Restore Packages` (Восстановить пакеты).

Объединение в пакеты и минификация

Объединение в пакеты и минификация — важные инструменты для увеличения производительности приложения. Разнесение специального кода JavaScript и CSS по небольшим функционально ориентированным файлам способствует облегчению сопровождения, но может привести к проблемам с производительностью. У браузеров есть встроенный предел на количество файлов, которые разрешено загружать из одного и того же URL, так что слишком большое число файлов (даже совсем маленьких) может замедлить загрузку страниц. Объединение в пакеты — это процесс помещения мелких файлов в более крупные файлы. В результате ограничения на количество загружаемых файлов обходятся, но само объединение должно проводиться с умом, т.к. гигантские файлы также могут стать причиной проблем.

Минификация представляет собой процесс сокращения размера файла за счет назначения переменным и функциям более коротких имен, устранения необязательных пробельных символов и удаления переносов строк. Целью минификации является уменьшение размера файла до возможного минимума без ущерба для функциональности.

Проект *BundlerMinifier*

В ранних версиях объединение в пакеты и минификация для проектов .NET Core постоянно изменялись. Команда разработчиков ASP.NET Core остановилась на проекте *BundlerMinifier*, созданном Мэдсом Кристенсеном. Он доступен в виде расширения Visual Studio и в форме пакета NuGet для .NET Core CLI. Дополнительные сведения можно найти по адресу <https://github.com/madskristensen/BundlerMinifier>.

На заметку! В .NET Core и Visual Studio 2017 по-прежнему поддерживаются Gulp, Grunt, WebPack и многие другие инструменты построения. В Microsoft отказались от выбора специфической инфраструктуры с открытым кодом, которая применялась бы по умолчанию, из-за долговечности поддержки шаблонов (три года для версий LTS (long-term support — долгосрочная поддержка)) и крутых колебаний настроения в сообществе JavaScript.

Конфигурирование объединения в пакеты и минификации

В файле `bundleconfig.json` конфигурируются входные и выходные файлы. Вот стандартная конфигурация:

```
// Конфигурирует объединение в пакеты и минификацию для проекта.
// Дополнительная информация доступна по адресу
// https://go.microsoft.com/fwlink/?LinkId=808241.
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    // Массив относительных путей для входных файлов.
    // Поддерживается универсализация имен.
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    // Необязательное указание параметров минификации.
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    // Необязательное генерирование файла .map.
    "sourceMap": false
  }
]
```

Каждый блок разметки JSON содержит (как минимум) определения для входных файлов (с поддержкой универсализации имен), а также имя и местоположение выходного упакованного и/или минифицированного файла. Например, в первом блоке показанного выше листинга для файла `site.css` создается минифицированная версия по имени `site.min.css`. Если в массиве `inputFiles` имеется более одного файла (либо используются групповые символы), тогда в выходной файл упаковываются все перечисленные файлы. Во втором блоке то же самое делается для файла `site.js` с дополнительными параметрами `minify` и `sourceMap`. Оба параметра необязательны. Они были установлены в свои стандартные значения в целях иллюстрации. Параметр `sourceMap` предназначен для создания исходных файлов отображения из файлов JavaScript.

Интеграция с Visual Studio

Выясним версию расширения `BundlerMinifier`, установленного в Visual Studio, путем выбора пункта меню `Tools` ⇒ `Extensions and Updates` (Сервис ⇒ Расширения и обновления) и затем варианта `Installed` (Установленные) в левой части открывшегося диалогового окна. Если расширение `BundlerMinifier` не установлено, тогда выберем пункт меню `Online` ⇒ `Visual Studio Gallery` (Онлайн ⇒ Галерея Visual Studio). Расширение должно располагаться на первой странице `Most Popular` (Самые популярные), но если нет, то его можно найти за счет ввода слова *Bundler* в поле поиска.

При наличии более новой версии по сравнению с версией, установленной на машине, она отобразится в разделе `Updates` (Обновления). Установим или обновим расширение до последней версии, которой на момент подготовки книги была 2.7.385. Затем может потребоваться перезапуск Visual Studio.

Объединение в пакеты при изменении

Чтобы настроить объединение в пакеты и минификацию на отслеживание изменений, щелкнем правой кнопкой мыши на имени проекта в окне Solution Explorer и выберем в контекстном меню пункт Bundler & Minifier (Объединение в пакеты и минификация). Удостоверимся в отметке пункта Produce Output Files (Выпустить выходные файлы), как показано на рис. 33.4.

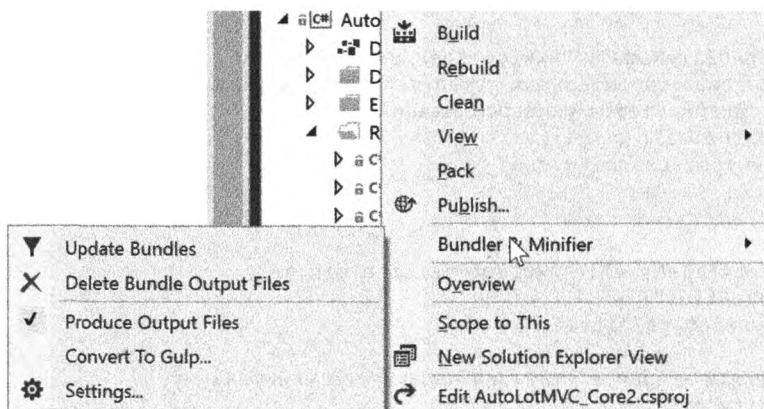


Рис. 33.4. Конфигурирование расширения BundlerMinifier для Visual Studio

Отметка пункта Produce Output Files обеспечивает добавление средства слежения за файлами, на которые производится ссылка в `bundleconfig.json`. Для его проверки перейдем в папку `css` внутри `wwwroot` и удалим файл `site.min.css`. Откроем файл `site.css`, внесем в него незначительное изменение (вроде добавления пробела или ввода и последующего удаления какого-то символа) и сохраним файл. Файл `site.min.css` будет создан заново.

Объединение в пакеты при построении

Существуют два способа включить объединение в пакеты при построении проекта средой Visual Studio 2017. Первый способ — щелкнуть правой кнопкой мыши на файле `bundleconfig.json`, выбрать в контекстном меню пункт Bundler & Minifier и отметить пункт Enable bundle on build (Включить объединение в пакеты при построении), что видно на рис. 33.5.

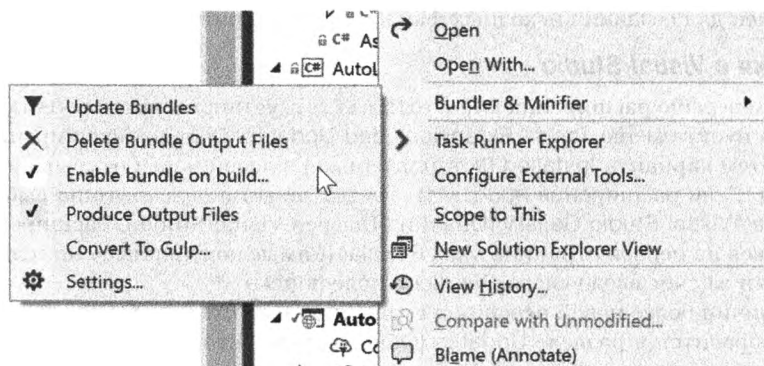


Рис. 33.5. Включение объединения в пакеты при построении

В результате загрузится еще один пакет NuGet, который вовлечет BundlerMinifier в процесс MSBuild. Установку понадобится подтвердить (рис. 33.6).

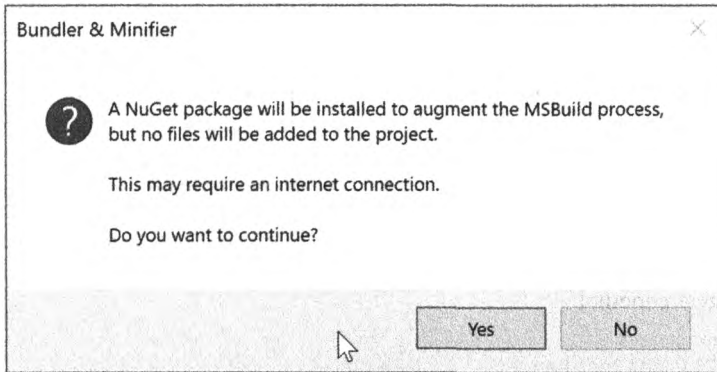


Рис. 33.6. Добавление поддержки интеграции с MSBuild

Использование проводника запускаемых задач

Второй способ предусматривает применение проводника запускаемых задач (Task Runner Explorer) для связывания задач объединения в пакеты с процессом построения. Проводник Task Runner Explorer — это еще одно расширение Visual Studio, созданное Мэдсом Кристенсеном и теперь являющееся частью стандартной установки Visual Studio 2017. Он делает возможным привязку обращений командной строки к процессу MSBuild. Откроем его, выбрав пункт меню Tools⇒Task Runner Explorer (Сервис⇒Проводник запускаемых задач). В левой части окна проводника Task Runner Explorer отображаются доступные команды, а в правой — назначенные привязки.

Чтобы назначить привязку для обновления всех файлов после каждого построения, щелкнем правой кнопкой мыши на команде Update all files (Обновить все файлы) и выберем в контекстном меню пункт Bindings⇒After Build (Привязки⇒После построения), как показано на рис. 33.7.

Для задачи Clean output files (Очистить выходные файлы) назначим привязки Before Build (Перед построением) и Clean Build (Очистка построения), что обеспечит удаление упакованных и минифицированных файлов перед началом построения или при выполнении команды очистки.

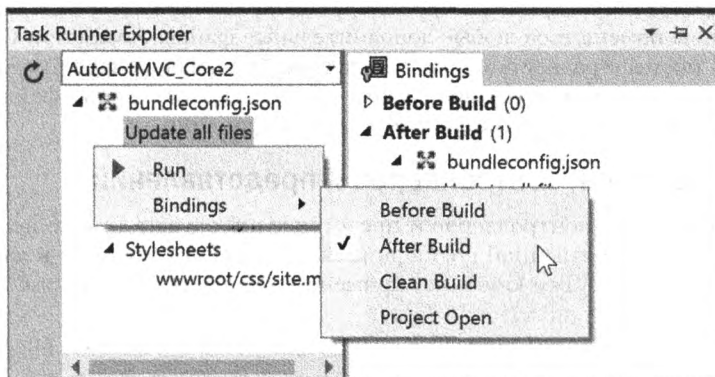


Рис. 33.7. Обновление всех файлов после каждого построения

Интеграция с .NET Core CLI

В дополнение к интеграции с Visual Studio расширение BundlerMinifier также работает с интерфейсом командной строки .NET Core CLI за счет использования NuGet-пакета BundlerMinifier.Core. Выясним доступную версию, открыв диспетчер пакетов NuGet и проведя поиск BundlerMinifier.Core, но не будем ее устанавливать. Инструментарий Visual Studio (на момент написания главы) не располагал встроенной поддержкой для добавления в проект ссылок на пакеты CLI, так что придется делать все вручную.

Откроем файл AutoLotMVC_Core2.csproj, щелкнув правой кнопкой мыши на имени проекта в окне Solution Explorer и выбрав в контекстном меню пункт Edit AutoLotMVC_Core2.csproj (Редактировать AutoLotMVC_Core2.csproj). Добавим пакет как элемент DotNetCliToolReference для Microsoft.VisualStudio.Web.CodeGeneration.Tools (указав текущую версию):

```
<ItemGroup>
  <DotNetCliToolReference
    Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
    Version="2.0.0" />
  <DotNetCliToolReference Include="BundlerMinifier.Core" Version="2.5.357" />
</ItemGroup>
```

Такое действие настроит выполнение привязки и/или очистки с применением .NET CLI либо из консоли диспетчера пакетов, либо из обычного окна командной строки. Откроем консоль диспетчера пакетов, перейдем в папку, содержащую файл проекта, и введем следующую команду:

```
dotnet bundle
```

Чтобы удалить все сгенерированные файлы, необходимо ввести команду clean:

```
dotnet bundle clean
```

Содержимое клиентской стороны (папка wwwroot)

Папка wwwroot предназначена для хранения файлов CSS, изображений и другого содержимого сайта, которое будет визуализироваться напрямую. Стандартный шаблон создает для содержимого клиентской стороны четыре папки — css, images, js и lib. Специальный файл CSS (по имени site.css) и специальный файл JavaScript (под названием site.js) содержат начальный стиль CSS и код JavaScript для сайта. Папка lib используется инструментом Bower для хранения инфраструктур клиентской стороны, указанных в файле bower.json.

Сюда должны помещаться любые дополнительные файлы JavaScript, CSS или изображений, т.к. процесс развертывания ожидает найти здесь содержимое такого рода. Кроме того, не следует забывать о добавлении любых дополнительных файлов JavaScript и/или CSS в процесс объединения в пакеты и минификации.

Папки для моделей, контроллеров и представлений

Папки для моделей, контроллеров и представлений служат тем же целям, что и их аналоги в MVC 5. Единственной специфичной для проекта моделью в данном приложении выступает ErrorViewModel.cs, применяемая контроллером HomeController и представлением Error.cshtml.

Контроллеры и представления обсуждаются в соответствующих разделах далее в главе.

Контроллеры и действия

Точно как в MVC 5 и ASP.NET Web API, контроллеры и действия являются основополагающими компонентами приложения ASP.NET Core. Они подробно рассматривались в главе, посвященной ASP.NET MVC 5, поэтому мы лишь раскроем их особенности в версии ASP.NET Core.

Базовый класс Controller

Как уже упоминалось, версия ASP.NET Core унифицирует ASP.NET MVC 5 и ASP.NET Web API. В результате базовые классы `Controller`, `ApiController` и `AsyncController` из MVC 5 и Web API 2.2 объединяются в один новый базовый класс `Controller`.

В базовом классе `Controller` также существуют многочисленные вспомогательные методы для возвращения кодов состояния HTTP, наиболее распространенные из которых кратко описаны в табл. 33.7.

Таблица 33.7. Распространенные вспомогательные методы, предлагаемые базовым классом `Controller`

Вспомогательный метод	Результирующий код состояния HTTP	Эквивалентный код
<code>NoContent</code>	<code>NoContentResult</code>	204
<code>Ok</code>	<code>OkResult</code>	200
<code>NotFound</code>	<code>NotFoundResult</code>	404
<code>BadRequest</code>	<code>BadRequestResult</code>	400
<code>Created</code>	<code>CreatedResult</code>	201
<code>CreateAtAction</code>	<code>CreatedAtActionResult</code>	
<code>CreatedAtRoute</code>	<code>CreateAtRouteResult</code>	
<code>Accepted</code>	<code>AcceptedResult</code>	202
<code>AcceptedAtAction</code>	<code>AcceptedAtActionResult</code>	
<code>AcceptedAtRoute</code>	<code>AcceptedAtRouteResult</code>	

В качестве примера приведенный ниже код возвращает код состояния 400 (неправильный запрос), если значение параметра равно `null`, и код состояния 404 (не найдено), если запись не найдена:

```
public IActionResult Edit(int? id)
{
    if (id == null)
    {
        return BadRequest();
    }
    var inventory = _repo.GetOne(id);
    if (inventory == null)
    {
        return NotFound();
    }
    return View(inventory);
}
```

Действия

Действия ASP.NET Core возвращают тип `IActionResult` (или `Task<IActionResult>` для асинхронных операций) или класс, реализующий `IActionResult`, такой как

ActionResult. В последующих разделах будут описаны наиболее часто используемые классы, которые реализуют интерфейс IActionResult.

Результаты действий

В табл. 33.8 перечислены некоторые распространенные типы, производные от класса ActionResult.

Таблица 33.8. Типовые классы, производные от ActionResult

Результат действия	Описание
ViewResult	Возвращает представление (или частичное представление) как веб-страницу
PartialViewResult	
RedirectResult	Перенаправляет на другое действие
RedirectToRouteResult	
JsonResult	Возвращает клиенту сериализованный результат JSON
FileResult	Возвращает клиенту содержимое двоичного файла
ContentResult	Возвращает клиенту тип содержимого, определяемый пользователем
HttpStatusCodeResult	Возвращает специфический код состояния HTTP

Результаты в виде форматированных ответов

Чтобы вернуть объект (либо коллекцию объектов) в формате JSON, объект (или объекты) необходимо поместить в оболочку объекта Json (или объекта JsonResult), который создает JsonResult. Вот пример:

```
public IActionResult GetAll()
{
    IEnumerable<Customer> customers = _repo.GetAllCustomers();
    return Json(customers);
}
```

Действия, возвращающие тип (или строго типизированную коллекцию), делают то же самое, что и предыдущий метод. Предшествующее действие можно написать по-другому:

```
public IEnumerable<Customer> GetAll()
{
    return Repo.GetAll();
}
```

Результаты в виде форматированных ответов могут комбинироваться с кодами состояния HTTP за счет помещения объектного результата внутрь оболочки результата действия с кодом состояния:

```
public IActionResult GetAll()
{
    return Ok(_repo.GetAllCustomers());
}
```

Результаты в виде перенаправлений

Результат действия RedirectToActionResult перенаправляет пользователя на другое действие. В следующем примере пользователь перенаправляется на действие Index текущего контроллера:

```
return RedirectToAction(nameof(Index));
```

Существуют перегруженные версии, позволяющие указывать другой контроллер и значения маршрута.

Типы ViewResult

Подобно MVC 5, тип ViewResult — это ActionResult, инкапсулирующий представление Razor. Тип PartialViewResult — это ViewResult, который предназначен для визуализации внутри другого представления.

Добавление контроллера Inventory

С выходом Visual Studio 2017 15.3 и ASP.NET Core 2.0 вернулись многие вспомогательные средства (такие как создание шаблонов контроллеров и представлений). Щелчком правой кнопкой мыши на папке Controllers и выберем в контекстном меню пункт Add⇒Controller (Добавить⇒Контроллер), как показано на рис. 33.8.

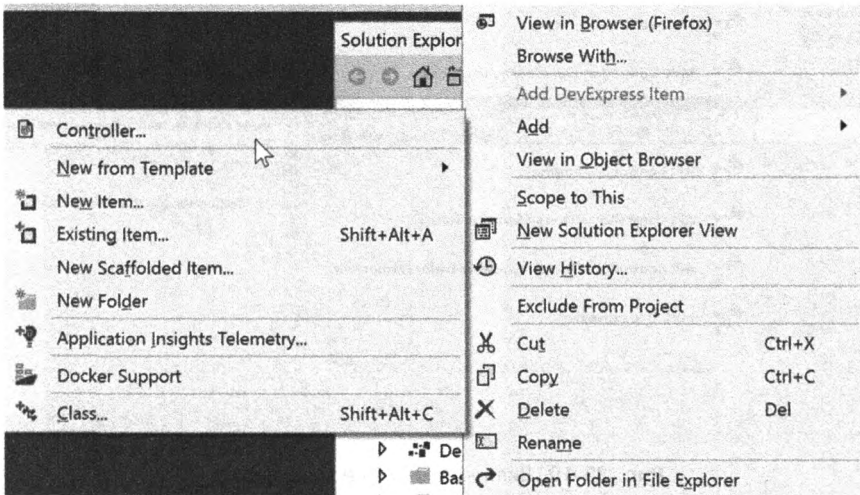


Рис. 33.8. Открытие диалогового окна создания шаблонов Add New Controller (Добавление нового контроллера)

Если это делается в проекте впервые, тогда будет предложено добавить зависимости для создания шаблонов (рис. 33.9). Выберем переключатель Minimal Dependencies (Минимальные зависимости) и щелкнем на кнопке Add (Добавить).

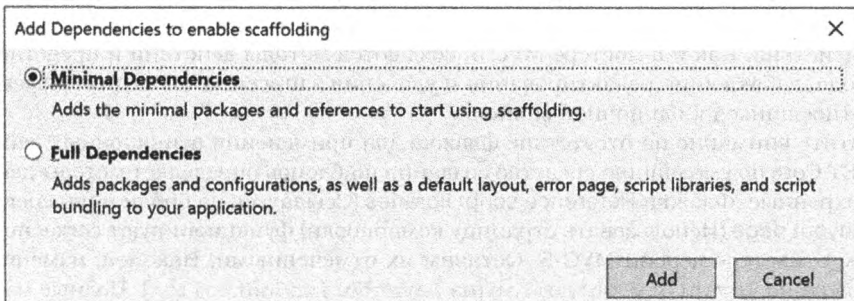


Рис. 33.9. Включение создания шаблонов

В результате появится текстовый файл (по имени `ScaffoldingReadMe.txt`), который в основном можно проигнорировать, т.к. мы уже сделали все, что в нем советуется. Кроме того, обновится файл проекта путем добавления в него следующего пакета:

```
<PackageReference
  Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0" />
```

Теперь, когда зависимости установлены, снова щелкнем правой кнопкой мыши на папке `Controllers` и выберем в контекстном меню пункт `Add Controller`. Откроется диалоговое окно `Add Scaffold` (Добавить шаблон), изображенное на рис. 33.10. В нем доступно несколько вариантов, из которых нужно выбрать `MVC Controller with views, using Entity Framework` (Контроллер MVC с представлениями, использующий Entity Framework).

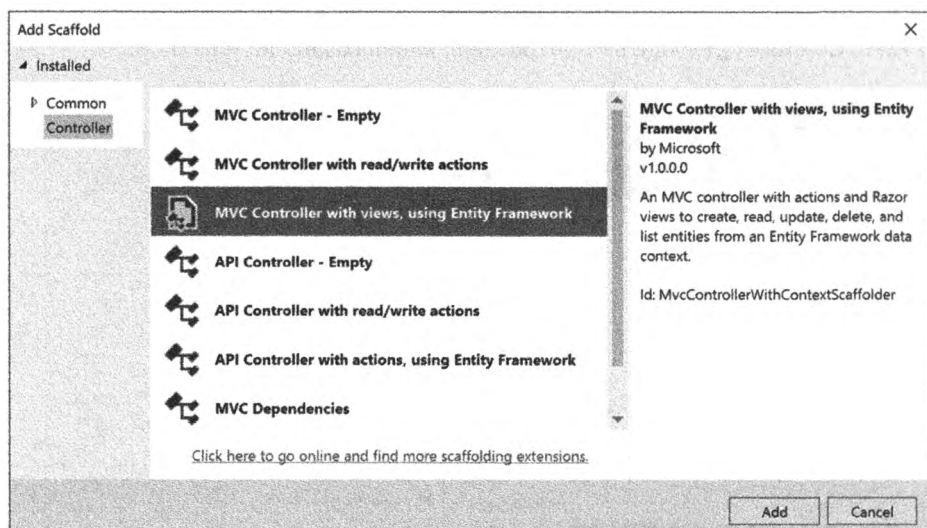


Рис. 33.10. Диалоговое окно `Add Scaffold`

Откроется дополнительное диалоговое окно (рис. 33.11), которое позволяет сконфигурировать контроллер и представления. Первая задача связана с указанием класса модели, который определяет тип для контроллеров и методов действий. Выберем в раскрывающемся списке класс `Inventory`.

Далее будет предложено указать класс контекста. Если класс контекста не выбирался, то мастер его создаст. В качестве контекста данных выберем `AutoLotContext`.

По умолчанию отмеченный флажок `Generate views` (Генерировать представления) инструктирует мастер создать связанное представление для каждого генерируемого метода действия. Как и в мастере MVC 5, создаются методы действий и представления для вывода, добавления, редактирования и удаления записей, а также отображения деталей, относящихся к одиночной записи.

Обратите внимание на отсутствие флажка для применения асинхронных действий. В ASP.NET Core по умолчанию средство создания шаблонов определяет методы действий как асинхронные. Флажки `Reference script libraries` (Ссылаться на библиотеки сценариев) и `Use a layout page` (Использовать страницу компоновки) функционируют согласно своим аналогам в мастере версии MVC 5. Оставим их отмеченными. Наконец, изменим имя контроллера на `InventoryController` (из `InventoriesController`). Полные настройки в мастере приведены на рис. 33.11. Для продолжения щелкнем на кнопке `Add`.

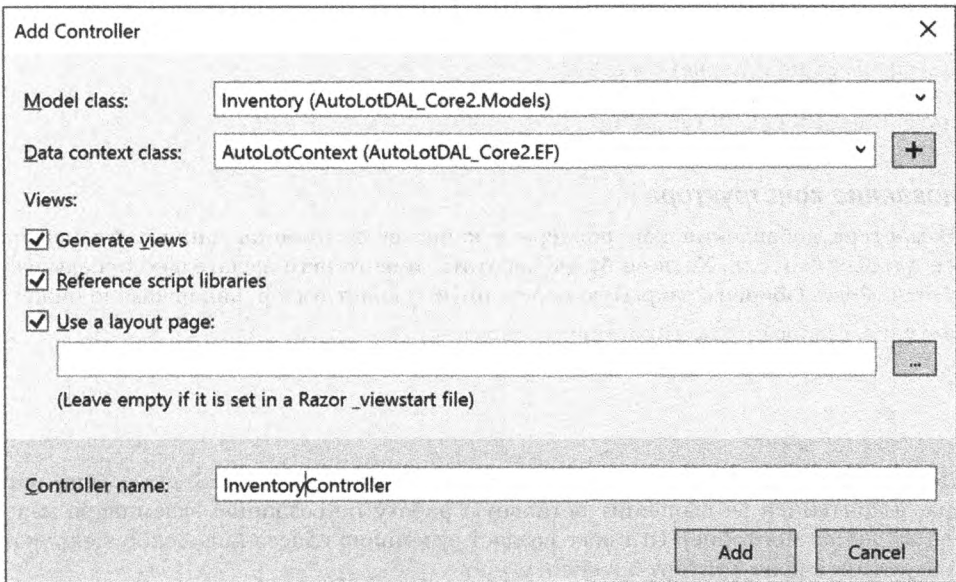


Рис. 33.11. Выбор модели, контекста и других параметров

Результаты довольно многообразны. Прежде всего, в папке `Controllers` создан класс `InventoryController`. Вдобавок в папке `Views` создана папка `Inventory`, куда помещены пять представлений.

Обновление класса `InventoryController`

Основная часть сгенерированного контроллера похожа на свою версию из MVC 5. Фактически в этом приложении можно было бы применить основную массу кода класса `InventoryController` из главы 29, внося лишь незначительные корректировки. Главное отличие — по умолчанию для методов действий используется модификатор `async`. Еще одно отличие касается применения новых вспомогательных методов контроллера для возвращения кодов состояния вместо того, чтобы создавать вручную объекты `HttpStatusCodeResult`. В последующих разделах мы модернизируем контроллер для использования уровня доступа к данным `AutoLotDAL_Core2`, точнее говоря, `InventoryRepo`. Также будут внесены изменения в синхронные версии методов действий, связанные с возвращением экземпляров реализаций `ActionResult` вместо `Task<ActionResult>`.

На заметку! Здесь не даются рекомендации о том, какие методы применять — синхронные или асинхронные. Отличия между ними подробно обсуждались ранее в книге, и вы сами должны принять решение о том, делать методы действий синхронными либо асинхронными. Независимо от выбранной версии методов действий обеспечьте тщательное тестирование приложения.

В конце класса контроллера определен метод по имени `InventoryExists()`, который можно удалить. Он использовался в шаблонном коде, но не будет применяться в обновленном коде.

Обновление операторов `using`

Первым делом модифицируем операторы `using` следующим образом:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using AutoLotDAL_Core2.Models;
using AutoLotDAL_Core2.Repos;
```

Обновление конструктора

В мастере добавления контроллера в качестве источника данных был выбран класс `AutoLotContext`. Хотя он будет работать, вместо него желательно использовать `InventoryRepo`. Обновим закрытую переменную и конструктор, как показано ниже:

```
private readonly IInventoryRepo _repo;
public InventoryController(IInventoryRepo repo)
{
    _repo = repo;
}
```

Достаточно лишь обеспечить передачу интерфейса `IInventoryRepo` как параметра, а контейнер DI выполнит остальную работу по созданию экземпляра класса `InventoryRepo`. Контейнер DI также создаст экземпляр класса `AutoLotContext`, который передается конструктору `InventoryRepo`:

```
public InventoryRepo(AutoLotContext context) : base(context)
{
}
```

Обновление методов действий `Index()` и `Details()`

Обновление методов действий `Index()` и `Details()` сводится к простой замене `AutoLotContext` экземпляром `InventoryRepo`:

```
// GET: Inventory
public IActionResult Index()
{
    return View(_repo.GetAll());
}

// GET: Inventory/Details/5
public IActionResult Details(int? id)
{
    if (id == null)
    {
        return BadRequest();
    }
    var inventory = _repo.GetOne(id);
    if (inventory == null)
    {
        return NotFound();
    }
    return View(inventory);
}
```

Паттерн “Отправка-перенаправление-получение”

Как и в MVC 5, для предотвращения проблемы с дублированием отправок формы в инфраструктуре ASP.NET Core применяется паттерн “Отправка-перенаправление-получение” (Post-Redirect-Get). Он используется в методах `Create()`, `Edit()` и `Delete()`.

Паттерн “Отправка-перенаправление-получение” был описан в главе 29. Вспомните, что он предусматривает перенаправление каждого успешного HTTP-действия POST на HTTP-действие GET, поэтому если пользователь щелкнет на кнопке отправки еще раз, то лишь обновится страница, выданная действием GET, а повторная отправка запроса POST не произойдет.

Обновление методов действий *Create ()*

Версия GET метода действия `Create ()` изменений не требует:

```
// GET: Inventory/Create
public IActionResult Create()
{
    return View();
}
```

Версия POST метода действия `Create ()` должна быть модифицирована. Как мы поступали в главе 29, удалим поля `Id` и `Timestamp` из атрибута `Bind`. Значения для полей `Id` и `Timestamp` генерируются сервером, а не вводятся пользователем при создании новой записи. Сделаем метод синхронным и изменим код с целью улучшения обработки ошибок.

```
// POST: Inventory/Create
// Для защиты от атак чрезмерными отправками указывайте только свойства,
// которые желательно привязывать; дополнительные сведения доступны
// по ссылке http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create([Bind("Make,Color,PetName")] Inventory inventory)
{
    if (!ModelState.IsValid) return View(inventory);
    try
    {
        _repo.Add(inventory);
    }
    catch (Exception ex)
    {
        // Создать запись невозможно.
        ModelState.AddModelError(string.Empty,
            $"{@"Unable to create record: {ex.Message}"}");
        return View(inventory);
    }
    return RedirectToAction(nameof(Index));
}
```

Обновление методов действий *Edit ()*

Версия GET метода действия `Edit ()` должна быть сделана синхронной и работать с хранилищем:

```
// GET: Inventory/Edit/5
public IActionResult Edit(int? id)
{
    if (id == null)
    {
        return BadRequest();
    }
    var inventory = _repo.GetOne(id);
```

```

    if (inventory == null)
    {
        return NotFound();
    }
    return View(inventory);
}

```

Версию POST метода действия `Edit()` также потребуется модифицировать, чтобы сделать ее синхронной и обеспечить применение в ней хранилища:

```

// POST: Inventory/Edit/5
// Для защиты от атак чрезмерными отправками указывайте только свойства,
// которые желательно привязывать; дополнительные сведения доступны
// по ссылке http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(int id, [Bind("Make,Color,PetName,Id,Timestamp")]
    Inventory inventory)
{
    if (id != inventory.Id)
    {
        return BadRequest();
    }
    if (!ModelState.IsValid) return View(inventory);
    try
    {
        _repo.Update(inventory);
    }
    catch (DbUpdateConcurrencyException ex)
    {
        // Сохранить запись невозможно из-за ее обновления другим пользователем.
        ModelState.AddModelError(string.Empty,
            $"Unable to save the record. Another user has updated it. {ex.Message}");
        return View(inventory);
    }
    catch (Exception ex)
    {
        // Сохранить запись невозможно.
        ModelState.AddModelError(string.Empty,
            $"Unable to save the record. {ex.Message}");
        return View(inventory);
    }
    return RedirectToAction(nameof(Index));
}

```

Обновление методов действий *Delete()*

Версия GET метода действия `Delete()` обновляется для использования хранилища:

```

// GET: Inventory/Delete/5
public IActionResult Delete(int? id)
{
    if (id == null)
    {
        return BadRequest();
    }
    var inventory = _repo.GetOne(id);
}

```

```

    if (inventory == null)
    {
        return NotFound();
    }
    return View(inventory);
}

```

Версия POST метода действия `Delete()` подвергается той же модификации, которая предпринималась в главе 29: обновление сигнатуры с целью приема записи `Inventory`, а также изменение имени с `DeleteConfirmed()` на `Delete()`.

```

// POST: Inventory/Delete
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Delete([Bind("Id, Timestamp")] Inventory inventory)
{
    try
    {
        _repo.Delete(inventory);
    }
    catch (DbUpdateConcurrencyException ex)
    {
        // Удалить запись невозможно из-за ее обновления другим пользователем.
        ModelState.AddModelError(string.Empty,
            $"{ex.Message}Unable to delete record. Another user updated the record. {ex.Message}");
    }
    catch (Exception ex)
    {
        // Удалить запись невозможно.
        ModelState.AddModelError(string.Empty,
            $"{ex.Message}Unable to delete record: {ex.Message}");
    }
    return RedirectToAction(nameof(Index));
}

```

Итоговые сведения о контроллерах и действиях

Опытные разработчики приложений ASP.NET MVC заметят, насколько приведенный ранее код контроллера и действий похож на код, который применялся годами.

Представления

Наряду с тем, что контроллеры и действия в ASP.NET Core должны выглядеть знакомыми разработчикам приложений MVC 5, представления используют в своих интересах множество новых функциональных возможностей. Вспомогательные методы HTML в стиле MVC 5 по-прежнему поддерживаются, но интенсивно применяются новые вспомогательные функции дескрипторов (рассмотренные ранее в главе), делая представления в ASP.NET Core гораздо более дружелюбными к HTML. Несколько следующих разделов посвящены обновлению представлений и шаблонов для приложения `AutoLotMVC_Core2`.

Обновление файла импортирования представлений

Откроем файл `_ViewImports.cshtml` и добавим в него показанные ниже операторы `using`:

```

@using AutoLotMVC_Core2
@using AutoLotMVC_Core2.Models

```

На заметку! Из-за очевидной проблемы в инфраструктуре, существующей на момент написания главы, представления на самом деле не подхватывали операторы `using`, заданные в файле `_ViewImports.cshtml`. Помимо файла `_ViewImports.cshtml` необходимые операторы `using` также указываются в самих представлениях.

Представление компоновки

Представление `_Layout.cshtml` (в папке `Views\Shared`) аналогично своей версии MVC 5, включая использование Bootstrap для стилизации и отзывчивого дизайна. Главные отличия связаны с применением вспомогательных функций дескрипторов. Например, вспомогательная функция дескриптора `environment` используется для определения, какие файлы CSS и JavaScript нужно загружать:

```
<environment include="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment exclude="Development">
  <link rel="stylesheet"
    href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/
    bootstrap.min.css"
    asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
    asp-fallback-test-class="sr-only"
    asp-fallback-test-property="position"
    asp-fallbacktest-value="absolute" />
  <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
<!-- Для краткости код не показан. -->
<environment include="Development">
  <script src="~/lib/jquery/dist/jquery.js"></script>
  <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
  <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment exclude="Development">
  <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
    asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
    asp-fallback-test="window.jQuery"
    crossorigin="anonymous"
    integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsFRV2a+AfHI19k8z8l9ggpc8X+Yts
t4yBo/hH+8Fk">
  </script>
  <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
    asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
    crossorigin="anonymous"
    integrity="sha384-Tc5IQ1b027qvyjSMfHjOMALkfUWVxZxUPnCJA712mCWNIPG9mGCD8
wGNicPD7Txa">
  </script>
  <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>
```

В дополнение к вспомогательной функции дескриптора `environment` в приведенном выше коде также применяются вспомогательные функции дескрипторов `script` и `link`. Меню строится с использованием вспомогательной функции дескриптора `link`. Добавим новый пункт меню для страницы `Inventory`:

```

<div class="navbar-collapse collapse">
  <ul class="nav navbar-nav">
    <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
    <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
    <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
    <li><a asp-area="" asp-controller="Inventory"
      asp-action="Index">Inventory</a></li>
  </ul>
</div>

```

Частичное представление со сценариями проверки достоверности

Хотя представление `ValidationScriptsPartialView.cshtml` в изменениях не нуждается, оно является великолепным примером применения вспомогательных функций дескрипторов `environment` и `script`:

```

<environment include="Development">
  <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
  <script
src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js">
  </script>
</environment>
<environment exclude="Development">
  <script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/
jquery.validate.min.js"
    asp-fallback-src="~/lib/jquery-validation/dist/jquery.validate.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.validator"
    crossorigin="anonymous"
    integrity="sha384-Fnqn3nxp3506LP/7Y3j/25B1WeA3PXTyTl178LjECcPaKCV12TsZ
P7yyMxOe/G/k">
  </script>
  <script src="https://ajax.aspnetcdn.com/ajax/jquery.validation.
unobtrusive/3.2.6/jquery.validate.unobtrusive.min.js"
    asp-fallback-src="~/lib/jquery-validation-unobtrusive/jquery.validate.
unobtrusive.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.validator && window.
jQuery.validator.unobtrusive"
    crossorigin="anonymous"
    integrity="sha384-JrXK+k53HACyavUKOsL+NkmSesD2P+73eDMrbTtTk0h4RmOF8hF8
apPlkp26JlyH">
  </script>
</environment>

```

Шаблон для отображения складских данных

Как и MVC 5, инфраструктура ASP.NET Core поддерживает шаблоны для отображения и редактирования. Создадим внутри папки `Views\Inventory` новую папку под названием `DisplayTemplates` и поместим в нее новое частичное представление по имени `InventoryList.cshtml`. Заменим его содержимое следующей разметкой:

```

@using AutoLotDAL_Core2.Models
@model Inventory
<tr>
  <td>
    @Html.DisplayFor(model => model.Make)
  </td>

```



```

<td>
    @Html.DisplayFor(model => model.Color)
</td>
<td>
    @Html.DisplayFor(model => model.PetName)
</td>
<td>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Details" asp-route-id="@Model.Id">Details</a> |
    <a asp-action="Delete" asp-route-id="@Model.Id">Delete</a>
</td>
</tr>

```

В разметке легко заметить, что вспомогательные методы HTML все еще представляют собой важную часть разработки приложений ASP.NET Core, особенно при отображении данных. Применимы все правила, которые использовались в MVC 5. В новой разметке задействована вспомогательная функция дескриптора `a`, демонстрирующая атрибуты `asp-action` и `asp-route-имя_переменной`.

Шаблон для редактирования складских данных

Создадим внутри папки `Views\Inventory` новую папку под названием `EditorTemplates` и добавим в нее новое частичное представление по имени `Inventory.cshtml`. Заменяем его содержимое приведенной ниже разметкой:

```

@using AutoLotDAL_Core2.Models
@model Inventory
<h4>Inventory</h4>
<hr />
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Make" class="control-label"></label>
    <input asp-for="Make" class="form-control"/>
    <span asp-validation-for="Make" class="text-danger"></span>
</div>
<div class="form-group">
    <label asp-for="Color" class="control-label"></label>
    <input asp-for="Color" class="form-control"/>
    <span asp-validation-for="Color" class="text-danger"></span>
</div>
<div class="form-group">
    <label asp-for="PetName" class="control-label"></label>
    <input asp-for="PetName" class="form-control"/>
    <span asp-validation-for="PetName" class="text-danger"></span>
</div>

```

В шаблоне применяются вспомогательные функции дескрипторов для проверки достоверности, а также вспомогательные функции дескрипторов `label` и `input`. Чтобы можно было сравнить с MVC 5, вот версия Razor того же шаблона:

```

@model AutoLotDAL.Models.Inventory
<div class="form-horizontal">
    <h4>Inventory</h4>
    <hr />
    @Html.ValidationSummary(false, "", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(model => model.Make,
            htmlAttributes: new { @class = "control-label col-md-2" })
    </div>

```

```

<div class="col-md-10">
    @Html.EditorFor(model => model.Make,
        new { htmlAttributes = new { @class = "form-control" } })
    @Html.ValidationMessageFor(model => model.Make, "",
        new { @class = "text-danger" })
</div>
</div>
<div class="form-group">
    @Html.LabelFor(model => model.Color,
        htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.Color,
            new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Color,
            "", new { @class = "text-danger" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(model => model.PetName,
        htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.PetName,
            new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.PetName,
            "", new { @class = "text-danger" })
    </div>
</div>
</div>

```

Такое несходство между двумя версиями разметки очень ясно демонстрирует преимущество использования вспомогательных функций дескрипторов перед вспомогательными методами HTML. Имейте в виду, что вторая разметка также будет работать в ASP.NET Core.

Представление Index

Модифицируем файл `Index.cshtml` с целью применения функции Razor и шаблона для отображения `InventoryList.cshtml`:

```

@using AutoLotDAL_Core2.Models
@model IEnumerable<Inventory>
@functions
{
    public IList<Inventory> SortCars(IList<Inventory> cars)
    {
        var list = from s in cars orderby s.PetName select s;
        return list.ToList();
    }
}
@{
    ViewData["Title"] = "Index";
}
<h2>Index</h2>
<p>
    <a asp-action="Create">Create New</a>
</p>

```

```

<table class="table">
  <thead>
    <tr>
      <th>@Html.DisplayNameFor(model => model.Make)</th>
      <th>@Html.DisplayNameFor(model => model.Color)</th>
      <th>@Html.DisplayNameFor(model => model.PetName)</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    @foreach (var item in SortCars(Model.ToList()))
    {
      @Html.DisplayFor(modelItem=>item, "InventoryList")
    }
  </tbody>
</table>

```

Представление Details

Единственное изменение, которое мы внесем в представление Details.cshtml, связано с удалением отображения данных Timestamp. Обновленное содержимое файла Details.cshtml **выглядит следующим образом**:

```

@model AutoLotDAL_Core2.Models.Inventory
@{
    ViewData["Title"] = "Details";
}
<h2>Details</h2>
<div>
  <h4>Inventory</h4>
  <hr />
  <dl class="dl-horizontal">
    <dt>@Html.DisplayNameFor(model => model.Make)</dt>
    <dd>@Html.DisplayFor(model => model.Make)</dd>
    <dt>@Html.DisplayNameFor(model => model.Color)</dt>
    <dd>@Html.DisplayFor(model => model.Color)</dd>
    <dt>@Html.DisplayNameFor(model => model.PetName) </dt>
    <dd>@Html.DisplayFor(model => model.PetName)</dd>
  </dl>
</div>
<div>
  <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
  <a asp-action="Index">Back to List</a>
</div>

```

Представление Create

Представление Create.cshtml объединяет вспомогательные методы HTML и шаблоны Razor для редактирования со вспомогательными функциями дескрипторов, чтобы свести к минимуму объем разметки HTML. Модифицируем представление, как показано далее:

```

@model AutoLotDAL_Core2.Models.Inventory
@{
    ViewData["Title"] = "Create";
}

```

```

<h2>Create</h2>
<div class="row">
  <div class="col-md-4">
    <form asp-action="Create">
      @Html.EditorForModel()
      <div class="form-group">
        <input type="submit" value="Create" class="btn btn-default"/>
      </div>
    </form>
  </div>
</div>
<div>
  <a asp-action="Index">Back to List</a>
</div>
@section Scripts {
  @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Обратите внимание на использование вспомогательной функции дескриптора `form` и отсутствие вызова `HTML.AntiforgeryToken()`. Вспомните, что маркер противодействия подделке предоставляется вспомогательной функцией дескриптора `form` автоматически.

Представление Edit

Представление `Edit.cshtml` похоже на `Create.cshtml`, но в него добавляются два скрытых поля с применением вспомогательной функции дескриптора `input`. Обновим разметку следующим образом:

```

@model AutoLotDAL_Core2.Models.Inventory
@{
  ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<div class="row">
  <div class="col-md-4">
    <form asp-action="Edit">
      <input type="hidden" asp-for="Id"/>
      <input type="hidden" asp-for="Timestamp"/>
      @Html.EditorForModel()
      <div class="form-group">
        <input type="submit" value="Save" class="btn btn-default"/>
      </div>
    </form>
  </div>
</div>
<div>
  <a asp-action="Index">Back to List</a>
</div>
@section Scripts {
  @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Представление Delete

В представлении Delete.cshtml мы сначала удалим отображение столбца Timestamp и затем добавим внутрь дескриптора form скрытое поле input для столбца Timestamp. Окончательное представление выглядит так:

```
@model AutoLotDAL_Core2.Models.Inventory
@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>
<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Inventory</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>@Html.DisplayNameFor(model => model.Make)</dt>
        <dd>@Html.DisplayFor(model => model.Make)</dd>
        <dt>@Html.DisplayNameFor(model => model.Color)</dt>
        <dd>@Html.DisplayFor(model => model.Color)</dd>
        <dt>@Html.DisplayNameFor(model => model.PetName)</dt>
        <dd>@Html.DisplayFor(model => model.PetName)</dd>
    </dl>
    <form asp-action="Delete">
        <input type="hidden" asp-for="Id" />
        <input type="hidden" asp-for="Timestamp" />
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-action="Index">Back to List</a>
    </form>
</div>
```

Итоговые сведения о представлениях

В версии ASP.NET Core появились вспомогательные функции дескрипторов как альтернатива использованию вспомогательных методов HTML для многих дескрипторов, связанных с формами. Наряду с тем, что вспомогательные методы HTML из MVC 5 по-прежнему поддерживаются, в предшествующих разделах демонстрировались преимущества применения вспомогательных функций дескрипторов там, где это возможно. В результате получается более чистая и читабельная разметка, которая в итоге повышает удобство сопровождения кода.

Компоненты представлений

Как упоминалось ранее, компоненты представлений являются нововведением ASP.NET Core. Они могут использоваться везде, где нужно комбинировать код серверной стороны с разметкой пользовательского интерфейса, которая не должна быть конечной точкой для действия пользователя. Типичным примером служит система меню, строящаяся динамически на основе записей из базы данных. Нет никаких причин, по которым пользователь напрямую обращался бы к такому коду. В рассматриваемом примере мы собираемся заменить представление Index.cshtml компонентом представления.

Написание кода серверной стороны

Компонент представления образован из класса серверной стороны, производного от `ViewComponent`, и подчиняется соглашению об именовании вида `<специальное_имя>ViewComponent`. Код может находиться где угодно в проекте, хотя мы обычно помещаем его в папку под названием `ViewComponents`. Создадим папку по имени `ViewComponents` в корневой папке проекта `AutoLotMVC_Core2` и добавим в нее новый файл класса `InventoryViewComponent.cs`. Добавим в начало файла следующие операторы `using`:

```
using System.Threading.Tasks;
using AutoLotDAL_Core2.Repos;
using Microsoft.AspNetCore.Mvc;
```

Сделаем класс открытым и унаследованным от `ViewComponent`. Компоненты представлений не обязательно наследовать от базового класса `ViewComponent`, но подобно базовому классу `Controller` поступать так полезно. Модифицируем класс, как показано ниже:

```
public class InventoryViewComponent : ViewComponent
{
}
```

Чтобы скопировать представление `Index` из `InventoryController`, данный компонент представления нуждается в обращении к методу `GetAll()` класса `InventoryRepo`. К счастью, компоненты представлений также задействуют контейнер `DI`, встроенный в `ASP.NET Core`. Создадим переменную уровня класса для хранения экземпляра реализации `IInventoryRepo` и конструктор, который принимает экземпляр реализации этого интерфейса:

```
public class InventoryViewComponent : ViewComponent
{
    private readonly IInventoryRepo _repo;
    public InventoryViewComponent(IInventoryRepo repo)
    {
        _repo = repo;
    }
}
```

Компоненты представлений визуализируются из представлений путем вызова открытого метода `InvokeAsync()`. Метод `InvokeAsync()` возвращает экземпляр `Task<ViewComponentResult>`, который концептуально похож на экземпляр реализации `IActionResult`. Добавим в класс `InventoryViewComponent` следующий код:

```
public async Task<ViewComponentResult> InvokeAsync()
{
    var cars = _repo.GetAll(x => x.Make, true);
    if (cars != null)
    {
        return View("InventoryPartialView", cars);
    }
    return new ContentViewComponentResult("Unable to locate records.");
}
```

Метод `InvokeAsync()` получает список всех записей об автомобилях, отсортированный по полю `Make` (в порядке возрастания) и в случае успеха возвращает `ViewComponentResult` с применением вспомогательного метода `View()` из базового класса `ViewComponent`, передавая ему список записей `Inventory` в качестве модели

представления. Если извлечь список не удалось, тогда компонент представления возвращает `ContentViewComponentResult` с сообщением об ошибке.

Вспомогательный метод `View()` из базового класса `ViewComponent` работает подобно методу базового класса `Controller` с тем же самым именем за исключением пары основных отличий. Первое отличие в том, что имя файла стандартного представления выглядит как `Default.cshtml`. Подобно методу `View()` класса `Controller` для частичного представления может использоваться другое имя, передаваемое в первом параметре. Второе отличие касается местоположения визуализируемого частичного представления, которым должно быть одна из следующих двух папок:

```
Views/<parent_controller_name_calling_view >/
Components/<view_component_name>/<view_name>
Views/Shared/Components/<view_component_name>/<view_name>
```

Ниже приведен полный код класса:

```
public class InventoryViewComponent : ViewComponent
{
    private readonly IInventoryRepo _repo;
    public InventoryViewComponent(IInventoryRepo repo)
    {
        _repo = repo;
    }
    public async Task<IViewComponentResult> InvokeAsync()
    {
        var cars = _repo.GetAll(x => x.Make, true);
        return View("InventoryPartialView", cars);
    }
}
```

Написание кода клиентской стороны

Представления, визуализируемые из компонентов представлений, являются все-го лишь частичными представлениями. Создадим внутри папки `Views\Shared` новую папку под названием `Components`, а в ней еще одну папку по имени `Inventory`. Имя последней папки должно совпадать с именем созданного ранее класса компонента представления за вычетом суффикса `ViewComponent`. Создадим в папке `Inventory` частичное представление под названием `InventoryPartialView.cshtml`.

Удалим существующий код и поместим в файл `InventoryPartialView.cshtml` следующую разметку:

```
@using AutoLotDAL_Core2.Models
@model IEnumerable<Inventory>
<table class="table">
    <thead>
        <tr>
            <th>@Html.DisplayNameFor(model => model.Make)</th>
            <th>@Html.DisplayNameFor(model => model.Color)</th>
            <th>@Html.DisplayNameFor(model => model.PetName)</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.ToList())
        {
            <tr>
                <td>@Html.DisplayFor(modelItem => item.Make)</td>
```

```
 @Html.DisplayFor(modelItem => item.Color)</td>  @Html.DisplayFor(modelItem => item.PetName)</td>   | | |
```

Разметка должна выглядеть уже знакомой. Представления компонентов представлений — это просто представления в обозначенном местоположении, построенные тем же способом, что и частичные представления.

Обращение к компонентам представлений

К компонентам представлений обычно обращаются из представления (хотя их можно визуализировать также из метода действия контроллера). Синтаксис прямолинеен: `Component.Invoke(<строковое_имя_компонента_представления>)`. Как и в случае контроллеров, при обращении к компоненту представления суффикс `ViewComponent` должен быть отброшен:

```
@await Component.InvokeAsync("Inventory")
```

Обращение к компонентам представлений как к специальным вспомогательным функциям дескрипторов

В версии ASP.NET Core 1.1 появилась возможность обращаться к компонентам представлений как к вспомогательным функциям дескрипторов. Вместо применения `Component.InvokeAsync()` к компоненту представления можно просто обратиться примерно так:

```
<vc:Inventory></vc:Inventory>
```

Чтобы использовать данный способ обращения к компонентам представления, они должны быть добавлены как вспомогательные функции дескрипторов с применением команды `@addTagHelper`. Она уже добавлялась в файл `_ViewImports.cshtml` ранее в главе для включения специального компонента представления `EmailTagHelper`:

```
@addTagHelper *, AutoLotMVC_Core2
```

Добавление компонента представления в AutoLotMVC_Core2

Мы заменим текущее представление `Index` компонентом представления `InventoryViewComponent`. Скопируем файл `Index.cshtml` в папку `Views\Inventory` и назовем новому файлу имя `IndexWithViewComponent.cshtml`. Очистим код в новом файле и заменим его следующим кодом:

```

@{
    ViewData["Title"] = "Index";
}
<h2>Index</h2>

```



```
<p>
  <a asp-action="Create">Create New</a>
</p>
<vc:Inventory></vc:Inventory>
```

Модифицируем метод действия `Index()` в классе `InventoryController`, как показано далее:

```
public IActionResult Index()
{
    return View("IndexWithViewComponent");
}
```

Если теперь запустить приложение и щелкнуть на пункте `Inventory` в главном меню, то метод `Index()` будет обращаться к простому представлению, которое визуализирует компонент представления `InventoryViewComponent` и `InventoryPartialView.cshtml`, отображая все записи из таблицы `Inventory`.

Исходный код. Решение `AutolotMVC_Core2` доступно в подкаталоге `Chapter_33`.

Резюме

В главе были исследованы многочисленные аспекты построения веб-приложений с помощью ASP.NET Core. После демонстрации в действии шаблона ASP.NET Core Web Application вы узнали о новых функциональных средствах ASP.NET Core, включая новую систему конфигурации на основе среды, встроенную поддержку внедрения зависимостей, веб-сервер Kestrel, вспомогательные функции дескрипторов и многое другое. Наконец, вы получили более глубокое представление об инфраструктуре ASP.NET Core, попутно создав веб-приложение `AutoLotMVC_Core2`.

В следующей главе вы научитесь строить службы REST, используя ASP.NET Core.

ГЛАВА 34

Введение в приложения служб ASP.NET Core

В предыдущей главе была подробно раскрыта инфраструктура ASP.NET Core применительно к построению веб-приложения `AutoLotMVC_Core2`. В настоящей главе рассмотрение ASP.NET Core завершается созданием службы REST под названием `AutoLotAPI_Core2`.

Мы начнем с использования шаблона ASP.NET Core Web API (да, он по-прежнему именован как Web API, хотя теперь существует только одна инфраструктура) для создания решения и проекта. Затем мы выясним организацию проекта и параметры запуска, обновим пакеты NuGet и добавим библиотеку доступа к данным `AutoLotDAL_Core2` из главы 32. Далее мы исследуем изменения в версии ASP.NET Web API 2.2, которые не были охвачены в главе 33, и завершим службу REST.

Наконец, мы модернизируем веб-приложение `AutoLotMVC_Core2`, чтобы задействовать в нем службу `AutoLotAPI_Core2` как серверный источник данных вместо взаимодействия с `AutoLotDAL_Core2`.

На заметку! В главе предполагается, что вы уже имеете практические знания ASP.NET Web API 2.2 (см. главу 30) и проработали материал главы 33, посвященный ASP.NET Core. Если это не так, тогда перед продолжением просмотрите главы 28 и 33.

Шаблон ASP.NET Core Web API

Как обсуждалось в главе 33, для веб-приложений (в стиле MVC, применяющих Razor Pages либо использующих разнообразные инфраструктуры JavaScript) и приложений служб в Visual Studio предусмотрены отдельные шаблоны. Поскольку имеется единственная инфраструктура ASP.NET Core, отличия между шаблонами не структурные, а семантические. Все шаблоны в ASP.NET Core направлены на изменение первоначально создаваемых файлов.

Мастер создания проекта

Чтобы создать решение для данной главы, запустим Visual Studio и выберем пункт меню `File⇒New⇒Project` (Файл⇒Создать⇒Проект). В боковой панели слева выберем элемент `Web (Веб)` внутри `Visual C#`, затем в центральной панели выберем шаблон `ASP.NET Core Web Application (Веб-приложение ASP.NET Core)`, после чего в поле `Name (Имя)` изменим имя на `AutoLotAPI_Core2` (рис. 34.1).

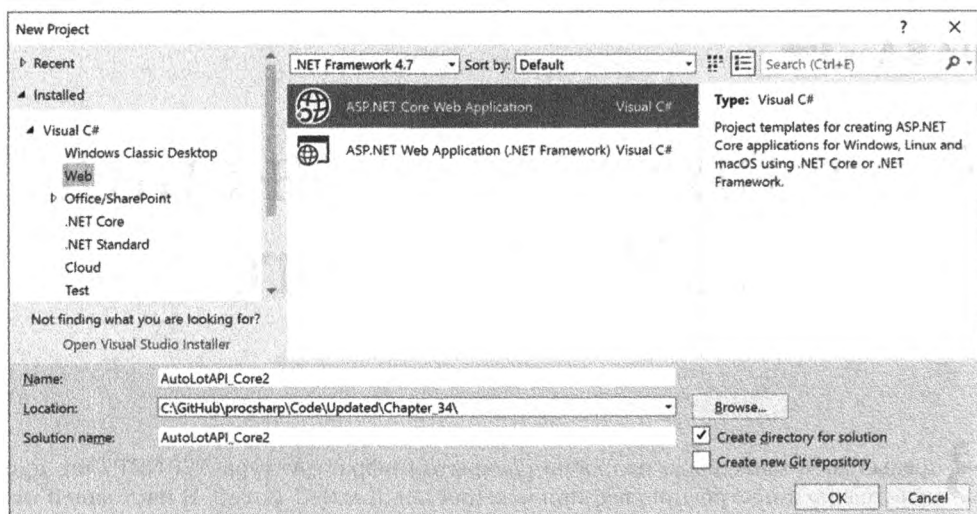


Рис. 34.1. Создание нового проекта службы REST в ASP.NET Core

На следующем экране мастера выберем шаблон Web API. Оставим флажок Enable Docker Support (Включить поддержку Docker) неотмеченным и установим аутентификацию в No Authentication (Аутентификация отсутствует), как показано на рис. 34.2.

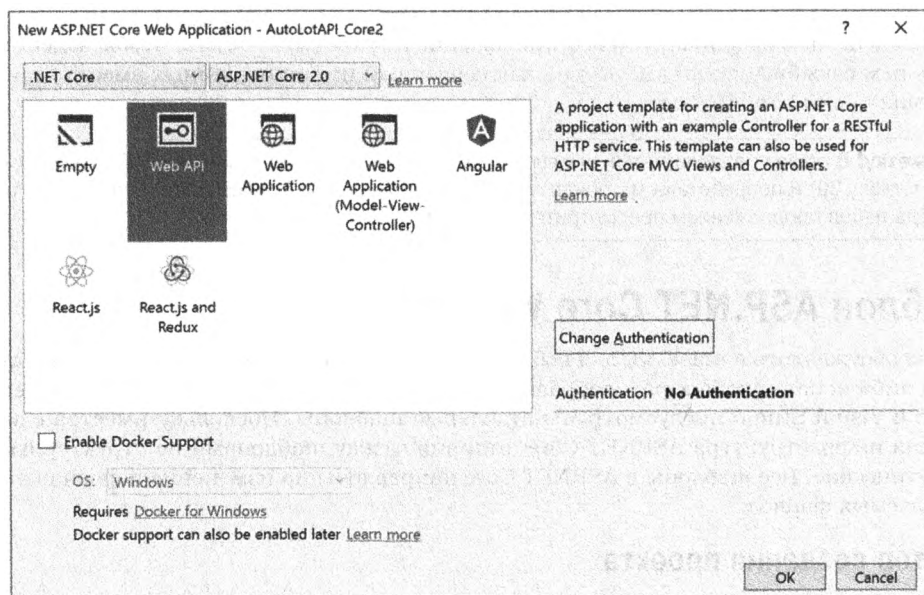


Рис. 34.2. Выбор шаблона ASP.NET Core Web API

На заметку! Docker — это технология контейнеризации, рассмотрение которой выходит за рамки данной книги. Полная информация доступна по адресу <https://docs.microsoft.com/ru-ru/azure/vs-azure-tools-docker-hosting-web-apps-in-docker>.

Организация проекта приложения службы ASP.NET Core

К настоящему времени новая организация проектов ASP.NET Core должна быть знакомой. Главное отличие между проектами веб-приложений и служб ASP.NET Core в том, что шаблон службы создает папку `wwwroot`, но оставляет ее пустой (что и можно было ожидать от приложения службы).

Добавление библиотеки доступа к данным

Последовательность действий аналогична процессу, рассмотренному в главе 33. Скопируем проекты `AutoLotDAL_Core2` и `AutoLotDAL_Core2.Models` из главы 32 в папку, где находится проект `AutoLotAPI_Core2`, добавим ссылки из проекта `AutoLotAPI_Core2` на проекты `AutoLotDAL_Core2` и `AutoLotDAL_Core2.Models` и удостоверимся в том, что проект `AutoLotDAL_Core2` все еще ссылается на проект `AutoLotDAL_Core2.Models`.

Обновление и добавление пакетов NuGet

Щелчком правой кнопкой мыши на проекте `AutoLotAPI_Core2` и выберем в контекстном меню пункт `Manage NuGet Packages` (Управление пакетами NuGet). Перейдем на вкладку `Updates` (Обновления) и обновим любые пакеты, которые нуждаются в обновлении.

Создаваемое приложение будет применять NuGet-пакет `AutoMapper` для трансформации классов моделей и удаления связанных экземпляров, как делалось в проекте `Web API 2.2` из главы 30. Чтобы установить его, выберем в поле со списком `Filter` (Фильтр) вариант `Browse` (Обзор), введем в поле поиска строку `AutoMapper` и установим пакет.

Запуск и развертывание приложений служб

Подобно веб-приложениям ASP.NET Core приложения служб ASP.NET Core могут запускаться из Visual Studio (с использованием IIS или Kestrel) и в командной строке (с применением Kestrel). Все варианты управляются настройками внутри файла `launchsettings.json`.

Приложения служб ASP.NET Core развертываются теми же способами, что и веб-приложения ASP.NET Core.

Изменения в приложениях служб ASP.NET Core

По сравнению с ASP.NET Web API 2.2 в приложения служб ASP.NET Core внесены два значительных изменения. Первое изменение касается формата данных JSON, возвращаемых из методов действий, а второе связано с требованием явной маршрутизации методов HTTP.

Формат возвращаемых данных JSON

В предшествующих версиях ASP.NET Web API методы действий возвращали данные JSON, сформатированные согласно *стилю Pascal*, при котором первая буква и буквы последующих слов записываются в верхнем регистре. Он является стандартным стилем для открытых свойств .NET. Например, в приведенном далее классе имена свойств `FirstName` и `JobTitle` определены в стиле *Pascal*:

```
public class Customer
{
    public string FirstName { get; set; }
    public string JobTitle { get; set; }
}
```

При возвращении экземпляра класса `Customer` как данных JSON в Web API 2.2 он форматировался следующим образом:

```
{"FirstName":"Bob","JobTitle":"Boss"}
```

Инфраструктура ASP.NET Core присоединилась практически ко всей остальной части Интернета и теперь возвращает данные, сформатированные в *“верблюжьем”* стиле (первая буква записывается в нижнем регистре, а буквы последующих слов — в верхнем регистре). Показанные выше данные возвращаются из службы ASP.NET Core в таком виде:

```
{"firstName":"Bob","jobTitle":"Boss"}
```

У вас может возникнуть вопрос: разве это крупная проблема? В мире, где работа производится полностью в рамках стека технологий Microsoft, проблема вовсе не крупная. Тем не менее, если служба REST поставляет данные в инфраструктуры, в которых регистр символов важен, тогда указанное изменение может нарушить (и действительно нарушит) работу приложений. Хорошая новость в том, что службу можно сконфигурировать на функционирование подобно Web API 2.2.

Откроем файл `Startup.cs`, относящийся к приложению `AutoLotAPI_Core2`, и добавим в его начало следующий оператор `using`:

```
using Newtonsoft.Json.Serialization;
```

Отыщем код метода `ConfigureServices()` и модифицируем вызов `services.AddMvc()`, как показано ниже:

```
services.AddMvc()
    .AddJsonOptions(options =>
    {
        // Вернуться к стилю Pascal при форматировании данных JSON.
        options.SerializerSettings.ContractResolver = new DefaultContractResolver();
    });
```

В результате данные JSON, возвращаемые службой, будут форматироваться в стиле `Pascal`.

Явная маршрутизация для методов HTTP

Версия ASP.NET Web API 2.2 позволяет не указывать метод HTTP, если имя метода действия начинается с `GET`, `PUT`, `DELETE` или `POST`. Как упоминалось в главе 30, такое соглашение в целом сочли неудачной идеей для проектов Web API, а в инфраструктуре ASP.NET Core от него вообще отказались.

Когда для метода действия метод HTTP не указан, он будет вызываться с использованием HTTP-метода `GET`.

Управление привязкой моделей в ASP.NET Core

Привязка моделей в ASP.NET Core, как и привязка моделей в MVC 5 и Web API 2.2, просматривает данные в отправленной форме (при их наличии), затем данные маршрутизации и, наконец, параметры строки запроса. Управлять привязкой моделей можно за счет применения к параметрам методов действий атрибутов, перечисленных в табл. 34.1.

Для приложений служб ASP.NET Core обычно будут использоваться атрибуты `FromRoute` и `FromBody`.

Таблица 34.1. Атрибуты привязки моделей ASP.NET Core 2.0

Атрибут	Описание
BindingRequired	Если привязка не может произойти, тогда будет добавлена ошибка состояния модели
BindNever	Сообщает средству привязки моделей о том, что параметр привязываться не должен
FromHeader	Используются для указания точного источника привязки, подлежащего применению (заголовок HTTP, строка запроса, параметры маршрута или значения формы)
FromQuery	
FromRoute	
FromForm	
FromServices	Это значение будет добавляться с использованием внедрения зависимостей
FromBody	Привязывает данные из тела запроса. Форматер выбирается на основе содержимого запроса (скажем, JSON, XML и т.д.)
ModelBinder	Применяется для переопределения стандартного средства привязки моделей (для специальной привязки моделей)

Построение AutoLotAPI_Core2

Пришло время заняться построением AutoLotAPI_Core2. Материал данного раздела не должен вызывать удивление. Все, что вы узнали в главе 33 о конфигурации проектов ASP.NET Core, внедрении зависимостей и действиях, по-прежнему применимо. Многое будет похожим на настройку проекта AutoLotMVC_Core2.

Добавление строки подключения

Откроем файл appsettings.Development.json и добавим в него следующую строку подключения (она должна совпадать со строкой подключения, используемой в главах 32 и 33):

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "ConnectionStrings": {
    "AutoLot":
      "server=(LocalDb)\\MSSQLLocalDB;database=AutoLotCore2;integrated
      security=True;MultipleActiveResultSets=True;App=EntityFramework;"
  }
}
```

Обновление файла Program.cs для инициализации данных

Как и в AutoLotMVC_Core2, файл Program.cs представляет собой место, куда помещается код инициализации данных. Добавим в начало файла Program.cs такие операторы using:

```
using AutoLotDAL_Core2.DataInitialization;
using AutoLotDAL_Core2.EF;
using Microsoft.Extensions.DependencyInjection;
```

Модифицируем код в методе `Main()`, как показано далее:

```
var webHost = BuildWebHost(args);
using (var scope = webHost.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<AutoLotContext>();
    MyDataInitializer.RecreateDatabase(context);
    MyDataInitializer.InitializeData(context);
}
webHost.Run();
```

На заметку! Попытка запуска проекта прямо сейчас закончится неудачей, поскольку не был сконфигурирован контейнер DI. Мы сделаем это следующим.

Обновление файла `Startup.cs`

Файл `Startup.cs` для приложений служб ASP.NET Core требует меньшей настройки по сравнению с аналогичным файлом для веб-приложений. Некоторые отличия вполне очевидны; например, все, что приходилось делать с пользовательским интерфейсом, здесь не делается. Легко заметить, что метод `Configure()` не создает таблицу маршрутов. Причина в том, что приложения служб опираются на маршрутизацию с помощью атрибутов, а не на таблицы маршрутов.

Откроем файл `Startup.cs` и приведем операторы `using` к следующему виду:

```
using AutoLotDAL_Core2.EF;
using AutoLotDAL_Core2.Repos;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
```

Обновление конструктора

Конструктор также может принимать экземпляр реализации `IHostingEnvironment`, который понадобится позже в главе. Добавим переменную `private readonly`, предназначенную для хранения этого экземпляра, и модифицируем конструктор:

```
private readonly IHostingEnvironment _env;
public Startup(IConfiguration configuration, IHostingEnvironment env)
{
    _env = env;
    Configuration = configuration;
}
```

Обновление метода `ConfigureServices()`

Мы уже обновили вызов метода `services.AddMvc()` для применения распознавателя JSON из Web API 2.2:

```
services.AddMvc()
    .AddJsonOptions(options =>
```

```
{
    // Вернуться к стилю Pascal при форматировании данных JSON.
    options.SerializerSettings.ContractResolver =
        new DefaultContractResolver();
});
```

Следующая задача — сконфигурировать `AutoLotContext` и добавить `IInventoryRepo` в контейнер DI.

Конфигурирование `AutoLotContext` и `InventoryRepo` для поддержки внедрения зависимостей

Службы ASP.NET Core также поддерживают организацию пула классов `DbContext`. Добавим после вызова `services.AddMvc()` представленный далее код:

```
services.AddDbContextPool<AutoLotContext>(
    options => options.UseSqlServer(Configuration.GetConnectionString("AutoLot"),
        o => o.EnableRetryOnFailure()
            .ConfigureWarnings(warnings => warnings.Throw(RelationalEventId.
                QueryClientEvaluationWarning)));
```

Поместим после конфигурации `AutoLotContext` новую строку для добавления `InventoryRepo` в контейнер DI:

```
services.AddScoped<IInventoryRepo, InventoryRepo>();
```

Добавление обработки исключений на уровне приложений

Подобно MVC 5 и Web API 2.2 инфраструктура ASP.NET Core поддерживает фильтры и при создании приложения службы полезно добавить глобальный фильтр исключений, чтобы инкапсулировать обработку ошибок для службы.

Добавление `AutoLotExceptionHandler`

Щелкнем правой кнопкой мыши на проекте `AutoLotAPI_Core2` и добавим новую папку под названием `Filters`, а в ней создадим новый класс по имени `AutoLotExceptionHandler`. Модифицируем операторы `using`, как показано ниже:

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.EntityFrameworkCore;
```

Сделаем класс открытым и реализующим интерфейс `IExceptionHandler`. Интерфейс `IExceptionHandler` имеет один метод `OnException()`, который запускается, когда появилось необработанное исключение. Обновим класс следующим образом:

```
public class AutoLotExceptionHandler : IExceptionHandler
{
    public void OnException(ExceptionContext context)
    {
    }
}
```

Цель созданного фильтра — по существу повторить страницу исключений разработчика в веб-приложениях ASP.NET Core. В случае среды разработки фильтр должен возвращать трассировку стека и полные сведения об исключении в виде данных JSON, а иначе дружественное сообщение без каких-либо технических подробностей.

Для выяснения среды времени выполнения планируется задействовать экземпляр реализации интерфейса `IHostingEnvironment`, который будет передан через внедрение

посредством конструктора. Такой прием используется для присваивания значения булевой переменной уровня класса по имени `_isDevelopment`. Добавим переменную и конструктор:

```
private readonly bool _isDevelopment;
public AutoLotExceptionHandler(IHostingEnvironment env)
{
    _isDevelopment = env.IsDevelopment();
}
```

В методе `OnException()` из `ExceptionContext` извлекается фактическое исключение. Оно дает возможность получить трассировку стека и точно определить происшедшую ошибку. В рассматриваемом примере фильтра мы собираемся проверять на предмет исключения `DbUpdateConcurrency` как специальный случай и обрабатывать все остальные ошибки без изменений.

Далее в методе создается экземпляр реализации `ActionResult` либо как `BadRequestObjectResult` (для ошибок, связанных с параллелизмом), либо как обобщенный `ObjectResult` (для всех других ошибок). Модифицируем метод:

```
public void OnException(ExceptionContext context)
{
    var ex = context.Exception;
    string stackTrace =
        (_isDevelopment) ? context.Exception.StackTrace : string.Empty;
    IActionResult actionResult;
    string message = ex.Message;
    if (ex is DbUpdateConcurrencyException)
    {
        // Возвратить код 400.
        if (!_isDevelopment)
        {
            message = "There was an error updating the database. Another user has altered the record.";
            // При обновлении базы данных возникла ошибка.
            // Запись была изменена другим пользователем.
        }
        actionResult = new BadRequestObjectResult(
            new { Error = "Concurrency Issue.", Message = ex.Message,
                StackTrace = stackTrace });
        // Проблема параллелизма.
    }
    else
    {
        if (!_isDevelopment)
        {
            message = "There was an unknown error. Please try again.";
            // Возникла неизвестная ошибка. Повторите действие.
        }
        actionResult = new ObjectResult(
            new { Error = "General Error.", Message = ex.Message, StackTrace = stackTrace });
        // Ошибка общего характера.
        {
            StatusCode = 500
        };
    }
    context.Result = actionResult;
}
```

Регистрация фильтра исключений

Далее фильтр необходимо зарегистрировать с помощью инфраструктуры MVC. Откроем файл `Startup.cs` и добавим следующий оператор `using`:

```
using AutoLotAPI_Core2.Filters;
```

Найдем код метода `ConfigureServices()` и модифицируем вызов `services.AddMvc()`:

```
services.AddMvc(config =>
{
    config.Filters.Add(new AutoLotExceptionFilter(_env));
})
.AddJsonOptions(options =>
{
    // Вернуться к стилю Pascal при форматировании данных JSON.
    options.SerializerSettings.ContractResolver =
        new DefaultContractResolver();
});
```

Теперь любые необработанные ошибки в конвейере приложения будут перехватываться `AutoLotExceptionFilter`.

Добавление контроллера Inventory

Щелкнем правой кнопкой мыши на папке `Controllers` и выберем в контекстном меню пункт `Add⇒Controller` (Добавить⇒Контроллер), как показано на рис. 34.3.

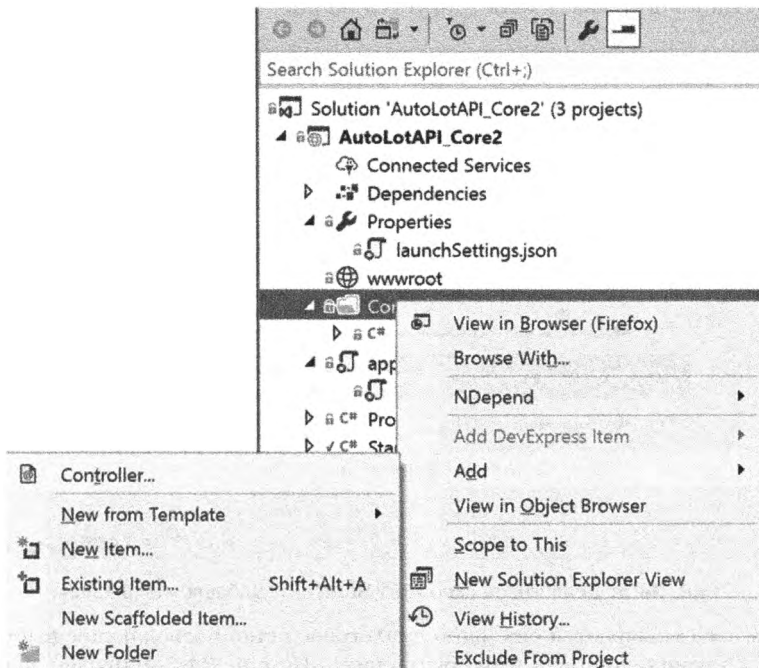


Рис. 34.3. Открытие диалогового окна `Add New Controller` (Добавление нового контроллера)

Если это делается впервые для проекта, тогда будет предложено добавить шаблонные зависимости (рис. 34.4). Выберем переключатель Minimal Dependencies (Минимальные зависимости) и щелкнем на кнопке Add (Добавить).

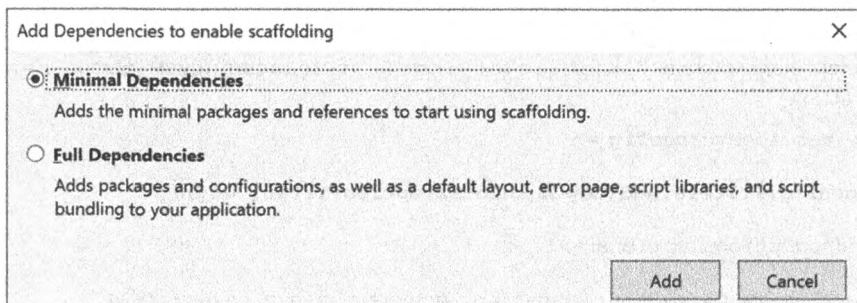


Рис. 34.4. Добавление шаблонных зависимостей

В точности как для проектов веб-приложений ASP.NET Core данный мастер создает файл ScaffoldingReadMe.txt и добавляет следующий пакет (если он отсутствует в файле проекта):

```
<PackageReference
  Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0" />
```

Чтобы добавить контроллер, снова щелкнем правой кнопкой мыши на папке Controllers и выберем в контекстном меню пункт Add⇒Controller. Выберем шаблон API Controller with actions, using Entity Framework (Контроллер API с действиями, использующий Entity Framework), как демонстрируется на рис. 34.5.

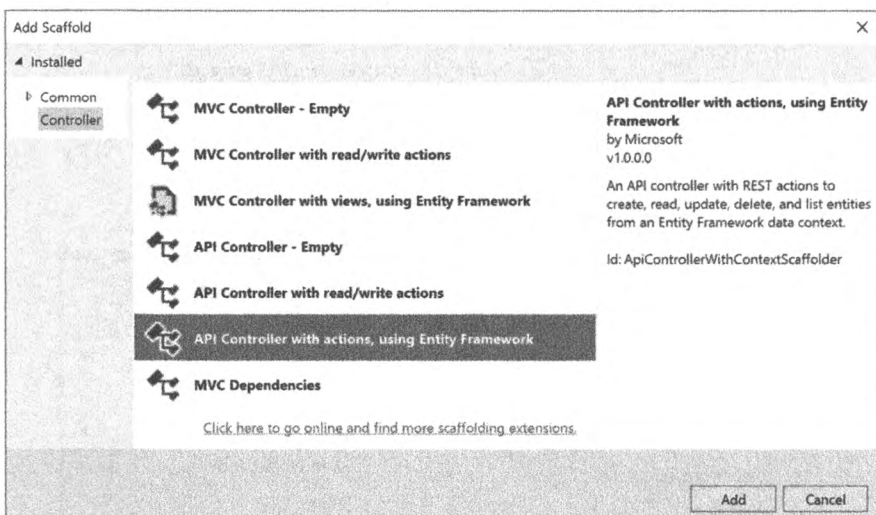


Рис. 34.5. Диалоговое окно Add Scaffold (Добавление шаблона)

В результате открывается еще одно диалоговое окно, позволяющее выбрать модель и контекст, а также назначить имя контроллеру. На рис. 34.6 приведен завершающий экран мастера. Для продолжения щелкнем на кнопке Add.

Как и в случае мастера в проекте веб-приложения ASP.NET Core, в папке Controllers создается класс InventoryController.

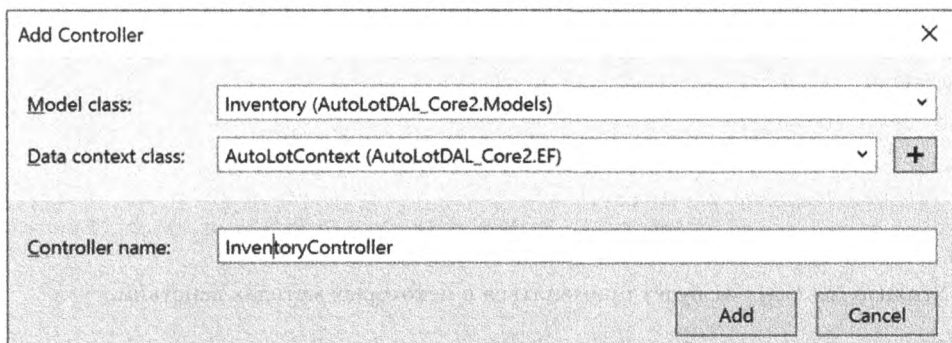


Рис. 34.6. Выбор модели и контекста и назначение имени контроллеру

Удаление метода *InventoryExists()*

Данный метод в разрабатываемой версии кода не применяется, поэтому его можно удалить из контроллера.

Обновление *InventoryController*

Мастер создал контроллер, код которого напрямую работает с *AutoLotContext*. Основная масса изменений, вносимых в контроллер, связана с заменой *AutoLotContext* на *InventoryRepo*.

Рассмотрим атрибут в начале контроллера:

```
[Produces("application/json")]
```

Это фильтр, который указывает, что все содержимое, возвращаемое из контроллера, будет форматироваться как данные JSON. По умолчанию в ASP.NET Core для возвращаемых из контроллера данных принят формат JSON, но в клиенте с помощью заголовка *Accept* формат можно было бы изменять. Атрибут *Produces* предотвращает запрашивание клиентами другого формата.

Следующий атрибут определяет маршрут для контроллера:

```
[Route("api/Inventory")]
```

Обновление операторов *using*

Добавим в начало файла показанные ниже операторы *using*:

```
using AutoLotDAL_Core2.Models;
using AutoLotDAL_Core2.Repos;
using AutoMapper;
using Newtonsoft.Json;
```

Обновление конструктора

В мастере добавления контроллера для источника данных был выбран класс *AutoLotContext*. Хотя он будет работать, вместо него желательно использовать *InventoryRepo*. Модифицируем закрытую переменную и конструктор следующим образом:

```
private readonly IInventoryRepo _repo;
public InventoryController(IInventoryRepo repo)
{
    _repo = repo;
}
```

Затем понадобится сконфигурировать AutoMapper для очистки любых навигационных свойств, как делалось в проекте Web API 2.2 из главы 30. Добавим в конструктор такой код:

```
Mapper.Initialize(
    cfg =>
    {
        cfg.CreateMap<Inventory,
            Inventory>().ForMember(x => x.Orders, opt => opt.Ignore());
    });
```

Утилита AutoMapper будет применяться в некоторых методах действий.

Обновление метода действия GetCars ()

С обновлением метода действия GetCars () связано два аспекта. Во-первых, нужно заменить обращение к AutoLotContext вызовом метода GetAll () класса InventoryRepo. Во-вторых, необходимо использовать AutoMapper для удаления всех экземпляров навигационных свойств. Вспомните, что инфраструктура EF Core не поддерживает ленивую загрузку, поэтому если явно не указано включать связанные данные (что не делалось), то когда сериализатор JSON попытается вернуть запись Inventory, возникнут ошибки. Дополнительные сведения можно найти в главе 30. Модифицируем метод, как показано ниже:

```
// GET: api/Inventory
[HttpGet]
public IEnumerable<Inventory> GetCars()
{
    var inventories = _repo.GetAll();
    return Mapper.Map<List<Inventory>, List<Inventory>>(inventories);
}
```

Обновление метода действия GetInventory ()

В метод действия GetInventory () потребуется внести сразу несколько изменений. Первым делом маршруту нужно назначить имя (оно будет применяться позже в контроллере). При маршрутизации с помощью атрибутов ASP.NET Core маршруты именуются в объявлениях методов HTTP. Приведем атрибут к такому виду:

```
[HttpGet("{id}", Name="DisplayRoute")]
```

Затем модифицируем метод с целью использования InventoryRepo и возвращения очищенной (посредством AutoMapper) записи Inventory в качестве содержимого для сообщения с кодом состояния HTTP 200. Обратите внимание на применение вспомогательного метода Ok() базового класса Controller. Вот полный код метода действия GetInventory():

```
// GET: api/Inventory/5
[HttpGet("{id}", Name="DisplayRoute")]
public async Task<IActionResult> GetInventory([FromRoute] int id)
{
    Inventory inventory = _repo.GetOne(id);
    if (inventory == null)
    {
        return NotFound();
    }
    return Ok(Mapper.Map<Inventory, Inventory>(inventory));
}
```

Параметр `FromRoute` используется для гарантирования того, что `id` поступает из маршрута, а не какого-то другого источника.

Обновление метода действия `PutInventory()`

В мире HTTP метод PUT представляет собой обновление существующей записи. Метод действия `PutInventory()` получает параметр `id` записи, подлежащей удалению, из маршрута и значения записи `Inventory` (включая параметр `id`) из тела сообщения запроса. Метод сначала проверяет, находится ли модель в допустимом состоянии (с участием системы проверки достоверности, предлагаемой средством привязки моделей), после чего проверяет, совпадает ли параметр `id` из маршрута с параметром `id` в записи `Inventory` из тела сообщения. Если любая из проверок не проходит, тогда с применением вспомогательного метода `BadRequest()` из базового класса `Controller` возвращается соответствующий ответ HTTP.

В этот метод вносится простое изменение — код, который обновляет запись с использованием `AutoLotContext`, заменяется вызовом метода `Update()` класса `InventoryRepo`:

```
// PUT: api/Inventory/5
[HttpPut("{id}")]
public async Task<IActionResult> PutInventory([FromRoute] int id,
    [FromBody] Inventory inventory)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    if (id != inventory.Id)
    {
        return BadRequest();
    }
    _repo.Update(inventory);
    return NoContent();
}
```

Код обработки ошибок был удален, т.к. `AutoLotExceptionHandler` будет перехватывать все исключения и обрабатывать их, возвращая подходящие данные JSON (на основе среды).

Обновление метода действия `PostInventory()`

Метод действия `PostInventory()` добавляет новую запись `Inventory` в базу данных. Если модель находится в недопустимом состоянии, то он возвращает результат вызова `BadRequest()` с ошибками в теле. Если состояние модели является допустимым, тогда метод добавляет запись `Inventory` с применением `InventoryRepo`. В случае успешного добавления возвращается URL метода действия `GetInventory()` в виде заголовка. Полный код метода действия `PostInventory()` показан ниже:

```
// POST: api/Inventory
[HttpPost]
public async Task<IActionResult> PostInventory([FromBody] Inventory inventory)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    _repo.Add(inventory);
    return CreatedAtRoute("DisplayRoute", new { id = inventory.Id }, inventory);
}
```

Обновление метода действия DeleteInventory()

Чтобы удалить запись с использованием проверки параллелизма, необходимы значения Id и Timestamp. Запросы DELETE в HTTP не имеют тела, а потому такую информацию придется добавить в маршрут. Модифицируем маршрут (внутри атрибута HttpDelete) и сигнатуру метода следующим образом:

```
[HttpDelete("{id}/{timestamp}")]
public async Task<IActionResult>
    DeleteInventory([FromRoute] int id, [FromRoute] string timestamp)
```

Свойство Timestamp представляет собой (как уже хорошо известно) массив байтов, который не может указываться как часть маршрута в URL. Метод действия DeleteInventory() должен кодировать значение Timestamp в строку, для чего будет применяться инфраструктура Json.NET от Newtonsoft. Позже в главе мы декодируем значение; понадобится лишь преобразовать строку обратно в массив байтов. Важно отметить, что для обеспечения корректной работы процесса в строку нужно добавить кавычки:

```
if (!timestamp.StartsWith("\""))
{
    timestamp = $"\"{timestamp}\"";
}
var ts = JsonConvert.DeserializeObject<byte[]>(timestamp);
```

Наконец, внутри DeleteInventory() вызывается метод Delete() класса InventoryRepo с передачей ему значений Id и Timestamp и возвращается результат вызова Ok(). Ниже представлен полный код метода действия DeleteInventory():

```
// DELETE: api/Inventory/5
[HttpDelete("{id}/{timestamp}")]
public async Task<IActionResult> DeleteInventory([FromRoute]
    int id, [FromRoute] string timestamp)
{
    if (!timestamp.StartsWith("\""))
    {
        timestamp = $"\"{timestamp}\"";
    }
    var ts = JsonConvert.DeserializeObject<byte[]>(timestamp);
    _repo.Delete(id, ts);
    return Ok();
}
```

Обработкой ошибок занимается класс AutoLotExceptionHandler.

Итоговые сведения о приложениях служб ASP.NET Core

Как было показано в настоящей главе, веб-приложения и приложения служб ASP.NET Core задействуют одну и ту же инфраструктуру. Конфигурация, запуск, контроллеры, фильтры и другие аспекты ASP.NET Core (кроме, конечно же, пользовательского интерфейса) полностью переносимы между веб-приложениями и службами REST.

Обновление AutoLotMVC_Core2 для использования AutoLotAPI_Core2

Подобно обновлению сайта ASP.NET MVC под названием CarLotMVC для работы с CarLotWebAPI теперь мы обновим AutoLotMVC_Core2 с целью применения службы AutoLotAPI_Core2 вместо AutoLotDAL_Core2 в качестве уровня доступа к данным.

Копирование и добавление приложения AutoLotMVC_Core2

Скопируем папку проекта AutoLotMVC_Core2 (из главы 33) в папку решения AutoLotAPI_Core2. Добавим проект в решение, щелкнув правой кнопкой мыши на имени решения и выбрав в контекстном меню пункт Add⇒Existing Project (Добавить⇒Существующий проект). Установим проекты AutoLotAPI_Core2 и AutoLotMVC_Core2 как запускаемые (в указанном порядке). Для этого щелкнем правой кнопкой мыши на имени решения, выберем в контекстном меню пункт Set StartUp Projects (Установить как запускаемые проекты) и приведем диалоговое окно к виду, показанному на рис. 34.7.

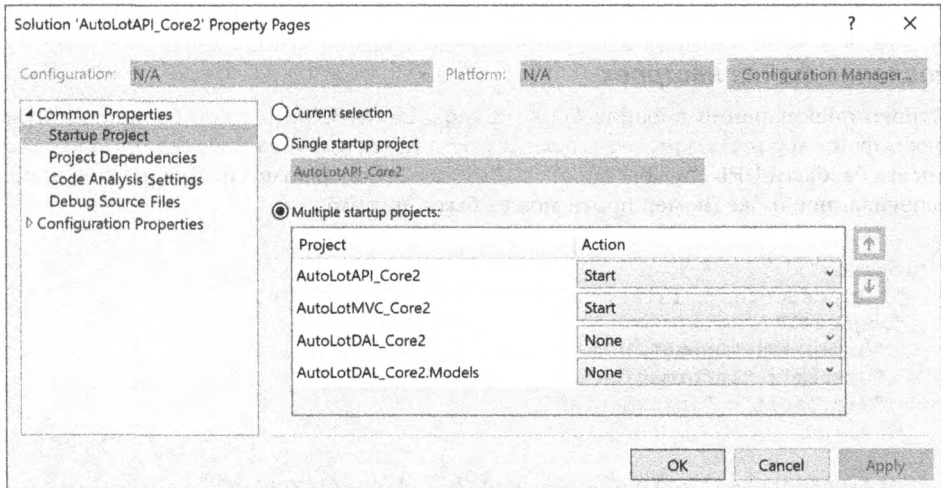


Рис. 34.7. Установка множества запускаемых проектов

Удаление AutoLotDAL_Core2 из AutoLotMVC_Core2

Ссылка на AutoLotDAL_Core2 больше не нужна, но по-прежнему необходим доступ к моделям в AutoLotDAL_Models. Удалим ссылку на AutoLotDAL_Core2, раскрыв узел Dependencies\Projects ниже проекта AutoLotMVC_Core2.

Обновление Program.cs

Откроем файл Program.cs и удалим код, который инициализирует базу данных. В итоге метод Main() будет выглядеть так:

```
public static void Main(string[] args)
{
    var webHost = BuildWebHost(args);
    webHost.Run();
}
```


Кроме того, удалим излишние операторы `using`. Должны остаться только два оператора:

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
```

Обновление *Startup.cs*

Откроем файл `Startup.cs`. Удалим из метода `ConfigureServices()` код, инициализирующий `AutoLotContext`, и удалим строку, которая добавляет `InventoryRepo` в контейнер DI. После завершения метод будет выглядеть следующим образом:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
}
```

Также понадобится удалить два оператора `using`, ссылающиеся на `AutoLotDAL_Core2`, и два оператора `using`, ссылающиеся на `EntityFrameworkCore`:

```
using AutoLotDAL_Core2.EF;
using AutoLotDAL_Core2.Repos;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;
```

Обновление файла настроек

Строка подключения в файле `appsettings.Development.json` больше не нужна. Откроем файл `appsettings.Development.json` и удалим узел `ConnectionString`, заменив его базовым URL службы `AutoLotAPI_Core2`. Содержимое файла должно напоминать показанное ниже (номер порта может быть другим):

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "ServiceAddress": "http://localhost:51714/api/Inventory"
}
```

Для установления порта необходимо открыть файл `launchSettings.json`, относящийся к службе `AutoLotAPI_Core2`, определить веб-сервер, который будет использоваться при тестировании (IIS или Kestrel), и выбрать подходящий номер порта.

Создание нового контроллера *Inventory*

Мы собираемся создать новый класс `InventoryController` полностью с нуля. Удалим существующий файл класса `InventoryController.cs` и создадим новый файл с тем же самым именем. Поместим в начало файла следующие операторы `using`:

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
```

```
using AutoLotDAL_Core2.Models;
using Microsoft.Extensions.Configuration;
using Newtonsoft.Json;
```

Создание конструктора

Сделаем класс открытым и унаследованным от класса `Controller`. Добавим закрытую переменную для хранения URL службы и конструктор, который принимает экземпляр класса, реализующего `IConfiguration`. Экземпляр реализации `IConfiguration` применяется для извлечения базового URL из файла настроек. В данный момент класс имеет такой вид:

```
public class InventoryController : Controller
{
    private readonly string _baseUrl;
    public InventoryController(IConfiguration configuration)
    {
        _baseUrl = configuration.GetSection("ServiceAddress").Value;
    }
}
```

Добавление метода *GetInventoryRecord()*

Мы создадим закрытый вспомогательный метод для получения одиночной складской записи от службы. Код будет создавать новый экземпляр `HttpClient` и вызывать метод `GetInventory()` класса `InventoryController` в `AutoLotAPI_Core2`. Если вызов проходит успешно, тогда с использованием `Json.NET` из возвращенных данных JSON будет создан экземпляр записи `Inventory`. Если вызов не удался, то метод возвратит `null`. Ниже приведен полный код метода:

```
private async Task<Inventory> GetInventoryRecord(int id)
{
    var client = new HttpClient();
    var response = await client.GetAsync($"{_baseUrl}/{id}");
    if (response.IsSuccessStatusCode)
    {
        var inventory = JsonConvert.DeserializeObject<Inventory>(
            await response.Content.ReadAsStringAsync());
        return inventory;
    }
    return null;
}
```

Добавление метода действия *Index()*

Метод действия `Index()` создает новый экземпляр `HttpClient` и вызывает стандартный HTTP-метод GET службы. Вот его код:

```
public async Task<IActionResult> Index()
{
    var client = new HttpClient();
    var response = await client.GetAsync(_baseUrl);
    if (response.IsSuccessStatusCode)
    {
        var items = JsonConvert.DeserializeObject<List<Inventory>>(
            await response.Content.ReadAsStringAsync());
        return View(items);
    }
    return NotFound();
}
```

В случае применения версии, которая задействует `InventoryViewComponent`, метод действия `Index()` потребуется обновить:

```
public async Task<IActionResult> Index()
{
    return View("IndexWithViewComponent");
}
```

Так или иначе, `InventoryViewComponent` необходимо обновить, поскольку текущая версия все еще ссылается на проект `AutoLotDAL_Core2`.

Обновление `InventoryViewComponent`

Класс `InventoryViewComponent` нуждается в обновлении с целью использования службы REST вместо `AutoLotDAL_Core2`. Модифицируем операторы `using` следующим образом:

```
using System.Collections.Generic;
using System.Net.Http;
using System.Threading.Tasks;
using AutoLotDAL_Core2.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.Extensions.Configuration;
using Newtonsoft.Json;
```

Далее обновим конструктор для принятия экземпляра реализации `IConfiguration` и базового URL, как делалось в классе `InventoryController`:

```
private readonly string _baseUrl;
public InventoryViewComponent(IConfiguration configuration)
{
    _baseUrl = configuration.GetSection("ServiceAddress").Value;
}
```

Наконец, модифицируем метод `InvokeAsync()`, чтобы в нем применялся новый экземпляр класса `HttpClient` для получения списка записей `Inventory`, передаваемых частичному представлению:

```
public async Task<IViewComponentResult> InvokeAsync()
{
    var client = new HttpClient();
    var response = await client.GetAsync(_baseUrl);
    if (response.IsSuccessStatusCode)
    {
        var items = JsonConvert.DeserializeObject<List<Inventory>>(
            await response.Content.ReadAsStringAsync());
        return View("InventoryPartialView", items);
    }
    return new ContentViewComponentResult("Unable to return records.");
}
```

Добавление метода действия `Details()`

Теперь добавим метод действия `Details()`. Он сначала проверяет, был ли передан параметр `id`, и если нет, то возвращает результат вызова `BadRequest()`. Если параметр `id` был указан, тогда используется метод `GetInventoryRecord()`, чтобы вернуть представление `Details.cshtml` с записью `Inventory` или результат вызова `NotFound()`, когда возвращаемым значением метода `GetInventoryRecord()` оказывается `null`. Добавим следующий код:

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return BadRequest();
    }
    var inventory = await GetInventoryRecord(id.Value);
    return inventory != null ? (IActionResult) View(inventory) : NotFound();
}
```

Добавление методов действий *Create()*

Версия GET метода действия *Create()* остается той же, что и ранее:

```
// GET: Inventory/Create
public IActionResult Create()
{
    return View();
}
```

Версия POST метода действия *Create()* проверяет состояние модели и затем применяет новый экземпляр *HttpClient* для отправки службе запроса POST. Если все прошло успешно, тогда осуществляется перенаправление на метод действия *Index()*, а в противном случае пользователю возвращается представление *Create.cshtml* для исправления любых ошибок.

```
// POST: Inventory/Create
// Для защиты от атак чрезмерными отправками указывайте только свойства,
// которые желательно привязывать; дополнительные сведения доступны
// по ссылке http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
    Create([Bind("Make,Color,PetName")] Inventory inventory)
{
    if (!ModelState.IsValid) return View(inventory);
    try
    {
        var client = new HttpClient();
        string json = JsonConvert.SerializeObject(inventory);
        var response = await client.PostAsync(_baseUrl,
            new StringContent(json, Encoding.UTF8, "application/json"));
        if (response.IsSuccessStatusCode)
        {
            return RedirectToAction(nameof(Index));
        }
    }
    catch (Exception ex)
    {
        // Создать запись невозможно.
        ModelState.AddModelError(string.Empty,
            $"Unable to create record: {ex.Message}");
    }
    return View(inventory);
}
```

Добавление методов действий *Edit()*

Версия GET метода действия `Edit()` проверяет, передавался ли параметр `id`, и если это так, то использует метод `GetInventoryRecord()` для получения экземпляра `Inventory` и передает его представлению `Edit.cshtml`:

```
// GET: Inventory/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return BadRequest();
    }
    var inventory = await GetInventoryRecord(id.Value);
    return inventory != null ? (IActionResult)View(inventory) : NotFound();
}
```

Версия POST метода действия `Edit()` работает почти так же, как версия POST метода действия `Create()`, но только создает запрос PUT к службе:

```
// POST: Inventory/Edit/5
// Для защиты от атак чрезмерными отправками указывайте только свойства,
// которые желательно привязывать; дополнительные сведения доступны
// по ссылке http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
                                     [Bind("Make, Color, PetName, Id, Timestamp")]
                                     Inventory inventory)
{
    if (id != inventory.Id) return BadRequest();
    if (!ModelState.IsValid) return View(inventory);
    var client = new HttpClient();
    string json = JsonConvert.SerializeObject(inventory);
    var response = await client.PutAsync($"({_baseUrl}/{inventory.Id}",
        new StringContent(json, Encoding.UTF8, "application/json"));
    if (response.IsSuccessStatusCode)
    {
        return RedirectToAction(nameof(Index));
    }
    return View(inventory);
}
```

Добавление методов действий *Delete()*

Версия GET метода действия `Delete()` функционирует практически аналогично версии GET метода действия `Edit()`:

```
// GET: Inventory/Delete/5
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return BadRequest();
    }
    var inventory = await GetInventoryRecord(id.Value);
    return inventory != null ? (IActionResult)View(inventory) : NotFound();
}
```

Версия POST метода действия `Delete()` применяет `Json.NET` для преобразования значения свойства `Timestamp` в строку, чтобы его можно было добавить в маршрут. Затем используется экземпляр `HttpClient` для создания запроса `DELETE`. Если все прошло успешно, то пользователь перенаправляется на метод действия `Index()`.

```
// POST: Inventory/Delete/5
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete([Bind("Id, Timestamp")]
    Inventory inventory)
{
    var client = new HttpClient();
    var timeStampString = JsonConvert.SerializeObject(inventory.Timestamp);
    HttpRequestMessage request =
        new HttpRequestMessage(HttpMethod.Delete,
            $"{_baseUrl}/{inventory.Id}/{timeStampString}");
    await client.SendAsync(request);
    return RedirectToAction(nameof(Index));
}
```

Итоговые сведения о миграции `AutoLotMVC_Core2`

Как и можно было ожидать, внесенные в `AutoLotMVC_Core2` изменения с целью применения службы `REST` вместо `AutoLotDAL_Core2`, оказались похожими на аналогичное упражнение, которое выполнялось в главе 30. Большинство того же самого кода удалось бы использовать для взаимодействия с проектом `CarLotAPI` взамен `AutoLotAPI_Core2`.

Исходный код. Скомбинированное решение `AutolotAPI_Core2` и `AutolotMvc_Core2` доступно в подкаталоге `Chapter_34`.

Резюме

В этой главе рассмотрение `ASP.NET Core` завершилось построением службы `REST`. После создания решения вы узнали о дополнительных отличиях между `Web API 2.2` и `ASP.NET Core`, которые не раскрывались в главе 33.

Затем мы занялись построением приложения службы `AutoLotAPI_Core2` для поддержки операций `CRUD` в отношении записей `Inventory`. После добавления строки подключения в файл `appsettings.Development.json` мы внесли изменения в файлы `Startup.cs` и `Program.cs`. Мы создали класс `AutoLotExceptionHandler` для обработки исключений в методах действий приложения и зарегистрировали фильтр с помощью инфраструктуры `MVC`.

Далее мы добавили класс `InventoryController` с применением встроенного средства формирования шаблонов и обновили контроллер для использования `InventoryRepo` вместо `AutoLotContext`. Наконец, мы модифицировали `AutoLotMVC_Core2` с целью обращения к URL службы `AutoLotAPI_Core2`, применяя `Json.NET` для сериализации и десериализации записей.

Предметный указатель

A

ACID, 826
ADO (Active Data Objects), 783
ADO.NET, 836
API-интерфейс, 884; 939
 симметричный, 940
AppID, 539
ASP.NET Core, 1249; 1295
ASP.NET Core Web Application, 1244
ASP.NET MVC, 1146
ASP.NET Web API, 1189

B

BCL (base class library), 48
Bower, 1269

C

CAS (Code Access Security), 1218
CIL (Common Intermediate Language), 48; 56; 527
CLI (Common Language Infrastructure), 77
CLR (Common Language Runtime), 48; 50; 67; 606
CLS (Common Language Specification), 48; 50
COM (Component Object Model), 48
CTS (Common Type System), 48; 64

D

DAL (Data Access Layer), 1106
Dependency injection (DI), 1228
DLR (Dynamic Language Runtime), 606
Docker, 1296

E

EDM (Entity Data Model), 838
EDMX (Entity data model XML), 836
Entity Framework, 839; 865
Entity Framework 6, 836
Entity Framework Core, 1220
Entity Framework (EF), 48; 782; 836
Extensible Application Markup Language
 (XAML), 938

F

Fluent API, 1231

G

GAC (Global Assembly Cache), 73; 510; 518; 537
GUID (Globally unique identifier), 539

H

HTML, 1174

I

IDE (Integrated Development
 Environment), 57
IntelliSense, 202; 402; 436
IntelliSense в Visual Studio, 169
Intermediate Language (IL), 55

J

JSON (JavaScript Object Notation), 1194
Just-In-Time (JIT), 48

L

LINQ (Language Integrated Query), 51; 462;
 463; 482; 452
LINQ to DataSet, 457
LINQ to Entities, 457; 837
LINQ to Objects, 452; 457; 458
LINQ to XML, 457

M

Microsoft .NET Core, 79
MSIL (Microsoft Intermediate Language), 56
MVC (Model-View-Controller), 1146
MVVM (Model View ViewModel), 1105

N

.NET Core, 1210
NuGet, 933; 1223; 1247; 1297

O

ORM (Object-relational mapping), 782
Orderby, 476

P

Parallel LINQ (PLINQ), 457; 728
Plain old CLR object (POCO), 836; 1146
Platform Invocation Services (P/Invoke), 495
Prism, 1107
Process identifier (PID), 621

R

Razor, 1172; 1176
RCW (Runtime Callable Wrapper), 613

S

Snk (Strong Name Key), 539
 SOA (Service-oriented architecture), 886
 SQL Server 2016, 796
 SQL Server Express, 882
 SQL Server Management Studio, 796

T

TLS (Thread Local Storage), 622

U

UDT (User-defined type), 62

V

Visual Basic, 530
 Visual Studio, 248; 525; 530; 542; 586; 589;
 888; 976; 988
 инструменты Visual Studio, 84
 Visual Studio 2017 Enterprise, 95
 Visual Studio 2017 Professional, 95
 Visual Studio Community, 82
 Visual Studio Enterprise, 82
 Visual Studio Professional, 82
 Visual Studio Tools for Office (VSTO), 84

W

WAS (Windows Activation Service), 891
 WCF (Windows Communication Foundation),
 48; 555; 620; 827; 884; 1189
 WCF Service Library, 920
 WPF (Windows Presentation Foundation), 48;
 938; 941; 959; 972; 976; 1031

X

Xamarin, 79
 XAML, 938; 949; 957

A

Агрегация, 253
 Адаптер данных, 791
 Адрес WCF, 896
 Анимация, 1066
 в коде C#, 1080
 в разметке XAML, 1083
 с использованием дискретных ключевых
 кадров, 1085
 Аннотации данных, 1127
 Entity Framework, 842
 Аргумент
 именованный, 166
 командной строки, 102
 необязательный, 165; 206

Атрибут, 526; 645

CIL, 656
 .NET, 526
 используемый вместе с директивой .class, 659
 ограничение использования атрибутов, 584
 построение специальных атрибутов, 583
 потребители атрибутов, 579
 применение в C#, 580
 рефлексия атрибутов с использованием
 позднего связывания, 588
 роль атрибутов .NET, 578
 сокращенная система обозначения атрибу-
 тов C#, 581
 специальный, 583
 указание параметров конструктора для
 атрибутов, 581
 уровня сборки, 585

B

База данных
 AutoLot, 796
 инициализация, 863
 начальное заполнение базы данных, 864
 транзакции базы данных, 826
 Безопасность
 к типам, 350
 Библиотека
 AutoLotDAL, 1161
 CarLibrary.dll, 525
 CoreFX, 1215
 mscorlib.dll, 510
 .NET, 528
 базовых классов (BCL), 48
 классов, 510; 517; 522
 конфигурирование, 510
 построение, 522
 специальная, 510
 классов .NET, 510
 кода, 517
 параллельных задач (TPL), 718
 Блок
 finally, 302
 try/catch, 604

B

Ввод-вывод
 базовый, 106
 файловый, 738
 Визуализация графических данных, 1059
 с использованием фигур, 1033
 Визуальный конструктор Visual Studio, 248
 Время
 выполнения, 270
 квант времени, 622
 существования объектов, 483

Вставка

данных, 851
множества записей, 852

Вывод инвентаризационных записей, 875

Вызов

групповой, 389
обратный, 379

Выражение

LINQ
строго типизированое, 457
дерево выражения, 607
запроса, 457; 460; 479
с использованием типа Enumerable и
анонимных методов, 481
с использованием типа Enumerable и
лямбда-выражений, 479
с использованием типа Enumerable и
низкоуровневых делегатов, 481
с применением операций запросов, 479
лямбда, 479
сортировки, 476

Вытаскивание, 649

Г

Глобальный кеш сборок (GAC), 518; 537

Граф объекта, 487; 764; 838

Д

Данные

XML, 773
абстрагирование поставщиков данных с
использованием интерфейсов, 792
адаптеры данных, 791
аннотации данных, 842; 843; 1127
вставка данных, 851
встроенный тип данных, 450
графические, 1059
добавление данных, 1109
документов XML, 457
доступ к данным с помощью ADO.NET, 782
закрытые, 221
индексация данных
с использованием строковых значений, 419
инициализация данных
с помощью синтаксиса инициализации, 236
контракты данных WCF, 930
метаданные, 457; 559
модель привязки данных WPF, 1015
поля данных, 208; 663
работа с данными константных полей, 237
параллелизм данных, 719
подмножество данных, 473
поставщики данных ADO.NET, 784
привязанные
форматирование, 1018

реляционные, 457

создание данных, 507

стек, 178

тип данных

в библиотеке базовых классов .NET, C# и
CIL, 662

встроенный, 450

проецирование новых типов данных, 474

трансформация данных Canvas, 1048

удаление данных, 858

уровня приложения, 968

фабрики поставщиков данных, 804

фильтрация данных
с использованием метода OfType<T>(), 470

числовые

форматирование, 108

Делегат, 63; 379; 380; 396

Razor, 1176

безопасность в отношении типов, 385

низкоуровневый, 481

обобщенный, 393

EventHandler<T>, 405

объект делегата, 385

пример простейшего делегата, 384

роль делегата AsyncCallback, 694

типа делегата .NET, 379

Делегация, 253

Дерево

визуальное, 1095

выражения, 607

логическое, 1093

Дефекты, 280

Диаграмма Венна, 476

Дизассемблер промежуточного языка, 74

Динамическая загрузка сборок, 571

Директива, 645

CIL, 656

.class, 659

.maxstack, 667

using, 458; 514

сборки, 657

Диспетчер задач Windows, 621

Документация

WPF, 972

по схеме конфигурационного файла, 555

Документирование

определяемой сборки, 561

ссылаемых сборок, 561

строковых литералов, 562

Домен, 687

приложения, 620; 638; 687

.NET, 631

выгрузка доменов приложений програм-
мным образом, 639

Дубликат

устранение дубликатов, 477

3

- Заголовок файла
 - CLR, 520
 - Widows, 519
- Загрузка
 - динамическая загрузка сборок, 571
 - изображения, 1068
 - ленивая, 856
 - энергичная, 857
 - явная, 857
- Задача
 - параллелизм задач, 725
- Запись
 - вставка множества записей, 852
 - выборка записей, 852
 - инвентаризационная, 875
 - вывод, 875
 - добавление, 875
 - обновление, 860
 - редактирование, 875
 - удаление множества записей, 859
- Заполнители, 354
- Запрос
 - LING, 466; 470
 - применение к объектам коллекций, 468
 - применение к элементарным массивам, 458
 - Parallel LING (PLING), 457; 728
 - выражения запросов, 457; 460
 - построение с использованием типа Enumerable и анонимных методов, 481
 - построение с использованием типа Enumerable и низкоуровневых делегатов, 481
 - построение с применением операций запросов, 479
 - на загрузку внешней сборки
 - неявный, 533
 - явный, 533
 - на отмену, 723
 - обновления, 816
 - операции запросов, 471; 482
 - создания, 816
 - удаления, 816
- Заталкивание, 649
- Зондирование, 533

И

- Идентификатор приложения (AppID), 539
- Издатель
 - политика издателя, 551
 - отключение политики издателя, 552
- Имя
 - дружественное, 533
 - полностью заданное, 512, 517
 - строгое, 526; 539
 - генерация в Visual Studio, 542
 - генерация в командной строке, 540

Индексатор

- многомерный, 421
- определения индексаторов в интерфейсных типах, 422
- перегрузка методов индексаторов, 420
- Инициализация
 - коллекций
 - синтаксис, 360
 - словаря, 366
- Инкапсуляция, 194; 215; 221
 - с использованием свойств .NET, 224
- Инструмент
 - Bower, 1269
 - ildasm.exe, 231
 - peverify.exe, 656
 - Visual Studio, 84
- Интерполяция строк, 53
- Интерфейс, 62; 306
 - CTS, 62
 - Fluent API, 1231
 - IDataAdapter, 791
 - IDataParameter, 790
 - IDataReader, 791; 832
 - IDataRecord, 791
 - IDbCommand, 790
 - IDbCommandInterceptor, 877
 - IDbConnection, 789
 - IDbDataAdapter, 791
 - IDbDataParameter, 790
 - IDbTransaction, 789
 - IRepo, 871
 - иерархия интерфейсов, 322
 - полиморфный, 217; 242; 261
 - реализация интерфейса, 311
 - в CIL, 659
 - с использованием Visual Studio, 319
 - рефлексия реализованных интерфейсов, 567
 - специальный, 309
 - типы интерфейсов CTS, 62
 - явная реализация интерфейсов, 320
- Исключение, 280; 281; 1117
 - внутреннее, 301
 - времени выполнения, 270
 - генерация
 - общего исключения, 285
 - повторная, 300
 - конфигурирование состояния, 288
 - многочисленные исключения, 297
 - обработка исключений, 281
 - структурированная, 280
 - отладка необработанных исключений
 - с использованием Visual Studio, 304
 - перехват исключений, 287
 - построение специальных исключений, 293; 295; 296
 - строго типизированное, 293

- уровня приложения, 293
- уровня системы, 292
- фильтры исключений, 303
- Итератор, 328
 - именованный, 330

К

Каталог

- создание подкаталогов с помощью типа
DirectoryInfo, 743

Квант времени, 622

Кеш

- загрузки, 552
- сборок
 - глобальный (GAC), 73; 518; 537

Кисти WPF, 1042

Класс, 194; 483

- Activator, 575
- AllTracks, 505
- AppDomain, 632
- ApplicationException, 293
- Array, 102; 150; 155; 464
- ArrayList, 344; 345; 349
- Assembly, 563; 571
- AssemblyName, 563; 573
- Attribute, 579
- AutoLotContext, 1227
- Base, 429
- BaseRepo, 1237
- BasicMath<T>, 377
- BigInteger, 119
- BinaryOp, 381
- BitArray, 344
- BitVector32, 346
- BundleConfig, 1154
- Car, 195; 243
- Collections, 344
- Console, 106
- ContentResult, 1162
- Controller, 1275
- CSharpModule, 593
- Customer, 1225
- DataSet, 420
- DataTable, 421
- DateTime, 166
- Derived, 429
- Dictionary<TKey, TValue>, 366
- Employee, 222; 248; 251
- EntityBase, 1141; 1224
- Enum, 173; 175
- Enumerable, 455
- Environment, 104
- EventInfo, 563
- Exception, 282
- FieldInfo, 563
- FileResult, 1162

- FileStream, 499
- GC, 483
- HashSet, 359
- Hashtable, 344
- HttpStatusCodeResult, 1162
- HybridDictionary, 346
- Inventory, 1225
- InventoryController, 1191
- InventoryMetaData, 1226
- InventoryPartial, 1226
- InventoryRepo, 1240
- JsonResult, 1162
- Lazy<>, 506
- ListDictionary, 346
- List<T>, 354; 361; 454
- Manager, 249
- MemberInfo, 563
- MessageBox, 561
- MethodInfo, 563; 566
- MiniVan, 243
- Module, 563
- MulticastDelegate, 383
- Object, 153; 273; 494
- ObservableCollection<T>, 367; 1113
- Order, 1227
- ParameterInfo, 563
- PartialViewResult, 1162
- PeopleCollection, 352
- Person, 182; 276; 278
- PersonCollection, 418; 419
- Point, 423; 427
- PointRef, 180
- Predicate<T>, 410
- Process, 623
- Program, 100; 111; 158; 166; 268; 461; 467
485; 513; 566; 600; 634
- PropertyInfo, 563
- PTSalesPerson, 252
- Queue, 344
- Queue<T>, 363
- ReadOnlyObservableCollection<T>, 367
- Rectangle, 360
- RedirectResult, 1162
- RedirectToRouteResult, 1162
- SalesPerson, 258
- SavingsAccount, 209; 210; 211
- Shape, 261
- SimpleMath, 385
- SortedList, 344
- SortedSet<T>, 364
- SportsCar, 94
- Stack, 344
- Stack<T>, 362
- String, 120; 121; 278
- StringBuilder, 127
- StringCollection, 346

- System, 273
- SystemException, 293
- TimeUtilClass, 214
- Type, 173; 563
- Validation, 1117
- ValueType, 178
- VehicleDescriptionAttribute, 583
- ViewResult, 1162
- WhereArrayIterator<T>, 463
- абстрактный, 260
 - базовый, 307
- базовый, 243; 662
- дочерний, 243
- запечатанный, 252
- иерархия классов, 114
- обобщенный, 372
- обслуживающий, 208; 213
- операции приведения классов, 268
- преобразования между связанными типами классов, 428
- родительский, 216; 243
- статический, 213
- суперкласс, 243
- тип класса
 - CIL, 658
 - CTS, 61; 62
- частичный, 240
 - сценарии использования, 241
- Клонирование, 331
- Ключ
 - открытый, 539
 - маркер открытого ключа, 537
 - секретный, 539
 - строгого имени, 539
- Ключевое слово
 - as, 269; 314
 - async, 731
 - await, 731
 - base, 249
 - const, 237
 - default, 373
 - delegate, 381, 383
 - dynamic, 600; 604; 608
 - event, 398
 - explicit, 429
 - fixed, 444; 449
 - implicit, 429; 433
 - interface, 310
 - is, 271; 314
 - lock, 709
 - namespace, 510
 - new, 152; 196; 485
 - operator, 42
 - override, 256
 - protected, 251
 - ref, 161; 163
 - sealed, 245; 259
 - sizeof, 450
 - stackalloc, 444; 449
 - static, 207; 213; 229; 425; 455
 - this, 201
 - unsafe, 445
 - using, 214; 500; 513
 - var, 467
 - virtual, 256
 - where, 375
 - yield, 328
 - контекстное, 225
- Ключевые слова
 - C#, 662
 - XAML, 952
- Код
 - CIL, 521; 527; 646; 665; 667
 - компиляция с помощью ilasm.exe, 655
 - неуправляемый, 54
 - ошибки приложения, 101
 - перенос кода, 1139
 - управляемый, 54
 - хеш-код, 539
- Коллекция, 342; 1107
 - наблюдаемая, 1112
 - обобщенная, 353
 - объектов, 457; 775
 - инициализация коллекций, 360
- Команда
 - CLI, 1216
 - WPF, 997
 - объекты команд, 812
- Командная строка
 - аргументы командной строки, 102
- Компилятор
 - C# (csc.exe), 445; 539
 - JIT, 59
- Компиляция
 - кода CIL с помощью ilasm.exe, 655
 - файла CILTypes.il, 661
- Компоновки, 974; 1177
- Конкатенация строк, 122
- Конструктор, 128; 197
 - Visual Studio, 988
 - WPF, 976
 - вызов с помощью синтаксиса инициализации, 235
 - выпуск конструкторов, 681
 - добавление, 817
 - обновление, 1280
 - построение цепочки конструкторов, 202
 - специальный, 177; 198
 - стандартный, 113; 177; 197
- Контейнеризация, 1213
- Контекст, 687

Контракт
WCF, 892
службы, 934
Контроллер, 1147; 1162
Конфигурирование
закрытых сборок, 534
объектов для сериализации, 765
разделяемых сборок, 546
состояния исключения, 288
стандартного пространства имен, 516
Копирование
массовое, 833
с помощью ADO.NET, 831
Кортеж, 53; 189; 191
деконструирование кортежей, 192
Куча
очищенная и сжатая, 488
управляемая, 483
размещение объектов в управляемой куче, 486

Л

Литерал
строковый
документирование, 562
Лямбда-выражение, 379; 408; 410; 454;
479

М

Манифест
SharedCarLibClient, 546
строго именованной сборки, 541
Манифест сборки, 60; 521; 522; 525
Маркер
открытого ключа, 537
Маршрутизация, 1158; 1192
с помощью атрибутов, 1192
Массив, 150
интерфейсных типов, 318
использование в качестве аргументов, 154
использование в качестве возвращаемых значений, 154
локальный
неявно типизированный, 152
многомерный, 153
применение запросов LINQ к элементарным массивам, 458
прямоугольный, 153
синтаксис инициализации массивов C#, 151
Меню
построение системы меню, 992
Метаданные, 457; 559
типов, 521; 528; 557
Метод
HTML, 1174
Main(), 100; 968

абстрактный, 218
анонимный, 406; 416; 481
асинхронный, 101
виртуальный, 257
вызов
асинхронный, 692
без параметров, 577
расширяющих методов, 435
с параметрами, 577
индексаторный, 417
перегрузка, 420
модификаторы параметров, 157
перегрузка методов, 167; 199
средство IntelliSense в Visual Studio для перегруженных методов, 169
переопределение метода, 256
преобразование метода, 392
групповое, 392
расширяющий, 434; 455; 463
вызов, 435
импортирование, 436
поддержка средством IntelliSense, 436
рефлексия методов, 566
синтаксис групповых преобразований методов, 401
статический, 386
Миграции Entity Framework, 865
Минификация, 1155
Модель, 1106; 1146
анемичная, 1106
включения/делегации (агрегация), 253
генерация модели, 843
использование классов моделей в коде, 851
наблюдаемая, 1107; 1111
обновление модели, 867
отделение модели от уровня доступа к данным, 882
представления, 1106
привязки данных WPF, 1015
сущностных данных (EDM), 838
Модификатор
out, 159; 160
params, 163
ref, 161
Модификаторы
доступа
C#, 219
private, 220
protected, 220
protected internal, 220
параметров в C#, 158
для методов, 157
Модуль, 628
набор модулей, 679

Н

Наследование, 215; 216; 242
 классическое, 243; 432
 межъязыковое, 531
 множественное, 245; 324
 с помощью интерфейсных типов, 324

О

Области, 1177
 Обобщения, 342
 в CIL, 661
 Обратные вызовы, 379
 Общая система типов (CTS), 61
 Общеязыковая исполняющая среда (CLR), 67
 Объект, 483
 Window, 970
 время жизни объекта, 484
 время существования объектов, 483
 граф объектов, 487; 838
 десериализация объектов с использованием BinaryFormatter, 771
 инициализация объектов, 234; 454
 синтаксис, 454
 коллекция объектов, 457
 команды, 812
 контекстно-связанный, 642
 контекстные границы объектов, 640
 конфигурирование объектов для сериализации, 765
 освобождаемый, 494; 498
 подключения, 789; 809
 поколения объектов, 489
 эфемерные, 490
 приложения, 99
 размещение в управляемой куче, 486
 сериализация, 738; 762
 коллекций объектов, 775
 с использованием BinaryFormatter, 769
 создание объектов
 ленивое (отложенное), 504
 финализируемый, 492; 497
 чтения данных, 814
 Оператор
 catch, 300
 fixed, 450
 Операция
 -, 444
 --, 425; 444
 -=, 391; 425
 ->, 444; 448
 !=, 444
 ?, 403
 [], 444

* , 444
 &, 444
 +, 444
 ++, 425; 444
 +=, 389; 425
 <, 444
 <=, 444
 ==, 444
 перегрузка, 427
 ==>, 157; 415; 454
 >, 444
 >=, 444
 descending, 476
 nameof, 1112
 new, 113
 null-условная, 188
 orderby, 476
 агрегирования LINQ, 478
 бинарная
 перегрузка, 423
 запроса, 479; 482
 LINQ, 471
 лямбда (=>), 157; 415; 454
 объединения с null, 187
 отсутствие ограничений операций, 377
 перегрузка операций, 422; 426; 427
 приведения классов, 268
 присваивания, 179
 сокращенного, 425
 унарная
 перегрузка, 425
 эквивалентности
 перегрузка, 426
 Оснастка, 594
 построение
 на C#, 594
 на Visual Basic, 594
 Отбрасывание, 53
 использование с кортежами, 191
 переменных, 158
 Очередь, 363
 финализации, 497
 Ошибка
 на этапе компиляции, 112
 пользовательская, 280

П

Пакет
 NuGet, 933
 объединение в пакеты, 1155
 Панели
 вложенные, 991
 Парадигма Code First, 841; 843
 Параллелизм, 707; 876
 данных, 719

Параметр

- выходной, 159
- именованный, 166
- модификаторы параметров
в C#, 158
- для методов, 157
- отображение параметров на локальные
переменные в CIL, 668
- передача параметров по значению, 158
- рефлексия параметров, 569
- типа, 354

Паттерн

- MVC, 1146
- MVVM, 1105
- "Отправка-перенаправление-
получение", 1280

Перегрузка

- метода, 167; 199
- индексаторов, 420
- операций, 422
- сравнения, 427
- эквивалентности, 426

Переменная

- внешняя, 407
- инициализация, 111
- локальная, 668
- неявно типизированная, 453; 462
- объявление, 111; 668
- ссылочная, 162
- отбрасывание переменных, 158
- типа перечисления, 172
- типа структуры, 177

Перехват, 877

- исключений, 287
- событий клавиатуры, 972
- событий мыши, 971

Перечисление, 63; 170

- в CIL, 660
- файлов с помощью типа DirectoryInfo, 742

Перечислитель, 170

Перья WPF, 1042

Платформа

- .NET, 48
- .NET Compiler, 1216

Подключение

- открытие, 817
- закрытие, 817

Подключенный уровень ADO.NET, 809

Подобъект, 469

Позднее связывание, 557; 575

Поиск с помощью Find(), 854

Поле

- допускающее только чтение, 239
- доступ через указатели, 448
- закрытое, 766
- константное, 237

- определение полей данных в CIL, 663
- открытое, 766
- рефлексия полей, 566
- статическое, 239

Полиморфизм, 215; 217; 218

Политика издателя

- отключение политики издателя, 552

Поставщики данных

- ADO.NET, 784
- абстрагирование поставщиков данных
с использованием интерфейсов, 792
- фабрики поставщиков данных, 804

Поток, 687

- главный, 621
- закрытие потока, 752
- освобождение потока, 752
- переднего плана, 706
- пул потоков CLR, 716
- рабочий, 622
- ручное создание вторичных потоков, 702
- фоновый, 706

Представление, 1106; 1147; 1283

- MVC, 1177

Преобразование

- между связанными типами классов, 428
- неявное, 432
- процедуры преобразования, 429
- числовое
- неявное, 428
- явное, 428

Преобразователи типов, 954

Привязка

- HTTP, 894
- TCP, 895
- WCF, 893; 974
- добавление привязок, 1109

Приложение

- базовая структура приложения WCF, 891
- домен приложения, 638
- идентификаторы приложений (AppID), 539
- код ошибки приложения, 101
- копирование, 532
- объект приложения, 99
- построение клиентского приложения
Visual Basic, 530
- WCF, 909
- построение приложений .NET, 84
- в среде Windows, 81
- под управлением операционной системы,
отличающейся от Windows, 96
- построение расширяемого консольного
приложения, 597
- развертывание с помощью Xsoru, 532
- служб ASP.NET Core, 1295
- создание консольного клиентского прило-
жения, 825

создание проекта консольного приложения
 Visual Basic, 531
 удаление, 532
 шаблон приложения ASP.NET MVC, 1148
 Проверка достоверности, 1105; 1117
 Программа
 структура простой программы C#, 98
 Программирование
 асинхронное, 686
 в стиле черного ящика, 222
 многопоточное, 686
 на основе атрибутов, 557
 объектно-ориентированное, 215
 параллельное, 686
 с использованием TPL, 717
 с использованием обратных вызовов Timer, 714
 Проект
 Mono, 78
 Xamarin, 79
 Проектирование
 возвратное, 650
 Производительность, 347
 Пространства имен
 ADO.NET, 787
 CIL, 658
 XML, 950
 Пространство имен, 69
 System.Collections, 344; 351
 System.Collections.Concurrent, 346
 System.Collections.Generic, 353; 358
 System.Collections.ObjectModel, 367
 System.Collections.Specialized, 346; 351
 System.Configuration, 554
 System.Data, 788
 System.Diagnostics, 623
 System.Dynamic, 607
 System.IO, 738
 System.Reflection, 563
 System.Reflection.Emit, 675
 System.Threading, 698
 System.Threading.Tasks, 718
 конфигурирование стандартного пространства имен, 516
 определение специальных пространств имен, 510
 создание вложенных пространств имен, 515
 стандартное пространство имен Visual Studio, 516
 Процедура
 хранимая, 823
 Процесс, 620; 687
 Псевдоним
 разрешение конфликтов имен с помощью псевдонимов, 513
 Пул потоков CLR, 716

Р

Развертывание
 в SQL Server Express с использованием миграций, 882
 Распаковка (unboxing), 347; 1084
 Расширяемость, 589
 Редактор
 XAML, 959
 трансформаций Visual Studio, 1050
 Ресурс, 1066
 WPF, 1066
 неуправляемый, 491; 495
 Рефлексия, 461; 562
 атрибутов
 с использованием позднего связывания, 588
 с использованием раннего связывания, 587
 возвращаемых значений методов, 569
 методов, 566
 обобщенных типов, 569
 параметров, 569
 полей, 566
 разделяемых сборок, 573
 реализованных интерфейсов, 567
 свойств, 566
 типов, 557

С

Сборка
 CILCarClient.exe, 673
 CILCars.dll, 671
 *.dll, 522
 Microsoft.CSharp.dll, 603
 .NET, 55; 670
 формат, 518
 System.Core.dll, 607
 System.Data.dll, 783
 System.Numerics.dll, 118
 WCF, 888
 WPF, 942
 атрибуты уровня сборки, 585
 взаимодействия, 612
 основная (PIA), 613
 внешняя, 522; 656
 выпуск сборки, 679
 глобальный кеш сборок, 537
 динамическая, 600; 645; 674; 677; 683
 директивы, связанные со сборками, 657
 загруженная, 635
 динамически, 571
 закрытая, 518; 532
 конфигурирование, 534
 замораживание текущей версии сборки, 547
 запрос на загрузку внешней сборки
 неявный, 533
 явный, 533

- конфигурируемая, 518
- манифест сборки, 60; 521; 522
- манифест строго именованной сборки, 541
- определяемая, 561
 - документирование определяемой сборки, 561
- основные сборки, связанные с LINQ, 458
- подчиненная, 522
- политики издателя, 551
- разделяемая, 518; 537; 544; 549; 556
 - конфигурирование, 546
- рефлексия разделяемых сборок, 573
- роль сборок .NET, 516
- самоописательная, 518
- ссылаемая, 561
 - документирование, 561
- строго именованная, 517
 - установка в GAC, 543
- текущая, 657
- тестирование сборки AutoLotDAL, 865
- удостоверение закрытой сборки, 533
- Сборка мусора
 - параллельная, 490
 - принудительный запуск, 492
 - фоновая, 490
- Сборщик мусора, 485
- Свойства
 - Priority, 701
 - автоматические, 230; 232
 - инициализация, 233
 - допускающие только чтение, 228
 - допускающие только запись, 228
 - зависимости, 949
 - именованные
 - синтаксис, 584
 - навигационные, 855
 - определение свойств в CIL, 664
 - открытые, 766
 - присоединяемые, 956
 - рефлексия свойств, 566
 - сжатые до выражений, 227
- Связывание
 - позднее, 557; 575
 - рефлексия атрибутов с использованием позднего связывания, 588
 - раннее
 - рефлексия атрибутов с использованием раннего связывания, 587
- Сериализация
 - выбор формatera сериализации, 767
 - двоичная, 776
 - коллекций объектов, 775
 - настройка сериализации
 - с использованием ISerializable, 777
 - с использованием атрибутов, 780
 - объектов
 - .NET, 763
 - с использованием BinaryFormatter, 769
- "Синтаксический сахар", 157
- Система
 - CTS, 50
 - уведомлений привязки WPF, 1107
- Словарь
 - инициализация словаря, 366
- Служба
 - WCF
 - хостинг, 900
 - асинхронный вызов из клиента, 928
 - визуализации графики WPF, 1031
 - реализация контракта службы, 934
 - тестирование службы, 936
- Событие, 379; 974
 - C#, 396
 - прослушивание входящих событий, 400
 - маршрутизируемое, 1002; 1003
 - перехват событий
 - клавиатуры, 972
 - мышь, 971
 - пузырьковое, 1003
 - туннельное, 1003
- Сокрытие членов, 266
- Сортировка, 339
 - выражения сортировки, 476
- Спецификация
 - CLS, 65
 - CTS, 50
- Среда
 - CLR, 50; 533
 - IDE, 589
 - Visual Studio, 84; 525; 530
 - исполняющая, 67
- Средство IntelliSense, 402; 436
 - активизация, 202
 - для перегруженных методов, 169
- Ссылка, 483
 - this, 669
 - на внешние сборки в CIL, 656
 - на тип, 558
 - объектная, 486
- Стек, 178; 649
 - виртуальный стек выполнения, 649
 - коды операций CIL,
 - связанные с загрузкой в стек, 666
 - связанные с извлечением из стека, 667
- Стили, 1066
 - анимированные, 1091
 - применение стилей в коде, 1091
- Строка, 123
 - интерполяция строк, 53; 128
 - конкатенация строк, 122
- Структура, 175
 - в CIL, 660
 - создание, 372
- Суперкласс, 243

Сущности (entity), 836
роль сущностей, 838

Т

Таблица

маршрутов, 1158
метаданных, 457
объектов
доступных для финализации, 497

Технология

COM, 48
Docker, 1296
LING, 51; 452

Тип, 98

enum, 170
Enumerable, 481
анонимный, 438; 456
безопасность к типам, 350
вложенный, 220
встроенные типы данных CTS, 64
делегата, 379; 380
допускающий значение null, 185
закрепление типа
 посредством ключевого слова fixed, 449
значения, 178; 184; 484
содержащий ссылочные типы, 180
интерфейсный, 306
 CTS, 62
 массивы интерфейсных типов, 318
класса, 194
 CIL, 658
 CTS, 61
контекстно-свободный, 641
контекстно-связанный, 641
метаданные типов, 521; 528; 557
обобщенный, 354
общая система типов (CTS), 61
определение типа, 558
освобождаемый, 501
параметр типа, 354
преобразование типов, 428
преобразователи типов, 954
реализующий специфичные интерфейсы
 расширение, 437
рефлексия типов, 557; 569
сериализируемый, 765
ссылка на тип, 558
ссылочный, 124; 178; 182
 отличия между типами значений и
 ссылочными типами, 184
 передача по значению, 182
 передача по ссылке, 183
структур CTS, 62
точность воспроизведения типов, 768
указателя, 444
финализируемый, 501; 508

Тип данных

C#, 110
bool, 110
byte, 110
char, 111
decimal, 111
double, 111
float, 111
int, 110
long, 111
object, 111
sbyte, 110
short, 110
string, 111
uint, 110
ulong, 111
ushort, 110
в библиотеке базовых классов .NET, 662
встроенный, 450
проецирование новых типов данных, 474
Типизация
ADO.NET, 827
неявная, 600
Трансформация, 1046
данных Canvas, 1048

У

Уведомления, 1105

Указатель

доступ к полям через указатели (->), 448
на новый объект, 486
на следующий объект, 486

Упаковка (boxing), 347

Управляющие последовательности, 123

Уровень

автономный, 784
подключенный, 783
ADO.NET, 809

Установка

SQL Server 2016, 796
SQL Server Management Studio, 796
Visual Studio 2017, 82

Утилита

al.exe, 551
AutoMapper, 1196
Class Designer, 92
csc.exe, 445
dumpbin.exe, 519; 520
gacutil.exe, 543; 552; 556
ilasm.exe, 645; 655
ildasm.exe, 57; 74; 231; 485; 487; 525; 527;
 541; 557; 580
peverify.exe, 656
sn.exe, 539; 556
svcutil.exe, 909

Ф

Фабрика поставщиков данных, 804; 807
 Файл
 App.xaml, 965
 AssemblyInfo.cs, 526; 541; 586
 BundleConfig.cs, 1154
 CILTypes.il, 661
 CLR, 520
 *.config, 510; 535; 554
 *.dll, 510; 517; 522
 *.exe, 517
 FilterConfig.cs, 1155
 Global.asax.cs, 1151
 *.il, 654
 JavaScript, 1157
 *.snk, 539; 542
 Web.config, 935
 Widows
 заголовок файла, 519
 XML, 510
 документация по схеме конфигурацион-
 но файла, 555
 перечисление файлов с помощью типа
 DirectoryInfo, 742
 программное слежение за файлами, 760
 Фильтры, 1155
 в ASP.NET MVC, 1156
 исключений, 303
 Финализация, 497
 Форма, 1184
 Формат
 JSON, 1194; 1297
 сборки .NET, 518
 Форматер сериализации, 767
 Функция
 Razor, 1176
 локальная, 169; 329; 735
 обмена, 447
 обратного вызова, 379

Х

Хеш-код, 539
 Хостинг службы WCF, 900
 внутри Windows-службы, 923
 Хранилище
 добавление хранилищ для многократного
 использования кода, 871
 управление хранилищем, лежащим в осно-
 ве перечисления, 171
 Хранимая процедура, 823

Ц

Цифровая подпись, 540

Ч

Член
 виртуальный, 218
 запечатывание виртуальных членов, 259
 статический
 импортирование, 214
 типа CTS, 64

Ш

Шаблон
 ASP.NET Core Web API, 1295
 ASP.NET Core Web Application, 1244
 ASP.NET MVC, 1148
 Razor, 1181
 URL, 1158
 WCF Service, 920; 931
 WCF в Visual Studio, 888; 959
 WPF, 1066
 встраивание шаблонов в стили, 1103
 как ресурс, 1100
 освобождения
 формализованный, 502
 элемента управления, 1099

Э

Элемент
 дочерний, 954
 корневой, 487
 Элемент управления, 974
 ComboBox, 1012
 Ink API, 974
 InkCanvas, 1009; 1011
 RadioButton, 1008
 TabControl, 1006
 WPF, 975; 980
 WPF в Visual Studio, 977

Я

Язык
 C#, 51; 165
 CIL, 56; 59; 527; 645; 656; 666
 коды операций CIL, 646
 CLR, 606
 DLR, 606
 IL, 55
 LINQ, 51
 MSIL, 56

ПРОФЕССИОНАЛАМ ОТ ПРОФЕССИОНАЛОВ

Язык программирования C# 7 и платформы .NET и .NET Core

Эта классическая книга представляет собой всеобъемлющий источник сведений о языке программирования C# и о связанной с ним инфраструктуре. В 8-м издании книги вы найдете описание функциональных возможностей самых последних версий C# 7.0 и 7.1 и .NET 4.7, а также совершенно новые главы, посвященные легковесной межплатформенной инфраструктуре .NET Core. Перепроектированные инфраструктуры ASP.NET Core 2.0 и Entity Framework (EF) Core 2.0 рассматриваются наряду с последними обновлениями, внесенными в .NET 4.7, которые затронули Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), ASP.NET MVC и ASP.NET Web API.

Погрузитесь в книгу и выясните, почему на протяжении более 15 лет она была лидером у разработчиков по всему миру. Сформируйте прочный фундамент в виде знания приемов объектно-ориентированной обработки, атрибутов и рефлексии, обобщений и коллекций, а также множества более сложных тем, которые не раскрываются в других книгах (коды операций CIL, выпуск динамических сборок и т.д.). С помощью настоящей книги вы сможете уверенно использовать язык C# на практике и хорошо ориентироваться в мире NET.

В книге рассматриваются следующие темы

- Новейшие возможности версий C# 7.0 и 7.1, от кортежей до сопоставления с образцом
- Базовые основы легковесной платформы Microsoft с открытым кодом .NET Core, включая ASP.NET Core MVC, веб-службы ASP.NET Core и Entity Framework Core
- Полное описание XAML, .NET 4.7 и Visual Studio 2017
- Философия, лежащая в основе .NET и новой межплатформенной версии .NET Core

НА ВЕБ-САЙТЕ

Исходные коды всех примеров, рассмотренных в книге, а также дополнительные приложения на русском языке в формате PDF можно загрузить с веб-сайта издательства по адресу: <http://www.williamspublishing.com/Books/978-5-6040723-1-8.html>.

Категория: языки программирования/C#

Предмет рассмотрения: C# 7.0 и 7.1

Уровень: для пользователей средней и высокой квалификации



ISBN= 978-5-6040723-1-8



www.williamspublishing.com

Apress®