

Министерство образования и науки Российской Федерации  
Федеральное агентство по образованию  
Ярославский государственный университет им. П. Г. Демидова  
Кафедра компьютерной безопасности и математических методов  
обработки информации

**О. П. Якимова**

# **Языки программирования**

## **Часть 1**

*Лабораторный практикум*

*Рекомендовано*

*Научно-методическим советом университета для студентов,  
обучающихся по специальности Компьютерная безопасность*

Ярославль 2010

УДК 004.43  
ББК 3 973.2–018.1я73  
Я 45

*Рекомендовано  
Редакционно-издательским советом университета  
в качестве учебного издания. План 2009/10 года*

Рецензент  
кафедра компьютерной безопасности и математических методов обработки  
информации Ярославского государственного университета  
им. П. Г. Демидова

**Якимова, О. П. Языки программирования. Ч. 1:** лаборатор-  
Я 45 ный практикум / О. П. Якимова; Яросл. гос. ун-т им. П. Г. Де-  
мидова. – Ярославль : ЯрГУ, 2010. – 68 с.

Предназначен для студентов, обучающихся по специальности  
090102.65 Компьютерная безопасность (дисциплина «Языки  
программирования», блок ОПД), очной формы обучения.

УДК 004.43  
ББК 3 973.2–018.1я73

© Ярославский государственный университет  
им. П. Г. Демидова, 2010

---

Учебное издание

**Якимова Ольга Павловна**  
**Языки программирования**  
**Часть 1**

*Лабораторный практикум*

Редактор, корректор И. В. Бунакова  
Верстка Е. Л. Шелехова

Подписано в печать 06.05.10. Формат 60×84 <sup>1</sup>/<sub>16</sub>.  
Бум. офсетная. Гарнитура "Times NewRoman".  
Усл. печ. л. 3,95. Уч.-изд. л. 2,16.  
Тираж 50 экз. Заказ

Оригинал-макет подготовлен  
в редакционно-издательском отделе Ярославского  
государственного университета им. П. Г. Демидова.

Отпечатано на ризографе.  
Ярославский государственный университет им. П. Г. Демидова.  
150000, Ярославль, ул. Советская, 14.

## **Введение**

Целью лабораторного практикума по курсу «Языки программирования» является получение практических навыков прикладного программирования с применением объектно ориентированного подхода (ООП) к проектированию и реализации программного обеспечения на единой универсальной платформе разработки приложений Microsoft .NET.

В задачи настоящего лабораторного практикума входит освоение таких основополагающих концепций ООП, как абстракция, наследование, инкапсуляция и полиморфизм. Разработка приложений ведется на языке объектно ориентированного программирования C# в среде Microsoft Visual Studio .NET.

Важным разделом практикума является исследование систем типизации среды .NET. При этом наибольшее внимание уделяется таким важным аспектам, как пространства имен, абстрактные типы данных, классы и методы, интерфейсы и обобщенные коллекции. В практикуме раскрываются вопросы, связанные с событийно-управляемым программированием. Краткое введение в теорию обработки событий сопровождается рядом примеров из практики программирования на языке C#. Отдельно исследуется обработка исключительных ситуаций.

В результате освоения практикума студенты получают возможность самостоятельной разработки широкого спектра прикладных программных решений в условиях современной компонентной архитектуры.

# Лабораторная работа 1. Объекты и классы (конструкторы, инкапсуляция, свойства, перегрузка операций)

*Цель работы:* познакомиться с основой объектного подхода в языке C#, созданием объектов, классов и механизмом инкапсуляции на основе свойств.

## **Необходимые теоретические сведения**

### **Классы и объекты**

Формально класс – это пользовательский тип, состоящий из полей данных и методов, которые работают с этими данными. Множество полей данных представляет «состояние» экземпляра класса. Экземпляры класса иначе называются объектами.

Сила объектно ориентированных языков состоит в том, что в одном пользовательском типе за счет группировки данных и функциональности вы можете смоделировать поведение некоторой сущности реального мира. Например, для представления сотрудника в системе расчета зарплаты, необходимо создать класс, содержащий имя, текущую зарплату и идентификатор (табельный номер). Кроме того, класс Employee должен содержать методы GiveBonus() для начисления премии и DisplayStats() для вывода информации о сотруднике.

Синтаксис класса:

```
тип_доступа class имя_класса
{
    тип_доступа тип имя_переменной1;
    тип_доступа тип имя_переменной2;
    ...
    тип_доступа возвращаемый_тип имя_метода1(список_параметров)
    {
        тело_метода
    }
    ...
}
```

Модификаторы доступа определяют поле видимости данного класса. Для классов предназначены два модификатора или типа доступа:

- `public` – класс доступен для других компонент (сборок);
- `internal` – класс видим только внутри данной сборки (приложения).

По умолчанию применяется модификатор `internal`. Модификаторы доступа также указываются и перед полями и методами класса: `private` (по умолчанию), `public`, `protected`, `internal` и `protected internal`. Члены класса с типом доступа `public` доступны везде за пределами данного класса, с типом доступа `protected` – внутри членов данного класса и производных, с типом доступа `private` – только для других членов данного класса. Тип доступа `internal` применяется для типов, доступных в пределах одной сборки.

Классы в C# могут определять любое количество конструкторов. Конструктор класса – метод для инициализации переменных экземпляра класса объекта при его создании. Он имеет то же имя, что и его класс. В конструкторах тип возвращаемого значения не указывается явно. Конструкторы используются для присваивания начальных значений переменным экземпляра и для выполнения любых других процедур инициализации, необходимых для создания объекта.

Все классы имеют конструкторы независимо от того, определен он или нет. По умолчанию в C# предусмотрено наличие конструктора, который присваивает нулевые значения всем переменным экземпляра (для переменных обычных типов) и значения `null` (для переменных ссылочного типа). Но если конструктор явно определен в классе, то конструктор по умолчанию использоваться не будет. Приведем пример создания класса `Employee`, описанного выше.

```
public class Employee
{
    private string fullName; // Имя сотрудника
    private int empID; // табельный номер
    private float currPay; // зарплата
    // Конструктор по умолчанию
}
```

```

public Employee(){
    // Пользовательский конструктор
    public Employee(string FullName, int emplID, float currPay)
    {
        this.fullName = FullName;
        this.emplID = emplID;
        this.currPay = currPay;
    }
    // Еще один пользовательский конструктор с одним
    // параметром, который перенаправляет вызов конструктору с
    // тремя параметрами
    public Employee(string fullName)
        : this(fullName, 3333, 0.0F){}
    // Методы класса
    public void GiveBonus(float amount)
    { currPay += amount;
    }
    public virtual void DisplayStats()
    {
        Console.WriteLine("Name: {0}", fullName);
        Console.WriteLine("Pay: {0}", currPay);
        Console.WriteLine("ID: {0}", emplID);
    }
}

```

После создания внутренних данных состояния и набора конструкторов класса следующий шаг состоит в реализации деталей *открытого интерфейса* класса. Этим термином называют множество членов, которые непосредственно доступны из объектной переменной с помощью оператора `.` (точка). С точки зрения программиста, открытый интерфейс – это любой член, объявленный в классе с использованием модификатора доступа `public`. Это могут быть поля данных, константы/поля только для чтения, методы, свойства.

## Инкапсуляция на основе свойств класса

Во всех объектно ориентированных языках используются три основных принципа ООП, которые часто называют «столпами ООП»: инкапсуляция (реализованный в данном языке механизм скрытия внутренней реализации объекта), наследование (реализованный в данном языке механизм многократного использования кода), полиморфизм (реализованный в данном языке механизм трактовки связанных объектов одинаковым образом).

Концепция инкапсуляции вращается вокруг идеи, что поля данных объекта не должны быть доступными непосредственно через открытый интерфейс. Вместо определения открытых полей (которые пользователь может непреднамеренно изменить недопустимым образом, из-за чего произойдёт сбой в работе программы) определяются закрытые поля данных и связанные с ними *свойства*. Свойство – пара методов со специальными именами. Метод `set()` вызывается при задании значения свойства, метод `get()` – при получении значения свойства. Обращение к свойству выглядит как обращение к полю данных, но транслируется в вызов одного из двух методов. Определяя в свойстве только один из двух методов, получаем свойства только для чтения и для записи. Каждый из методов может иметь модификатор доступа. Приведем пример для класса `Employee`: свойство `Pay` инкапсулирует поле `currPay`, свойство `ID` – поле `empID`, свойство `Name` инкапсулирует поле `fullName`.

```
public class Employee
{
    private string fullName; // Имя сотрудника
    private int empID; // табельный номер
    private float currPay; // зарплата
    ....
    // свойство для поля empID
    public int ID
    {
        get { return empID; }
        set { // проверка значения на корректность
            if (value > 0) && (value < 9999)

```

```

        emplID = value;
    else { emplID = 3333;
        Console.WriteLine("Задано значение по
умолчанию");
    }
}

//Свойство для поля currPay.
public float Pay
{
    get {return currPay;}
    set { if( (value >=0)&&(value < 100 000))
        currPay = value;
        else currPay =0;
    }
}

// Свойство для поля fullName.
public string Name
{
    get { return fullName; }
    set { fullName = value; } // здесь должна быть проверка на
        // допустимость
}

.....
} // к классу Employee

```

## Создание объекта

При создании объекта класса происходит вызов соответствующего конструктора. Следующий метод Main() создаёт несколько объектов Employee, используя наши пользовательские конструкторы, и демонстрирует работу со свойствами класса.

```

class Program
{
    static void Main(string[] args)
    {

```



```

// Создание сотрудника пользовательским конструктором
Employee ann = new Employee("Ivanova Anna I.", 1001,
12000.0f);
ann.GiveBonus(400);
ann.DisplayStats();
Employee piter = new Employee("Petrov Petr A.", 1002,
14000.0f);
piter.DisplayStats();
// Использование конструктора по умолчанию и свойств
Employee brenner = new Employee();
brenner.ID = 2022;
brenner.Pay = 7500;
brenner.Name = "Smirnov Boris P.";
brenner.Pay += 500;
brenner.DisplayStats();
}
}

```

## Перегрузка операторов C#

Язык C# позволяет выполнять перегрузку операторов для их использования в собственных классах. Это позволяет добиться естественного вида определяемого пользователем типа данных и использовать его в качестве основного типа данных. Например, можно создать новый тип данных с именем `ComplexNumber`, представляющий комплексное число, и определить методы выполнения математических операций над такими числами с использованием стандартных арифметических операторов, например оператора `+` для сложения двух комплексных чисел.

Чтобы выполнить перегрузку оператора, необходимо написать функцию с указанием имени оператора, а затем символа оператора, для которого выполняется перегрузка. Например, так выполняется перегрузка оператора `+`:

```

public static ComplexNumber operator+(ComplexNumber a,
ComplexNumber b)

```

Все перегрузки операторов являются статическими методами класса. Кроме того, следует учесть, что если перегружается

оператор равенства (==), то необходимо перегрузить и оператор неравенства (!=). Операторы < и >, а также <= и >= тоже следует перегружать парами.

Ниже приведен полный список операторов, которые можно перегрузить:

- Унарные операторы: +, -, !, ~, ++, --, true, false
- Бинарные операторы: +, -, \*, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=

Приведенный ниже пример кода создает класс ComplexNumber, который перегружает операторы + и -:

```
public class ComplexNumber
{
    private int real;
    private int imaginary;
    public ComplexNumber() : this(0, 0) //конструктор
    {}
    public ComplexNumber(int r, int i) // конструктор
    {
        real = r;
        imaginary = i;
    }
    // Перекрытие метода ToString(), чтобы выводить
    // комплексное число в привычной форме:
    public override string ToString()
    {
        return(System.String.Format("{0} + {1}i", real, imaginary));
    }
    // Перегрузка оператора '+' :
    public static ComplexNumber operator+(ComplexNumber a,
    ComplexNumber b)
    {
        return new ComplexNumber(a.real + b.real, a.imaginary +
    b.imaginary);
    }
    //Перегрузка оператора '-' :
```

```

    public static ComplexNumber operator-(ComplexNumber a,
ComplexNumber b)
    {
        return new ComplexNumber(a.real - b.real, a.imaginary -
b.imaginary);
    }
}

```

Этот класс позволяет создавать комплексные числа и выполнять операции с двумя комплексными числами:

```

class TestComplexNumber
{
    static void Main()
    {
        ComplexNumber a = new ComplexNumber(10, 12);
        ComplexNumber b = new ComplexNumber(8, 9);
        System.Console.WriteLine("Complex Number a = {0}",
a.ToString());
        System.Console.WriteLine("Complex Number b = {0}",
b.ToString());
        ComplexNumber sum = a + b;
        System.Console.WriteLine("Complex Number sum = {0}",
sum.ToString());
        ComplexNumber difference = a - b;
        System.Console.WriteLine("Complex Number difference = {0}",
difference.ToString());
    }
}

```

Как наглядно показано в программе, теперь можно использовать операторы «плюс» и «минус» для объектов, принадлежащих к классу ComplexNumber. Ниже приведены выходные данные:

```

Complex Number a = 10 + 12i
Complex Number b = 8 + 9i
Complex Number sum = 18 + 21i
Complex Number difference = 2 + 3i

```

## **Контрольные вопросы**

1. Что понимается под термином «класс»?
2. Какие элементы определяются в составе класса?
3. Каково соотношение понятий «класс» и «объект»?
4. Что понимается под термином «члены класса»? Какие члены класса Вам известны?
5. Какие члены класса содержат код, а какие – данные?
6. Дайте определение инкапсуляции.
7. С какой целью используются свойства класса?
8. Перечислите пять разновидностей членов класса, специфичных для языка C#.
9. Что понимается под термином «конструктор»? В чем состоит его назначение?
10. Сколько конструкторов может содержать класс языка C#?
11. Приведите синтаксис описания класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
12. Какие модификаторы типа доступа Вам известны?
13. В чем заключаются особенности доступа членов класса с модификатором `public`? `private`? с модификатором `protected`? `internal`?
14. Каждый ли класс языка C# имеет конструктор?
15. Зачем необходимо перегружать операции?

## **Задание для лабораторной работы 1**

В соответствии с вариантом задания самостоятельно разработать класс и программу, иллюстрирующую его возможности. Требования к классу:

- обязательно наличие закрытой (`private`) и общедоступной (`public`) частей;
- класс должен иметь *по крайней мере два* конструктора, определенных программистом: конструктор по умолчанию и конструктор с параметрами;
- необходимо задать набор свойств для получения значений и модификации полей данных, находящихся в закрытой части класса;

– для разработанного класса должна быть перегружена одна операция: арифметическая, сравнения, присваивания. Выбор перегружаемых операций определяется семантикой предметной области.

Для последнего варианта приведен рекомендуемый перечень операций над объектами класса. Для остальных вариантов сделать по аналогии.

### **Варианты заданий**

- 1) общественная организация,
- 2) обитатель моря,
- 3) птица,
- 4) изделие,
- 5) организация,
- 6) печатное издание,
- 7) испытание (как обобщенный экзамен, зачет),
- 8) место (как административная единица – город, село),
- 9) товар,
- 10) документ,
- 11) транспортное средство,
- 12) двигатель,
- 13) государство,
- 14) животное,
- 15) корабль,
- 16) автотранспорт,
- 17) спецмашина,
- 18) судно,
- 19) автомобиль,
- 20) учебное заведение,
- 21) магазин,
- 22) персона (в отделе кадров).

Класс персона. Поля: ФИО, год рождения, год поступления на работу или учебу, базовый размер з/п или стипендии, размер надбавки (за отличную учебу или заведование кафедрой). Действия над объектами класса: начисление стипендии (зарплаты), вывод личных данных, операции сравнения объектов, присваивание.

## Лабораторная работа 2. Семейства классов и программирование полиморфных методов (наследование, абстрактные классы, виртуальные методы)

*Цель работы:* познакомиться с созданием семейств связанных классов и программированием полиморфных методов при объектно ориентированном подходе при использовании языка C#.

### ***Необходимые теоретические сведения***

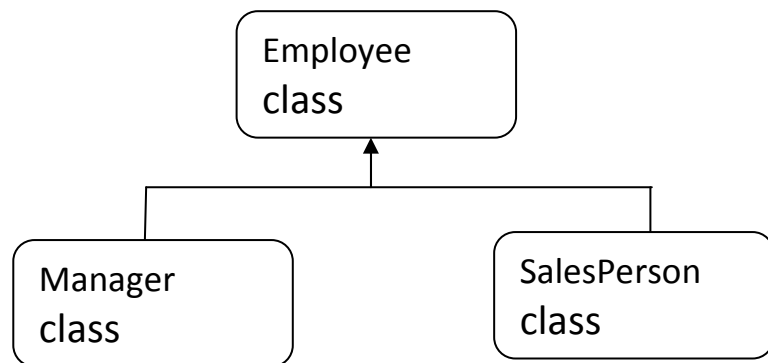
#### **Наследование**

Наследование – это тот аспект ООП, который способствует многократному использованию кода. Основная идея наследования состоит в том, что новые классы могут использовать и/или расширять функциональность других классов. При этом поддерживается концепция иерархической классификации, имеющей направление сверху вниз. Используя наследование, объект должен определить только те качества, которые делают его уникальным в пределах своего класса.

Синтаксис:

```
class имя_класса : имя_родительского_класса  
{тело_класса}
```

Проиллюстрируем это положение на примере класса сотрудника Employee: используем его функциональность для создания двух новых классов – класса менеджера Manager и класса продавца SalesPerson. В этом случае иерархия классов будет выглядеть так:



Пусть класс Manager расширяет класс сотрудника Employee, добавляя запись о количестве акций, которыми владеет конкретный менеджер, а класс SalesPerson содержит поле со значением числа продаж.

```
public class Manager : Employee
{
    private ulong numberOfOptions; // число акций
    public Manager(){}
    // здесь вызывается конструктор базового класса
    public Manager(string FullName, int emplID, float currPay, ulong
numbOfOpts)
        : base(FullName, emplID, currPay)
    {
        numberOfOptions = numbOfOpts;
    }
    public ulong NumbOpts // свойство для числа акций
    {
        get {return numberOfOptions;}
        set { numberOfOptions = value;} // здесь д.б. проверка на
// корректность
    }
}
public class SalesPerson : Employee
{
    protected int numberOfSales; // число продаж
```

```

    public SalesPerson(){}
    // здесь вызывается конструктор базового класса
    public SalesPerson(string FullName, int emplID, float currPay, int
numbOfSales)
        : base(FullName, emplID, currPay)
    {
        numberOfSales = numbOfSales; }
    public int NumbSales
    {
        get {return numberOfSales;}
        set { numberOfSales = value;}
    }
}

```

Чтобы поля базового класса были непосредственно доступны его наследникам, необходимо изменить модификатор доступа полей базового класса на *protected*.

С помощью наследования можно не только создавать иерархию классов (отношение «являться»), но и построить еще одну структуру – вложенные объекты (тогда, когда один объект является частью другого – отношение «часть – целое»).

## Полиморфизм

Полиморфизм – одна из основных составляющих объектно ориентированного программирования, позволяющая определять в базовом классе методы, которые будут общими для всех классов-наследников, при этом класс-наследник может определять специфическую реализацию некоторых или всех этих методов. Основным инструментом для реализации принципа полиморфизма является использование виртуальных методов и абстрактных классов.

## Виртуальные методы

Метод, при определении которого в базовом классе было указано ключевое слово *virtual*, *может быть* переопределен (или перекрыт) в классах-наследниках. Например, в классе *Employee* сделаем метод *GiveBonus()* виртуальным:

```

public class Employee
{

```



```

...
public virtual void GiveBonus(float amount)
    {currPay += amount;}
...
}

```

Теперь классы Manager и SalesPerson могут иметь собственную версию этого метода. В идеале в премии продавца должен учитываться объём продаж, а менеджеры получают дополнительные акции кроме увеличения зарплаты. Для переопределения виртуального метода используется ключевое слово `override`.

```

public class Manager : Employee
{
...
public override void GiveBonus(float amount)
{
    base.GiveBonus(amount); // увеличение зарплаты
    // дополнительные акции
    Random r = new Random();
    numberOfOptions += (ulong)r.Next(500);
}
...
}
public class SalesPerson : Employee
{
...
public override void GiveBonus(float amount)
{
    int salesBonus = 0;
    if(numberOfSales >= 0 && numberOfSales <= 100)
        salesBonus = 10;
    else salesBonus = 15;
    base.GiveBonus(amount * salesBonus);
}
...
}

```

Для экземпляров разных классов будет вызываться при выполнении программы своя версия виртуального метода.

Переопределять виртуальный метод не обязательно. Если класс-наследник не предоставляет собственную версию виртуального метода, то используется метод базового класса.

## Абстрактные классы

В семействе классов базовый класс часто является обобщенным (в нашем примере таков класс сотрудника Employee). Его задача состоит в определении общих полей, свойств и методов для классов-наследников, а создание экземпляров обобщенного класса не имеет смысла. В этом случае в C# используют ключевое слово *abstract*.

Назначение абстрактного класса заключается в предоставлении общего определения для базового класса, которое могут совместно использовать несколько производных классов. Создавать экземпляры абстрактного класса нельзя. Абстрактные классы могут определять абстрактные методы. Для этого перед типом возвращаемого значения метода необходимо поместить ключевое слово *abstract*. Абстрактные методы не имеют реализации, поэтому определение такого метода заканчивается точкой с запятой вместо обычного блока метода. Классы, производные от абстрактного класса, должны реализовывать все абстрактные методы. Если абстрактный класс наследует виртуальный метод из базового класса, абстрактный класс может переопределить виртуальный метод с помощью абстрактного метода.

Приведем пример. Сделаем класс Employee абстрактным и определим в нём абстрактный метод DoWork(). Производные классы Manager и SalesPerson обязаны явно реализовать метод DoWork().

```
abstract public class Employee
{
    ...
    public abstract void DoWork();
    ...
}
public class Manager : Employee
```

```

{
...
// класс-наследник содержит явную реализацию метода
public override void DoWork()
{ Console.WriteLine("Менеджер работает"); }
...
}
public class SalesPerson : Employee
{
...
// класс-наследник содержит явную реализацию метода
public override void DoWork()
{ Console.WriteLine("Продавец работает"); }
...
}

```

## ***Контрольные вопросы***

1. Что понимается под термином «наследование»?
2. Что общего имеет дочерний класс с родительским?
3. В чем состоит различие между дочерним и родительским классами?
4. Приведите синтаксис описания наследования классов в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
5. Что понимается под термином «полиморфизм»?
6. Какие механизмы используются в языке C# для реализации концепции полиморфизма?
7. С какой целью используются виртуальные методы?
8. В чем состоит особенность виртуальных методов в производных (дочерних) классах?
9. Какие условия определяют выбор версии виртуального метода?
10. Какое ключевое слово (модификатор) языка C# используется для определения виртуального метода в базовом (родительском) классе? А в производном (дочернем) классе?

11. В какой момент трансляции программы осуществляется выбор вызываемого переопределенного метода?

12. Приведите синтаксис виртуального метода в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

13. Что понимается под термином «абстрактный класс»?

14. В чем заключаются особенности абстрактных классов?

15. Являются ли абстрактные методы виртуальными?

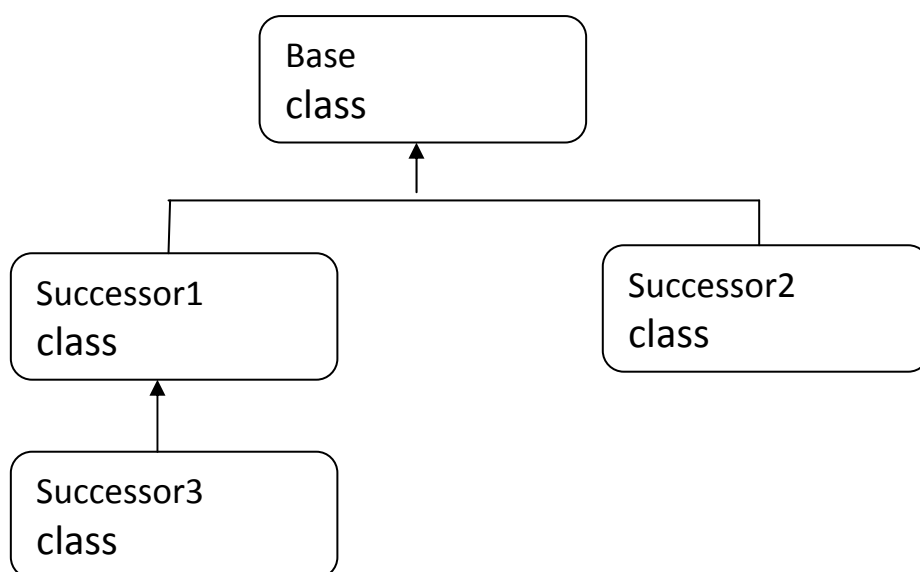
16. Возможно ли создание иерархии классов посредством абстрактного класса?

17. Возможно ли создание объектов абстрактного класса?

18. Приведите синтаксис абстрактного класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

## Задание для лабораторной работы 2

Построить иерархию классов в соответствии с вариантом задания. Схематичное изображение иерархии:



В качестве базового абстрактного класса взять класс, разработанный Вами в лабораторной работе 1. В базовый класс добавить абстрактный метод и реализовать его в производных классах. Другие методы базового класса сделать виртуальными и переопределить их в классах-наследниках. Производные классы должны иметь собственные поля данных, отличные от полей базового класса.

Для разработанной Вами иерархии классов в методе Main:

- 1) описать массив объектов базового класса;
- 2) занести в этот массив из файла (!) объекты дочерних классов;
- 3) продемонстрировать работу методов класса у всех элементов этого массива.

### **Варианты заданий**

- 1) общественная организация, партия, клуб, объединение,
- 2) обитатель моря, морские млекопитающие, дельфины, рыбы,
- 3) птицы, хищные птицы, орлы, воробьи,
- 4) деталь, механизм, изделие, узел,
- 5) организация, страховая компания, нефтегазовая компания, завод,
- 6) печатное издание, журнал, книга, учебник,
- 7) тест, экзамен, выпускной экзамен, испытание,
- 8) место, село, город, мегаполис,
- 9) товар, канцтовары, ручки, диски,
- 10) квитанция, накладная, счет, документ,
- 11) автомобиль, поезд, транспортное средство, экспресс,
- 12) двигатель, двигатель внутреннего сгорания, дизель, реактивный двигатель,
- 13) республика, монархия, конституционная монархия, государство,
- 14) млекопитающее, парнокопытное, пресмыкающееся, животное,
- 15) корабль, пароход, парусник, корвет,
- 16) автотранспорт, автобус, микроавтобус, грузовик,
- 17) спецмашина, пожарная машина, пожарная автолестница, скорая помощь,
- 18) судно, баржа, скоростной корабль, метеор,
- 19) автомобиль, спортивный автомобиль (болид), легковой, кабриолет,
- 20) учебное заведение, школа, вуз, академия,
- 21) магазин, киоск, минимаркет, супермаркет,
- 22) студент, преподаватель, персона, заведующий кафедрой.

## Лабораторная работа 3.

### Обработка исключительных ситуаций

*Цель работы:* разобраться в том, как путём структурной обработки исключений бороться с непредвиденными ситуациями, возникающими во время выполнения приложений на С#.

### **Необходимые теоретические сведения**

#### **Исключение**

Исключение – это отклонение от нормального выполнения программы, любое ошибочное условие или непредвиденное поведение, с которым сталкивается программа в процессе выполнения. Исключения могут возникать вследствие сбоя в вашем коде или в вызванном коде (таком, как общая библиотека), недоступности ресурсов операционной системы, неожиданных условий, с которыми сталкивается общезыковая среда выполнения (такими, как код, который не может быть проверен), и т. д. При возникновении некоторых из этих условий приложение пользователя может выполнить восстановление самостоятельно, однако это возможно не всегда. Восстановление возможно для большинства случаев исключений приложений, но оно невозможно для подавляющей части исключений среды выполнения.

В платформе .NET Framework исключение – это объект типа, производного от System.Exception. Исключение посылается из области кода, где возникла проблема. Исключение передается в стек до тех пор, пока его не обработает приложение или не завершится выполнение программы.

#### **Обработка исключений**

Функции обработки исключений на языке С# помогают обрабатывать любые непредвиденные или исключительные ситуации, происходящие при выполнении программы. При этом следует руководствоваться следующими правилами.

Блок кода, начинающийся с ключевого слова try, используется для заключения в него инструкций, которые могут выдать исключения. При возникновении исключения в блоке try поток

управления немедленно переходит к первому соответствующему обработчику исключений, присутствующему в стеке вызовов. В языке C# ключевое слово `catch` используется для определения обработчика исключений. В блоке `catch` происходит обработка ошибки.

Код в блоке `finally` выполняется даже при возникновении исключения. Блок `finally` используется для освобождения ресурсов, например для закрытия потоков или файлов, открытых в блоке `try`, поскольку код, следующий после конструкции `try/catch`, не будет выполнен при возникновении исключения.

Блок `try` следует использовать с блоком `catch` либо `finally`; в него может входить несколько блоков `catch`. Оператор `try` без `catch` или `finally` вызовет ошибку компилятора. Общая схема выглядит следующим образом:

```
try
{
    // код, который может вызвать исключение
}
catch (SomeSpecificException ex)
{
    // код, обрабатывающий ошибку
}
finally
{
    // код, который будет выполнен после блока try/catch
}
```

Блок `catch` указывает тип перехватываемого исключения. Этот тип называется фильтром исключений, он должен иметь тип `Exception` либо быть его производным. В программе может быть несколько блоков `catch` с различными фильтрами исключений. Первыми должны быть размещены блоки `catch` с самыми конкретными производными классами исключений, последним – блок `catch` без фильтра. Для каждого вызванного исключения выполняется первый блок `catch`, указывающий точный тип или базовый класс созданного исключения. Если блок `catch`, указывающий соответствующий фильтр исключения, отсутствует, будет выполнен блок `catch` без фильтра (если таковой имеется).

Если обработчик для определенного исключения не существует, выполнение программы завершается с сообщением об ошибке.

Перехват исключений возможен при выполнении следующих условий.

- Понимание причины возникновения исключения и реализация особого восстановления, например перехват объекта `FileNotFoundException` и вывод запроса на ввод нового имени файла.

- Возможность создания и вызова нового, более конкретного исключения. Например:

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch(System.IndexOutOfRangeException e)
    {
        throw new System.ArgumentOutOfRangeException("Parameter
index is out of range.");
    }
}
```

- Частичная обработка исключения. Например, блок `catch` можно использовать для добавления записи в журнал ошибок, но затем нужно повторно вызвать исключение, чтобы выполнить его последующую обработку.

```
try
{
    // попытка доступа к ресурсу
}
catch (System.UnauthorizedAccessException e)
{
    LogError(e); // запись в журнал
    throw e; // повторное возбуждение исключения
}
```



Исключения могут явно генерироваться программой с помощью ключевого слова `throw`. Объекты исключения содержат подробные сведения об ошибке, такие как состояние стека вызовов и текстовое описание ошибки.

В примере ниже метод тестирует деление на ноль и выполняет перехват соответствующей ошибки. Без обработки исключений эта программа была бы завершена с ошибкой `DivideByZeroException was unhandled` (не обработано исключение «деление на ноль»).

```
class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new System.DivideByZeroException();
        return x / y;
    }
    static void Main()
    {
        double a = 98, b = 0; // данные введены для тестирования
                             // исключений
        double result = 0;
        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Инструкция `finally` предназначена для обеспечения немедленного выполнения необходимой очистки объектов, обычно занимающих внешние ресурсы, даже в случае, когда генери-

руется исключение. Примером подобной очистки является вызов метода `Close` для объекта класса `FileStream` сразу после его использования, не дожидаясь, когда этот объект будет уничтожен сборщиком мусора среды CLR, как показано ниже:

```
static void CodeWithCleanup()
{
    System.IO.FileStream file = null;
    System.IO.FileInfo fileInfo = null;
    try
    {
        fileInfo = new System.IO.FileInfo("C:\\file.txt");
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch(System.UnauthorizedAccessException e)
    {
        System.Console.WriteLine(e.Message);
    }
    finally
    {
        if (file != null)
        {
            file.Close();
        }
    }
}
```

Так как исключение может произойти в любой момент внутри блока `try` до вызова метода `OpenWrite()` или ошибкой может завершиться выполнение самого метода `OpenWrite()`, при выполнении попытки закрыть файл нет гарантии, что он открыт. Блок `finally` добавляет проверку, чтобы убедиться, что объект класса `FileStream` не имеет значения `null`, прежде чем вызывать метод `Close`.

## Пользовательские исключения

Для того чтобы сигнализировать об ошибке времени выполнения, можно создавать собственные исключения, специфичные для вашего приложения. Для этого нужно создать пользовательский класс исключения, производный от класса `System.Exception`. Шаблон подобного класса приведён ниже.

```
[global::System.Serializable]
public class MyException : Exception
{
    public MyException() { }
    public MyException(string message) : base(message) { }
    public MyException(string message, Exception inner) :
base(message, inner) { }
    protected MyException(
        System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context)
        : base(info, context) { }
}
```

Для генерации пользовательского исключения необходимо создать экземпляр нашего класса исключения и воспользоваться оператором `throw`.

```
...
MyException ex = new MyException("Здесь описание
ошибки");
// здесь служба поддержки
ex.HelpLink = "http://www.adres.ru";
throw ex;
...
```

## **Контрольные вопросы**

1. Что такое исключение?
2. Какие операторы заключаются в блок try?
3. Сколько блоков catch может быть расположено подряд? В чем их назначение?
4. В чем состоит значение механизма исключений в языке C#?
5. Какие операторы языка C# используются для обработки исключений? Приведите синтаксис блока try...catch в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
6. Что происходит в случае неудачного перехвата исключения?
7. В каком случае возможно использование оператора языка C# catch без параметров?
8. Каким образом осуществляется возврат в программу после обработки исключительной ситуации?
9. Приведите синтаксис блока finally (в составе оператора try...catch) в общем виде. Проиллюстрируйте его фрагментом программы на языке C#. Объясните применение ключевого слова finally.
10. С помощью какого ключевого слова происходит генерация исключений?
11. С какой целью создаются пользовательские исключения? Приведите фрагмент кода с описанием класса исключения.

## **Задания для лабораторной работы 3**

1. В код лабораторной работы 2 добавить обработку исключений «файл не найден», «нет прав доступа к файлу».
2. Создать пользовательское исключение согласно Вашей предметной области. Написать метод, генерирующий пользовательское исключение и обработать его.

## Лабораторная работа 4. Интерфейсы, коллекции и обобщенные коллекции

*Цель работы:* познакомиться с программированием на базе интерфейсов, в том числе с получением ссылок на интерфейсы, их явной реализацией, созданием иерархий интерфейсов и некоторыми предопределенными интерфейсами. Исследовать классы коллекций и обобщенных коллекций.

### **Необходимые теоретические сведения**

#### **Интерфейс**

Интерфейс – это набор семантически связанных абстрактных членов. Количество членов, определенных в конкретном интерфейсе, зависит от того, какое поведение мы пытаемся смоделировать при помощи этого интерфейса. С точки зрения синтаксиса интерфейсы в С# определяются следующим образом:

// Этот интерфейс определяет возможности работы с  
// вершинами геометрической фигуры

```
public interface IPointy
{
    byte GetNumberOfPoints(); //автоматически этот член
    // интерфейса становится абстрактным
}
```

Интерфейсы .NET также могут поддерживать любое количество свойств (и событий). Например, интерфейс *IPointy* может вместо метода содержать такое свойство для чтения:

```
public interface IPointy
{
    // Чтобы сделать это свойство свойством "только для чтения"
    // или "только для записи", достаточно просто удалить
    // соответствующий блок set или get
    byte Points { get; }
}
```

В любом случае интерфейс – это не более чем именованный набор абстрактных членов, а это значит, что любой класс,

реализующий этот интерфейс, должен самостоятельно полностью определять каждый из членов этого интерфейса. Таким образом, интерфейсы – это еще один способ реализации полиморфизма в приложении: поскольку в разных классах члены одних и тех же интерфейсов будут реализованы по-разному, в результате эти классы будут реагировать на одни и те же вызовы по-своему.

В качестве примера мы будем использовать иерархию геометрических фигур, производных от базового класса `Shape`. Наш интерфейс `IPointy` – это элементарный интерфейс, возвращающий количество углов у геометрической фигуры.

Как мы уже выяснили, интерфейс – это набор абстрактных членов. Однако `C#` позволяет нам использовать абстрактные члены и в обычном классе. Так зачем же вообще нужны интерфейсы, если мы можем создать набор абстрактных методов, реализовав его как класс, и сделать этот класс базовым для наших пользовательских классов?

Ответ будет прост: класс – это класс, а интерфейс – это интерфейс. В классах помимо абстрактных методов, свойств и событий определяются также переменные класса и обычные (не абстрактные) методы, появление которых в интерфейсе исключено.

Интерфейс – это чистая синтаксическая конструкция, которая предназначена только для определенных целей. Интерфейсы никогда не являются типами данных, и в них не бывает реализаций методов по умолчанию. Каждый член интерфейса (будь то свойство или метод) автоматически становится абстрактным. Кроме того, в `C#` наследование одного класса более чем от одного базового класса (то есть множественное наследование) запрещено. В то же время реализация в классе сразу нескольких интерфейсов – это обычное дело.

## **Реализация интерфейса**

Когда в `C#` какой-либо класс должен реализовать нужные нам интерфейсы, названия этих интерфейсов должны быть помещены (через запятую) после двоеточия, следующего за именем этого класса. Обратите внимание, что имя базового класса всегда должно стоять перед именами любых интерфейсов:

// Любой класс может реализовывать любое количество  
// интерфейсов, но он имеет только один базовый класс:

```
public class Hexagon : Shape, IPointy
{
    public Hexagon() { }
    public byte Points
    {
        get { return 6; }
    }
}

public class Triangle : Shape, IPointy
{
    public Triangle() { }
    public byte Points
    {
        get {return 3;}
    }
}
```

Теперь и Hexagon, и Triangle, когда их об этом попросят, предоставят информацию о том, сколько у них углов. Обратите внимание, что вы не можете выбирать, какие методы в интерфейсе вам реализовывать, а какие – нет. В классе либо должны быть реализованы все методы данного интерфейса, либо этот интерфейс вообще не реализуется – половинчатого решения быть не может.

## **Получение ссылки на интерфейс**

C# позволяет обращаться к членам интерфейсов несколькими способами. Предположим, что мы создаем объект одного из наших классов и нам необходимо узнать, сколько у него углов.

Первый способ – воспользоваться явным приведением типов:

```
// Получаем ссылку на интерфейс IPointy, используя явное  
// приведение типов
Hexagon hex = new Hexagon("Bill");
IPointy itfPt = (IPointy)hex;
Console.WriteLine(itfPt.Points);
```

Здесь мы получаем ссылку на интерфейс IPointy, явно приводя объект класса Hexagon к типу IPointy. Если класс Hexagon поддерживает интерфейс IPointy, мы получим ссылку на интерфейс (у нас она называется itfPt) и все будет хорошо. Однако, когда мы пытаемся получить ссылку на интерфейс путем явного приведения типов для объекта класса, не поддерживающего данный интерфейс, система генерирует исключение InvalidCastException. Чтобы избежать проблем с исключением, исключение нужно перехватить:

```
// Используя программные средства, поэтапно  
// перехватываем исключение
```

```
Circle c = new Circle("Lisa");  
IPointy itfPt;  
try {  
    itfPt = (IPointy)c;  
    Console.WriteLine(itfPt.Points());  
}  
catch (InvalidCastException e)  
{ Console.WriteLine("OPS! Not pointy..."); }
```

Второй способ получить ссылку на интерфейс – использовать ключевое слово as:

```
// Еще один способ получить ссылку на интерфейс  
Hexagon hex2 = new Hexagon("Peter");  
IPointy itfPt2;  
itfPt2 = hex2 as IPointy;  
if ( itfPt2 != null )  
    Console.WriteLine(itfPt2.Points());  
else  
    Console.WriteLine("OOPS! Not pointy...");
```

Если при использовании ключевого слова as мы попробуем создать ссылку на интерфейс через объект, который этот интерфейс не поддерживает, ссылка будет просто установлена в null, и при этом никаких исключений генерироваться не будет.

Третий способ получения ссылки на интерфейс – воспользоваться оператором is. Если объект не поддерживает интерфейс, условие станет равно false:



```
// Есть ли у тебя углы?
Triangle t = new Triangle();
if( t is IPointy)
Console.WriteLine( t.Points());
else
Console.WriteLine("OOPS! Not pointy...");
```

Если у нас имеется массив разных объектов и нам необходимо будет выяснить в процессе выполнения, какие именно объекты из этого массива поддерживают определенный интерфейс, это можно сделать любым из приведенных выше способов.

## **Создание иерархий интерфейсов**

В С# один интерфейс может наследовать другому. Как обычно, базовый интерфейс определяет общее поведение, в то время как производный интерфейс – более конкретное и специфическое. Простая иерархия интерфейсов может выглядеть следующим образом:

```
// Базовый интерфейс
interface IDrawable
{
void Draw();
}
interface IPrintable : IDrawable
{
void Print();
}
interface IMetaFileRender : IPrintable
{
void Render();
}
```

Если наш класс должен поддерживать поведение, определенное во всех трех интерфейсах, то базовым для него должен быть интерфейс самого нижнего уровня (в нашем случае – IMetaFileRender). Все методы, определенные в интерфейсах

более высокого уровня, будут автоматически включены в производные интерфейсы.

## **Наследование от нескольких базовых интерфейсов**

C# допускает наследование сразу от нескольких базовых интерфейсов. При этом еще раз заметим, что множественное наследование между классами (когда один класс является производным одновременно от нескольких классов) в C# запрещено – множественное наследование разрешается только для интерфейсов. Проиллюстрируем возможности множественного наследования интерфейсов на примере.

Предположим, что в нашем распоряжении есть набор интерфейсов, моделирующих поведение автомобиля:

```
interface ICar
{
    void Drive(); // ехать
}
interface IUnderwaterCar
{
    void Dive(); } // нырять
// Этот интерфейс имеет ДВА базовых.
interface IJamesBondCar : ICar, IUnderwaterCar
{
    void TurboBoost(); // разгоняться
}
```

Если мы захотим создать класс, который реализует интерфейс IJamesBondCar, нам придется реализовать в нем все методы каждого из интерфейсов.

```
public class JamesBondCar : IJamesBondCar
{
    public void Drive() { Console.WriteLine("Speeding up..."); }
    public void Dive() { Console.WriteLine("Submerging..."); }
    public void TurboBoost() { Console.WriteLine("Blast off!"); }
}
```

## Стандартные интерфейсы

В библиотеку базовых классов .Net встроено большое количество стандартных интерфейсов. Для создания сравниваемых объектов используется интерфейс `Comparable`. Если мы реализуем этот стандартный интерфейс для своего класса, то сможем воспользоваться встроенным методом сортировки для массива объектов своего класса.

Формальное определение интерфейса выглядит следующим образом:

```
// Этот интерфейс позволяет определять место объекта  
// среди других аналогичных объектов
```

```
interface Comparable  
{  
    int CompareTo(object o);  
}
```

Поскольку этот интерфейс состоит из единственного метода `CompareTo()`, вся суть заключается в том, как будет реализован этот метод. Прежде всего мы должны определить, по значению какой внутренней переменной будет производиться сортировка. Для типа `Employee`, описанного в первой лабораторной работе, самая подходящая переменная – это табельный номер.

```
// Такая реализация метода CompareTo() позволит  
// сортировать объекты работников по значению табельного  
// номера – empID
```

```
public class Employee: Comparable  
{  
    private string fullName; // Имя сотрудника  
    private int empID; // табельный номер  
    ...  
    // Реализация Comparable  
    int Comparable. CompareTo (object o)  
    {  
        Employee temp = (Employee)o;  
        if( this.empID > temp. empID)  
            return 1;
```

```

if(( this.empID < temp. empID )
return -1;
else   return 0;
}
...
}

```

Как видно из этого кода, метод CompareTo() сравнивает значение empID для текущего объекта (того, для которого вызван этот метод) со значением empID для принимаемого объекта (того объекта, который передан этому методу в качестве входящего параметра). В зависимости от результатов сравнения выдается одно из трех возможных значений. Заметим, что может возвращаться любое число больше нуля, если значение переменной для текущего объекта больше, чем у принимаемого.

Теперь мы можем спокойно производить сортировку массива объектов.

```

static void Main()
{
Employee[] otdel = new Employee[4];
otdel[0] = new Manager("Chuckiy H.", 1020, 19000.Of, 34);
otdel[1] = new Manager("Petrov Petr A.", 1002, 14000.Of, 40);
otdel[2] = new SalesPerson("Ivanov A.", 2101, 16500.Of, 2400);
otdel[3] = new SalesPerson("Elin Al.", 2002, 13300.Of, 3600);
// используем возможности IComparable
Array.Sort(otdel);
...
}

```

Кроме IComparable, на практике наиболее часто используются стандартные интерфейсы IEnumerable – для создания перечислимых объектов, ICloneable – для глубокого копирования объектов. Найти подробную информацию об их реализации вы можете на сайте MSDN или в книге [1].

## Коллекции

Платформа .NET Framework предоставляет кроме массивов еще и специализированные классы для хранения и извлечения данных. Эти классы обеспечивают поддержку стеков, очередей, списков и хэш-таблиц. Большинство классов коллекций реализуют одинаковые интерфейсы, и эти интерфейсы могут наследоваться для создания новых классов коллекций, соответствующим более специализированным потребностям в хранении данных. Классы коллекций заданы как часть пространства имен System.Collections. Большинство классов коллекций являются производными от интерфейсов ICollection, IComparer, IEnumerable, IList, IDictionary и IDictionaryEnumerator.

Наиболее часто на практике используются следующие классы коллекций:

ArrayList	Динамически изменяющий свой размер массив объектов.
Hashtable	Представляет набор взаимосвязанных ключей и значений, основанных на хэш-коде ключа.
Queue	Стандартная очередь, реализованная по принципу FIFO (first-in-first-out, «первым пришел, первым ушел»).
Sorted List	Похож на словарь, однако к элементам можно также обратиться по их порядковому номеру (индексу).
Stack	Стек, реализованный по принципу LIFO (last-in-first-out, «последним пришел, первым ушел»), обеспечивающий возможности по проталкиванию данных в стек, выталкиванию данных из стека и считыванию данных.

Динамический массив ArrayList очень удобен в применении, так как позволяет включать произвольное число элементов любого типа и хранит все элементы как объекты наиболее общего типа System.Object. Но при этом ArrayList не обеспечивает типовую безопасность. Если по ошибке в коллекцию элементов одного типа мы добавим экземпляр другого, то ошибка будет обнаружена только во время выполнения, при попытке вызвать несуществующий метод. Выходом в этом случае является использование обобщенных коллекций.

## Обобщенные коллекции

Обобщенные классы коллекций обеспечивают повышенную безопасность типа. Их описание содержится в пространстве имен `System.Collections.Generic`. Это пространство имен содержит множество классов и интерфейсов, позволяющих помещать элементы в различные контейнеры, а именно:

<code>List&lt;T&gt;</code>	Список элементов типа <code>T</code> с динамически изменяемым размером.
<code>Dictionary &lt;K,V&gt;</code>	Обобщенная коллекция пар имя/значение.
<code>Queue&lt;T&gt;</code>	Обобщенная очередь элементов типа <code>T</code> .
<code>SortedDictionary &lt;K,V&gt;</code>	Обобщенная реализация сортированного множества пар имя/значение.
<code>Stack&lt;T&gt;</code>	Обобщенный стек элементов типа <code>T</code> .
<code>LinkedList&lt;T&gt;</code>	Обобщенная реализация двусвязного списка.

Обобщенные классы являются объектами, память для которых выделяется в куче, поэтому они должны создаваться с помощью оператора `new` с необходимыми аргументами конструктора. Кроме того, для них нужно указать тип (типы), который будет подставлен в параметр (параметры) обобщенного класса. Поэтому если мы хотим создать список целочисленных значений и список объектов класса `Employee`, то соответствующий код будет выглядеть так:

```
static void Main(string[] args)
{
    // создаем список, содержащий целочисленные значения
    List<int> myInts = new List<int>();
    // помещаем в список элементы
    myInts.Add(5);
    int x = 10;
    myInts.Add(x);
    // создаем список, содержащий объекты Employee
    List<Employee> otdel = new List<Employee>();
    otdel.Add( new Manager());
    otdel.Add( new SalesPerson());
    Console.WriteLine("В отделе работает {0} людей", otdel.count);
}
```

Мы можем создавать собственные обобщенные методы, структуры, классы, интерфейсы. Ключевое слово `where`, стоящее после описания обобщенного метода (класса, интерфейса), задает ограничение на параметр типа `T`. `T` может быть только значимым типом или иметь в качестве базового какой-то определенный класс. Подробнее о возможных значениях для ограничений см. [1]. Приведем несколько примеров.

// Этот метод осуществляет обмен значений двух значимых  
// типов

```
public static void Swap<T>(ref T a, ref T b) where T : struct
{
    Console.WriteLine("You sent the Swap() method a {0}",
        typeof(T));
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

Ниже приведен пример обобщенной структуры.

```
public struct Point<T>
{
    // обобщенные поля данных
    private T xPos;
    private T yPos;
    // 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()
{
    return string.Format("[{0}, {1}]", xPos, yPos);
}
}

```

Наконец, рассмотрим пример обобщенного интерфейса и класса, который этот интерфейс реализует.

```

public interface IBinaryOperations<T>
{
    T Add(T arg1, T arg2);
    T Subtract(T arg1, T arg2);
    T Multiply(T arg1, T arg2);
    T Divide(T arg1, T arg2);
}
public class BasicMath : IBinaryOperations<int>
{
    public BasicMath() {}
    // IBinaryOperations<int> Members

    public int Add(int arg1, int arg2)
    { return arg1 + arg2; }
    public int Subtract(int arg1, int arg2)
    { return arg1 - arg2; }
    public int Multiply(int arg1, int arg2)
    { return arg1 * arg2; }
    public int Divide(int arg1, int arg2)
    { return arg1 / arg2; }
}

```



## **Контрольные вопросы**

1. Дайте определение интерфейса. Для каких целей применяются интерфейсы?
2. Как можно получить ссылку на интерфейс? Приведите примеры.
3. В чем особенность иерархий интерфейсов?
4. Какие стандартные интерфейсы вы знаете? Приведите примеры их реализации.
5. Объясните сходства и различия классов коллекций и обобщенных коллекций.
6. Приведите примеры обобщенных структур данных.
7. Чем отличается синтаксис интерфейса от синтаксиса абстрактного класса?
8. Какие объекты языка C# могут быть членами интерфейсов?
9. Каким количеством классов может быть реализован интерфейс?
10. Может ли класс реализовывать множественные интерфейсы?
11. Какой модификатор доступа соответствует интерфейсу?
12. Приведите синтаксис интерфейса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
13. Возможно ли наследование интерфейсов?
14. Насколько синтаксис наследования интерфейсов отличается от синтаксиса наследования классов?

## **Задания для лабораторной работы 4**

1. Для разработанной Вами в лабораторной работе 1 иерархии классов добавить реализацию интерфейса `IComparable` и отсортировать массив объектов базового класса.
2. Из объектов Вашей иерархии классов создать коллекцию (`ArrayList`, очередь, стек или список – по вариантам) и продемонстрировать методы работы с этой коллекцией (вставка элементов, удаление, поиск и т. д.).
3. Создайте обобщенный класс списка, двусвязного списка и т. п. – по вариантам (`SortedDictionary<K,V>`, `List<T>`, `LinkedList<T>`, `Stack<T>`, `Queue<T>`), который сможет работать только с

объектами, имеющими в качестве базового класс, разработанный Вами в лабораторной работе 1. В обобщенном классе должны быть следующие методы: печати всего списка, добавления/исключения элементов, проверки существования элемента в списке, очистки списка, сортировки (если это возможно).

## **Лабораторная работа 5**

### **Знакомство с .Net сборками**

*Цель работы:* исследовать ключевые детали создания, развертывания и настройки .Net сборок.

### ***Необходимые теоретические сведения***

#### **Сборки. Утилита ildasm.exe**

Приложение всегда состоит из одной или более сборок. Сборка – это функциональная единица, имеющая версию и описание, которую можно использовать в других приложениях. В самом простом случае, сборка – это приложение, которое состоит из одного функционального модуля – класса. В большинстве случаев в приложении может быть несколько сборок, и каждая может иметь свои вспомогательные файлы. При выполнении все сборки приложения должны существовать и к ним должен быть открыт доступ. Каждая сборка должна быть независимой. У любой сборки есть метаданные, которые описывают сборку и содержат версию. Содержимое сборки можно просмотреть, запустив дизассемблер «Microsoft Intermediate Language Disassembler» (ildasm.exe).

#### **Частные сборки**

Программы, которые написаны на языках, поддерживаемых библиотекой .NET Framework, и на C# в частности, компилируются в код MSIL, которые затем среда CLR (Common Language Runtime) преобразует в машинный код.

Частные сборки отвечают тем же требованиям, что и обычные, но предназначены они для использования только в одном

приложении и, как правило, лежат в той же папке, что и использующее их приложение. Частная сборка может находиться не только в корневой папке, но и во вложенных папках корневого каталога. Приложение ссылается на частную сборку по ее частному имени, которое содержится в метаданных. Когда среда CLR ищет частную сборку для применения, она использует частное имя сборки для поиска, причем поиск осуществляется в корневом каталоге и дочерних подкаталогах.

## **Сборки со строгим именем**

Обычные сборки могут быть без труда декомпилированы, и код в них может быть повторно использован. Для коммерческих приложений это недопустимо. Сборки, подписанные строгим именем, позволяют обеспечить безопасность, защиту кода, облегчить применение их в нескольких приложениях, а также управлять версионностью сборок. Строгое имя является уникальным обозначением сборки. Оно гарантирует невозможность замены вашей сборки другой.

Строгое имя сборки включает в себя частное имя сборки, ее версию, открытый ключ для клиентского приложения и цифровую подпись безопасности. Если сборка была локализована, то в нее также войдет описание культуры локализации.

При компиляции сборки данные о ней помещаются в зашифрованный файл, ключ которого и представляет собой цифровую подпись. Открытый ключ хранится в этом же файле, и с его помощью клиентское приложение расшифровывает цифровую подпись. Строгое имя сборки гарантирует ее уникальность и защиту от декомпиляции.

Сборка со строгим именем должна ссылаться только на сборку со строгим именем. Когда мы используем сборку, подписанную строгим именем, мы надеемся получить все преимущества подписанных сборок. Если же подписанная сборка ссылается на частную, то в первую очередь под угрозу ставится конфиденциальность приложения, кроме того, возможен конфликт версий частных сборок. В обычной жизни это похоже на использование банковского хранилища с бронированными стенами и дверями, но слабыми окнами.

Сборка со строгим именем может располагаться в любых местах – корневой папке приложения, произвольной папке локального или удаленного компьютеров, в Интернете.

Одни и те же сборки могут быть использованы в нескольких приложениях. Можно не дублировать эти сборки, а разместить их в так называемом глобальном КЭШе сборок (Global Assembly Cache) – централизованном хранилище сборок. В результате получается значительный выигрыш в размере приложения. В GAC может храниться несколько версий одной сборки, и он может управлять ими. Если сборку разместили в GAC, то она автоматически становится публичной – доступной другим приложениям. Если, напротив, использование подписанной сборки другими приложениями не требуется – достаточно поместить ее в корневую папку приложения.

Приложение, использующее строгую сборку, содержит информацию о ее строгом имени и версии. В результате достигается невозможность подмены сборки в корневой папке приложения: если просто заменить сборку на более новую, приложение не будет запускаться, поскольку строгое имя и версия сборок будут отличаться.

### **Создание сборки со строгим именем**

Создание сборки со строгим именем сводится к созданию закрытого ключа для шифрования хэш-кода сборки. Для этого используется утилита `sn.exe`, которая запускается из командной строки. Для создания закрытого ключа вводим команду:

`sn.exe -k "Путь к папке для сохранения ключа\Название ключа.snk"`

После того как закрытый ключ создан, необходимо прикрепить его к приложению. В окне Solution Explorer проекта дважды щелкаем на файле `AssemblyInfo.cs`. Если атрибут `[assembly: AssemblyKeyFile("")]` содержит путь к закрытому ключу, компилятор использует его для шифрования данных. Также с помощью атрибута `[assembly]` можно указать версию сборки, настройки культуры и другие параметры. Главное преимущество использования сборок, подписанных строгим именем, – защита их от декомпиляции.

Создать ключ и подписать сборку можно непосредственно в среде разработки MS Visual Studio. Для этого в окне Solution Explorer проекта щелкаем правой кнопкой мыши на имени проекта и выбираем Properties – Signing. В появившемся окне указываем файл ключа или необходимость его генерации и подписываем сборку.

## **Глобальный КЭШ сборок GAC (Global Assembly Cache). Утилита gacutil.exe**

Глобальный КЭШ сборок GAC – это хранилище сборок, одновременно используемых несколькими приложениями. Такие сборки называются публичными. GAC может содержать в себе несколько сборок, отличающихся друг от друга только версией. На вашем компьютере GAC находится в каталоге C:\WINDOWS\assembly. Все сборки, находящиеся в GAC, подписаны строгим именем – при установке сборки среда Common Language Runtime проверяет сборку на уникальность и сравнивает ее с другими, уже имеющимися сборками.

Управлять глобальным хранилищем сборок можно несколькими способами. Первый способ – с помощью утилиты gacutil.exe, которая запускается из командной строки Visual Studio.NET. Из всех команд утилиты нас интересуют всего три:

/i или – i: установка сборки в GAC;

/l или – l: вывод списка установленных сборок;

/u или – u: удаление сборки.

Более широкие возможности управления сборками предоставляет консоль MMC (Microsoft Management Console).

## **Настройка сборки**

.Net приложения можно разворачивать простым копированием всех необходимых сборок в одну папку на жестком диске, но на практике обычно выделяются отдельные подкаталоги для связанного содержимого, например, библиотек. В файлах конфигурации .NET можно указать подкаталоги, в которых среда выполнения будет искать частные сборки при запуске приложения. Для вставки в проект конфигурационного файла необходимо выполнить команду Project – Add New Item – Application Configuration File.

Файл конфигурации должен начинаться с корневого элемента под названием `<configuration>`. Вложенный элемент `<runtime>` может задавать элемент `<assemblyBinding>`, внутри которого в атрибуте `<privatePath>` и указывается нужный подкаталог с частной сборкой (если вам необходимо указать несколько подкаталогов, то они перечисляются через точку с запятой). Заметим, что атрибут `<privatePath>` служит для определения подкаталогов относительно каталога приложения. Если вы хотите указать каталог вне каталога приложения, то вам необходимо использовать элемент `<codeBase>`.

Предположим, что мы хотим разместить частные сборки в подкаталоге `MyLibraries` приложения. Тогда файл конфигурации может выглядеть так:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-
      microsoft-com:asm.v1">
      <probing privatePath="MyLibraries"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

## ***Контрольные вопросы***

1. Что такое сборка?
2. В чем различие между частными и общими сборками?
3. Как получить сборку со строгим именем?
4. Каково назначение GAC ?
5. Как настроить сборку?

## **Задания для лабораторной работы 5**

1. Создайте из класса, определенного в лабораторной работе 1, библиотеку (сборку). Подпишите ее.
2. Создайте приложение, которое будет использовать Вашу сборку. Настройте ее с помощью конфигурационного файла.

## Лабораторная работа 6. Делегаты и события

*Цель работы:* познакомиться с вопросами создания делегатов и манипулирования ими, исследовать ключевое слово `event`, которое упрощает работу с делегатами.

### **Необходимые теоретические сведения**

#### **Делегаты**

Делегат – это объект, имеющий ссылку на метод. Делегат позволяет выбрать вызываемый метод во время выполнения программы. Фактически значение делегата – это адрес области памяти, где находится точка входа метода.

Делегат позволяет указать в коде программы вызов метода, но фактически вызываемый метод определяется во время работы программы, а не во время компилирования.

Делегат объявляется с помощью ключевого слова `delegate`, за которым указывается тип возвращаемого значения, имя делегата и список параметров вызываемых методов. Примеры объявления классов делегатов:

```
delegate int ClassDelegate(int key);  
delegate void XXX(int intKey, float fKey);
```

Характерной особенностью делегата является возможность его использования для вызова любого метода, который соответствует сигнатуре делегата. Это дает возможность определить во время выполнения программы, какой из методов должен быть вызван. Вызываемый метод может быть методом экземпляра, ассоциированным с объектом, либо статическим методом, ассоциированным с классом. Метод можно вызвать только тогда, когда его сигнатура (тип возвращаемого значения, набор параметров) соответствует сигнатуре делегата.

Пример объявления и применения делегата представлен ниже.

```
// Данный делегат может указывать на любой метод,  
// принимающий два целых числа и возвращающий целое число  
public delegate int BinaryOp(int x, int y);
```

```

// Этот класс содержит методы, на которые будет указывать
// BinaryOp
public class SimpleMath
{
    public int Add(int x, int y)
    { return x + y; }
    public int Subtract(int x, int y)
    { return x - y; }
    public static int SquareNumber(int a)
    { return a * a; }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Simple Delegate Example *****\n");
        // Создадим экземпляр делегата BinaryOp, указывающий
        // на метод SimpleMath.Add()
        SimpleMath m = new SimpleMath();
        BinaryOp b = new BinaryOp(m.Add);
        // вызовем метод Add(), используя делегат
        Console.WriteLine("\n10 + 10 is {0}", b(10, 10));
        // Этот код является ошибочным:
        // BinaryOp b = new BinaryOp(m.SquareNumber);
        Console.ReadLine();
    }
}

```

## Многоадресность делегатов

Многоадресность – это способность делегата хранить несколько ссылок на различные методы, что позволяет при вызове делегата инициировать эту цепочку методов. Для создания цепочки методов необходимо создать экземпляр делегата и, пользуясь операторами `+` или `+=`, добавлять методы к цепочке. Для удаления метода из цепочки используется оператор `-` или `-=`.

```
delegate void Del(string s);
```



```

class TestClass
{
    static void Hello(string s)
    {
        System.Console.WriteLine(" Hello, {0}!", s);
    }
    static void Goodbye(string s)
    {
        System.Console.WriteLine(" Goodbye, {0}!", s);
    }
    static void Main()
    {
        Del a, b, c, d;
        // Делегат a указывает на метод Hello:
        a = Hello;
        // Делегат b хранит ссылку на метод Goodbye:
        b = Goodbye;
        // Делегат c хранит ссылки на оба метода:
        c = a + b;
        // Делегат d хранит ссылку только на метод Goodbye:
        d = c - a;
        System.Console.WriteLine("Invoking delegate a:");
        a("A");
        System.Console.WriteLine("Invoking delegate b:");
        b("B");
        System.Console.WriteLine("Invoking delegate c:");
        c("C");
        System.Console.WriteLine("Invoking delegate d:");
        d("D");
    }
}
/* Output:
Invoking delegate a: Hello, A!
Invoking delegate b: Goodbye, B!
Invoking delegate c: Hello, C! Goodbye, C!

```

Invoking delegate d: Goodbye, D!

\*/

## События

Делегаты представляют собой довольно интересные конструкции в том плане, что позволяют двум объектам в памяти участвовать в двустороннем диалоге. Так как способность одного объекта вызывать другой весьма полезна, C# предоставляет ключевое слово `event`, позволяющее упростить работу с делегатами. Когда компилятор обрабатывает ключевое слово `event`, вам автоматически предоставляются методы регистрации и отмены регистрации, а также все необходимые переменные члены для ваших типов делегатов.

Чаще всего события (events) используются в приложениях под Windows с графическим интерфейсом пользователя, в которых такие элементы управления, как `Button` (кнопка) или `Calendar` (календарь), реагируя на события, выдают информацию на той же панели, где они расположены. В качестве примера такого события можно привести, например, щелчок мышью на кнопке. Однако применение событий вовсе не ограничено приложениями с графическим интерфейсом – они могут быть исключительно полезными и в обычных консольных программах, как мы убедимся в наших примерах. Если в классе объявить член-событие, то объект – представитель этого класса сможет уведомлять объекты других классов о данном событии.

Определение события состоит из двух шагов. Во-первых, необходимо объявить делегат, содержащий методы, вызываемые при возникновении события. Во-вторых, нужно объявить события в терминах связанного делегата.

Для определения типа, который может отправлять события, применяется следующий эталон:

```
public class SenderOfEvents
{
    public delegate retval AssociatedDelegate(args);
    // retval – тип возвращаемого значения, args – список
    // параметров
    public event AssociatedDelegate NameOfEvent;
```

```
...  
}
```

В следующем примере определяется событие с тремя методами, которые связаны с ним. При инициировании события методы выполняются. Затем один метод удаляется из события и событие иницируется еще раз.

```
// Объявление делегата, связанного с событием:  
public delegate void MyEventHandler();  
class TestEvent  
{  
    // Объявление события через делегат MyEventHandler.  
    public event MyEventHandler TriggerIt;  
    // Объявление метода, который иницирует событие:  
    public void Trigger()  
    {  
        TriggerIt();  
    }  
    // Объявим методы, которые будут подписаны на событие.  
    public void MyMethod1()  
    {  
        System.Console.WriteLine("Hello!");  
    }  
    public void MyMethod2()  
    {  
        System.Console.WriteLine("Hello again!");  
    }  
    public void MyMethod3()  
    {  
        System.Console.WriteLine("Good-bye!");  
    }  
    static void Main()  
    {  
        // Создадим экземпляр класса TestEvent.  
        TestEvent myEvent = new TestEvent();  
        // Подпишем на событие три метода:
```

```

myEvent.TriggerIt += new MyEventHandler(myEvent.MyMethod1);
myEvent.TriggerIt += new MyEventHandler(myEvent.MyMethod2);
myEvent.TriggerIt += new MyEventHandler(myEvent.MyMethod3);
    // Инициируем событие:
    myEvent.Trigger();
    // Отменим подписку на событие у второго метода:
myEvent.TriggerIt -= new MyEventHandler(myEvent.MyMethod2);
    System.Console.WriteLine("\nHello again!\n unsubscribed from
the event.");
    // Инициируем новое событие:
    myEvent.Trigger();
}
}
// Результаты работы программы: Hello! Hello again! Good-bye!
// "Hello again!" unsubscribed from the event.
// Hello! Good-bye!

```

## **Создание событий базового класса в производных классах**

В следующем примере показан стандартный способ объявления событий в базовом классе таким образом, чтобы они могли создаваться и из производного класса. Этот принцип широко используется в классах Windows Forms в библиотеке классов .NET Framework.

При создании класса, который может служить базовым для других классов, следует учитывать, что события являются делегатами особого типа, которые могут быть вызваны только из класса, который их объявил. Производные классы не могут напрямую создавать события, объявленные в базовом классе. Иногда нужно, чтобы событие могло создаваться только в базовом классе, однако чаще всего следует обеспечить производному классу возможность создания событий базового класса. Для этого следует создать в базовом защищенный метод, предоставляющий оболочку для события. Путем вызова или переопределения этого метода производные классы могут опосредованно вызывать событие.

```

namespace BaseClassEvents
{
    // Специальный EventArgs клас для хранения информации о
    // фигуре.
    public class ShapeEventArgs : EventArgs
    {
        private double newArea;
        public ShapeEventArgs(double d)
        {
            newArea = d;
        }
        public double NewArea
        {
            get { return newArea; }
        }
    }
    // Базовый класс, который содержит событие
    public abstract class Shape
    {
        protected double area;
        public double Area
        {
            get { return area; }
            set { area = value; }
        }
    }
    // Событие. Заметим, что мы используем обобщенный тип
    // события EventHandler<T> и поэтому можем не объявлять
    // отдельный делегат
    public event EventHandler<ShapeEventArgs> ShapeChanged;
    public abstract void Draw();
    // Метод, иницирующий событие, который производный
    // класс может переопределить.
    protected virtual void OnShapeChanged(ShapeEventArgs e)
    {

```

```
// Создадим временную копию события, чтобы избежать
// ошибки, если последний подписчик отпишется от события
// немедленно после проверки на null и перед возбуждением
// события.
```

```
    EventHandler<ShapeEventArgs> handler = ShapeChanged;
    if (handler != null)
    {
        handler(this, e);
    }
}
```

```
// производный класс окружности
```

```
public class Circle : Shape
{
    private double radius;
    public Circle(double d)
    {
        radius = d;
        area = 3.14 * radius;
    }
    public void Update(double d)
    {
        radius = d;
        area = 3.14 * radius;
        OnShapeChanged(new ShapeEventArgs(area));
    }
    protected override void OnShapeChanged(ShapeEventArgs e)
    {
        // Выполним действия, специфичные для circle.
        ...
        // Вызов метода базового класса, инициирующего событие
        base.OnShapeChanged(e);
    }
    public override void Draw()
    {

```

```

    Console.WriteLine("Drawing a circle");
}
}
// производный класс прямоугольник
public class Rectangle : Shape
{
    private double length;
    private double width;
    public Rectangle(double length, double width)
    {
        this.length = length;
        this.width = width;
        area = length * width;
    }
    public void Update(double length, double width)
    {
        this.length = length;
        this.width = width;
        area = length * width;
        OnShapeChanged(new ShapeEventArgs(area));
    }
    protected override void OnShapeChanged(ShapeEventArgs e)
    {
        // Выполнение действий, специфичных для прямоугольника
        ...
        // Вызов метода базового класса, иницирующего событие
        base.OnShapeChanged(e);
    }
    public override void Draw()
    {
        Console.WriteLine("Drawing a rectangle");
    }
}
// Класс, имитирующий поверхность, на которой
// отображаются фигуры

```

```

    // Этот класс подписан на события фигур,
    // чтобы знать, когда выполнять их перерисовку
    public class ShapeContainer
    {
        List<Shape> _list;
        public ShapeContainer()
        {
            _list = new List<Shape>();
        }
        public void AddShape(Shape s)
        {
            _list.Add(s);
            // Подписка на событие базового класса
            s.ShapeChanged += HandleShapeChanged;
        }
        // ...Другие методы для рисования, изменения размера и т.д.
        private void HandleShapeChanged(object sender,
        ShapeEventArgs e)
        {
            Shape s = (Shape)sender;
            // Сообщение для демонстрации
            Console.WriteLine("Received event. Shape area is now {0}",
            e.NewArea);
            // Перерисовка фигуры
            s.Draw();
        }
    }
    class Test
    {
        static void Main(string[] args)
        {
            // Создадим экземпляры классов, иницирующих событие и
            // подписчика
            Circle c1 = new Circle(54);

```



```

Rectangle r1 = new Rectangle(12, 9);
ShapeContainer sc = new ShapeContainer();
// Добавим фигуры в контейнер
sc.AddShape(c1);
sc.AddShape(r1);
// Инициализируем события
c1.Update(57);
r1.Update(7, 7);
// Задержка окна консоли для просмотра результатов
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();
}
}
}
/* Output: Received event. Shape area is now 178.98
   Drawing a circle
   Received event. Shape area is now 49
   Drawing a rectangle
   */

```

## ***Контрольные вопросы***

1. Дайте определение делегата. Приведите фрагмент кода, иллюстрирующий создание и использование делегата.
2. Что понимается под многоадресностью делегата?
3. В чем состоят преимущества использования делегатов?
4. В какой момент осуществляется выбор вызываемого метода в случае использования делегатов?
5. Что является значением делегата?
6. В чем состоит практическое значение многоадресности?
7. Каким образом осуществляется создание цепочки методов для многоадресных делегатов?
8. Для чего служит ключевое слово `event`? Являются ли события членами классов?
9. Приведите синтаксис описания события в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

10. Как подписаться на событие и отписаться от него?
11. Как создать событие базового класса в производном?

### **Задание для лабораторной работы 6**

Добавьте в иерархию классов лабораторной работы 2 обработку событий. События должны быть определены в базовом классе, но инициироваться в производных классах. Добавьте обработчики событий, продемонстрируйте подписку на события и отказ от неё.

## **Лабораторная работа 7**

### **Отражение типов и позднее связывание**

*Цель работы:* познакомиться ключевыми элементами служб отражения классом `System.Type` и пространством имен `System.Reflection` и механизмом позднего связывания, позволяющего создавать расширяемые приложения.

### ***Необходимые теоретические сведения***

#### **Отражение**

Отражением называется процесс нахождения типов во время выполнения. Классы в пространстве имен `System.Reflection` вместе с `System.Type` позволяют получать сведения о загруженных сборках и определенных в них типах, таких как классы, интерфейсы и типы значений. Отражение можно также использовать для создания экземпляров типов во время выполнения, для вызова этих экземпляров и получения доступа к ним.

#### **Класс `System.Type`**

Данный класс определяет несколько членов, которые можно задействовать для исследования метаданных типов. Многие из этих членов возвращают типы из пространства имен `System.Reflection`. Приведем некоторые из них.

<i>Член</i>	<i>Описание</i>
IsAbstract IsArray IsClass IsEnum IsValueType	Эти свойства позволяют получать различную основную информацию о типе: является ли он массивом, абстрактным, значимым и т. д.
GetConstructors() GetEvents() GetFields() GetInterfaces() GetMethods() GetProperties() GetNestedTypes()	Эти методы позволяют получить массив, содержащий требуемые элементы (интерфейсы, методы, свойства и т. д.). Каждый метод возвращает соответствующий массив (например, GetFields() возвращает массив FieldInfo, GetMethods() возвращает массив MethodInfo). Каждый из этих методов имеет форму в единственном числе, например GetMethod().
GetType()	Статический метод возвращает экземпляр класса Type на основании переданного имени
InvokeMember()	Метод позволяет выполнять позднее связывание для заданного элемента

Получить экземпляр класса Type можно разными способами, но непосредственно создать объект этого типа нельзя, так как Type – это абстрактный класс.

// получение информации о типе, используя экземпляр этого  
// типа

```
Manager Bill = new Manager();
```

```
Type t = Bill.GetType();
```

// получение информации о типе статическим методом  
// GetType() параметры – имя типа в сборке, имя сборки

```
Type t = Type.GetType("Employers.Manager, Employers ");
```

// получение информации о типе с помощью оператора  
// typeof()

```
Type t = typeof(Manager);
```

Для иллюстрации механизма отражения приведем пример приложения, которое будет выводить информацию о методах, свойствах, полях для любого типа в библиотеке mscorlib.dll. Данный пример взят из книги [1].

```

    // Статический метод, который выводит информацию о
// различных свойствах исследуемого типа
    public static void ListVariousStats(Type t)
    {
        Console.WriteLine("***** Various Statistics *****");
        Console.WriteLine("Base class is: {0}", t.BaseType);
        Console.WriteLine("Is type abstract? {0}", t.IsAbstract);
        Console.WriteLine("Is type sealed? {0}", t.IsSealed);
        Console.WriteLine("Is type generic? {0}", t.IsGenericType
Definition);
        Console.WriteLine("Is type a class type? {0}", t.IsClass);
        Console.WriteLine("");
    }
    // Статический метод, который выводит список всех методов
// исследуемого типа, их параметры и тип возвращаемого
// значения
    public static void ListMethods(Type t)
    {
        Console.WriteLine("***** Methods *****");
        MethodInfo[] mi = t.GetMethods();
        foreach (MethodInfo m in mi)
        {
            // Get return value.
            string retVal = m.ReturnType.FullName;
            string paramInfo = "(";
            // Get params.
            foreach (ParameterInfo pi in m.GetParameters())
            {
                paramInfo += string.Format("{0} {1} ", pi.ParameterType,
pi.Name);
            }
            paramInfo += ")";
            // Now display the basic method sig.
            Console.WriteLine("->{0} {1} {2}", retVal, m.Name, paramInfo);
        }
    }

```

```

    Console.WriteLine("");
}
// Статический метод, который выводит список полей
// исследуемого типа
public static void ListFields(Type t)
{
    Console.WriteLine("***** Fields *****");
    FieldInfo[] fi = t.GetFields();
    foreach (FieldInfo field in fi)
        Console.WriteLine("->{0}", field.Name);
    Console.WriteLine("");
}
// Статический метод, который выводит список свойств
// исследуемого типа
public static void ListProps(Type t)
{
    Console.WriteLine("***** Properties *****");
    PropertyInfo[] pi = t.GetProperties();
    foreach (PropertyInfo prop in pi)
        Console.WriteLine("->{0}", prop.Name);
    Console.WriteLine("");
}
// Статический метод, который выводит список
// реализованных в типе интерфейсов
public static void ListInterfaces(Type t)
{
    Console.WriteLine("***** Interfaces *****");
    Type[] ifaces = t.GetInterfaces();
    foreach (Type i in ifaces)
        Console.WriteLine("->{0}", i.Name);
}
// Приложение
static void Main(string[] args)
{
    Console.WriteLine("***** Welcome to MyTypeViewer *****");
}

```

```

string typeName = "";
bool userIsDone = false;
do
{
    Console.WriteLine("\nEnter a type name to evaluate");
    Console.Write("or enter Q to quit: ");
    // Получение имени типа для исследования
    typeName = Console.ReadLine();
    // Проверка окончания работы с программой
    if (typeName.ToUpper() == "Q")
    {
        userIsDone = true;
        break;
    }
    // Try to get named type.
    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");
    }
} while (!userIsDone);
}

```

## Динамическая загрузка сборок

Динамическая загрузка сборок – это процесс загрузки внешних сборок по запросу. Пространство имен System.Reflection определяет класс Assembly, используя который можно динамически загружать как общие, так и частные сборки и получать их свойства. Приведем фрагмент кода, который по введенному имени сборки выводит о ней информацию.

```
// Получим имя сборки
string asmName = Console.ReadLine();
// Динамическая загрузка
try
{
    asm = Assembly.Load(asmName);
    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);
}
catch
{
    Console.WriteLine("Sorry, can't find assembly.");
}
...
```

## Позднее связывание

Позднее связывание – это техника, позволяющая создать экземпляр некоторого типа и вызывать его члены во время выполнения, не имея информации о нем во время компиляции. Прежде всего эта техника используется для создания расширяемых приложений (plug-in), то есть приложений, которые позволяют сторонним производителям встроить в них свои модули. Примером расширяемого приложения является среда разработки MS Visual Studio.

Понятно, что расширяемое приложение должно предоставить некоторый механизм, позволяющий указать, какой модуль

встроить, а это динамическая загрузка. Расширяемое приложение должно определить, поддерживает ли модуль нужную функциональность, здесь требуется отражение. И наконец, расширяемое приложение должно получить ссылку на необходимый класс или интерфейс и вызвать его члены. Для этих действий и необходимо позднее связывание.

Приведем пример расширяемого приложения. Сначала определим общий интерфейс, который будет реализован каждым встраиваемым модулем.

```
public interface IAppFunctionality
{
    void Dolt();
}
```

Код встраиваемого модуля может быть таким:

```
public class TheCSharpModule : IAppFunctionality
{
    void IAppFunctionality.Dolt()
    {
        MessageBox.Show("You have just used the C# snap in!");
    }
}
```

Теперь приведем код формы расширяемого приложения Windows Forms.

```
partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    private bool LoadExternalModule(string path)
    {
        bool foundSnapIn = false;
        IAppFunctionality itfAppFx;
        // Динамическая загрузка выбранного модуля.
        Assembly theSnapInAsm = Assembly.LoadFrom(path);
```



```

// Получение всех типов сборки
Type[] theTypes = theSnapInAsm.GetTypes();
// Поиск типа, реализующего интерфейс IAppFunctionality.
for (int i = 0; i < theTypes.Length; i++)
{
    Type t = theTypes[i].GetInterface("IAppFunctionality");
    if (t != null)
    {
        foundSnapIn = true;
        // Позднее связывание для создания экземпляра типа
        object o = theSnapInAsm.CreateInstance(theTypes[i].FullName);

        // Вызов метода Dolt() интерфейса
        itfAppFx = o as IAppFunctionality;
        itfAppFx.Dolt();
        lstLoadedSnapIns.Items.Add(theTypes[i].FullName);
    }
}
return foundSnapIn;
}

// Обработчик события выбора пункта меню «загрузка
// модуля»
private void snapInModuleToolStripMenuItem_Click_1(object
sender, EventArgs e)
{
    // Открытие диалога для выбора модуля для загрузки
    OpenFileDialog dlg = new OpenFileDialog();
    if (dlg.ShowDialog() == DialogResult.OK)
    {
        if (LoadExternalModule(dlg.FileName) == false)
            MessageBox.Show("Nothing implements IAppFunctionality!");
    }
}
}

```

## **Контрольные вопросы**

1. Что такое отражение?
2. Как можно получить экземпляр класса `System.Type`?
3. Когда применяется динамическая загрузка сборок?
4. Приведите фрагмент кода, иллюстрирующий механизм отражения.
5. Для какой цели используется позднее связывание?
6. Как создать расширяемое приложение?

## **Задания для лабораторной работы 7**

1. Объявите интерфейс, содержащий 3 метода: печати (вывода), сортировки по числовому полю, сортировки по строковому полю.
2. Измените код лабораторной работы 4 таким образом, чтобы производные классы реализовывали вышеописанный интерфейс.
3. Создайте расширяемое приложение `Windows.Forms`. Приложение должно загружать любую библиотеку классов, реализующую интерфейс пункта 1, и отображать на форме информацию, отсортированную различными способами.

## Список литературы

1. Троелсен, Э. С# и платформа .Net 3.0, специальное издание / Э. Троелсен. – СПб.: Питер, 2008. – 1456 с.
2. Рихтер, Д. CLR via C#. Программирование на платформе Microsoft .Net Framework 2.0 на языке C#. Мастер-класс / Д. Рихтер. – СПб.: Питер, 2007. – 656 с.
3. Нортроп, Т. Основы разработки приложений на платформе Microsoft .Net Framework. Учебный курс Microsoft / Т. Нортроп, Ш. Уилдермьюс, Б. Райан. – СПб.: Питер, 2007. – 864 с.
4. Главная страница MSDN : <http://www.msdn.com>

## Оглавление

<b>Введение .....</b>	<b>3</b>
<b>Лабораторная работа 1. Объекты и классы (конструкторы, инкапсуляция, свойства, перегрузка операций) .....</b>	<b>4</b>
<b>Лабораторная работа 2. Семейства классов и программирование полиморфных методов (наследование, абстрактные классы, виртуальные методы) .....</b>	<b>14</b>
<b>Лабораторная работа 3. Обработка исключительных ситуаций .....</b>	<b>22</b>
<b>Лабораторная работа 4. Интерфейсы, коллекции и обобщенные коллекции .....</b>	<b>29</b>
<b>Лабораторная работа 5. Знакомство с .Net сборками .....</b>	<b>42</b>
<b>Лабораторная работа 6. Делегаты и события .....</b>	<b>47</b>
<b>Лабораторная работа 7. Отражение типов и позднее связывание .....</b>	<b>58</b>
<b>Список литературы.....</b>	<b>67</b>



**О. П. Якимова**

**Языки программирования  
Часть 1**