Кириченко А.А.

Объектно-ориентированное программирование на алгоритмическом языке C#.

Москва 2015.

Кириченко А.А.

Объектно-ориентированное программирование на алгоритмическом языке C#.

– сетевое электронное издание учебного пособия; М., издательство "Высшая школа экономики" , 215стр., формат PDF.

ISBN 978-5-9904911-4-4

Рецензент: Фомин Г.П., к.т.н., профессор кафедры Математических методов в экономике РЭУ им. Г.В.Плеханова

Объектно-ориентированный стиль программирования требует от программиста повышенной квалификации из-за сложности языка.

В учебном пособии проводится анализ архитектуры таких стилей программирования, как визуальное (точнее – событийное), и объектно-ориентированное программирование, вводятся ограничения на использование пакета Visual Studio при объектно-ориентированном программировании и формулируются способы коррекции программного проекта для сохранения базовых принципов инкапсуляции. Изложенная методика создания и приведенные примеры позволят приобрести устойчивые навыки объектно-ориентированного программирования.

Книга может быть полезна студентам, бакалаврам, магистрам, аспирантам, специализирующимся на использовании компьютеров при решении экономических задач.

Кириченко А.А. профессор департамента программной инженерии факультета компьютерных наук Федерального государственного автономного образовательного учреждения высшего профессионального образования “Национальный исследовательский университет "Высшая школа экономики" при правительстве РФ”.

ISBN 978-5-9904911-4-4

Содержание.

Архитектура различных стилей программирования.

По мере развития вычислительной техники изменялась не только вычислительная среда, но и способ мышления программистов, и методы, и средства программирования. Появлялись новые алгоритмические языки, модели, стили и технологии программирования.

Разработанный фирмой Microsoft язык C# сориентирован на использование проверенных временем самых современных технологий: предшественниками его были язык C, разработанный в рамках модели структурного программирования, языки C++, реализующий модель объектно-ориентированного программирования, и Java с его межъязыковой возможностью взаимодействия.

C# имеет встроенную поддержку средств написания программ, например, таких, как пакет Visual Studio, облегчающий создание программ с асинхронным интерфейсом, в основе которых лежит получившее развитие в операционной системе Windows использование событий.

Пакет Visual Studio является универсальным пакетом, позволяющим писать программы на разных языках и реализовать их в виде различных моделей программирования.

Например, разработка на C# программ с асинхронным интерфейсом предусматривает создание такой конструкции, как формы, и нанесение на неё компонент и элементов управления. При комплектовании формы приходится определять размеры элементов управления, их локализацию на форме, цвет, поясняющие надписи, обработчики событий, возникающих при активизации элементов управления. Если программирование ведётся на Visual Studio, то наносимые на форму элементы управления размещают свой код в классе, характеризующем форму.

Но код, определяющий размеры, локализацию, цвет, надписи для формы необходим, не является для неё посторонним. А обработчики возникающих при активизации размещённых на форме элементов управления событий содержат код, не имеющий к форме никакого отношения. Для программ, управляемых событиями (визуальное программирование), включение обработчиков событий, возникающих при активизации элементов управления допустимо и не нарушает требований этого стиля программирования.

Включение же такого кода в состав класса, характеризующего форму, нарушает один из основных принципов объектно – ориентированного программирования – инкапсуляцию и является недопустимым для программирования в стиле ООП.

Visual Studio позволяет разрабатывать программы фон-Неймановского стиля, стиля, использующего библиотеки стандартных подпрограмм, программы структурного программирования, компонентные программы, программы, управляемые событиями (визуальное программирование), и объектно-ориентированные программы. Но среди них только визуальное и объектно-ориентированное программирование создают трудности для реализации друг друга.

Проанализируем особенности следующих стилей программирования: программирование, управляемое событиями (визуальное программирование) и объектно-ориентированное программирование.

К основным конструктивным элементам программы, управляемой событиями, относятся классы, методы, делегаты, события. Это универсальные конструктивные элементы, используемые в различных моделях программирования.

К конструктивным элементам объектно-ориентированных программ так же относятся классы и объекты. Причём, классы и объекты могут использоваться и в других моделях программирования, но в объектно-ориентированных программах они имеют свойства, не характерные для других моделей программирования, значительно отличающиеся от элементов с аналогичными названиями в других моделях.

Существенное отличие объектно-ориентированные программы получают из-за наличия специфических принципов программирования в объектно-ориентированной модели. Таких принципов три: инкапсуляция, наследование и полиморфизм.

Программирование, управляемое событиями.

Современное программирование ориентировано на интерактивный способ работы с программой, когда конечный пользователь сидит непосредственно за экраном компьютера и управляет процессом выполнения программы. В таких условиях роль пользовательского интерфейса становится крайне важной.

Для создания интерфейса используется графическое представление, способствующее интерактивному взаимодействию пользователя с программой. Это графическое представление создаётся с помощью специального инструментария, позволяющего визуализировать процесс работы.

Строго говоря, визуальное программирование — это способ **создания** программы для ЭВМ путём манипулирования графическими объектами **вместо** написания её текста. Именно создания программы, а не управления её работой. Это название иногда неправильно применяют для модели программирования, управляемого событиями.

По существу в современном понимании визуальное программирование является программированием, управляемым событиями.

Упрощённо процесс визуальной разработки приложений заключается в графическом проектировании внешнего вида (дизайна) приложений с последующей привязкой программного кода к элементам пользовательского интерфейса.

Создавая новую программу, разработчик вызывает на экран форму и перемещает мышью на неё необходимые компоненты и элементы управления. За счёт этого быстро формируется рабочий скелет программы. При этом размещение элементов интерфейса, их локализация на форме и настройка внешнего вида производится мышью простым и понятным образом.

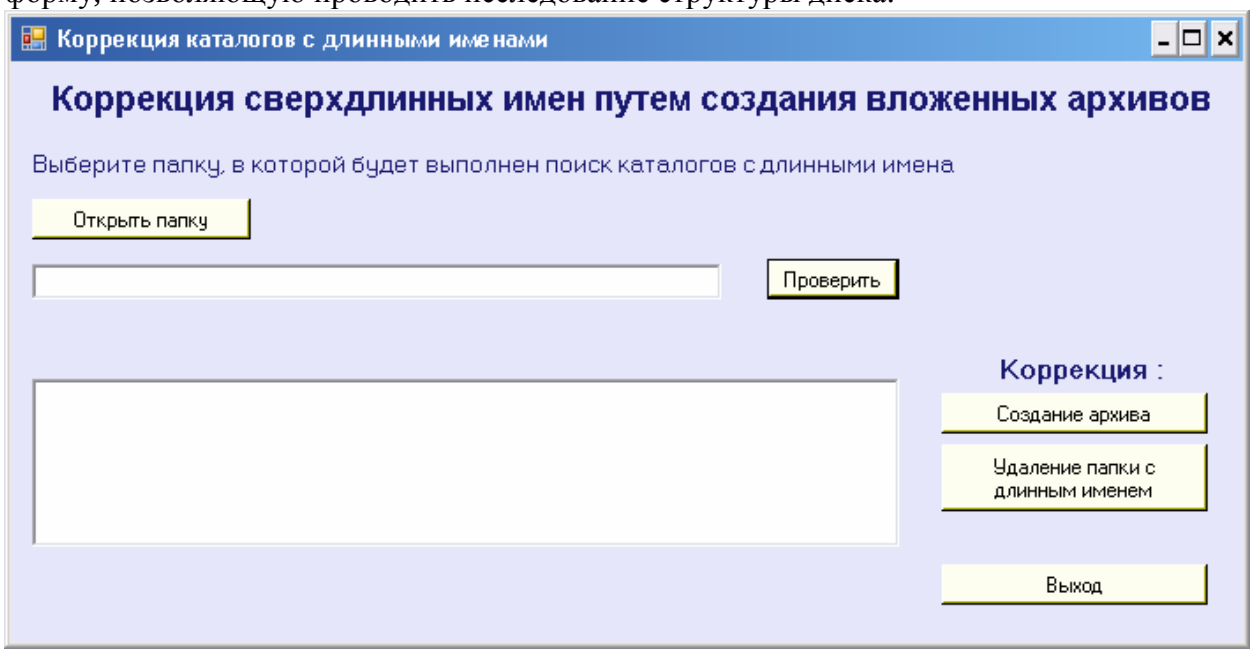
Когда скелет программы создан, его надо наполнить необходимой функциональностью. Щёлкая мышью элементы интерфейса, нанесенные на форму, программист с помощью редактора текста вводит программный код, предназначенный для работы с этими элементами. Такая технология во много раз ускоряет разработку приложений со сложным пользовательским интерфейсом.

Термин «программирование, управляемое событиями» означает следующее: в ходе выполнения программы могут возникать ситуации, идентифицируемые как события. В этом случае выполнение программы приостанавливается, операционная система получает сообщение от программы о возникновении в ней «события» и вызывает предусмотренный обработчик события. В результате начинает выполняться программа обработчика события.

Windows-приложение в полной мере отвечает этой модели программирования, позволяя достаточно просто и естественно создавать визуальную, управляемую событиями программу. По умолчанию в этом приложении создается форма, которую можно населять элементами управления, определяющими интерфейс пользователя.

В главной процедуре Main, с которой начинается выполнение программы, создается объект, задающий спроектированную форму, и эта форма открывается на экране. Дальнейший ход вычислений определяется конечным пользователем, – какие данные он будет вводить в текстовые окна, какие командные кнопки будет нажимать.

Рассмотрим пример, реализованный как Windows-приложение и демонстрирующий форму, позволяющую проводить исследование структуры диска:



При построении интерфейса использовались такие элементы управления, как

- метки окна редактирования,
- текстовые окна, позволяющие выводить и вводить в них текст и редактировать его,

- командные кнопки.

Для удобства пользователя использовались также окна, позволяющие группировать информацию, отделяя в данном случае исходную информацию от получаемых результатов.

Код программы, реализующей форму, приведен в следующем примере:

Пример текста программы Windows-приложения, демонстрирующий форму, позволяющую проводить исследование структуры диска

```
using System;
using System.IO;
using System.Diagnostics;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace _2term_work
{
    public partial class Form1 : Form
    {

//Определение переменных
        string[] a;
        string str;

//Конструктор
        public Form1()
        {
            InitializeComponent();
        }

//Обработчики событий
//----------------------------------------------------------------
private void button1_Click(object sender, EventArgs e)
        {
if (textBox1.Text.Equals("")) { MessageBox.Show("Выберите каталог для
проверки"); }
            else
            {
                a = new string[100];
                try
                {
string[] subdir = Directory.GetDirectories(textBox1.Text, "*",
System.IO.SearchOption.AllDirectories);
                    int k = 0;
                    int j = 0;
                    for (int i = 0; i < subdir.Length; i++)
                    {
                        if (subdir[i].Length > 200)
                        {
   label1.Text = "В папке обнаружены имена длиной более 200 символов";
                            k += 1;
                            a[j] = subdir[i]; j++;
                        }
                    }
                    listBox1.DataSource = a;
                    if (k == 0)
                    {
    label1.Text = "Папок с длиной имени более 200 символов не обнаружено";
```

```
                    }
                }
  catch { MessageBox.Show("Выбранная папка содержит защищенные каталоги"); }
            }
        }
//-------------------------------------------------------------------------
        private void button2_Click(object sender, EventArgs e)
        {
            FolderBrowserDialog d = new FolderBrowserDialog();
            d.Description = "Выберите папку для проверки";
            d.SelectedPath = @"c:\";
            if (d.ShowDialog() == DialogResult.OK)
            {
                textBox1.Text = d.SelectedPath;
            }
        }

        private void button3_Click(object sender, EventArgs e)
        {
            string dir = "", dir1 = "";
if (listBox1.SelectedIndex == -1) { MessageBox.Show("Выберите в списке слева
папку для реконструкции"); }
            else
            {
                int k = listBox1.SelectedIndex;
                string[] d = a[k].Split('\\');
                string[] g = textBox1.Text.Split('\\');
                dir = textBox1.Text + "\\" + d[g.Length];
                dir1 = dir + ".rar";
                Process pro = new Process();
                pro.StartInfo.CreateNoWindow = true;
                pro.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
         pro.StartInfo.Arguments = "a -y -inul -ibck " + dir1 + " " + dir;
                pro.StartInfo.FileName = "WinRAR";
                pro.Start();
                str = dir;
            }
        }
//-------------------------------------------------------------------------
        private void button4_Click(object sender, EventArgs e)
        {
            Dispose();
        }
//-------------------------------------------------------------------------
        private void button5_Click(object sender, EventArgs e)
        {
            int k = 0;
            Directory.Delete(str, true);
            str = "";
            a = new string[100];
            listBox1.DataSource = a;
string[] subdir = Directory.GetDirectories(textBox1.Text, "*",
System.IO.SearchOption.AllDirectories);
            int p = 0;
            int j = 0;
            for (int i = 0; i < subdir.Length; i++)
            {
                if (subdir[i].Length > 200)
                {
     label1.Text = "В папке обнаружены имена длиной более 200 символов!";
                    p += 1;
                    a[j] = subdir[i]; j++;
                }
            }
```

```
            listBox1.DataSource = a;
            if (k == 0)
            {
    label1.Text = "Папок с длиной имени более 200 символов не обнаружено";
            }
        }
    }       //Конец класса Form1
}   //Конец пространства имён _2term_work
```

Этот код содержит класс, в котором спроектирована выводимая на экран форма, код для выбора проверяемого каталога, выбора папки для реконструкции, проверки длины имён файлов и папок, и т.д.

Несмотря на то, что весь проект состоит из двух классов: Form1 и Program, практически весь код сосредоточен в одном классе Form1. Класс Program содержит модуль Main (главная точка входа для приложения), состоящий только из операторов запуска созданной формы.

Для этого проекта характерно, что в нём используются обработчики событий, вызываемых нажатием соответствующих кнопок на форме.

Данная модель программирования вначале разрабатывалась для реализации языка типа Visual Basic, и к размещению кода строгих требований не предъявляет.

Использование Visual Studio для создания асинхронных программ предусматривает автоматическое создание объектов для используемых компонент и элементов управления в специальной зоне класса Form1, ограниченной операторами #region Windows Form Designer generated code – и #endregion. .

При этом делается предупреждение, что в этой зоне программист ничего менять не должен (Required method for Designer support - do not modify the contents of this method with the code editor). Хотя эта вставка дизайнера делается в отдельном файле (не Form1.cs, а Form1.Designer.cs), получается, что часть всей асинхронной программы размещается в классе Form1.

При создании компонент и элементов управления определяются некоторые их атрибуты, в число которых входит и функциональность элемента – т.е. определение того, что должно происходить при активизации этого элемента. Для этого создаются обработчики событий, которые системой VS размещаются так же в классе Form1.

К структуре классов и к их количеству в программе при визуальном программировании никаких требований не предъявляется.

Конструктивные элементы программы, управляемой событиями

К конструктивным элементам программы, управляемой событиями, относятся пространства имён, классы, методы, делегаты, события. Это универсальные конструктивные элементы, используемые в различных моделях программирования. Самые крупные конструктивные элементы – это пространства имён. Они состоят из классов различных типов, например, таких, как структуры, интерфейсы, перечисления индексаторы, делегаты. Методы представляют собой блоки кода, содержащие наборы инструкций (команд).

Классы.

Модульность построения - основное свойство системы визуального программирования. Класс, как модуль, является основной архитектурной единицей построения программной системы этого типа. Модуль может не представлять собой

содержательную единицу - его размер и содержание определяется архитектурными соображениями, а не семантическими. Ничто не мешает построить монолитную систему, состоящую из одного модуля - она может решать ту же задачу, что и система, состоящая из многих модулей.

Методы.

В программировании все программы принято представлять в виде процедур и функций.

Процедуры и функции обычно связываются с классом, они обеспечивают функциональность данных класса и называются методами класса. Метод представляет собой блок кода, содержащий набор инструкций.

В C# процедуры и функции не существуют вне класса, они существуют только как методы некоторого класса. (В данном контексте понятие класс распространяется и на все его частные случаи - структуры, интерфейсы, делегаты).

В языке C# нет специальных ключевых слов - procedure и function, но присутствуют сами эти понятия.

Чем является метод - процедурой или функцией, позволяет однозначно определить синтаксис объявления метода.

Функция отличается от процедуры двумя особенностями:

- вызывается в выражениях.
- всегда вычисляет некоторое значение (обычно – только одно), возвращаемое в качестве результата функции по оператору return.

Процедура C# имеет свои особенности:

вызов процедуры является оператором языка;

процедура возвращает формальный результат void, указывающий на отсутствие результата;

процедура имеет входные и выходные аргументы, причем выходных аргументов - ее результатов - может быть достаточно много.

Хорошо известно, что одновременное существование в языке процедур и функций в каком-то смысле избыточно. Добавив еще один выходной аргумент, любую функцию можно записать в виде процедуры. Справедливо и обратное.

Однако значительно удобнее иметь обе формы реализации метода: и процедуры, и функции.

Обычно метод предпочитают реализовать в виде функции тогда, когда он имеет один выходной аргумент, рассматриваемый как результат вычисления значения функции.

Возможность вызова функций в выражениях также влияет на выбор в пользу реализации метода в виде функции.

В других случаях метод реализуют в виде процедуры.

Методы не существуют вне класса. Обычно для их использования с помощью оператора new создаются объекты – т.е. экземпляры класса, через которые осуществляется доступ к методам. Но в некоторых случаях методы объявляются с помощью оператора static “методами класса” – при этом для доступа к методу объект создавать не надо.

Если, учитывая возможность наследования класса, нельзя заранее определить тело метода, в классе может быть создан **абстрактный** метод с отсроченным определением его тела.

Для создания экземпляров класса используются специальные методы – **конструкторы**. Отличительной особенностью конструктора является то, что его имя совпадает с именем класса. Основная задача конструктора – создание объекта и настройка начальных значений его полей.

Несколько методов, имеющих одинаковое имя, могут иметь разные тела. Такие методы называются **перегруженными**. Основным отличием их описаний друг от друга является разный состав параметров, передаваемых методам.

В программе на C# должен присутствовать особый метод – **«Main»**, в котором начинается и заканчивается управление. В методе Main создаются объекты и выполняются другие методы.

Метод Main является статическим методом.

Метод Main не возвращает никакого значения или может возвращать значение типа int, может принимать ограниченный состав аргументов (параметр метода Main является массивом значений типа string, представляющим аргументы командной строки, используемые для вызова программы).

Метод может быть виртуальным, допускающим переопределение в производных классах.

Взаимодействие методов по управлению и по обмену данными.

Выполнение программы можно рассматривать как взаимодействие методов с точки зрения двух аспектов:

- по управлению (вызов/возврат)
- по обмену данными

Управление методами включает в себя объявление (подпись) методов, их описание и доступ к ним.

Обмен данными затрагивает возвращаемые значения, параметры и аргументы, особенности передачи параметров.

Управление методами.

Объявление метода

Методы объявляются в классе путем указания

уровня доступа, например public или private,
необязательных модификаторов, например abstract или sealed,
возвращаемого значения,
имени метода,
списка параметров этого метода.

Все вместе эти элементы образуют подпись метода.

Тип возвращаемого методом значения не является частью подписи метода с точки зрения перегрузки методов. В то же время он является частью подписи метода при определении совместимости между делегатом и методом, на который он указывает.

Параметры заключаются в круглые скобки и разделяются запятыми. Пустые скобки указывают на то, что у метода нет параметров.

Следующий класс содержит три метода.

```
abstract class Motorcycle
{
    // Это может использовать любой.
    public void StartEngine() {/* Тело метода находится здесь */ }

    // Это могут использовать только некоторые классы.
    protected void AddGas(int gallons) { /* Тело метода находится здесь */ }

    // Производные классы могут переопределить инструментарий базового класса.
    public virtual int Drive(int miles, int speed) { /* Тело метода находится
здесь */ return 1; }

    // Производные классы могут доработать это.
    public abstract double GetTopSpeed();
}
```

При объявлении метода обычно формируется его описание.

Описание методов (процедур и функций).

Синтаксически в описании метода различают две части - описание заголовка и описание тела метода.

Рассмотрим синтаксис заголовка метода:

[атрибуты][модификаторы]{void| тип_результата_функции}
имя_метода([список_формальных_аргументов])

Имя метода и список формальных аргументов составляют **сигнатуру** метода (в сигнатуру не входят имена формальных аргументов - здесь важны типы аргументов - и тип возвращаемого результата).

Квадратные скобки (метасимволы синтаксической формулы) показывают, что атрибуты и модификаторы могут быть опущены при описании метода.

Модификатор доступа может иметь четыре возможных значения, из которых пока рассмотрим только два - public и private.

Модификатор public показывает, что метод открыт и доступен для вызова клиентами и потомками класса.

Модификатор private говорит, что метод предназначен для внутреннего использования в классе и доступен для вызова только в теле методов самого класса.

Если модификатор доступа опущен, то по умолчанию предполагается, что он имеет значение private и метод является закрытым для клиентов и потомков класса.

Обязательным при описании заголовка является указание

типа результата,
имени метода,
круглых скобок,

наличие которых необходимо и в том случае, если сам список формальных аргументов отсутствует.

Формально тип результата метода указывается всегда, но значение void однозначно определяет, что метод реализуется процедурой.

Тип результата, отличный от void, указывает на функцию.

Вот несколько простейших примеров описания методов:

```
void A() {...};
int B(){...};
public void C(){...};
```

Методы A и B являются закрытыми, а метод C - открыт.

Методы A и C реализованы процедурами, а метод B - функцией, возвращающей целое значение.

Тело метода

Синтаксически тело метода является блоком, который представляет собой последовательность операторов и описаний переменных, заключенную в фигурные скобки.

Если речь идет о теле функции, то в блоке должен быть хотя бы один оператор перехода, возвращающий значение функции в форме return (выражение).

Доступ к методам

Вызов метода объекта очень похож на обращение к полю. После имени объекта ставится точка, затем имя метода и скобки.

В скобках перечисляются аргументы, разделенные запятыми.

Таким образом, методы класса Motorcycle можно вызывать так, как показано в следующем примере.

```
class TestMotorcycle : Motorcycle
{

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();   //Конструктор

        moto.StartEngine();   //Процедура
        moto.AddGas(15);      //Процедура
        moto.Drive(5, 20);    //Процедура
        double speed = moto.GetTopSpeed();    //Функция
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

Чтобы использовать возвращаемое методом значение в вызываемом методе, вызов метода можно поместить в любое место кода, где требуется значение соответствующего типа. Кроме того, возвращаемое значение можно присвоить переменной.

Например, следующие два примера кода выполняют одну и ту же задачу.

```
int result = obj.AddTwoNumbers(1, 2);
obj.SquareANumber(result);
```

```
   obj.SquareANumber(obj.AddTwoNumbers(1, 2)); //Передача результата от одного
                                                //метода другому
```

Вызов метода. Синтаксис

Как уже отмечалось, метод может вызываться в выражениях или быть вызван как оператор.
В качестве оператора может использоваться любой метод - как процедура, так и функция.

Если же попытаться вызвать процедуру в выражении, то это приведет к ошибке еще на этапе компиляции. Возвращаемое процедурой значение void несовместимо с выражениями. Так что в выражениях могут быть вызваны только функции.

Сам вызов метода, независимо от того, процедура это или функция, имеет один и тот же синтаксис:
имя_метода([список_фактических_аргументов])

Если это оператор, то вызов завершается точкой с запятой.

Формальный аргумент, задаваемый при описании метода - это всегда имя аргумента (идентификатор).

Фактический аргумент - это выражение, значительно более сложная синтаксическая конструкция. Вот точный синтаксис фактического аргумента:
[ref|out]выражение

Вызов метода. Семантика

Что происходит в момент вызова метода?

Выполнение начинается с вычисления фактических аргументов, которые являются выражениями.

Вычисление этих выражений может приводить, в свою очередь, к вызову других методов, так что этот первый этап может быть довольно сложным и требовать больших временных затрат.

В чисто функциональном программировании все вычисление по программе сводится к вызову одной функции, фактическими аргументами которой являются вызовы функций и так далее и так далее.

Обмен данными с методом.

Возвращаемые значения

Методы могут возвращать значения вызывающим их объектам.

Если тип возвращаемого значения, указываемый перед именем метода, не равен void, для возвращения значения используется ключевое слово return.

В результате выполнения инструкции с ключевым словом return, после которого указано значение нужного типа, вызвавшему метод объекту будет возвращено это значение.

Кроме того, ключевое слово return останавливает выполнение метода.

Если тип возвращаемого значения void, инструкцию return без значения все равно можно использовать для завершения выполнения метода.

Если ключевое слово return отсутствует, выполнение метода завершится, когда будет достигнут конец его блока кода.

Для возврата значений методами с типом возвращаемого значения отличным от void необходимо обязательно использовать ключевое слово return.

Например, в следующих двух методах ключевое слово return служит для возврата целочисленных значений.

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

В следующем примере метод возвращает переменную Area в виде значения double.

```
class ReturnTest
{
    static double CalculateArea(int r)
    {
        double area = r * r * Math.PI;
        return area;
    }

    static void Main()
    {
        int radius = 5;
        double result = CalculateArea(radius);
        Console.WriteLine("The area is {0:0.00}", result);
    }
}
```

```
// Output: The area is 78.54
```

Параметры и аргументы методов

Определение метода задает имена и типы любых необходимых параметров.

Пример:

```
public void Caller()
{
    int numA = 4;
    // Вызов метода с передачей переменной типа int.
    int productA = Square(numA);              //Функция

    int numB = 32;
    // Вызов метода с передачей другой переменной типа int.
    int productB = Square(numB);

    // Вызов метода с передачей переменной типа целочисленного литерала.
    int productC = Square(12);

    // Вызов метода с передачей выражения, эквивалентного типу int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Сохранение входного аргумента в локальной переменной.
    int input = i;
    return input * input;
}
```

Список формальных аргументов

Список формальных аргументов метода может быть пустым, и это довольно типичная ситуация для методов класса.

Список может содержать фиксированное число аргументов, разделяемых символом запятой.

Рассмотрим синтаксис объявления формального аргумента:
[ref|out|params]тип_аргумента имя_аргумента

Обязательным является указание типа и имени аргумента.

Никаких ограничений на тип аргумента не накладывается. Он может быть любым скалярным типом, массивом, классом, структурой, интерфейсом, перечислением, функциональным типом.

Несмотря на фиксированное число формальных аргументов, есть возможность при вызове метода передавать ему произвольное число фактических аргументов.

Для реализации этой возможности в списке формальных аргументов необходимо задать ключевое слово params.

Ключевое слово params позволяет определить параметр метода, принимающий аргумент, в котором количество аргументов является переменным.

В объявлении метода после ключевого слова params дополнительные параметры не допускаются, и в объявлении метода допускается только одно ключевое слово params.

Пример

```
public class MyClass
{
    public static void UseParams(params int[] list)
//Здесь list - это список параметров типа «массив целых чисел»
    {
        for (int i = 0 ; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
```

```
            Console.WriteLine();
        }

        public static void UseParams2(params object[] list)
    //Здесь list - это список параметров типа «объект»
        {
            for (int i = 0 ; i < list.Length; i++)
            {
                Console.Write(list[i] + " ");
            }
            Console.WriteLine();
        }

        static void Main()
        {
            UseParams(1, 2, 3);     //Создание массива
            UseParams2(1, 'a', "test");  //Использование массива

            // An array of objects can also be passed, as long as
            // the array type matches the method being called.
            int[] myarray = new int[3] {10,11,12};
            UseParams(myarray);
        }
    }
    /*
    Output:
        1 2 3
        1 a test
        10 11 12
    */
```

Содержательно, все аргументы метода разделяются на три группы: входные, выходные и обновляемые.

Аргументы первой группы передают информацию методу, их значения в теле метода только читаются.

Аргументы второй группы представляют собой результаты метода, они получают значения в ходе работы метода.

Аргументы третьей группы выполняют обе функции. Их значения используются в ходе вычислений и обновляются в результате работы метода.

Выходные аргументы всегда должны сопровождаться ключевым словом out, обновляемые - ref.

Что же касается входных аргументов, то, как правило, они задаются без ключевого слова, хотя иногда их полезно объявлять с параметром ref.

Если аргумент объявлен как выходной с ключевым словом out, то в теле метода обязательно должен присутствовать оператор присваивания, задающий значение этому аргументу. В противном случае возникает ошибка еще на этапе компиляции.

Передача параметров

В C# параметры могут быть переданы либо по значению, либо по ссылке.

Переменная типа значения содержит данные непосредственно, в противоположность переменной ссылочного типа, которая содержит ссылку на данные.

Поэтому передача переменной типа значения методу означает передачу методу копии переменной.

Любые изменения параметра, выполняемые внутри метода, не влияют на исходные данные, хранимые в переменной.

Если требуется, чтобы вызываемый метод изменял значение параметра, его следует передавать ссылкой с помощью ключевого слова ref или out.

Для простоты в следующем примере использовано ключевое слово ref.

Пример

В следующем примере демонстрируется передача параметров типа значения с помощью значения.

Переменная n передается с помощью значения в метод SquareIt.

Любые изменения, выполняемые внутри метода, не влияют на значение переменной.

```
class PassingValByVal
{
    static void SquareIt(int x)
    // Параметр x передаётся по значению.
    // Изменения x не влияют на его исходное значение.
    {
        x *= x;
        System.Console.WriteLine("Значение внутри метода: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("Значение перед вызовом метода: {0}", n);

        SquareIt(n);  // Passing the variable by value.
        System.Console.WriteLine("Значение после вызова метода: {0}", n);

        // Удерживаем окно консоли открытым.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Значение перед вызовом метода: 5
    Значение внутри метода: 25
    Значение после вызова метода: 5
*/
```

Переменная n, имеющая тип значения, содержит данные, значение 5.

При вызове метода SquareIt содержимое переменной n копируется в параметр x, который возводится в квадрат внутри метода.

Однако в методе Main значение переменной n остается одинаковым до и после вызова метода SquareIt.

Фактически, изменение, выполняемое внутри метода, влияет только на локальную переменную x.

Следующий пример похож на предыдущий за исключением того, что в нем параметр передается с помощью ключевого слова ref.

После вызова метода значение параметра изменяется.

```
class PassingValByRef
{
    static void SquareIt(ref int x)
    // Параметр x передаётся по значению.
    // Изменения x приведёт к изменению его исходного значения.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}",
n);
```

```
        SquareIt(ref n);  // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}",
n);

        // Удерживаем окно консоли открытым.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Значение перед вызовом метода: 5
    Значение внутри метода: 25
    Значение после вызова метода: 25
*/
```

В этом примере передается не значение переменной n а ссылка на переменную n.

Параметр x не является типом int; он является ссылкой на тип int, в данном случае ссылкой на переменную n.

Поэтому при возведении в квадрат параметра x внутри метода фактически в квадрат возводится переменная, на которую ссылается параметр x — переменная n.

Распространенным примером замены значений передаваемых параметров является метод Swap, в который передаются две переменные, x и y, и метод меняет местами их содержимое.

Параметры необходимо передавать в метод Swap с помощью ссылки.

В противном случае внутри метода будут использоваться локальные копии параметров.

Ниже приведен пример использования ссылочных параметров в методе Swap.

```
static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

При вызове этого метода следует использовать ключевое слово ref следующим образом.

```
static void Main()
{
    int i = 2, j = 3;
    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    SwapByRef (ref i, ref j);

    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
/* Output:
    i = 2  j = 3
    i = 3  j = 2
*/
```

Ключевое слово this.

Допустим, в классе «a» создан объект «в1» класса «в», в методе «d» которого есть переменная «с». Она локальная, существующая только в методе объекта «в1».

В классе «а» тоже есть переменная «с», которая передаётся методу «в1.d» в виде параметра:

```
в1.d(int c);   //с - это переменная класса «а»
{
this.c=c;        //this.c - это переменная текущего объекта, т.е. объекта «в1»
…
}
```

Таким образом, слово this – это ссылка на объект, которая всегда ссылается на текущий объект.

Примеры. Задача 1. Методы в одном файле.

Постановка задачи.

Дана вещественная таблица размером NxM элементов. Размер таблицы вводится с клавиатуры. Поменять местами строки таблицы по правилу:

строка с номером 0 меняется с последней, строка с номером 1 с предпоследней и т.д.

Пример: Таблица исходная -> 1.0 2.0 3.0 4.0
5.0 6.0 7.0 8.0
9.0 10.0 11.0 12.0

Таблица перевернутая -> 9.0 10.0 11.0 12.0
5.0 6.0 7.0 8.0
1.0 2.0 3.0 4.0

Таблица рассматривается как двухмерный массив. Набирается таблица поэлементно, а вводится построчно.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Пример
{
    class Перестановка
    {
        static void Прочитать(double[,] a, int n, int m)
        {
            int i, j;
            for (i = 0; i < n; i++)
                for (j = 0; j < m; j++)
                {
                    Console.Write("Элемент[{0},{1}]: ", i, j);
                    a[i, j] = double.Parse(Console.ReadLine());
                }
        }

        static void Вывести(double[,] a, int n, int m, string формат)
        {
            int i, j;
            for (i = 0; i < n; i++, Console.WriteLine())
                for (j = 0; j < m; j++)
                    Console.Write(формат, a[i, j]);
        }


        //Входные параметры
        //  a   -  исходная таблица
        //  n – количество строк
        //  ном1 - номер первой строки в паре
        //  ном2 - номер второй строки в паре
```

```
        //  m -   количество столбцов
        static void Переставить(double[,] a, int ном1, int ном2, int m)
        {
            int    j;    // номер столбца
            double b;    // буфер для временного хранения элемента таблицы

            for (j = 0; j < m; j++)
            {
                b = a[ном1, j];
                a[ном1, j] = a[ном2, j];
                a[ном2, j] = b;
            }
        }

        static void Main(string[] args)
        {
            int       n,     //Количество строк
                      m,     //Количество столбцов
                      mp,    //Количество пар строк
                      ном1,  //Номер первой строки в паре
                      ном2;  //Номер второй строки в паре
            double[,] a;     //Исходная таблица

            ConsoleKeyInfo клавиша; //Нажатая пользователем клавиша

            do
            {
                Console.Clear();
                Console.Write("Сколько строк: ");
                n = int.Parse(Console.ReadLine());
                Console.Write("Сколько столбцов: ");
                m = int.Parse(Console.ReadLine());

                a = new double[n, m];
                Прочитать(a, n, m);
                Console.WriteLine("\nИсходная таблица");
                Вывести(a, n, m, "{0,8:f2}");

                mp = n/2;
                for(ном1=0,ном2=n-1; ном1<mp; ном1++,ном2--)
                    Переставить(a, ном1, ном2 ,m);

                Console.WriteLine("\nТаблица после перестановки строк");
                Вывести(a, n, m, "{0,8:f2}");

                Console.WriteLine("\nДля выхода нажмите клавишу ESC");
                клавиша = Console.ReadKey(true);
            } while (клавиша.Key != ConsoleKey.Escape);
        }

    }
}
/*
Пример работы программы:
Сколько строк: 4
Сколько столбцов: 4
Элемент[0,0]: 1
Элемент[0,1]: 2
Элемент[0,2]: 3
Элемент[0,3]: 4
Элемент[1,0]: -1
Элемент[1,1]: -2
Элемент[1,2]: -3
Элемент[1,3]: -4
```

```
Элемент[2,0]: 11
Элемент[2,1]: 22
Элемент[2,2]: 33
Элемент[2,3]: 44
Элемент[3,0]: 111
Элемент[3,1]: 222
Элемент[3,2]: 333
Элемент[3,3]: 444

Исходная таблица
    1,00    2,00    3,00    4,00
   -1,00   -2,00   -3,00   -4,00
   11,00   22,00   33,00   44,00
  111,00  222,00  333,00  444,00

Таблица после перестановки строк
  111,00  222,00  333,00  444,00
   11,00   22,00   33,00   44,00
   -1,00   -2,00   -3,00   -4,00
    1,00    2,00    3,00    4,00

Для выхода нажмите клавишу ESC

*/
```

Структура программы.

В программе используются методы: **Прочитать, Вывести, Переставить, Main.** Все методы находятся в классе **Перестановка** пространства имён **Пример.**

Задача 2. Методы в отдельном файле

Постановка задачи.

```
Дана вещественная матрица размером NxM элементов. Размер матрицы вводится  с
клавиатуры (M не больше 10). Отсортировать строки матрицы по возрастанию
 их поэлементных сумм. Матрица заполняется случайными числами в диапазоне от
 -50 до +50. Пример:
 Матрица          10   2   3   4 --> 19
                   8   1   1   3 --> 13
                   2   3   1   4 --> 10

 Отсортированная   2   3   1   4 --> 10
 матрица           8   1   1   3 --> 13
                  10   2   3   4 --> 19

 Для ввода-вывода используются методы класса IOArray.
Для сортировки определить  класс с методами:
 - sum:    вычисление сумм строк и их запоминание в массиве
 - sort:   сортировка методом "пузырька"
 ВНИМАНИЕ: Матрица рассматривается как массив, состоящий из массивов, что
 позволяет при сортировке не выполнять физической перестановки строк,
 а ограничиться только перестановкой ссылок
```

Для того, чтобы при создании проекта в Visual Studio подключить оба файла необходимо дважды добавить тексты файлов через Add New Item.

Текст программы.

```
//Основной файл
using System;

namespace fvt
{
class MyArray
```

```
{
 public static void sum(double[][] a,  double[] s)
   {
       int i,   //Номер строки
             j;   //Номер столбца

       for(i=0; i<a.Length; i++)
          for(s[i]=0,j=0; j<a[i].Length; j++)
                   s[i] += a[i][j];

      }//Конец определения метода sum

 public static void sort(double[][] a, double[] s)
   {
      int i,            //Номер строки
            n;      //Количество строк
      bool flag;  //Индикатор перестановки
    double buf;
    double[] b;
      n = a.Length;
      do
       {
             n--;
             for(flag=false,i=0; i<n; i++)
                   if(s[i] > s[i+1])
                      {
                           b = a[i];
                           a[i] = a[i+1];
                           a[i+1] = b;
                           buf = s[i];
                           s[i] = s[i+1];
                           s[i+1] = buf;
                           flag = true;
                      }
       } while(flag);

   }//Конец определения метода sort
}//Конец объявления класса MyArray


class MyMetod
{
 public static void Main()
  {

   double[][]  a;           //Формируемая матрица
   double[]    s;           //Суммы строк
   double      min = 1.0,   //Нижняя граница диапазона генерируемых чисел
                 max = 5.0;   //Верхняя граница диапазона генерируемых чисел
   int         n,           //Число строк
                 m,           //Число столбцов
                 i;
   char        rep;         //Признак повторного выполнения
   string      sinp;        //Строка для приема данных

   do
       {
       Console.Write("Строк:     ");
       sinp = Console.ReadLine();
         n = int.Parse(sinp);
         Console.Write("Столбцов:  ");
         sinp = Console.ReadLine();
         m = int.Parse(sinp);
```

```
        a = new double[n][];
        for(i=0; i<n; i++)
         a[i] = new double[m];
        s = new double[n];

        IOArray.rand(a,min,max);
        IOArray.print(a,"{0,8:f2}");
      Console.WriteLine();

        MyArray.sum(a,s);
        MyArray.sort(a,s);

        IOArray.print(a,"{0,8:f2}");

        Console.Write("\nДля повтора намите клавишу Y: ");
        rep = char.Parse(Console.ReadLine());
        Console.WriteLine();
      }while(rep == 'Y' || rep == 'y');
  } //Конец определения метода Main
} //Конец объявления класса

//Файл IOARRAY.CS

//using System;
class IOArray
{
 public static void print(int[,] a, int n, int m, string fmt)
  {
      int i,j;
      for(i=0; i<n; i++, Console.WriteLine())
        for(j=0; j<m; j++)
            Console.Write(fmt,a[i,j]);
  }   //Конец объявления метода print(метод перегружен)

 public static void print(int[][] a, string fmt)
  {
      int i,j;
      for(i=0; i<a.Length; i++, Console.WriteLine())
      for(j=0; j<a[i].Length; j++)
           Console.Write(fmt,a[i][j]);
  }   //Конец объявления метода print(метод перегружен)

 public static void print(double[][] a, string fmt)
  {
      int i,j;
      for(i=0; i<a.Length; i++, Console.WriteLine())
         for(j=0; j<a[i].Length; j++)
              Console.Write(fmt,a[i][j]);
  }   //Конец объявления метода print(метод перегружен)

 public static void print(double[,] a, int n, int m, string fmt)
  {
      int i,j;
      for(i=0; i<n; i++, Console.WriteLine())
                 for(j=0; j<m; j++)
                       Console.Write(fmt,a[i,j]);
  }   //Конец объявления метода print(метод перегружен)

 public static void print(int[] a, int n, string fmt)
  {
      int i;
      for(i=0; i<n; i++)
         Console.Write(fmt,a[i]);
  }   //Конец объявления метода print(метод перегружен)
```

```
 public static void print(double[] a, int n, string fmt)
      {
            int i;
            for(i=0; i<n; i++)
                  Console.Write(fmt,a[i]);
      }     //Конец объявления метода print(метод перегружен)

 public static void rand(int[,] a, int n, int m, int min, int max)
   {
        Random ran = new  Random();
        int i,j;
        for(i=0; i<n; i++)
          for(j=0; j<m; j++)
              a[i,j] = ran.Next(min,max+1);
   }  //Конец объявления метода rand(метод перегружен)

 public static void rand(int[][] a, int min, int max)
  {
      Random ran = new  Random();
      int i,j;
      for(i=0; i<a.Length; i++)
        for(j=0; j<a[i].Length; j++)
            a[i][j] = ran.Next(min,max+1);
  }   //Конец объявления метода rand(метод перегружен)

public static void rand(double[][] a,  double min, double max)
   {
        Random ran = new  Random();
        int i,j;
        for(i=0; i<a.Length; i++)
            for(j=0; j<a[i].Length; j++)
               a[i][j] = min + (max-min)*ran.NextDouble();
   }  //Конец объявления метода rand(метод перегружен)

public static void rand(double[,] a,  int n, int m, double min, double max)
      {
            Random ran = new  Random();
            int i,j;
            for(i=0; i<n; i++)
                  for(j=0; j<m; j++)
                        a[i,j] = min + (max-min)*ran.NextDouble();
      }     //Конец объявления метода rand(метод перегружен)


}//Конец объявления класса IOArray
/*
L:\2>2
Строк:     5
Столбцов:  5
    1,78    1,19    2,27    1,75    2,87
    3,07    4,61    1,91    3,76    3,98
    2,82    2,48    2,56    4,27    3,48
    1,49    2,33    1,56    1,27    1,65
    3,60    3,78    2,96    2,56    3,90

    1,49    2,33    1,56    1,27    1,65
    1,78    1,19    2,27    1,75    2,87
    2,82    2,48    2,56    4,27    3,48
    3,60    3,78    2,96    2,56    3,90
    3,07    4,61    1,91    3,76    3,98

Для повтора намите клавишу Y:
```

```
Строк:
*/
}
```

Задание: составьте программу отдельного метода:

Напишите программу метода 1,

которому для работы передаётся одномерный массив данных. Проверьте работоспособность метода.

Напишите программу метода 2,

которому для работы передаётся по ссылке двумерный массив данных и по ссылке же возвращается такой же массив. Проверьте работоспособность метода.

Напишите программу метода 3,

функции для сложения двух целых чисел.

Main() и аргументы командной строки

Метод Main является точкой входа консольного приложения C# или приложения Windows. При запуске приложения метод Main является первым вызываемым методом. Кроме консольных приложений и приложений Windows в программах встречаются такие элементы, как библиотеки и службы. Для последних не требуется метод Main в качестве точки входа и они его просто не содержат.

В программе C# возможна только одна точка входа. Пример:

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments:
        System.Console.WriteLine(args.Length);
    }
}
```

Если в наличии имеется больше одного класса, который имеет метод Main, то необходимо скомпилировать программу с параметром компилятора **/main**, в котором следует указать, какой метод Main нужно использовать в качестве точки входа.

Общие сведения

- Метод Main является точкой входа EXE-программы, в которой начинается и заканчивается управление программой.
- Метод Main объявлен внутри класса или структуры. Main должен быть статический и он не должен быть открытый. (В предыдущем примере он получает доступ по умолчанию типа закрытый.) Включающий класс или структура не обязательно должна быть статической.
- Main может иметь возвращаемый тип либо void, либо int.
- Метод Main может быть объявлен с параметром string[], который содержит аргументы командной строки, или без него. При использовании Visual Studio для создания приложений Windows Forms, можно добавить параметр вручную или использовать класс Environment для получения аргументов командной строки. Индексация считываемых параметров командной строки начинается с нуля. В отличие от C и C++, имя программы не рассматривается как первый аргумент командной строки.

Аргументы командной строки

Можно отправлять аргументы методу Main, указав метод одним из следующих способов:

```
static int Main(string[] args)
```

или

```
static void Main(string[] args)
```

Параметр метода Main является массивом значений типа String, представляющим аргументы командной строки. Обычно наличие аргументов определяется проверкой свойства Length, например:

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

Кроме того, строковые аргументы можно преобразовать в числовые типы с помощью класса Convert или метода **Parse**. Например, следующая инструкция преобразует string в число типа long с помощью метода Parse:
long num = Int64.Parse(args[0]);

Также можно использовать тип long языка C#, являющийся псевдонимом типа Int64:
long num = long.Parse(args[0]);

Для выполнения этой операции также можно воспользоваться методом ToInt64 класса Convert:
long num = Convert.ToInt64(s);

Пример

В следующем примере показано, как аргументы командной строки можно использовать в консольном приложении. Приложение принимает аргументы по одному, преобразует каждый из них в целое число и вычисляет факториал этого числа. Если ни одного аргумента не предоставлено, приложение выдает сообщение, содержащее разъяснение правильного использования данной программы.

Для компиляции и запуска приложения из командной строки выполните следующие действия.

1. Вставьте следующий код в любой текстовый редактор и сохраните его как текстовый файл с именем Factorial.cs.

```
//Add a using directive for System if the directive isn't already
present.

public class Functions
{
    public static long Factorial(int n)
    {
        // Test for invalid input
        if ((n < 0) || (n > 20))
        {
            return -1;
```

```
        }

        // Calculate the factorial iteratively rather than recursively:
        long tempResult = 1;
        for (int i = 1; i <= n; i++)
        {
            tempResult *= i;
        }
        return tempResult;
    }
}

class MainClass
{
    static int Main(string[] args)
    {
        // Test if input arguments were supplied:
        if (args.Length == 0)
        {
            System.Console.WriteLine("Please enter a numeric
argument.");
            System.Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Try to convert the input arguments to numbers. This will
throw
        // an exception if the argument is not a number.
        // num = int.Parse(args[0]);
        int num;
        bool test = int.TryParse(args[0], out num);
        if (test == false)
        {
            System.Console.WriteLine("Please enter a numeric
argument.");
            System.Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Calculate factorial.
        long result = Functions.Factorial(num);

        // Print result.
        if (result == -1)
            System.Console.WriteLine("Input must be >= 0 and <= 20.");
        else
            System.Console.WriteLine("The Factorial of {0} is {1}.",
num, result);

        return 0;
    }
}
// If 3 is entered on command line, the
// output reads: The factorial of 3 is 6.
```

2. На экране Пуск или в меню Пуск откройте окно Командная строка разработчика Visual Studio, а затем перейдите в папку, содержащую созданный файл.
3. Введите следующую команду для компиляции приложения.

   csc Factorial.cs

Если приложение не содержит ошибок компиляции, создается исполняемый файл с именем Factorial.exe.

4. Для вычисления факториала 3 введите следующую команду:

   Factorial 3

5. Выходные результаты команды будут следующими: The factorial of 3 is 6.

```
D:\>cd D:\! 3 !

D:\! 3 !>csc Factorial.cs
Microsoft (R) Visual C# 2008 Compiler version 3.5.21022.8
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.


D:\! 3 !>Factorial
Please enter a numeric argument.
Usage: Factorial <num>

D:\! 3 !>Factorial 5
The Factorial of 5 is 120.

D:\! 3 !>
```

Отображение аргументов командной строки

Для доступа к аргументам, предоставленным для исполняемого файла в командной строке, можно использовать необязательный параметр для Main. Аргументы представляют собой массив строк. Каждый элемент массива содержит один аргумент. Пробел между элементами удален. Например, рассмотрим следующие вызовы вымышленного исполняемого файла из командной строки.

Данные, вводимые в командную строку	Массив строк, переданный в Main
executable.exe a b c	"a" "b" "c"
executable.exe один два	"один" "два"
executable.exe "один два" три	"один два" "три"

Пример

В этом примере показаны аргументы командной строки, переданные в приложение командной строки. Далее представлен результат для первой записи в расположенной выше таблице.

```
class CommandLine
{
```

```
    static void Main(string[] args)
    {
        // The Length property provides the number of array elements
        System.Console.WriteLine("parameter count = {0}", args.Length);

        for (int i = 0; i < args.Length; i++)
        {
            System.Console.WriteLine("Arg[{0}] = [{1}]", i, args[i]);
        }
    }
}
/* Output (assumes 3 cmd line args):
    parameter count = 3
    Arg[0] = [a]
    Arg[1] = [b]
    Arg[2] = [c]
*/
```

Доступ к аргументам командной строки с помощью оператора foreach

Другим методом итерации всех элементов массива является использование оператора foreach, как показано в следующем примере. Оператор foreach можно использовать для итерации всех элементов массива, класса коллекции .NET Framework или любого класса или структуры, реализующих интерфейс IEnumerable.

Пример

В данном примере показано, как напечатать аргументы командной строки с помощью оператора foreach.

```
// arguments: John Paul Mary


class CommandLine2
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("Number of command line parameters = {0}",
args.Length);

        foreach (string s in args)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Number of command line parameters = 3
    John
    Paul
    Mary
*/
```

Значения, возвращаемые методом Main()

Метод Main может возвращать значение void:

```
static void Main()
{
    //...
```

```
}
```

Он также может возвращать значение типа int:

```
static int Main()
{
    //...
    return 0;
}
```

Если значение, возвращаемое методом Main, не используется, то указание в качестве возвращаемого типа void несколько упрощает код. Однако возврат целого значения позволяет программе передавать информацию о своем состоянии другим программам и скриптам, которые вызывают исполняемый файл. В следующем примере показано, как получить доступ к значению, возвращаемому методом Main.

Пример

В этом примере с помощью пакетного файла запускается программа, после чего проверяется значение, возвращаемое функцией Main. При запуске программы в Windows значение, возвращаемое функцией Main, сохраняется в переменной среды, которая называется ERRORLEVEL. Пакетный файл может определить результат выполнения посредством проверки значения переменной ERRORLEVEL. В большинстве случаев на успешное выполнение указывает нулевое значение.

В следующем примере показана простая программа, в которой функция Main возвращает ноль. Нулевое значение указывает на успешное выполнение программы. Сохраните программу в файле MainReturnValTest.cs.

```
// Save this program as MainReturnValTest.cs.
class MainReturnValTest
{
    static int Main()
    {
        //...
        return 0;
    }
}
```

Поскольку в этом примере используется пакетный файл, рекомендуется выполнять компиляцию кода с помощью командной строки. Выполните инструкции из раздела Практическое руководство. Задание переменных среды для включения построений из командной строки или воспользуйтесь командной строкой Visual Studio, которую можно открыть с помощью пункта Средства Visual Studio в меню Пуск. В командной строке перейдите в папку, в которой сохранена программа. С помощью показанной ниже команды выполняется компиляция файла MainReturnValTest.cs и создается исполняемый файл MainReturnValTest.exe.

```
csc MainReturnValTest.cs
```

Далее создайте пакетный файл для запуска файла MainReturnValTest.exe и вывода результата. Вставьте следующий код в текстовый файл и сохраните его под именем test.bat в папке, содержащей файлы MainReturnValTest.cs и MainReturnValTest.exe. Введите в командной строке команду test, чтобы запустить командный файл.

Поскольку код возвращает нулевое значение, пакетный файл сообщает об успехе. Однако, если изменить файл MainReturnValTest.cs, чтобы он возвращал ненулевое значение, и затем перекомпилировать программу, при последующем выполнении пакетного файла будет выведено сообщение о неудаче.

```
rem test.bat
@echo off
```

```
MainReturnValTest
@if "%ERRORLEVEL%" == "0" goto good

:fail
    echo Execution Failed
    echo return value = %ERRORLEVEL%
    goto end

:good
    echo Execution succeeded
    echo Return value = %ERRORLEVEL%
    goto end

:end
/*

Execution succeeded

Return value = 0

*/
```

main (параметры компилятора C#)

Если метод **Main** содержится в нескольких классах, данный параметр указывает класс, который содержит точку входа в программу.

```
/main:class
```

Аргументы

class

Тип, в котором содержится метод **Main**.

Заметки

Если в компиляцию включено несколько типов с методом Main, можно указать, какой тип содержит метод **Main**, который необходимо использовать в качестве точки входа в программу.

Этот параметр используется при компиляции EXE-файлов.

Установка данного параметра компилятора в среде разработки Visual Studio

1. Откройте страницу Свойства проекта.
2. Выберите страницу свойств Приложение.
3. Измените значение свойства Начальный объект.

Пример

Компиляция файлов t2.cs и t3.cs; указывается, что метод **Main** находится в классе Test2:

```
csc t2.cs t3.cs /main:Test2
```

Построение и выполнение примеров параметров командной строки в среде Visual Studio

1. В обозревателе решений щелкните правой кнопкой мыши проект "CmdLine1" и выберите команду Назначить автозагружаемым проектом.

2. В обозревателе решений щелкните проект правой кнопкой мыши проект и выберите пункт Свойства.
3. Откройте папку Свойства конфигурации и щелкните Отладка.
4. В поле свойства Аргументы командной строки введите параметры командной строки и нажмите кнопку ОК. (Пример см. в руководстве.)
5. В меню Отладка выберите команду Запуск без отладки.
6. Повторите предыдущие действия для проекта "CmdLine2".

Делегаты.

В наши дни программная модель фон-Неймана, для которой характерно последовательное выполнение команд в том порядке, как они записаны в программе, устарела.

На смену ей пришло *программирование, управляемое событиями.*

Современная программа предоставляет пользователю интерфейс и ждет, когда он предпримет какое-либо действие. У пользователя богатый выбор таких действий.

Он может выбирать команды меню, нажимать кнопки, вносить изменения в текстовые поля, щелкать по значкам и т. д.

Каждое действие приводит к возникновению события.

Кроме того, существуют события, непосредственно не связанные с действиями пользователя, например срабатывание таймера, приход сообщения по электронной почте или окончание операции копирования файлов.

Событие инкапсулирует идею «произошло нечто важное», и программа должна на него отреагировать.

Для программирования, управляемого событиями в алгоритмическом языке C# введены специальные конструкции – делегаты и события.

События и делегаты являются тесно связанными понятиями, поскольку гибкая обработка событий требует точного выбора обработчика.

А обработчики событий реализуются, как правило, в виде делегатов.

Делегат предоставляет возможность инкапсулировать метод, а событие — это своего рода уведомление о том, что имело место некоторое действие.

Эти средства расширяют диапазон задач программирования, к которым можно применить язык C#.

Делегаты

Делегат (delegate) — это объект, который может ссылаться на метод.

Создавая делегат, вы создаете объект, который может содержать ссылку на метод. Благодаря этому делегат может вызывать метод, на который он ссылается.

На первый взгляд идея ссылки на метод может показаться странной, поскольку обычно мы имеем дело с ссылками, которые указывают на объекты, но в действительности здесь разница небольшая.

Ссылка по существу представляет собой адрес памяти. Следовательно, ссылка на объект — это адрес объекта.

Даже несмотря на то что метод не является объектом, он тоже имеет отношение к физической области памяти, а адрес его точки входа — это адрес, к которому происходит обращение при вызове метода.

Этот адрес можно присвоить делегату.

А раз делегат ссылается на метод, этот метод можно вызвать посредством данного делегата.

Во время выполнения программы один и тот же делегат можно использовать для вызова различных методов, просто заменив метод, на который ссылается этот делегат.

Благодаря этому, метод, который будет вызван делегатом, определяется не в период компиляции программы, а во время ее работы.

В этом и состоит достоинство делегата.

Делегат объявляется с помощью ключевого слова delegate.

Общая форма объявления делегата имеет следующий вид:

delegate *тип_возврата имя(список_параметров)*;

Здесь элемент *тип_возврата* представляет собой тип значений, возвращаемых методами, которые этот делегат будет вызывать.

Имя делегата указывается элементом *имя*.

Параметры, принимаемые методами, которые вызываются посредством делегата, задаются с помощью элемента *список_параметров*.

Делегат может вызывать только такие методы, у которых тип возвращаемого значения и список параметров (т.е. его сигнатура) совпадают с соответствующими элементами объявления делегата.

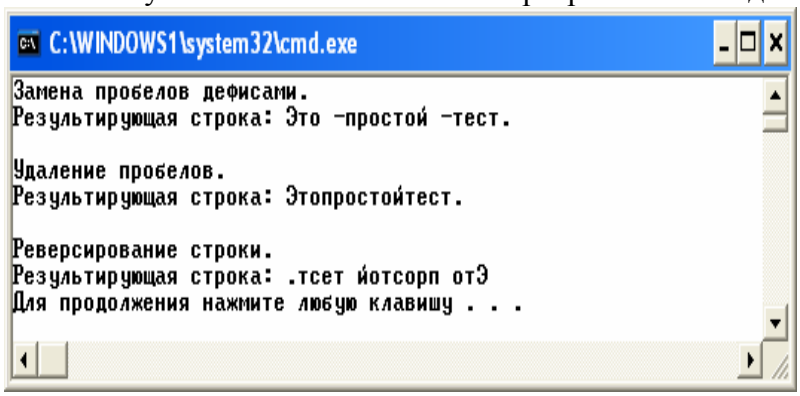
Делегат может вызывать либо **метод экземпляра класса**, связанный с объектом, или **статический метод**, связанный с классом.

Чтобы увидеть делегат в действии, начнем с простого примера.

```
// Простой пример использования делегата.
using System;
// Объявляем делегат:
delegate string strMod(string stx);

class DelegateTest
{
    // Метод заменяет в тексте пробелы дефисами:
    static string replaceSpaces(string a)
    {
        Console.WriteLine("Замена пробелов дефисами.");
        return a.Replace(" ", " -");
    }

    // Метод удаляет пробелы:
    static string removeSpaces(string a) {
    string temp = "";
    int i;
    Console.WriteLine("Удаление пробелов.");
    for(i=0; i < a.Length; i++)
    if (a[i] != ' ') temp += a[i];
    return temp;
}

    // Метод реверсирует строку:
    static string reverse(string a)
    {
        string temp = "";
        int i, j;
        Console.WriteLine("Реверсирование строки.");
        for (j = 0, i = a.Length - 1; i >= 0; i--, j++)
            temp += a[i];
        return temp;
    }

    public static void Main()
    {
        // Создание делегата:
        strMod strOp = new strMod(replaceSpaces);
        string str;
```

```
        // Вызов методов посредством делегата:
        str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();
        strOp = new strMod(removeSpaces);
        str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();
        strOp = new strMod(reverse);
        str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
    }
    }
```

Результаты выполнения этой программы выглядят так:



Итак, в программе объявляется делегат с именем strMod, который принимает один параметр типа s t r i n g и возвращает string-значение.

В классе DelegateTest объявлены три статических метода, сигнатура которых совпадает с сигнатурой, заданной делегатом.

Эти методы предназначены для модификации строк определенного вида.

Обратите внимание на то, что метод replaceSpaces () для замены пробелов дефисами использует метод Replace () — один из методов класса string.

В методе Main () создается ссылка типа strMod с именем strOp, и ей присваивается ссылка на метод replaceSpaces ().

Внимательно рассмотрите следующую строку:

strMod strOp = new strMod(replaceSpaces);

Обратите внимание на то, что метод replaceSpaces () передается делегату в качестве параметра.

Здесь используется только имя метода (параметры не указываются).

Это наблюдение можно обобщить: при реализации делегата задается только имя метода, на который должен ссылаться этот делегат.

Кроме того, объявление метода должно соответствовать объявлению делегата. В противном случае вы получите сообщение об ошибке еще во время компиляции.

Затем метод replaceSpaces () вызывается посредством экземпляра делегата с именем strOp, как показано в следующей строке:

str = strOp("Это простой т е с т . ") ;

Поскольку экземпляр strOp ссылается на метод replaceSpaces(), то вызывается именно метод replaceSpaces().

Затем экземпляру делегата strOp присваивается ссылка на метод removeSpaces(), после чего strOp вызывается снова.

На этот раз вызывается метод removeSpaces().

Наконец, экземпляру делегата strOp присваивается ссылка на метод reverse (), и strOp вызывается еще раз.

Это, как нетрудно догадаться, приводит к вызову метода reverse().

Главное в этом примере то, что вызов экземпляра делегата s t r o p трансформируется в обращение к методу, на который ссылается strOp при вызове.

Таким образом, решение о вызываемом методе принимается во время выполнения программы, а не в период компиляции.

Несмотря на то что в предыдущем примере используются статические методы, делегат может также ссылаться на методы экземпляров класса.

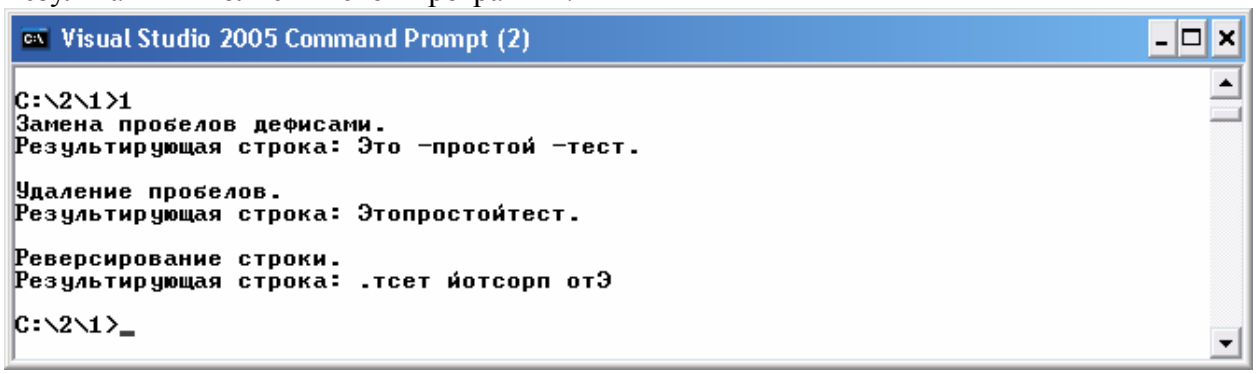
Однако он должен при этом использовать объектную ссылку.

Например, вот как выглядит предыдущая программа, переписанная с целью инкапсуляции операций над строками внутри класса StringOps: (program 1)

```
// Простой пример использования делегата.
using System;
// Объявляем делегат:
delegate string strMod(string stx);

class DelegateTest
{
    // Метод заменяет в тексте пробелы дефисами:
    static string replaceSpaces(string a)
    {
        Console.WriteLine("Замена пробелов дефисами.");
        return a.Replace(" ", " -");
    }

    // Метод удаляет пробелы:
    static string removeSpaces(string a) {
    string temp = "";
    int i;
    Console.WriteLine("Удаление пробелов.");
    for(i=0; i < a.Length; i++)
    if (a[i] != ' ') temp += a[i];
    return temp;
}

    // Метод реверсирует строку:
    static string reverse(string a)
    {
        string temp = "";
        int i, j;
        Console.WriteLine("Реверсирование строки.");
        for (j = 0, i = a.Length - 1; i >= 0; i--, j++)
            temp += a[i];
        return temp;
    }

    public static void Main()
    {
        // Создание делегата:
        strMod strOp = new strMod(replaceSpaces);
        string str;
        // Вызов методов посредством делегата:
        str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();
        strOp = new strMod(removeSpaces);
        str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();
        strOp = new strMod(reverse);
        str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
```

```
    }
  }
```

Результаты выполнения этой программы:



совпадают с результатами предыдущей версии, но в этом случае делегат ссылается на методы экземпляра класса StringOps.

Многоадресатная передача

Одна из самых интересных возможностей делегата — поддержка *многоадресатной передачи* (multicasting).

Выражаясь простым языком, Многоадресатная передача — это способность создавать список вызовов (или цепочку вызовов) методов, которые должны автоматически вызываться при вызове делегата.

Такую цепочку создать нетрудно.

Достаточно создать экземпляр делегата, а затем для добавления методов в эту цепочку использовать оператор "+=".

Для удаления метода из цепочки используется оператор " - = " . (Можно также для добавления и удаления методов использовать в отдельности операторы "+", " - " и "=", но чаще применяются составные операторы "+=" и "-=".)

Делегат с многоадресатной передачей имеет одно ограничение: он должен возвращать тип void.

Рассмотрим следующий пример многоадресатной передачи.

```
// Делегаты могут ссылаться также на методы экземпляров класса.
using System;
// Объявляем делегат.
delegate string strMod(string str);

class StringOps {
```

```
// Метод заменяет пробелы дефисами:
public string replaceSpaces(string a) {
Console.WriteLine("Замена пробелов дефисами.");
return a.Replace(' ', '-');
}
```

```
// Метод удаляет пробелы:
    public string removeSpaces(string a)
    {
        string temp = "";
        int i;
        Console.WriteLine("Удаление пробелов. ");
        for (i = 0; i < a.Length; i++)
            if (a[i] != ' ') temp += a[i];
        return temp;
```

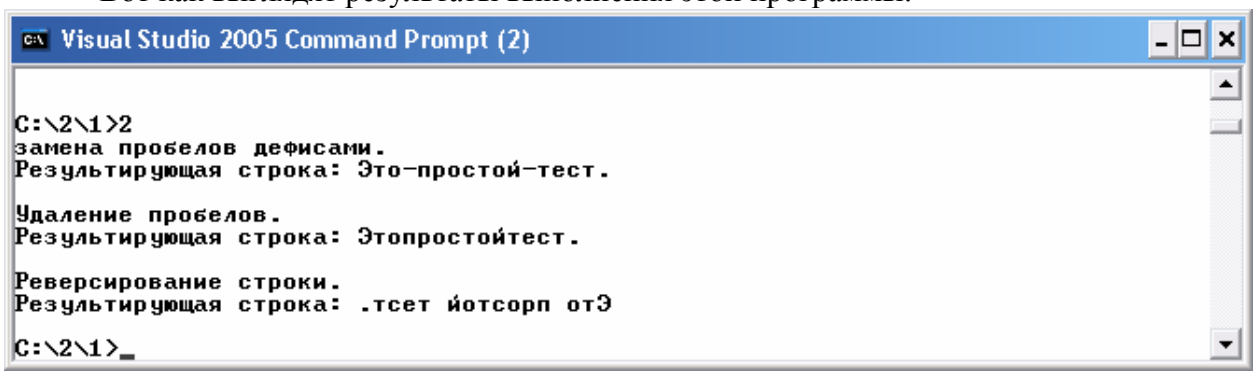
```
    }
```

```
// Метод реверсирует строку:
public string reverse(string a) {
string temp = "";
int i,j;
Console.WriteLine("Реверсирование строки.");
 for(j=0, i=a.Length-1; i >= 0; i--, j++)
temp += a[i];
return temp;
}
```

```
}
class DelegateTest {
public static void Main() {
StringOps so = new StringOps(); // Создаем экземпляр класса StringOps.
// Создаем делегат:
strMod strOp = new strMod(so.replaceSpaces);
string str;
// Вызываем методы с использованием делегатов:
str = strOp("Это простой тест.");
Console.WriteLine("Результирующая строка: " + str);
Console.WriteLine();
strOp = new strMod(so.removeSpaces);
str = strOp("Это простой тест.");
Console.WriteLine("Результирующая строка: " + str);
Console.WriteLine();
strOp = new strMod(so.reverse);
str = strOp("Это простой тест.");
Console.WriteLine("Результирующая строка: " + str);
}
}
```

Это — переработанный вариант предыдущих примеров, в котором тип s t r i n g для значений, возвращаемых методами обработки строк, заменен типом void, а для возврата модифицированных строк используется ref-параметр.

Вот как выглядят результаты выполнения этой программы:



В методе Main() создаются четыре экземпляра делегата.

Первый, strOp, имеет null-значение.

Три других ссылаются на методы модификации строк.

Затем организуется делегат для многоадресатной передачи, который вызывает методы removeSpaces () и reverse ().

Это достигается благодаря следующим строкам программы:

strOp = repiaceSp;
strOp += reverseStr;

Сначала делегату strOp присваивается ссылка replaceSp.

Затем, с помощью оператора "+=", в цепочку вызовов добавляется ссылка reverseStr.

При вызове делегата strOp в этом случае вызываются оба метода, заменяя пробелы дефисами и реверсируя строку.

Затем при выполнении строки программы

strOp -== replaceSp;

из цепочки вызовов удаляется ссылка replaceSp, а с помощью строки

strOp += removeSp;

в цепочку вызовов добавляется ссылка removeSp.

Затем делегат StrOp вызывается снова.

На этот раз из исходной строки удаляются пробелы, после чего она реверсируется.

Цепочки вызовов, организованные с помощью делегата, — мощный механизм, который позволяет определять набор методов, выполняемых "единым блоком".

Класс System.Delegate

Все делегаты представляют собой классы, которые неявным образом выводятся из класса System. Delegate.

Обычно его члены не используются напрямую.

Все же в некоторых ситуациях его члены могут оказаться весьма полезными.

Назначение делегатов

Несмотря на то что предыдущие примеры программ продемонстрировали, "как" работают делегаты, они не содержали ответа на вопрос "зачем это нужно?".

Так вот, делегаты используются по следующим причинам:

- они дают возможность определять вызываемый метод не при компиляции, а динамически во время выполнения программы;
- обеспечивают связь между объектами по типу «источник — наблюдатель»;
- позволяют создать универсальные методы, в которые можно передавать другие методы;
- поддерживают механизм обратных вызовов.

Рассмотрим сначала пример реализации первой из этих целей (получения возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы).

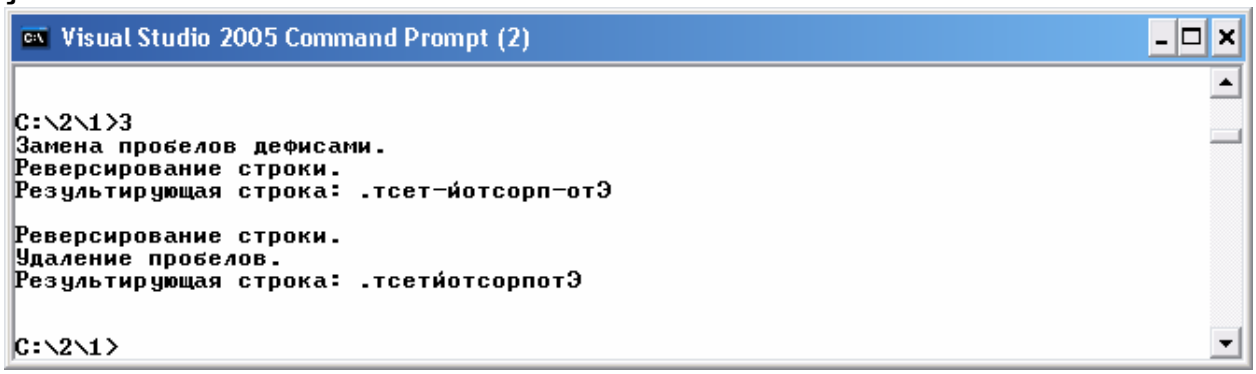
В листинге 3 объявляется делегат, с помощью которого один и тот же оператор используется для вызова двух разных методов (C00l и Hack).

```
// Демонстрация использования многоадресатной передачи.
using System;
// Объявляем делегат:
delegate void strMod(ref string str);

class StringOps
{
    // Метод заменяет пробелы дефисами:
    static void replaceSpaces(ref string a)
    {
        Console.WriteLine("Замена пробелов дефисами.");
        a = a.Replace(' ', '-');
    }
```

```
    // Метод удаляет пробелы:
    static void removeSpaces(ref string a)
    {
        string temp = "";
        int i;
        Console.WriteLine("Удаление пробелов.");
        for (i = 0; i < a.Length; i++)
            if (a[i] != ' ')
                temp += a[i];
        a = temp;
    }
```

```
    // Метод реверсирует строку:
    static void reverse(ref string a)
    {
        string temp = "";
        int i, j;
        Console.WriteLine("Реверсирование строки.");
        for (j = 0, i = a.Length - 1; i >= 0; i--, j++)
            temp += a[i];
        a = temp;
    }
```

```
    public static void Main()
    {
        // Создаем экземпляры делегатов:
        strMod strOp;
        strMod replaceSp = new strMod(replaceSpaces);
        strMod removeSp = new strMod(removeSpaces);
        strMod reverseStr = new strMod(reverse);
        string str = "Это простой тест.";
        // Организация многоадресатной передачи:
        strOp = replaceSp;
        strOp += reverseStr;
        // Вызов делегата с многоадресатной передачей:
        strOp(ref str);
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();
        // Удаляем метод замены пробелов и добавляем метод их удаления:
        strOp -= replaceSp;
        strOp += removeSp;
        str = "Это простой тест."; // Восстановление исходной строки:
        // Вызов делегата с многоадресатной передачей:
        strOp(ref str);
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();
    }
}
```



В начале программы **объявляется делегат**:
«delegate void Del (ref string s);»

Этим объявлением задаются его сигнатура и имя типа.

Затем **создаётся один или несколько методов такой же сигнатуры**: в рассматриваемом примере – методы C00l и Hack.

После этого в методе Main **создаётся экземпляр объявленного делегата** – при этом ему присваивается имя и делегат связывается с некоторым методом.

Сигнатура делегата и метода, с которым он связывается, должна быть одинаковой.

Создание экземпляра делегата включает в себя

объявление экземпляра делегата (Del d;) //- этим задаётся имя экземпляра

Инициализация (или регистрация), т.е. связывание делегата с методом

(например, d = new Del(C00l);)

После этого созданный и проинициализированный экземпляр делегата **запускается** в работу

«d(ref s);».

В цикле он запускается дважды, каждый раз – с новым загруженным в него методом.

Использование делегата имеет тот же синтаксис, что и вызов метода.

Результат работы программы:

c001 hackers

C001 hAcKeRs

Жизненный цикл делегата состоит из следующих событий:

1. В начале программы **объявляется делегат.** Этим объявлением задаются его сигнатура и имя типа.
2. Затем **создаётся один или несколько методов такой же сигнатуры**. Методы могут создаваться в разных частях программы, в разных классах.
3. После этого в методе Main **создаётся экземпляр объявленного делегата** – при этом ему присваивается имя и делегат связывается с некоторым методом.
   Сигнатура делегата и метода, с которым он связывается, должна быть одинаковой.
4. После этого созданный и проинициализированный экземпляр делегата **запускается** в работу.

Если делегат хранит ***ссылки на несколько методов***, они вызываются последовательно в том порядке, в котором были добавлены в делегат.

Добавление метода в список выполняется либо с помощью метода Combine, унаследованного от класса System.Delegate, либо, что удобнее, с помощью **перегруженной операции сложения**.

Вот как выглядит измененный метод Main из предыдущего листинга, в котором одним вызовом делегата выполняется преобразование исходной строки сразу двумя методами.

```
static void Main()
{
  string s = "cool ackers";

//создание экземпляра делегата
  Del d = new Del( C00l );
```

```
// добавление нового метода в экземпляр делегата:
d += new Del( Hack );

  //запуск делегата
d( ref s );
Console.WriteLine( s );
}

 // результат:
C001 hAcKeRs
```

Паттерн «наблюдатель»

Рассмотрим применение делегатов для обеспечения связи между объектами по типу «источник — наблюдатель».

В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных объектов.

При этом желательно избежать жесткой связанности классов, так как это часто негативно сказывается на возможности многократного использования кода.

Для обеспечения связи между объектами во время выполнения программы применяется следующая стратегия.

Объект, называемый *источником*, при изменении своего состояния, которое может представлять интерес для других объектов, посылает им уведомления о происшедшем событии. Получившие уведомление объекты называются *наблюдателями*.

Получив уведомление, наблюдатель запускает свой обработчик этого события – так он реагирует на событие.

Программисты часто используют одну и ту же схему организации и взаимодействия объектов в разных контекстах.

За такими схемами закрепилось название *паттерны*, или *шаблоны проектирования*.

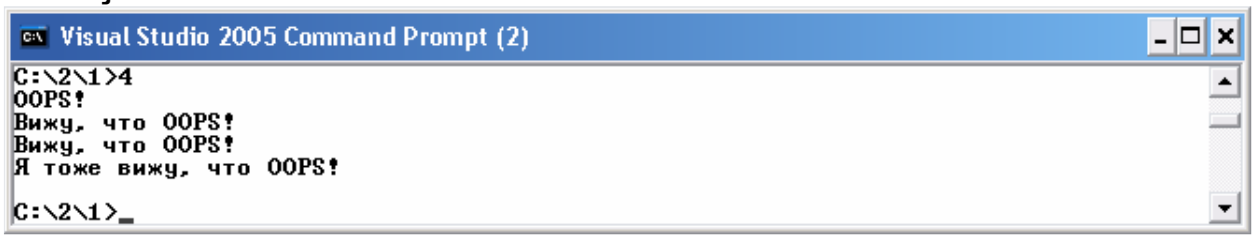
Описанная стратегия известна под названием *паттерн «наблюдатель»*.

Наблюдатель (observer) определяет между объектами зависимость типа «один ко многим», так что при изменении состояния одного объекта все зависящие от него объекты получают извещение и автоматически обновляются.

Рассмотрим пример (listing 4), в котором демонстрируется схема оповещения источником трех наблюдателей.

```
using System;
namespace ConsoleApplication1
{
public delegate void Del( object o );  // объявление делегата

class Subj                              // класс-источник
    {
Del dels;                          // объявление экземпляра делегата

public void Register( Del d )          // регистрация делегата
        {
            dels += d;
        }

public void OOPS()                    // что-то произошло
        {
            Console.WriteLine( "OOPS!" );
if ( dels != null ) dels( this );  // оповещение наблюдателей
        }
```

```
    }

class ObsA                          // класс-наблюдатель
    {
public void Do( object o )          // реакция на событие источника
        {
            Console.WriteLine( "Вижу, что OOPS!" );
        }
    }

class ObsB                          // класс-наблюдатель
    {
public static void See( object o )  // реакция на событие источника
        {
            Console.WriteLine( "Я тоже вижу, что OOPS!" );
        }
    }

    class Class1
    {
        static void Main()
        {
Subj s  = new Subj();               //  объект класса-источника

ObsA o1 = new ObsA();               //                  объекты
ObsA o2 = new ObsA();               //       класса-наблюдателя

s.Register( new Del( o1.Do ) );     //      регистрация методов
s.Register( new Del( o2.Do ) );     // наблюдателей в источнике
s.Register( new Del( ObsB.See ) );  //  ( экземпляры делегата )

s.OOPS();                           //    инициирование события
        }
    }
}
```



Гипотетическое изменение состояния объекта моделируется сообщением «OOPS!».

Один из методов в демонстрационных целях сделан статическим.

В источнике объявляется экземпляр делегата, в этот экземпляр заносятся методы тех объектов, которые хотят получать уведомление об изменении состояния источника.

Этот процесс называется *регистрацией делегатов*.

При регистрации имя метода добавляется к списку.

При наступлении «часа X» все зарегистрированные методы поочередно вызываются через делегат.

Результат работы программы:

OOPS!
Вижу, что OOPS!
Вижу, что OOPS!
Я тоже вижу, что OOPS!

Для обеспечения обратной связи между наблюдателем и источником делегат объявлен с параметром типа object, через который в вызываемый метод передается ссылка на вызывающий объект.

Следовательно, в вызываемом методе можно получать информацию о состоянии вызывающего объекта и посылать ему сообщения.

Связь «источник — наблюдатель» устанавливается во время выполнения программы для каждого объекта по отдельности.

Если наблюдатель больше не хочет получать уведомления от источника, можно удалить соответствующий метод из списка делегата с помощью метода Remove или перегруженной операции вычитания, например:

```
public void UnRegister( Del d )       // удаление делегата
{
    dels -= d;
}
```

Передача делегатов в методы

Поскольку делегат является классом, его можно передавать в методы в качестве параметра.

Таким образом обеспечивается *функциональная параметризация*: в метод можно передавать не только различные данные, но и различные функции их обработки.

Функциональная параметризация применяется для создания универсальных методов и обеспечения возможности обратного вызова.

В качестве простейшего примера *универсального метода* можно привести метод вывода таблицы значений функции, в который передается диапазон значений аргумента, шаг его изменения и вид вычисляемой функции.

Пример приведен в листинге 5.

```
//Listing5
using System;
namespace ConsoleApplication1
{
public delegate double Fun( double x );    // объявление делегата

    class Class1
    {
        public static void Table( Fun F, double x, double b )
        {
            Console.WriteLine( " ----- X ----- Y -----" );
            while (x <= b)
            {
      Console.WriteLine( "| {0,8:0.000} | {1,8:0.000} |", x, F(x));
               x += 1;
            }
            Console.WriteLine( " ---------------------" );
        }

        public static double Simple( double x )
        {
            return 1;
        }

        static void Main()
        {
            Console.WriteLine( " Таблица функции Sin " );
            Table( new Fun( Math.Sin ), -2, 2 );

            Console.WriteLine( " Таблица функции Simple " );
            Table( new Fun( Simple ), 0, 3 );
        }
    }
}
```

Результат работы программы:

```
Таблица функции Sin
 ----- X ----- Y -----
|    -2,000 |    -0,909 |
|    -1,000 |    -0,841 |
|     0,000 |     0,000 |
|     1,000 |     0,841 |
|     2,000 |     0,909 |
 ----------------------
 Таблица функции Simple
 ----- X ----- Y -----
|     0,000 |     1,000 |
|     1,000 |     1,000 |
|     2,000 |     1,000 |
|     3,000 |     1,000 |
 ----------------------
```

Обратный вызов

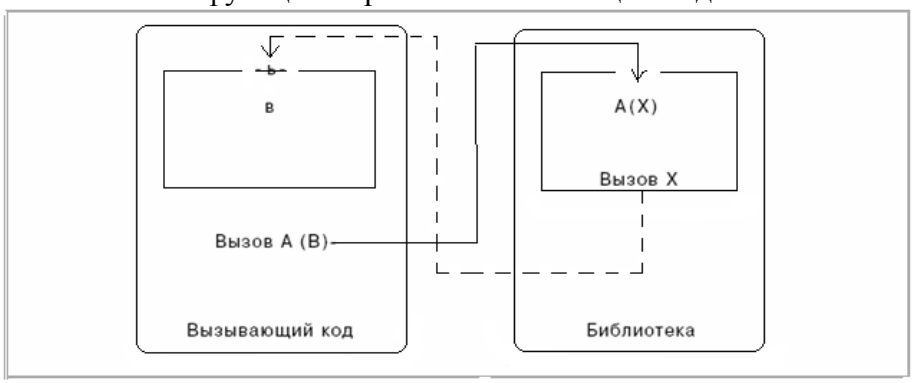
Обратный вызов (callback) представляет собой вызов функции, передаваемой в другую функцию в качестве параметра.

Рассмотрим рисунок (Механизм обратного вызова).

Допустим, в библиотеке описана функция A, параметром которой является имя другой функции.

В вызывающем коде описывается функция с требуемой сигнатурой (B) и передается в функцию A.

Выполнение функции A приводит к вызову B, то есть управление передается из библиотечной функции обратно в вызывающий код.



Механизм обратного вызова широко используется в программировании.

Например, он реализуется во многих стандартных функциях Windows.

В среде Visual Studio, использующей версию языка C#, можно применять упрощенный синтаксис для делегатов.

Первое упрощение заключается в том, что в большинстве случаев явным образом создавать экземпляр делегата не требуется, поскольку он создается автоматически по контексту.

Второе упрощение заключается в возможности создания так называемых *анонимных методов* — фрагментов кода, описываемых непосредственно в том месте, где используется делегат:

```
        static void Main()
        {
            Console.WriteLine( " Таблица функции Sin " );
    Table( Math.Sin, -2, 2 );                          // упрощение 1

            Console.WriteLine( " Таблица функции Simple " );
    Table( delegate (double x ){ return 1; }, 0, 3 ); // упрощение 2
        }
    }
}
```

В первом случае экземпляр делегата, соответствующего функции Sin, создается автоматически.
Чтобы это могло произойти, список параметров и тип возвращаемого значения функции должны быть совместимы с делегатом.
Во втором случае не требуется оформлять простой фрагмент кода в виде отдельной функции Simple, как это было сделано в предыдущем листинге, — код функции оформляется как анонимный метод и встраивается прямо в место передачи.

Приложение 1. Delegate - члены

Открытые свойства

Method	Получает метод, представленный делегатом.
Target	Получает экземпляр класса, из которого текущий делегат вызывает метод.

Открытые методы

Clone	Создает неполную копию делегата.
Combine	Перегружен. Последовательно соединяет списки вызовов заданных групповых (комбинированных) делегатов.
CreateDelegate	Перегружен. Создает делегат указанного типа.
DynamicInvoke	Динамически вызывает (с поздней привязкой) метод, представленный текущим делегатом.
Equals	Переопределен. Определяет, использует ли заданный объект и текущий одиночный (некомбинированный) делегат общий целевой объект, метод и список вызовов.
GetHashCode	Переопределен. Возвращает хеш-код делегата.
GetInvocationList	Возвращает список вызовов делегата.
GetObjectData	Реализует интерфейс ISerializable и возвращает данные, необходимые для сериализации делегата.
GetType (унаследовано от **Object**)	Возвращает Type текущего экземпляра.
Remove	Удаляет последний экземпляр списка вызовов делегата из списка вызовов другого

	делегата.
RemoveAll	Удаляет все экземпляры списка вызовов одного делегата из списка вызовов другого делегата.
ToString (унаследовано от **Object**)	Возвращает String, который представляет текущий Object.

Открытые операторы

Оператора равенства	Определяет, являются ли два заданных делегата эквивалентными.
Оператора неравенства	Определяет, являются ли заданные делегаты эквивалентными.

Защищенные конструкторы

Delegate - конструктор	Перегружен. Инициализирует новый делегат.

Защищенные методы

CombineImpl	Последовательно соединяет списки вызовов заданного группового (комбинированного) делегата и текущего группового (комбинированного) делегата.
DynamicInvokeImpl	Динамически вызывает (с поздней привязкой) метод, представленный текущим делегатом.
Finalize (унаследовано от **Object**)	Переопределен. Позволяет объекту Object попытаться освободить ресурсы и выполнить другие завершающие операции, перед тем как объект **Object** будет уничтожен в процессе сборки мусора. В языках C# и C++ для функций финализации используется синтаксис деструктора.
GetMethodImpl	Получает статический метод, представленный текущим делегатом.
MemberwiseClone (унаследовано от **Object**)	Создает неполную копию текущего Object.
RemoveImpl	Удаляет список вызовов одного делегата из списка вызовов другого делегата.

События.

Событие (event) — это автоматическое уведомление о выполнении некоторого действия.

События работают следующим образом. Объект, которому необходима информация о некотором событии, регистрирует обработчик для этого события. Когда ожидаемое событие происходит, вызываются все зарегистрированные обработчики. Обработчики событий представляются делегатами.

События — это члены класса, которые объявляются с использованием ключевого слова event. Наиболее распространенная форма объявления события имеет следующий вид:

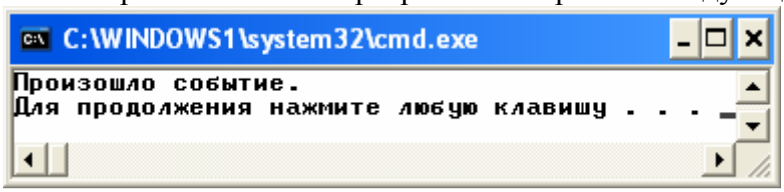
event *событийный_делегат объект;*

Здесь элемент *событийный_делегат* означает имя делегата, используемого для поддержки объявляемого события, а элемент объект — это имя создаваемого событийного объекта.

Начнем с рассмотрения очень простого примера.

```
// Демонстрация использования простейшего события.
using System;
// Объявляем делегат для события:
delegate void MyEventHandler();
// Объявляем класс события:
class MyEvent {
public event MyEventHandler SomeEvent;
// Этот метод вызывается для генерирования события:
public void OnSomeEvent() {
if(SomeEvent != null)
SomeEvent();
}
}
class EventDemo {
// Обработчик события:
    static void handler()
    {
        Console.WriteLine("Произошло событие.");
    }
public static void Main() {
MyEvent evt = new MyEvent();
// Добавляем метод handler() в список события:
evt.SomeEvent += new MyEventHandler(handler);
// Генерируем событие:
evt.OnSomeEvent();
}
}
```

При выполнении программа отображает следующие результаты:



Несмотря на простоту, программа содержит все элементы, необходимые для надлежащей обработки события. Рассмотрим их по порядку.

Программа начинается с такого объявления делегата для обработчика события:
delegate void MyEventHandler();

Все события активизируются посредством делегата. Следовательно, событийный делегат определяет сигнатуру для события. В данном случае параметры отсутствуют,

однако событийные параметры разрешены. Поскольку события обычно предназначены для многоадресатной передачи, они должны возвращать значение типа void.

Затем создается класс события MyEvent. При выполнении следующей строки кода, принадлежащей этому классу, объявляется событийный объект SomeEvent:

public event MyEventHandler SomeEvent;

Кроме того, внутри класса MyEvent объявляется метод OnSomeEvent (), который в этой программе вызывается, чтобы сигнализировать о событии. (Другими словами, этот метод вызывается, когда происходит событие.) Как показано в следующем фрагменте кода, он вызывает обработчик события посредством делегата SomeEvent.

if(SomeEvent != null)
SomeEvent();

Обратите внимание на то, что обработчик события вызывается только в том случае, если делегат SomeEvent не равен null-значению. Поскольку другие части программы, чтобы получить уведомлении о событии, должны зарегистрироваться, можно сделать так, чтобы метод OnSomeEvent () был вызван до регистрации любого обработчика события. Чтобы предотвратить вызов null-объекта, событийный делегат необходимо протестировать и убедиться в том, что он не равен null-значению.

Внутри класса EventDemo создается обработчик события handler (). В этом примере обработчик события просто отображает сообщение, но ясно, что другие обработчики могли бы выполнять более полезные действия.

В методе Main() создается объект класса MyEvent, а метод handler () регистрируется в качестве обработчика этого события.

```
MyEvent evt = new MyEvent();
// Добавляем метод handler() в список события,
evt.SomeEvent += new MyEventHandler(handler);
```

Обработчик добавляется в список с использованием составного оператора "+=". Следует отметить, что события поддерживают только операторы "+=" и " - = " .

В нашем примере метод handler () является статическим, но в общем случае обработчики событий могут быть методами экземпляров классов. Наконец, при выполнении следующей инструкции "происходит" событие, о котором мы так много говорили:

```
I/ Генерируем событие.
evt.OnSomeEvent();
```

При вызове метода OnSomeEvent () вызываются все зарегистрированные обработчики событий. В данном случае зарегистрирован только один обработчик, но их могло бы быть и больше.

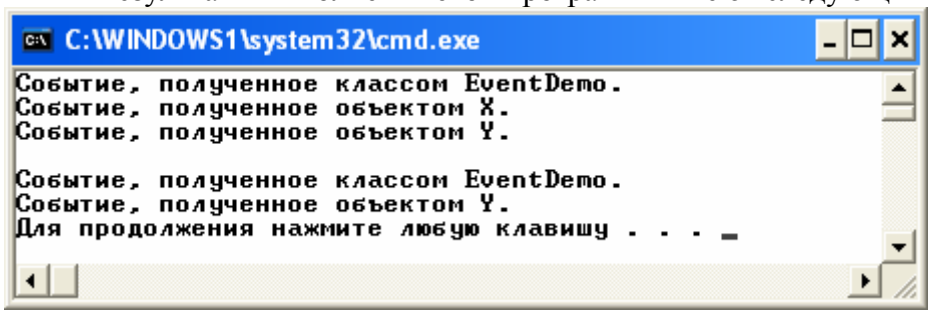
Пример события для многоадресатной передачи

Подобно делегатам события могут предназначаться для многоадресатной передачи. В этом случае на одно уведомление о событии может отвечать несколько объектов, рассмотрим пример.

```
// Демонстрация использования события, предназначенного
// для многоадресатной передачи.
using System;
// Объявляем делегат для события:
delegate void MyEventHandler();
// Объявляем класс события:
class MyEvent {
public event MyEventHandler SomeEvent;
// Этот метод вызывается для генерирования события:
public void OnSomeEvent() {
if(SomeEvent != null)
SomeEvent();
```

```
}
}
class X {
public void Xhandler() {
Console.WriteLine("Событие, полученное объектом X.");
}
}
class Y {
public void Yhandler() {
Console.WriteLine("Событие, полученное объектом Y.");
}
}
class EventDemo {
static void handler() {
Console.WriteLine("Событие, полученное классом EventDemo.");}
public static void Main() {
MyEvent evt = new MyEvent();
X xOb = new X();
Y yOb = new Y () ;
// Добавляем обработчики в список события:
evt.SomeEvent += new MyEventHandler(handler);
evt.SomeEvent += new MyEventHandler(xOb.Xhandler);
evt.SomeEvent += new MyEventHandler(yOb.Yhandler);
// Генерируем событие:
evt.OnSomeEvent();
Console.WriteLine();
// Удаляем один обработчик:
evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
evt.OnSomeEvent();
}
}
```

Результаты выполнения этой программы имеют следующий вид:



В этом примере создается два дополнительных класса X и Y, в которых также определяются обработчики событий, совместимые с сигнатурой делегата MyEventHandler.

Следовательно, эти обработчики могут стать частью цепочки событийных вызовов.

Обработчики в классах X и Y не являются статическими.

Это значит, что сначала должны быть созданы объекты каждого класса, после чего в цепочку событийных вызовов должен быть добавлен обработчик, связанный с каждым экземпляром класса. Различие между статическими обработчиками и обработчиками экземпляров классов рассматривается в следующем разделе.

Сравнение методов экземпляров классов со статическими методами, используемыми в качестве обработчиков событий

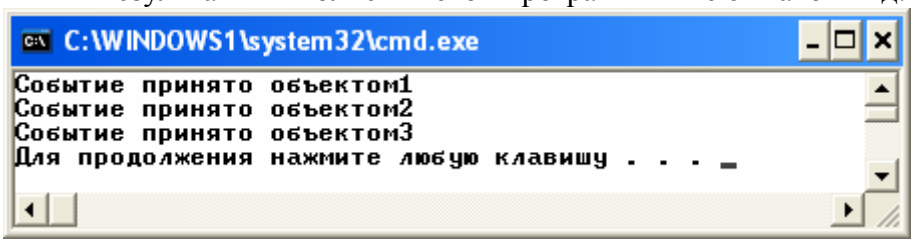
Несмотря на то что и методы экземпляров классов, и статические методы могут служить обработчиками событий, в их использовании в этом качестве есть существенные различия. Если в качестве обработчика используется статический метод, уведомление о событии применяется к классу (и неявно ко всем объектам этого класса). Если же в качестве обработчика событий используется метод экземпляра класса, события

посылаются к конкретным экземплярам этого класса. Следовательно, каждый объект класса, который должен получать уведомление о событии, необходимо регистрировать в отдельности. На практике в большинстве случаев "роль" обработчиков событий "играют" методы экземпляров классов, но, безусловно, все зависит от конкретной ситуации.

В следующей программе создается класс X, в котором в качестве обработчика событий определен метод экземпляра. Это значит, что для получения информации о событиях каждый объект класса X необходимо регистрировать отдельно. Для демонстрации этого факта программа готовит уведомление о событии для многоадресатной передачи трем обьектам типа X.

```
/* При использовании в качестве обработчиков событий
методов экземпляров уведомление о событиях принимают
отдельные объекты. */
using System;
// Объявляем делегат для события:
delegate void MyEventHandler();
// Объявляем класс события:
class MyEvent {
public event MyEventHandler SomeEvent;
// Этот метод вызывается для генерирования события:
public void OnSomeEvent() {
if(SomeEvent != null)
SomeEvent();
}
}
class X {
int id;
public X(int x) { id = x; }
// Метод экземпляра, используемый в качестве обработчика событий:
    public void Xhandler()
    {
        Console.WriteLine("Событие принято объектом" + id);
    }
}
class EventDemo {
public static void Main() {
MyEvent evt = new MyEvent();
X o1 = new X(1) ;
X o2 = new X(2) ;
X o3 = new X(3);
evt.SomeEvent += new MyEventHandler(o1.Xhandler);
evt.SomeEvent += new MyEventHandler(o2.Xhandler);
evt.SomeEvent += new MyEventHandler(o3.Xhandler);
// Генерируем событие,
evt.OnSomeEvent() ;
}
}
```

Результаты выполнения этой программы имеют такой вид:



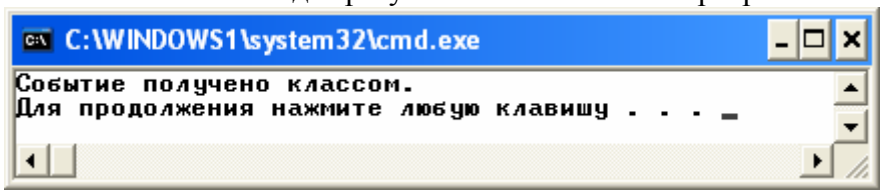
Как подтверждают эти результаты, каждый объект заявляет о своей заинтересованности в событии и получает о нем отдельное уведомление.

Если же в качестве обработчика событий используется статический метод, то, как показано в следующей программе, события обрабатываются независимо от объекта.

```
/* При использовании в качестве обработчиков событий
статического метода уведомление о событиях получает
```

```
класс. */
using System;
// Объявляем делегат для события:.
delegate void MyEventHandler();
// Объявляем класс события:
class MyEvent {
public event MyEventHandler SomeEvent;
// Этот метод вызывается для генерирования события:
public void OnSomeEvent() {
if(SomeEvent != null)
SomeEvent();
}
}
class X {
/* Это статический метод, используемый в качестве обработчика события: */
public static void Xhandler() {
Console.WriteLine("Событие получено классом.");
}
}
class EventDemo
{
    public static void Main()
    {
        MyEvent evt = new MyEvent();
        evt.SomeEvent += new MyEventHandler(X.Xhandler);
        // Генерируем событие,
        evt.OnSomeEvent();
    }
}
```

Вот как выглядят результаты выполнения программы:



Обратите внимание на то, что в программе не создается ни одного объекта типа x.

Но поскольку handler () — статический метод класса X, его можно связать с событием SomeEvent и обеспечить его выполнение при вызове метода OnSomeEvent ().

Смешанные средства обработки событий

События можно определять в интерфейсах.

"Поставкой" событий должны заниматься соответствующие классы.

События можно определять как абстрактные. Обеспечить реализацию такого события должен производный класс. Однако события, реализованные с использованием средств доступа add и remove, абстрактными быть не могут.

Любое событие можно определить с помощью ключевого слова sealed.

Событие может быть виртуальным, т.е. его можно переопределить в производном классе.

Рекомендации по обработке событий в среде .NET Framework

C# позволяет программисту создавать события любого типа. Однако в целях компонентной совместимости со средой .NET Framework необходимо следовать рекомендациям, подготовленным Microsoft специально для этих целей. Центральное место в этих рекомендациях занимает требование того, чтобы обработчики событий имели два параметра. Первый должен быть ссылкой на объект, который будет

генерировать событие. Второй должен иметь тип EventArgs и содержать остальную информацию, необходимую обработчику. Таким образом, .NET-совместимые обработчики событий должны иметь следующую общую форму записи:
void handler(object *source*, EventArgs *arg) {*

Обычно параметр source передается вызывающим кодом. Параметр типа EventArgs содержит дополнительную информацию, которую в случае ненадобности можнопроигнорировать.

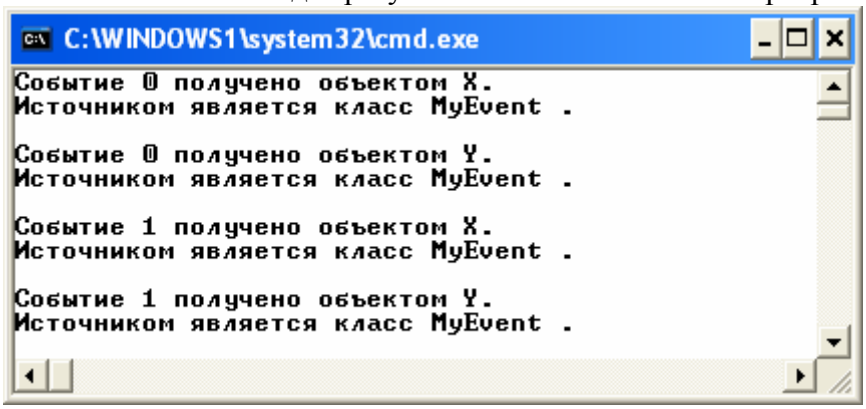
Класс EventArgs не содержит полей, которые используются при передаче дополнительных данных обработчику; он используется в качестве базового класса, из которого можно выводить класс, содержащий необходимые поля. Но поскольку многие обработчики обходятся без дополнительных данных, в класс EventArgs включено статическое поле Empty, которое задает объект, не содержащий никаких данных.

Ниже приведен пример, в котором создается .NET-совместимое событие.

```
// .NET-совместимое событие.
using System;
// Создаем класс, производный от класса EventArgs:
class MyEventArgs : EventArgs {
public int eventnum;
}
// Объявляем делегат для события:
delegate void MyEventHandler(object source,MyEventArgs arg);
// Объявляем класс события:
class MyEvent {
static int count = 0;
public event MyEventHandler SomeEvent;
// Этот метод генерирует SomeEvent-событие:
public void OnSomeEvent() {
MyEventArgs arg = new MyEventArgs();
if(SomeEvent != null) {
arg.eventnum = count++;
SomeEvent(this, arg);
}
}
class X {
public void handler(object source, MyEventArgs arg) {
Console.WriteLine("Событие " + arg.eventnum + " получено объектом X.");
Console.WriteLine("Источником является класс " + source + " . " ) ;
Console.WriteLine();
}
}
class Y {
public void handler(object source, MyEventArgs arg) {
Console.WriteLine("Событие " + arg.eventnum +
" получено объектом Y.");
Console.WriteLine("Источником является класс " + source + " . " ) ;
Console.WriteLine();
}
}
class EventDerno {
public static void Main() {
X obi = new X() ;
Y ob2 = new Y();
MyEvent evt = new MyEvent();
// Добавляем обработчик handler() в список событий:
evt. SomeEvent += new MyEventHandler (obi.handler) ;
evt.SomeEvent += new MyEventHandler(ob2.handler);
// Генерируем событие:
evt.OnSomeEvent();
evt.OnSomeEvent();
}
}
```

```
}
```

Вот как выглядят результаты выполнения этой программы:



В этом примере класс MyEventArgs выводится из класса EventArgs. В классе MyEventArgs добавлено только одно "собственное" поле — eventnum. В соответствии с требованиями .NET Framework делегат для обработчика событий MyEventHandler теперь принимает два параметра. Первый из них представляет собой объектную ссылку на генератор событий, а второй — ссылку на класс EventArgs или производный от класса EventArgs. В данном случае здесь используется ссылка на объект типа MyEventArgs.

Использование встроенного делегата EventHandler

Для многих событий параметр типа EventArgs не используется. Для упрощения процесса создания кода в таких ситуациях среда .NET Framework включает встроенный тип делегата, именуемый EventHandler. Его можно использовать для объявления обработчиков событий, которым не требуется дополнительная информация. Рассмотрим пример использования типа EventHandler.

```
// Использование встроенного делегата EventHandler.
using System;
// Объявляем класс события:
class MyEvent {
public event EventHandler SomeEvent;
// Здесь объявление использует делегат EventHandler.
// Следующий метод вызывается для генерирования SomeEvent-события:
public void OnSomeEvent() {
if(SomeEvent != null)
SomeEvent(this, EventArgs.Empty);
}
}
class EventDemo {
    static void handler(object source, EventArgs arg)
    {
        Console.WriteLine("Событие произошло.");
        Console.WriteLine("Источником является класс " + source + " . ");
    }
public static void Main() {
MyEvent evt = new MyEvent();
// Добавляем обработчик handler() в список событий:
evt.SomeEvent += new EventHandler(handler);
// Генерируем событие:
evt.OnSomeEvent();
}
}
```

В данном случае параметр типа EventArgs не используется и вместо него передается объект-заполнитель EventArgs.Empty. Результаты выполнения этой программы весьма лаконичны:

Учебный проект: использование событий

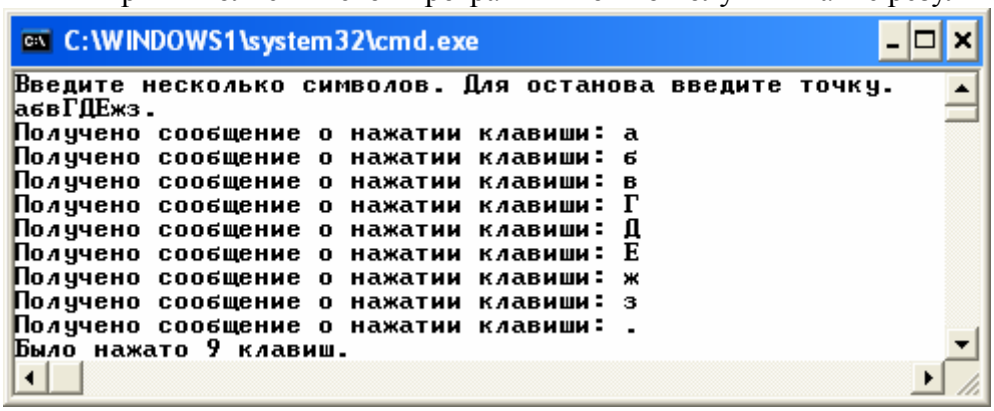
События часто используются в таких средах, как Windows с ориентацией на передачу сообщений. В подобной среде программа просто ожидает до тех пор, пока не получит сообщение, а затем выполняет соответствующие действия. Такая архитектура прекрасно подходит для обработки событий, позволяя создавать обработчики событий для различных сообщений и просто вызывать обработчик при получении определенного сообщения. Например, с некоторым событием можно было бы связать сообщение, получаемое в результате щелчка левой кнопкой мыши. Тогда после щелчка левой кнопкой мыши все зарегистрированные обработчики будут уведомлены о приходе этого сообщения.

Обрисуем в общих чертах работу этого механизма. В следующей программе создается обработчик событий нажатия клавиш. Событие называется KeyPress, и при каждом нажатии клавиши оно генерируется посредством вызова метода OnKeyPress ().

```
// Пример обработки события, связанного с нажатием клавиши на клавиатуре.
using System;
// Вводим собственный класс EventArgs, который будет хранить код клавиши:
class KeyEventArgs : EventArgs {
public char ch;
}
// Объявляем делегат для события:
delegate void KeyHandler(object source, KeyEventArgs arg);
// Объявляем класс события, связанного с нажатием клавиши на клавиатуре:
class KeyEvent
{
    public event KeyHandler KeyPress;
    // Этот метод вызывается при нажатии какой-нибудь клавиши:
    public void OnKeyPress(char key)
    {
        KeyEventArgs k = new KeyEventArgs();
        if (KeyPress != null)
        {
            k.ch = key;
            KeyPress(this, k);
        }
    }
}
// Класс, который принимает уведомления о нажатии клавиши:
class ProcessKey {
public void keyhandler(object source, KeyEventArgs arg) {
Console.WriteLine("Получено сообщение о нажатии клавиши: " + arg.ch);
}
}
// Еще один класс, который принимает уведомления о нажатии клавиши:
class CountKeys {
public int count = 0 ;
public void keyhandler(object source, KeyEventArgs arg) {
count++;
}
}
// Демонстрируем использование класса KeyEvent:
class KeyEventDemo {
public static void Main() {
KeyEvent kevt = new KeyEvent();
ProcessKey pk = new ProcessKey();
```

```
CountKeys ck = new CountKeys();
char ch;
kevt.KeyPress += new KeyHandler(pk.keyhandler);
kevt.KeyPress += new KeyHandler(ck.keyhandler);
Console.WriteLine("Введите несколько символов. " +
"Для останова введите точку.");
do {
ch = (char) Console.Read();
kevt.OnKeyPress(ch);
}
while(ch != '.');
Console.WriteLine("Было нажато " + ck.count + " клавиш.");
}
}
```

При выполнении этой программы можно получить такие результаты:



Эта программа начинается с создания класса-наследника KeyEventArgs, который используется для передачи сообщения о нажатии клавиши обработчику событий. Затем делегат KeyHandler определяет обработчик для событий, связанных с нажатием клавиши на клавиатуре. Эти события инкапсулируются в классе KeyEvent.

Программа для обработки нажатий клавиш создает два класса: ProcessKey и CountKeys. Класс ProcessKey включает обработчик с именем keyhandler (), который отображает сообщение о нажатии клавиши. Класс CountKeys предназначен для хранения текущего количества нажатых клавиш. В методе Main() создается объект класса KeyEvent. Затем создаются объекты классов ProcessKey и CountKeys, а ссылки на их методы keyhandler () добавляются в список вызовов, реализуемый с помощью событийного объекта kevt.KeyPress. Затем начинает работать цикл, в котором при каждом нажатии клавиши вызывается метод kevt.OnKeyPress(), в результате чего зарегистрированные обработчики уведомляются о событии.

Использование событийных средств доступа

Предусмотрены две формы записи инструкций, связанных с событиями. Форма, используемая в предыдущих примерах, обеспечивала создание событий, которые автоматически управляют списком вызова обработчиков, включая такие операции, как добавление обработчиков в список и удаление их из списка. Таким образом, можно было не беспокоиться о реализации операций по управлению этим списком. Поэтому такие типы событий, безусловно, являются наиболее применимыми. Однако можно и самим организовать ведение списка обработчиков событий, чтобы, например, реализовать специализированный механизм хранения событий.

Чтобы управлять списком обработчиков событий, используйте вторую форму event-инструкции, которая позволяет использовать средства доступа к событиям. Эти средства доступа дают возможность управлять реализацией списка обработчиков событий. Упомянутая форма имеет следующий вид:

```
event событийный_делегат имя__события {
add {
// Код добавления события в цепочку событий.
}
remove {
// Код удаления события из цепочки событий.
}
}
```

Эта форма включает два средства доступа к событиям: add и remove. Средство доступа add вызывается в случае, когда с помощью оператора "+=" в цепочку событий добавляется новый обработчик, а средство доступа remove вызывается, когда с помощью оператора " - = " из цепочки событий удаляется новый обработчик.

Средство доступа add или remove при вызове получает обработчик, который необходимо добавить или удалить, в качестве параметра. Этот параметр, как и в случае использования других средств доступа, называется value. При реализации средств доступа add и remove можно задать собственную схему хранения обработчиков событий.

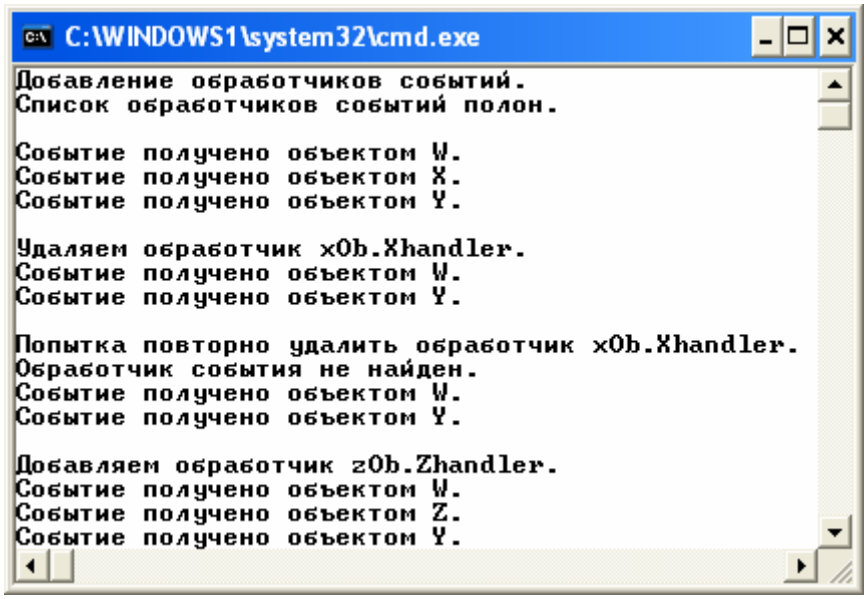
Например, для этого вы могли бы использовать массив, стек или очередь.

Рассмотрим пример использования событийных средств доступа. Здесь для хранения обработчиков событий взят массив. Поскольку этот массив содержит три элемента, в любой момент времени в событийной цепочке может храниться только три обработчика событий.

```
// Создание собственных средств управления списком событий.
using System;
// Объявляем делегат для события:
delegate void MyEventHandler();
// Объявляем класс события для хранения трех обработчиков событий:
class MyEvent {
MyEventHandler[] evnt = new MyEventHandler[3];
public event MyEventHandler SomeEvent {
// Добавляем обработчик события в список:
add {
int i;
for(i=0; i < 3; i++)
if(evnt[i] == null) {
evnt[i] = value;
break;
}
if (i == 3)
Console.WriteLine("Список обработчиков событий полон.");
}
// Удаляем обработчик события из списка:
remove {
int i ;
for(i=0; i < 3; i++)
if(evnt[i] == value){
evnt[i] = null;
break;
}
if (i == 3)
Console.WriteLine("Обработчик события не найден.");
}
}
// Этот метод вызывается для генерирования событий:
public void OnSomeEvent() {
for(int i=0; i < 3; i++)
if(evnt[i] != null) evnt[i]();
}
}
// Создаем классы, которые используют делегат MyEventHandler:
```

```
class W {
public void Whandler() {
Console.WriteLine("Событие получено объектом W.");
}
}
class X {
public void Xhandler() {
Console.WriteLine("Событие получено объектом X.");
}
}
class Y {
    public void Yhandler()
    {
        Console.WriteLine("Событие получено объектом Y.");
    }
}
class Z {
public void Zhandler() {
Console.WriteLine("Событие получено объектом Z.");
}
}
class EventDemo {
public static void Main() {
MyEvent evt = new MyEvent();
W wOb = new W();
X xOb = new X();
Y yOb = new Y () ;
Z zOb = new Z () ;
// Добавляем обработчики в список:
Console.WriteLine("Добавление обработчиков событий.");
evt.SomeEvent += new MyEventHandler(wOb.Whandler);
evt.SomeEvent += new MyEventHandler(xOb.Xhandler);
evt.SomeEvent += new MyEventHandler(yOb.Yhandler);
// Этот обработчик сохранить нельзя — список полон,
evt.SomeEvent += new MyEventHandler(zOb.Zhandler);
Console.WriteLine();
// Генерируем события:
evt.OnSomeEvent() ;
Console.WriteLine() ;
// Удаляем обработчик из списка:
Console.WriteLine("Удаляем обработчик xOb.Xhandler.");
evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
evt.OnSomeEvent();
Console.WriteLine() ;
// Пытаемся удалить его еще раз:
Console.WriteLine("Попытка повторно удалить обработчик xOb.Xhandler.");
evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
evt.OnSomeEvent();
Console.WriteLine();
// Теперь добавляем обработчик Zhandler:
Console.WriteLine("Добавляем обработчик zOb.Zhandler.");
evt. SomeEvent += new MyEventHandler (zOb.Zhandler) ;
evt.OnSomeEvent();
}
}
```

Вот результаты выполнения программы:



Рассмотрим внимательно код этой программы. Сначала определяется делегат обработчика события MyEventHandler. Код класса MyEvent, как показано в следующей инструкции, начинается с определения трехэлементного массива обработчиков событий evnt.

MyEventHandler[] evnt = new MyEventHandler[3];

Этот массив предназначен для хранения обработчиков событий, которые добавлены в цепочку событий. Элементы массива evnt инициализируются null-значениями по умолчанию.

Приведем event-инструкцию, в которой используются событийные средства доступа.

```
public event MyEventHandler SomeEvent {
// Добавляем обработчик события в список:
add {
int i;
for(i=0; i < 3; i++)
if(evnt[i] == null) {
evnt[i] = value;
break;
}
if (i == 3)
Console.WriteLine("Список обработчиков событий полон.");
}
// Удаляем обработчик события из списка:
remove {
int i ;
for(i=0; i < 3; i++)
if(evnt[i] == value){
evnt[i] = null;
break;
}
if (i == 3)
Console.WriteLine("Обработчик события не найден.");
}
}
```

При добавлении в список обработчика событий вызывается add-средство, и ссылка на этот обработчик (содержащаяся в параметре value) помещается в первый встретившийся неиспользуемый элемент массива evnt. Если свободных элементов нет, выдается сообщение об ошибке. Поскольку массив evnt рассчитан на хранение лишь трех элементов, он может принять только три обработчика событий. При удалении заданного обработчика событий вызывается remove-средство, и в массиве evnt выполняется поиск

ссылки на обработчик, переданной в параметре value. Если ссылка найдена, в соответствующий элемент массива помещается значение null , что равнозначно удалению обработчика из списка.

При генерировании события вызывается метод OnSomeEvent (). Он в цикле просматривает массив evnt, по очереди вызывая каждый обработчик событий.

Как показано в предыдущих примерах, при необходимости относительно нетрудно реализовать собственный механизм хранения обработчиков событий. Для большинства приложений все же лучше использовать стандартный механизм хранения, в котором не используются событийные средства доступа. Однако в определенных ситуациях форма event-инструкции, ориентированной на событийные средства доступа, может оказаться весьма полезной. Например, если в программе обработчики событий должны выполняться в порядке уменьшения приоритетов, а не в порядке их добавления в событийную цепочку, то для хранения таких обработчиков можно использовать очередь по приоритету.

Объектно-ориентированное программирование.

В отличии от событийнойй модели при объектно-ориентированном программировании необходимо учитывать дополнительные условия, предъявляющие к структуре программы специфические требования.

Характерные особенности объектно-ориентированных программ:

1. Объектная ориентация связана с использованием особой конструкции программы – объектов.

2. Объекты с похожими свойствами, то есть с одинаковыми наборами переменных состояния и методов, образуют класс.

3. Класс **(class)** – это шаблон, чертёж, схема объекта. Он определяет лишь типовые черты объектов.

4. **Объект (object)**- это конкретная реализация, экземпляр класса.

5. Для классов характерно, что методы и свойства взаимосвязаны, класс должен определять только одну логическую сущность. Обычно класс содержит Абстрактный тип данных - группу тесно связанных между собой данных и методов (функций), которые могут осуществлять операции над этими данными. Смешивать данные и методы их обработки для разных типов объектов в одном классе недопустимо.

6. В конечном итоге программа может содержать большое количество классов, как правило, слабо связанных между собой, или даже совсем не связанных.

7. Кроме этого в объектно-ориентированной программе должна существовать какая-то объединяющая конструкция, в которой из этих классов создаются объекты и в которой эти объекты живут, взаимодействуют, развиваются.

8. В языке C# такая конструкция создаётся в виде отдельного класса, отличительной чертой которого является находящийся в нём метод Main().

9. С метода Main() начинается исполнение любой самой сложной программы. Имя этого метода является зарезервированным. Главное его назначение – он является ведущим методом программного проекта, именно в нём должны создаваться, жить, развиваться и взаимодействовать все объекты.

10. Класс, в котором находится метод Main(), не имеет какого-либо специального имени, оно может быть любым, выбирает его пользователь. Кроме метода Main() в этом классе могут содержаться и другие методы и свойства, необходимые для работы программы и не имеющие отношения к другим классам проекта.

11. Для визуального отображения элементов управления создаётся специальный класс Windows.Forms.

12. Обычно Visual Studio размещает метод Main() в файле Program.cs, создавая для этого класс «static class Program».

Пример асинхронной объектно-ориентированной программы: Общая схема реализации паттерна «Издатель –Подписчик».

```
using System;
//-----------------------------------------------------------
//объявление делегата
 delegate void MyEventHandler();
//-----------------------------------------------------------
// объвление класса - издателя события
 class MyEvent {
 public event MyEventHandler SomeEvent;   //объявление события SomeEvent
 public void FireSomeEvent() {
 if(SomeEvent != null)         //проверка, можно ли зажигать событие
   SomeEvent();                 //метод для зажигания события SomeEvent
}
}
//конец класса - издателя
//-----------------------------------------------------------
// объявление класса - ресивера
 class Resiver {
 public static void handler()   {
    Console.WriteLine("Событие произошло");
                          }
}
// конец класса - ресивера
//-----------------------------------------------------------
// создание класса - демонстратора
 class EventDemo
  {
 public static void Main()   {
 // создание экземпляра класса - издателя
 MyEvent evt = new MyEvent();

// добавление обработчика события в общий список
 evt.SomeEvent += new MyEventHandler(Resiver.handler);

// зажигание события
evt.FireSomeEvent();

  }    // конец определения метода Main

}   // конец класса - демонстратора
```

В этом примере видна отличительная особенность объектно-ориентированной программы: объекты (т.е. экземпляры классов) создаются и взаимодействуют в классе, содержащем метод Main.

Пример синхронной объектно-ориентированной программы

```
//Учебный проект Шилдта:
using System;

class Stack {
  //  Эти члены закрытые.
  char[] stck; // Массив для хранения данных стека.
  int tos;     // Индекс вершины стека.

  // Создаем пустой класс Stack заданного размера.
  public Stack(int size) {
    stck = new char[size]; // Выделяем память для стека.
    tos = 0;
```

```
  }

  // Помещаем символы в стек.
  public void push(char ch) {
    if(tos==stck.Length) {
      Console.WriteLine(" -- Стек заполнен.");
      return;
    }

    stck[tos] = ch;
    tos++;
  }

  // Извлекаем символ из стека.
  public char pop() {
    if(tos==0) {
      Console.WriteLine(" -- Стек пуст.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }

  // Метод возвращает значение true, если стек полон.
  public bool full() {
    return tos==stck.Length;
  }

  // Метод возвращает значение true, если стек пуст.
  public bool empty() {
    return tos==0;
  }

  // Возвращает общий объем стека.
  public int capacity() {
    return stck.Length;
  }

  // Возвращает текущее количество объектов в стеке.
  public int getNum() {
    return tos;
  }
}

// Демонстрация использование класса Stack.


class StackDemo {
  public static void Main() {
    Stack stk1 = new Stack(10);
    Stack stk2 = new Stack(10);
    Stack stk3 = new Stack(10);
    char ch;
    int i;

    // Помещаем ряд символов в стек stkl.
    Console.WriteLine("Помещаем символы от A до Z в стек stkl.");
    for(i=0; !stk1.full(); i++)
      stk1.push((char) ('A' + i));

    if(stk1.full()) Console.WriteLine("Стек stkl полон.");

    // Отображаем содержимое стека stkl.
```

```
    Console.Write("Содержимое стека stkl: ");
    while( !stk1.empty() ) {
      ch = stk1.pop();
      Console.Write(ch);
    }

    Console.WriteLine();

    if(stk1.empty()) Console.WriteLine("Стек stkl пуст.\n");

    // Помещаем еще символы в стек stk1
    Console.WriteLine("Снова помещаем символы от A до Z в стек stk1.");
    for(i=0; !stk1.full(); i++)
      stk1.push((char) ('A' + i));

    /* Теперь извлекаем элементы из стека stkl и помещаем их в стек stk2.
В результате элементы стека stk2 должны быть расположены в обратном
порядке.*/
    Console.WriteLine("Теперь извлекаем элементы из стека stkl" +
                      "и помещаем их в стек stk2.");
    while( !stk1.empty() ) {
      ch = stk1.pop();
      stk2.push(ch);
    }

    Console.Write("Содержимое стека stk2: ");
    while( !stk2.empty() ) {
      ch = stk2.pop();
      Console.Write(ch);
    }

    Console.WriteLine("\n");

    // Помещаем 5 символов в стек
    Console.WriteLine("Помещаем 5 символов в стек stk3.");
    for(i=0; i < 5; i++)
      stk3.push((char) ('A' + i));

    Console.WriteLine("Вместимость стека stk3: " + stk3.capacity());
    Console.WriteLine("Количество объектов в стеке stk3: " +
                      stk3.getNum());

  }
}
/*                Результат работы программы:
Помещаем символы от A до Z в стек stkl.
Стек stkl полон.
Содержимое стека stkl: JIHGFEDCBA
Стек stkl пуст.

Снова помещаем символы от A до Z в стек stk1.
Теперь извлекаем элементы из стека stklи помещаем их в стек stk2.
Содержимое стека stk2: ABCDEFGHIJ

Помещаем 5 символов в стек stk3.
Вместимость стека stk3: 10
Количество объектов в стеке stk3: 5
*/
```

В книге Г.Шилдт, стр.182 содержится учебный проект:
«Стек — это классический пример объектно-ориентированного программирования, в котором сочетаются как средства хранения информации, так и методы получения доступа к этой информации».

Конструктивные элементы объектно-ориентированной программы

Классы и ООП.

Объектно-ориентированное программирование и проектирование построено на классах.
Любую программную систему, выстроенную в объектном стиле, можно рассматривать как совокупность классов, возможно, объединенных в проекты, пространства имен, решения, как это делается при программировании в Visual Studio .Net.

Две роли классов

У класса две различные роли: модуля и типа данных.

1. Класс - это модуль, архитектурная единица построения программной системы.

Модульность построения - основное свойство программных систем.

В ООП программная система, строящаяся по модульному принципу, состоит из классов, являющихся основным видом модуля.

Модуль может не представлять собой содержательную единицу - его размер и содержание определяется архитектурными соображениями, а не семантическими.

Ничто не мешает построить монолитную систему, состоящую из одного модуля - она может решать ту же задачу, что и система, состоящая из многих модулей.

2. Класс - это тип данных, задающий реализацию некоторой абстракции данных, характерной для задачи, в интересах которой создается программная система.

Вторая роль класса не менее важна.

Классы, задающие реализацию некоторой абстракции данных - не просто кирпичики, из которых строится система. Каждый кирпичик теперь имеет важную содержательную начинку.

Состав класса, его размер определяется не архитектурными соображениями, а той абстракцией данных, которую должен реализовать класс. **В основе класса лежит абстрактный тип данных.**

Если вы создаете класс Account, реализующий такую абстракцию как банковский счет, то в этот класс нельзя добавить поля из класса Car, задающего автомобиль, так как класс Car содержит другую абстракцию, другой тип данных.

Объектно-ориентированная разработка программной системы основана на стиле, называемом **проектированием от данных**.

Проектирование системы сводится к поиску абстракций данных, подходящих для конкретной задачи.

Каждая из таких абстракций реализуется в виде класса, которые и становятся модулями - архитектурными единицами построения нашей системы.

В хорошо спроектированной ОО-системе каждый класс играет обе роли, так что каждый модуль нашей системы имеет вполне определенную смысловую нагрузку.

Типичная ошибка – рассматривать класс только как архитектурную единицу, объединяя под обложкой класса разнородные поля и функции, после чего становится неясным, какой же тип данных задает этот класс.

Класс – это лишь шаблон объектов определённого типа, и только.

Синтаксис класса

Синтаксис описания класса:

[атрибуты][модификаторы]class имя_класса[:список_родителей]
{тело_класса}

Атрибуты рассмотрим позднее.
Возможными модификаторами в объявлении класса могут быть модификаторы new, abstract, sealed, о которых подробно будет говориться при рассмотрении наследования,
и четыре модификатора доступа, два из которых - private и protected - могут быть заданы только для вложенных классов.

Обычно класс имеет атрибут доступа public, открытый. Так что в простых случаях объявление класса выглядит так:
public class Rational {тело_класса}

В теле класса могут быть объявлены:
константы;
свойства;
поля;
конструкторы и деструкторы;
методы;
события;
делегаты;
индексаторы;
классы (структуры, интерфейсы, перечисления).

О событиях и делегатах предстоит подробный разговор впоследствии.

Из синтаксиса следует, что классы могут быть вложенными.
Такая ситуация - довольно редкая. Ее стоит использовать, когда некоторый класс носит вспомогательный характер, разрабатывается в интересах другого класса, и есть полная уверенность, что внутренний класс никому не понадобится, кроме класса, в который он вложен.

Внутренние классы обычно имеют модификатор доступа, отличный от public.

Основу любого класса составляют его конструкторы, поля и методы.

Поля класса

Поля класса синтаксически являются обычными переменными (объектами) языка. Их описание удовлетворяет обычным правилам объявления переменных.

Содержательно поля задают представление той самой абстракции данных, которую реализует класс. Поля характеризуют свойства объектов класса.
Два объекта одного класса имеют один и тот же набор полей, но разнятся значениями, хранимыми в этих полях.

Все объекты класса Person могут иметь поле, характеризующее рост персоны, но один объект может быть высокого роста, другой - низкого, а третий - среднего роста.

Доступ к полям

Каждое поле имеет модификатор доступа, принимающий одно из четырех значений: public, private, protected, internal.

public
Доступ к типу или члену возможен из любого другого кода в той же сборке или другой сборке, ссылающейся на него.
private
Доступ к типу или члену можно получить только из кода в том же классе или структуре.
protected
Доступ к типу или члену можно получить только из кода в том же классе или структуре, или в производном классе.
internal
Доступ к типу или члену возможен из любого кода в той же сборке, но не из другой сборки.
protected internal
Доступ к типу или члену возможен из любого кода в той же сборке, или из производного класса в другой сборке.

Атрибутом доступа по умолчанию является атрибут private. Независимо от значения атрибута доступа, все поля доступны для всех методов класса. Они являются для методов класса глобальной информацией, с которой работают все методы, извлекая из полей нужные им данные и изменяя их значения в ходе работы.
Если поля доступны только для методов класса, то они имеют атрибут доступа private, который можно опускать. Такие поля считаются закрытыми, но часто желательно, чтобы некоторые из них были доступны в более широком контексте.
Если некоторые поля класса A должны быть доступны для методов класса B, являющегося потомком класса A, то эти поля следует снабдить атрибутом protected. Такие поля называются защищенными.
Если некоторые поля должны быть доступны для методов классов B1, B2 и так далее, дружественных по отношению к классу A, то эти поля следует снабдить атрибутом internal, а все дружественные классы B поместить в один проект (assembly). Такие поля называются дружественными.
Наконец, если некоторые поля должны быть доступны для методов любого класса B, которому доступен сам класс A, то эти поля следует снабдить атрибутом public. Такие поля называются общедоступными или открытыми.

Методы класса

Методы класса синтаксически являются обычными процедурами и функциями языка. Их описание удовлетворяет обычным правилам объявления процедур и функций.

Содержательно методы определяют ту самую абстракцию данных, которую реализует класс.
Методы содержат описания операций, доступных над объектами класса.
Два объекта одного класса имеют один и тот же набор методов.

Доступ к методам
Каждый метод имеет модификатор доступа, принимающий одно из четырех значений: public, private, protected, internal.

Атрибутом доступа по умолчанию является атрибут private.

Независимо от значения атрибута доступа, все методы доступны для вызова при выполнении метода класса.
Если методы имеют атрибут доступа private, возможно, опущенный, то тогда они доступны только для вызова и только внутри методов самого класса. Такие методы считаются закрытыми.

Понятно, что класс, у которого все методы закрыты, абсурден, поскольку никто не смог бы вызвать ни один из его методов.
Как правило, у класса есть открытые методы, задающие интерфейс класса, и закрытые методы.

Интерфейс - это лицо класса и именно он определяет, чем класс интересен своим клиентам, что он может делать, какие сервисы предоставляет клиентам.

Закрытые методы составляют важную часть класса, позволяя клиентам не вникать во многие детали реализации.
Эти методы клиентам класса недоступны, они о них могут ничего не знать, и, самое главное, изменения в закрытых методах никак не отражаются на клиентах класса при условии корректной работы открытых методов.

Если некоторые методы класса A должны быть доступны для вызовов в методах класса B, являющегося потомком класса A, то такие методы следует снабдить атрибутом protected.

Если некоторые методы должны быть доступны только для методов классов B1, B2 и так далее, дружественных по отношению к классу A, то такие методы следует снабдить атрибутом internal, а все дружественные классы B поместить в один проект.

Наконец, если некоторые методы должны быть доступны для методов любого класса B, которому доступен сам класс A, то такие методы снабжаются модификатором public.
Методы-свойства
Методы, называемые свойствами (Properties), представляют специальную синтаксическую конструкцию, предназначенную для обеспечения эффективной работы со свойствами.

При работе со свойствами объекта (полями) часто нужно решить, какой модификатор доступа использовать, чтобы реализовать нужную стратегию доступа к полю класса.

Вот пять наиболее употребительных стратегий:
чтение, запись (Read, Write);
чтение, запись при первом обращении (Read, Write-once);
только чтение (Read-only);
только запись (Write-only);
ни чтения, ни записи (Not Read, Not Write).

Открытость свойств (атрибут public) позволяет реализовать только первую стратегию.

В языке C# принято, как и в других объектных языках, свойства объявлять закрытыми, а нужную стратегию доступа организовывать через методы.
Для эффективности этого процесса и введены специальные методы-свойства.

Рассмотрим пример - класс Person, у которого пять полей: fam, status, salary, age, health, характеризующих соответственно фамилию, статус, зарплату, возраст и здоровье персоны. Для каждого из этих полей может быть разумной своя стратегия доступа.
Возраст доступен для чтения и записи,
фамилию можно задать только один раз,
статус можно только читать,
зарплата недоступна для чтения,
а здоровье закрыто для доступа и только специальные методы класса могут сообщать некоторую информацию о здоровье персоны.

Пример 1.

```
public class Person
{
   //поля (все закрыты)
   string fam="", status="", health="";
   int age=0, salary=0;
   //методы - свойства
   /// <summary>
   ///стратегия: Read,Write-once (Чтение, запись при
   ///первом обращении)
   /// </summary>
   public string Fam
   {
      set {if (fam == "") fam = value;}
      get {return(fam);}
   }
   /// <summary>
   ///стратегия: Read-only(Только чтение)
   /// </summary>
   public string Status
   {
      get {return(status);}
   }
   /// <summary>
   ///стратегия: Read,Write (Чтение, запись)
   /// </summary>
   public int Age
   {
      set
      {
         age = value;
         if(age < 7)status ="ребенок";
         else if(age <17)status ="школьник";
         else if (age < 22)status = "студент";
         else status = "служащий";
      }
      get {return(age);}
   }
   /// <summary>
   ///стратегия: Write-only (Только запись)
   /// </summary>
   public int Salary
   {
      set {salary = value;}
   }
}
```

Вот пример, показывающий, как некоторый клиент создает и работает с полями персоны:

```
public void TestPersonProps()
{
  Person pers1 = new Person();
   pers1.Fam = "Петров";
   pers1.Age = 21;
   pers1.Salary = 1000;
   Console.WriteLine ("Фам={0}, возраст={1}, статус={2}",      pers1.Fam, pers1.Age,
pers1.Status);
   pers1.Fam = "Иванов";
 pers1.Age += 1;
   Console.WriteLine ("Фам={0}, возраст={1}, статус={2}",      pers1.Fam, pers1.Age,
pers1.Status);
}
//TestPersonProps
```

Заметьте, клиент работает с методами-свойствами так, словно они являются настоящими полями, вызывая их как в правой, так и в левой части оператора присваивания.

Заметьте также, что с каждым полем можно работать только в полном соответствии с той стратегией, которую реализует данное свойство.
Попытка изменения фамилии не принесет успеха, а изменение возраста приведет и к одновременному изменению статуса.
На рис. 1 показаны результаты работы этой процедуры.

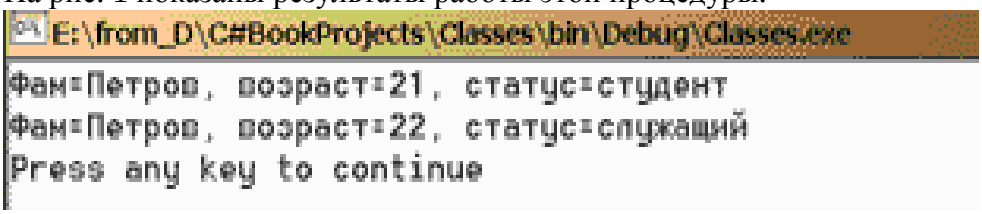


Рис. 1. Методы-свойства и стратегии доступа к полям

Индексаторы

Свойства являются частным случаем метода класса с особым синтаксисом.

Еще одним частным случаем является индексатор.

Метод-индексатор является обобщением метода-свойства.

Он обеспечивает доступ к закрытому полю, представляющему массив. Объекты класса индексируются по этому полю.

Синтаксически объявление индексатора - такое же, как и в случае свойств, но методы get и set приобретают аргументы по числу размерности массива, задающего индексы элемента, значение которого читается или обновляется.

Важным ограничением является то, что у класса может быть только один индексатор и у этого индексатора стандартное имя this.

Так что если среди полей класса есть несколько массивов, то индексация объектов может быть выполнена только по одному из них.

Добавим в класс Person свойство children, задающее детей персоны, сделаем это свойство закрытым, а доступ к нему обеспечит индексатор:

```
const int Child_Max = 20; //максимальное число детей
Person[] children = new Person[Child_Max];
int count_children=0; //число детей
public Person this[int i] //индексатор
{
   get {if (i>=0 && i< count_children)return(children[i]);
         else return(children[0]);}
   set
   {
      if (i==count_children && i< Child_Max)
       {children[i] = value; count_children++;}
   }
}
```

Имя у индексатора - this, в квадратных скобках в заголовке перечисляются индексы.

В методах get и set, обеспечивающих доступ к массиву children, по которому ведется индексирование, анализируется корректность задания индекса.

Закрытое поле count_children, хранящее текущее число детей, доступно только для чтения благодаря добавлению соответствующего метода-свойства. Запись в это поле происходит в методе set индексатора, когда к массиву children добавляется новый элемент.

Протестируем процесс добавления детей персоны и работу индексатора.

Индексатор создает из объекта как бы массив объектов, индексированный по соответствующему полю, в данном случае по полю children.

На рис. 2 показаны результаты вывода.

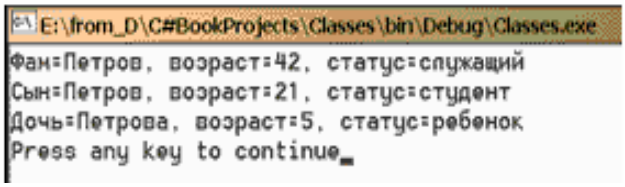


Рис. 2. Работа с индексатором класса

Операции

Еще одним частным случаем являются методы, задающие над объектами-классами бинарную или унарную операцию.

Введение в класс таких методов позволяет строить выражения, аналогичные арифметическим и булевым выражениям с обычно применяемыми знаками операций и сохранением приоритетов операций.

Статические поля и методы класса

Когда конструктор класса создает новый объект, то в памяти создается структура данных с полями, определяемыми классом. Уточним теперь это описание.

Не все поля отражаются в структуре объекта.

У класса могут быть поля, связанные не с объектами, а с самим классом. Эти поля объявляются как статические с модификатором static.

Статические поля доступны всем методам класса. Независимо от того, какой объект вызвал метод, используются одни и те же статические поля, позволяя методу использовать информацию, созданную другими объектами класса.

Статические поля представляют общий информационный пул для всех объектов классов, позволяя извлекать и создавать общую информацию.

Например, у класса Person может быть статическое поле message, в котором каждый объект может оставить сообщение для других объектов класса.

Аналогично полям, у класса могут быть и статические методы, объявленные с модификатором static.

Такие методы не используют информацию о свойствах конкретных объектов класса - они обрабатывают общую для класса информацию, хранящуюся в его статических полях.

Например, в классе Person может быть статический метод, обрабатывающий данные из статического поля message.

Другим частым случаем применения статических методов является ситуация, когда класс предоставляет свои сервисы объектам других классов.

Таковым является класс Math из библиотеки FCL, который не имеет собственных полей - все его статические методы работают с объектами арифметических классов.

Как вызываются статические поля и методы?

Всякий вызов метода в объектных вычислениях имеет вид x.F(...); где x - это цель вызова.

Обычно целью вызова является объект, который вызывает методы класса, не являющиеся статическими (динамическими или экземплярными).

В этом случае поля целевого объекта доступны методу и служат глобальным источником информации.

Если же необходимо вызвать статический метод (поле), то целью должен быть сам класс.

Можно полагать, что для каждого класса автоматически создается статический объект с именем класса, содержащий статические поля и обладающий статическими методами.

Этот объект и его методы доступны и тогда, когда ни один другой динамический объект класса еще не создан.

Константы

В классе могут быть объявлены константы.

Константы фактически являются статическими полями, доступными только для чтения, значения которых задаются при инициализации.

Однако задавать модификатор static для констант не только не нужно, но и запрещено.

В нашем классе Person была задана константа Child_Max, характеризующая максимальное число детей у персоны.

Никаких проблем не возникает, когда речь идет о константах встроенных типов, таких, как Child_Max.

Конструкторы класса

Конструктор - неотъемлемый компонент класса. Нет классов без конструкторов.

Конструктор представляет собой специальный метод класса, позволяющий создавать объекты класса.

Одна из синтаксических особенностей этого метода в том, что его имя должно совпадать с именем класса.

Если программист не определяет конструктор класса, то к классу автоматически добавляется конструктор по умолчанию - конструктор без аргументов.

Если программист сам создает один или несколько конструкторов, то автоматического добавления конструктора без аргументов не происходит.

Как и когда происходит создание объектов?

Чаще всего, при объявлении сущности в момент ее инициализации.

Рассмотрим создание трех объектов класса Person:

```
Person pers1 = new Person(),
pers2 = new Person();
Person pers3= new Person("Петрова");
```

Сущности pers1, pers2 и pers3 класса Person объявляются с инициализацией, задаваемой унарной операцией new, которой в качестве аргумента передается конструктор класса Person.

У класса может быть несколько конструкторов - это типичная практика, - отличающихся сигнатурой.

В данном примере в первой строке вызывается конструктор без аргументов,

во второй строке для сущности pers3 вызывается конструктор с одним аргументом типа string.

Разберем в деталях процесс создания объекта:

первым делом для сущности pers создается ссылка, пока висячая, со значением null;

затем в динамической памяти создается объект - структура данных с полями, определяемыми классом Person.

Поля объекта инициализируются значениями по умолчанию:

ссылочные поля - значением null,

арифметические - нулями,

строковые - пустой строкой.

Эту работу выполняет конструктор по умолчанию, который, можно считать, всегда вызывается в начале процесса создания.

Если инициализируется переменная значимого типа, то все происходит аналогичным образом, за исключением того, что объект создается в стеке;

если поля класса проинициализированы, как в нашем примере, то выполняется инициализация полей заданными значениями;

если вызван конструктор с аргументами, то начинает выполняться тело этого конструктора.

Как правило, при этом происходит инициализация отдельных полей класса значениями, переданными конструктору.

Так, поле fam объекта pers3 получает значение "Петрова";

На заключительном этапе ссылка связывается с созданным объектом.

Процесс создания объектов становится сложнее, когда речь идет об объектах, являющихся потомками некоторого класса.

В этом случае, прежде чем создать сам объект, нужно вызвать конструктор, создающий родительский объект.

Ключевое слово new используется в языке для двух разных целей.

Во-первых, это имя операции, запускающей только что описанный процесс создания объекта.

Во-вторых, это модификатор класса или метода.

Зачем классу нужно несколько конструкторов?

Дело в том, что, в зависимости от контекста и создаваемого объекта, может требоваться различная инициализация его полей. Перегрузка конструкторов и обеспечивает решение этой задачи.

Конструктор может быть объявлен с атрибутом private.

Понятно, что в этом случае внешний пользователь не может воспользоваться им для создания объектов. Но это могут делать методы класса, создавая объекты для собственных нужд со специальной инициализацией.

В классе можно объявить статический конструктор с атрибутом static.

Он вызывается автоматически - его не нужно вызывать стандартным образом.

Точный момент вызова не определен, но гарантируется, что вызов произойдет до создания первого объекта класса.

Такой конструктор может выполнять некоторую предварительную работу, которую нужно выполнить один раз, например,

- связаться с базой данных,
- заполнить значения статических полей класса,
- создать константы класса,
- выполнить другие подобные действия.

Статический конструктор, вызываемый автоматически, не должен иметь модификаторов доступа.

Таким образом, объекты создаются динамически в процессе выполнения программы - для создания объекта всегда вызывается тот или иной конструктор класса.

Деструкторы класса

Если задача создания объектов полностью возлагается на программиста, то задача удаления объектов, после того, как они стали не нужными, в Visual Studio .Net снята с программиста и возложена на соответствующий инструментарий - сборщик мусора.

В классическом варианте языка C++ деструктор так же необходим классу, как и конструктор.

В языке C# у класса может быть деструктор, но он не занимается удалением объектов и не вызывается нормальным образом в ходе выполнения программы.

Так же, как и статический конструктор, деструктор класса, если он есть, вызывается автоматически в процессе сборки мусора. Его роль - в освобождении ресурсов, например, файлов, открытых объектом.

Деструктор C# фактически является финализатором (finalizer), необходимым при обработке исключительных ситуаций.

Имя деструктора строится из имени класса с предшествующим ему символом ~ (тильда).

Как и у статического конструктора, у деструктора не указывается модификатор доступа.

Пример

В следующем примере на верхнем уровне пространства имен ProgrammingGuide определен класс MyCustomClass, содержащий три члена.

```
namespace ProgrammingGuide
{
    // Class definition.
    public class MyCustomClass
    {
        // Class members:
        // Property.
        public int Number { get; set; }

        // Method.
        public int Multiply(int num)
        {
            return num * Number;
        }

        // Instance Constructor.
        public MyCustomClass()
        {
            Number = 0;
        }
    }
    // Another class definition. This one contains
```

```
    // the Main method, the entry point for the program.
    class Program
    {
        static void Main(string[] args)
        {
            // Create an object of type MyCustomClass.
            MyCustomClass myClass = new MyCustomClass();

            // Set the value of a public property.
            myClass.Number = 27;

            // Call a public method.
            int result = myClass.Multiply(4);
        }
    }
}
```

Экземпляр (объект) класса MyCustomClass создается в методе Main класса Program, а для доступа к методам и свойствам используется точечная нотация.

Таким образом, в C# различаются следующие разновидности классов:

- Базовый (т.е. основной) – и производный (для которого существует модификатор доступа protected)
- Статические – (static) не требующие создания объектов для использования содержащихся в них методов.
- Вложенные – расположенные внутри других классов
- Разделяемые – хранящиеся в различных файлах, каждый из которых помечен словом partial
- Анонимные - состоящие из одного или более открытых свойств только для чтения. Все остальные виды членов класса, такие как методы или события, запрещены.
- Абстрактные – (abstract) неполные, подлежащие реализации в производном классе.
- Виртуальные – (virtual) обеспечивающий общую функциональность объектов, методы которых могут быть изменены.
- Запечатанные – (sealed) наследование которых запрещено.

Объекты.

Определение класса или структуры подобно чертежу, на котором указаны действия, выполняемые типом. Объект является блоком памяти, выделенной и настроенной в соответствии с чертежом. Объекты также называют экземплярами класса. Они могут храниться либо в именованной переменной, либо в массиве или коллекции.

Объект является элементом приложения и может представлять собой кроме экземпляра класса, ещё одну конструкцию - хранилище сложных данных (т.н. объекты данных).

Программа может создать множество объектов одного класса.

Как экземпляры класса объекты создаются с помощью ключевого слова new, за которым следует конструктор класса, для которого создаётся объект.

Например, создадим объект object1 класса Customer – клиент, потребитель:

```
Customer object1 = new Customer();
```

Здесь сначала создаётся ссылка на объект, а затем с помощью конструктора новый объект заполняется данными. Ссылка создаётся в левой части равенства.

Ссылка указывает на создаваемый объект, но не содержит данные этого объекта. Фактически, можно создать ссылку на объект без создания самого объекта:

```
Customer object2;
```

но создавать такие ссылки, которые не указывают на объект, не рекомендуется, так как попытка доступа к объекту по такой ссылке приведет к сбою во время выполнения.

Созданную ссылку можно сделать указывающей на объект, создав новый объект или назначив ее существующему объекту:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

здесь создаются две ссылки на объекты, которые указывают на один объект. Поэтому любые изменения объекта, выполненные посредством object3, будут видны при последующем использовании object4.

Как хранилище сложных данных переменные типа object могут принимать значения любого типа данных.

Объект является одним из самых сложных элементов языка C#. В связи с этим при работе с объектами необходимо учитывать такие их особенности, как

1. Тип объекта (экземпляр класса, или объект данных; ссылочный тип – или тип значения)
2. Структурные особенности объекта (контекст объекта – и метаданные)
3. Наследование возможностей класса Object
4. Варианты создания объектов
5. Особенности работы с членами объектов

Структурно объект состоит из **метаданных** (сведений о типе объекта, версии, языке, региональных параметрах, имени сборки, и т.д.) и **контекста** объекта (данных, которые определены в классе, который является чертежом данного объекта). Членами объектов, их составными частями являются поля, свойства, методы и события.

Метаданные могут содержать неявные и неочевидные члены, определяющие настройку объекта. Поэтому объекты не являются прозрачными элементами языка.

Объекты являются представителями ссылочных типов. При создании объекта память размещается в управляемой куче, и переменная хранит только ссылку на расположение объекта. Для типов в управляемой куче требуются служебные данные и при их размещении, и при их удалении в процессе сборки мусора.

Ссылочные типы и типы значений могут быть преобразованы друг в друга.

Когда переменная типа значения преобразуется в объект, говорят, что она упаковывается. Когда переменная типа object преобразуется в тип значения, говорят, что она распаковывается.

Во время **упаковки** типа значения средой CLR, она создает программу-оболочку значения внутри объекта и сохраняет ее в управляемой куче.

Операция **распаковки-преобразования** извлекает тип значения из объекта.

Концепция упаковки и распаковки лежит в основе единой системы типов C#, в которой значение любого типа можно рассматривать как объект.

Уничтожение обоих типов объектов в C# выполняется автоматически, но объекты на основе типов значений удаляются при выходе за рамки области действия, а объекты на основе ссылочных типов — в неуказанное время после удаления последней ссылки, указывающей на них.

Рассмотрим более подробно специальные операции по **преобразованию типов** обрабатываемых данных: упаковка (boxing) – и распаковка (unboxing).

Упаковка представляет собой неявное преобразование типа значения в тип object. При упаковке типа значения в куче выделяется экземпляр объекта и выполняется копирование значения в этот новый объект.

Рассмотрим следующее объявление переменной типа значения:

```
int i = 123;
```

Следующий оператор неявно применяет операцию упаковки к переменной i.

```
object o = i;  // Implicit boxing      (implicit - подразумевающийся)
```

Результат этого оператора создает ссылку на объект «o» в стеке, которая ссылается на значение типа int в куче. Это значение является копией значения типа значения, назначенного переменной i. Разница между двумя этими переменными, i и o, продемонстрирована на рисунке ниже.

Упаковка-преобразование

Можно также выполнять упаковку явным способом, как в следующем примере, однако явная упаковка не является обязательной.

```
int i = 123;
object o = (object)i;  // explicit boxing
(explicit - точный, полностью определённый)
```

Следующий пример преобразует целочисленную переменную i в объект «o» при помощи упаковки. Затем значение, хранимое переменной i, меняется с 123 на 456. В примере показано, что исходный тип значения и упакованный объект используют отдельные ячейки памяти, а значит, могут хранить разные значения.

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // Implicit boxing

        i = 456;  // Change the contents of i

        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```

Распаковка является явным преобразованием из типа object в тип значения. Операция распаковки состоит из следующих действий.

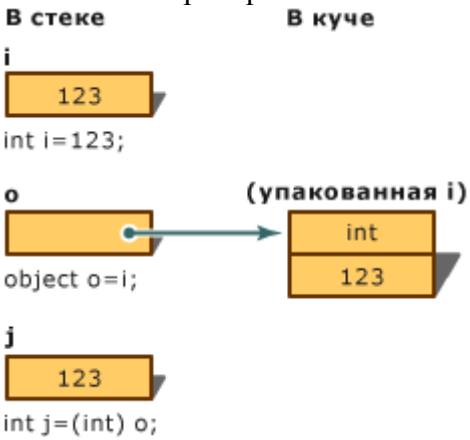
- Проверка экземпляра объекта на то, что он является упакованным значением заданного типа значения.
- Копирование значения из экземпляра в переменную типа-значения.

В следующих операторах показаны операции по упаковке и распаковке.

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;  // unboxing
```

На следующем рисунке представлен результат выполнения предыдущих операторов.

Распаковка-преобразование



Для успешной распаковки типов значений во время выполнения необходимо, чтобы экземпляр, который распаковывается, был ссылкой на объект, предварительно созданный с помощью упаковки экземпляра этого типа значения. Попытка распаковать null или ссылку в несовместимый тип значения вызовет InvalidCastException.

Все классы платформы .Net Framework являются наследниками System.Object. Класс Object поддерживает все классы в иерархии классов .NET Framework и предоставляет низкоуровневые службы для производных классов. Он является исходным базовым классом для всех классов платформы .NET Framework и корнем иерархии типов.

Все методы, поля (переменные-члены), константы, свойства и события объекта должны быть объявлены внутри типа; они называются членами класса, поскольку носителем типа является класс.

В следующем списке перечислены все возможные типы членов, которые можно объявлять в классе (или в структуре).

- Поля
- Константы
- Свойства
- Методы
- Конструкторы
- Деструкторы
- События
- Индексаторы
- Операторы
- Вложенные типы

При создании объекта все эти члены, составляющие ***«контекст объекта»,*** могут быть определены по-разному, причём иногда они получают определения в неявном виде.

Что в данном объекте должен делать оператор «+» - зависит от определения соответствующего оператора класса, представителем которого является созданный объект. Это может быть операция сложения целых чисел, а может быть и операцией сложения комплексных чисел, или операцией конкатенации.

Компилятор внедряет сведения о типе в исполняемый файл в качестве «***метаданных***».

Члены объекта и работа с ними

1. Объект представляет экземпляр класса. Перед получением доступа к членам класса, не являющимся public, необходимо создать объект. Для того, чтобы указать класс, из которого следует создать объект, используется зарезервированное слово new. Новый объект сохраняется в переменной объекта.

Для доступа к члену объекта нужно указать последовательно имя объекта, точку (.) и имя необходимого члена. В следующем примере устанавливается свойство Text объекта Label.

```
Label.Text = "Data not saved"
```

2. Хранящаяся в объекте информация предоставляется полями и свойствами. Можно получить и задать их значения с помощью инструкций присваивания так же, как получить и задать локальные переменные в процедуре. В следующем примере извлекается свойство Width и устанавливается свойство ForeColor объекта Label.

```
Integer = Label.Width
Label.ForeColor = System.Drawing.Color.Red
```

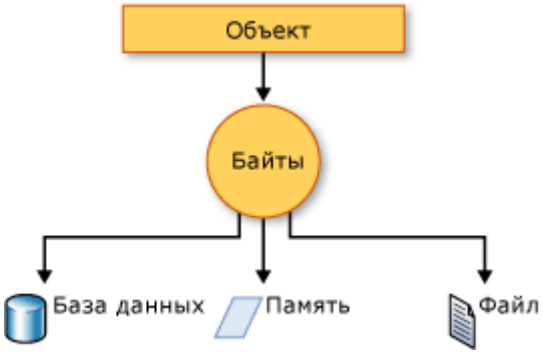
3. Действие, которое выполняет объект, называется методом. Например, Add является методом объекта ComboBox, он добавляет новую запись в поле со списком.

4. Наличие неявных и часто – неочевидных членов объекта затрудняет их использование, перенос или сохранение. Объект – это не просто набор данных, и не файл. Каждый объект имеет присущую только ему индивидуальную настройку, без учёта которой с этим объектом работать нельзя. Поэтому, если надо сохранить объект или передать его кому-то для обработки, используются не просто операции чтения – записи, как для обычных файлов или наборов данных, а специальные операции сериализации и десериализации.

Сериализация представляет собой процесс преобразования объекта в поток байтов с целью сохранения его в памяти, в базе данных или в файле. Ее основное назначение — сохранить состояние объекта для того, чтобы иметь возможность воссоздать его при необходимости.

Десериализация — это процесс получения сохраненных данных и восстановления из них объектов.

На этом рисунке изображён общий процесс сериализации.



Объект сериализуется в поток, который переносит не только данные (контекст объекта), но и метаданные - сведения о типе объекта, такие как его версию, язык и региональные параметры, а также имя сборки и др. Из этого потока объект можно сохранить в базе данных, файле или памяти.

Сериализация позволяет разработчику сохранять состояние объекта и воссоздавать его при необходимости, обеспечивая хранение объектов, а также обмен данными. С помощью сериализации разработчик может выполнять такие действия, как отправка объекта удаленному приложению посредством веб-службы, передача объекта из одного домена в другой, передача объекта через брандмауэр в виде XML-строки и хранение информации о безопасности или конкретном пользователе, используемой несколькими приложениями.

Например, нужно создать распределённое приложение, разные части которого должны обмениваться данными со сложной структурой. В таком случае для типов данных, которые предполагается передавать, пишется код, который осуществляет сериализацию и десериализацию. Объект заполняется нужными данными, затем

вызывается код сериализации, в результате получается, например, XML-документ. Результат сериализации передаётся принимающей стороне, например, по электронной почте или HTTP. Приложение-получатель создаёт объект того же типа и вызывает код десериализации, в результате получая объект с теми же данными, что были в объекте приложения-отправителя. По такой схеме работает, например, сериализация объектов через SOAP в Microsoft .NET.

Для сериализации объекта требуется объект, который будет сериализован, поток, который будет содержать сериализованный объект, и объект Formatter. Этот объект может быть получен из пространства имен System.Runtime.Serialization, которое содержит классы, необходимые для сериализации и десериализации объектов.

Примеры.

1. Пример простой объектно-ориентированной программы.

```
  using System;
  //класс
class Building {
  public int floors;    // количество этажей     I Эти переменные
  public int area;      // общая площадь здания  I могут быть
  public int occupants; // количество жильцов    I разных типов

  // Метод для отображения значения площади, приходящейся на одного человека.
  public void areaPerPerson() {
    Console.WriteLine("  " + area / occupants +
                      " приходится на одного человека");
  }
}

  //Другой класс
class BuildingDemo {
  public static void Main() {
    Building house = new Building();   //создание объекта
    Building office = new Building();   //создание объекта

 //Работа с объектами
    // присваиваем значения полям в объекте house
    house.occupants = 4;          // поля
    house.area = 2500;            // могут быть
    house.floors = 2;             // разных типов

    // присваиваем значения полям в объекте office
    office.occupants = 25;
    office.area = 4200;
    office.floors = 3;


    Console.WriteLine("Дом имеет:\n  " +
                      house.floors + " этажа\n  " +
                      house.occupants + " жильца\n  " +
                      house.area + " квадратных футов общей площади на них");
    house.areaPerPerson();

    Console.WriteLine();

    Console.WriteLine("Офис имеет:\n  " +
                      office.floors + " этажа \n  " +
                      office.occupants + " работников\n  " +
                      office.area + " квадратных футов общей площади на них
");
    office.areaPerPerson();
  }
```

}
Результаты выполнения этой программы выглядят так:
Дом имеет:
- 2 этажа
- 4 жильца
- 2500 квадратных футов общей площади на них
- 625 приходится на одного человека

Офис имеет:
- 3 этажа
- 25 работников
- 4200 квадратных футов общей площади на них
- 168 приходится на одного человека

2. Экземпляры классов создаются с помощью оператора new.

В следующем примере Person является типом, а person1 и person 2 — являются экземплярами или объектами этого типа.

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
        public Person(string name, int age)  //Конструктор с параметрами для создания
                                             //         объектов (экземпляров класса)
    {
        Name = name;
        Age = age;
    }
    //Other properties, methods, events...
}

class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Declare  new person, assign person1 to it.
        Person person2 = person1;

        //Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name, person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();

    }
}
/*
    Output:
    person1 Name = Leopold Age = 6
    person2 Name = Molly Age = 16
```

```
    person1 Name = Molly Age = 16
*/
```

Поскольку классы являются ссылочными типами, в переменной объекта класса хранится ссылка на адрес объекта в управляемой куче. Если первому объекту назначен второй объект того же типа, обе переменные ссылаются на объект, расположенный по данному адресу.

Поскольку структуры являются типами значений, в переменной объекта структуры хранится копия всего объекта.

3. Экземпляры структур также можно создать с помощью оператора new.

Структуры определяются с помощью ключевого слова struct, например:

```
public struct PostalAddress
{
    // Fields, properties, methods and events go here...
}
```

Структуры используют большую часть того же синтаксиса, что и классы, однако они более ограничены по сравнению с ними.

- Структуры являются Типами Значений, а классы — Ссылочными Типами.
- В отличие от классов, структуры можно создавать без использования оператора new.
- Структура не может быть унаследованной от другой структуры или класса и не может быть основой для других классов. Все структуры наследуют непосредственно от System.ValueType, который наследует от System.Object.

```
public struct Person
{
    public string Name;
    public int Age;
     public Person(string name, int age)     // Конструктор - требует строковый
                                             //               параметр - «имя»,
                                             //          и числовой - «возраст»
    {
        Name = name;
        Age = age;
    }
}

public class Application
{
    static void Main()
    {
        // Create  struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Create  new struct object. Note that  struct can be initialized
        // without using "new".
        Person p2 = p1;

        // Assign values to p2 members.
        p2.Name = "Spencer";
```

```
            p2.Age = 7;
            Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

            // p1 values remain unchanged because p2 is  copy.
            Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

            // Keep the console open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
    /*
      Output:
        p1 Name = Alex Age = 9
        p2 Name = Spencer Age = 7
        p1 Name = Alex Age = 9
    */
```

Память для p1 и p2 выделена в стеке потока. Эта память освобождена наряду с типом или методом, в котором она объявляется. Эта одна причина того, почему структуры копируются при присваивании. Напротив, при выходе всех ссылок на объект из области действия среда CLR выполняет автоматическое освобождение памяти (сборку мусора), выделенной для экземпляра класса. Возможность детерминированного уничтожения объекта класса, имеющаяся в C++, в данном случае отсутствует.

4. Сравнение объектов.

Сравнивая два объекта на предмет равенства, сначала необходимо определить, нужно ли узнать, представляют ли две переменные один объект в памяти или значения одного или нескольких их полей являются равными. Если планируется сравнить значения, следует решить, являются ли объекты экземплярами типов значений (структурами) или ссылочными типами (классами, делегатами, массивами).

- Чтобы определить, ссылаются ли два экземпляра класса на одно расположение в памяти (это значит, что они имеют одинаковый идентификатор), воспользуйтесь статическим методом Equals. (System.Object является неявным базовым классом для всех типов значений и ссылочных типов, включая структуры и классы, определенные пользователем).
- Чтобы определить, имеют ли поля экземпляра в двух экземплярах структур одинаковые значения, воспользуйтесь методом ValueType.Equals. Поскольку все структуры неявно наследуют от System.ValueType, метод может быть вызван непосредственно в объекте, как показано в следующем примере.

```
// Person is defined in the previous example.

//public struct Person
//{
//    public string Name;
//    public int Age;
//    public Person(string name, int age)
//    {
//        Name = name;
//        Age = age;
//    }
//}

Person p1 = new Person("Wallace", 75);
Person p2;
```

```
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

Реализация System.ValueTypeEquals использует отображение, поскольку необходимо определить поля, находящиеся в любой структуре. При создании собственных структур переопределите метод Equals для предоставления эффективного алгоритма равенства, соответствующего вашему типу.

Чтобы определить, равны ли значения полей в двух экземплярах класса, можно воспользоваться методом Equals или оператором ==. Однако их следует использовать, только если они переопределены или перегружены классом, чтобы предоставить пользовательское определение равенства для объектов этого типа.

5. Использование структур

Тип struct подходит для создания несложных объектов, таких как Point, Rectangle и Color. Хотя point удобно представить в виде класса с автоматически реализуемыми свойствами, в некоторых сценариях структура может оказаться более эффективной. Например, при объявлении массива из 1000 объектов Point потребуется выделить дополнительную память для хранения ссылок на все эти объекты, и структура в таком случае будет более экономичным решением. Поскольку .NET Framework уже содержит объект с именем Point, структура в данном примере называется вместо этого "CoOrds".

```
public struct CoOrds
{
    public int x, y;

    public CoOrds(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

При создании объекта структуры с помощью оператора new объект создается и вызывается соответствующий конструктор. В отличие от классов структуры можно создавать без использования оператора new. В таком случае вызов конструктора отсутствует, что делает выделение более эффективным. Однако поля остаются без значений и объект нельзя использовать до инициализации всех полей.

В отличие от классов структуры не поддерживают наследование. Структура не может быть унаследованной от другой структуры или класса и не может быть основой для других классов. Однако структуры наследуют от базового класса Object. Структуры могут реализовывать интерфейсы; этот механизм полностью аналогичен реализации интерфейсов для классов.

Класс нельзя объявить с помощью ключевого слова struct. В C# классы и структуры семантически различаются. Структура является типом значения, тогда как класс — это ссылочный тип.

Пример 1

В этом примере демонстрируется инициализация struct с помощью конструктора по умолчанию и конструктора с параметрами.

```
public struct CoOrds
{
    public int x, y;

    public CoOrds(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}


// Declare and initialize struct objects.
class TestCoOrds
{
    static void Main()
    {
        // Initialize:
        CoOrds coords1 = new CoOrds();
        CoOrds coords2 = new CoOrds(10, 10);

        // Display results:
        Console.Write("CoOrds 1: ");
        Console.WriteLine("x = {0}, y = {1}", coords1.x, coords1.y);

        Console.Write("CoOrds 2: ");
        Console.WriteLine("x = {0}, y = {1}", coords2.x, coords2.y);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    CoOrds 1: x = 0, y = 0
    CoOrds 2: x = 10, y = 10
*/
```

Пример 2

В следующем примере демонстрируется уникальная возможность структур. В нем создается объект CoOrds без использования оператора new. Если заменить слово struct словом class, программа не будет скомпилирована.

```
public struct CoOrds
{
    public int x, y;

    public CoOrds(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}


// Declare a struct object without "new."
class TestCoOrdsNoNew
```

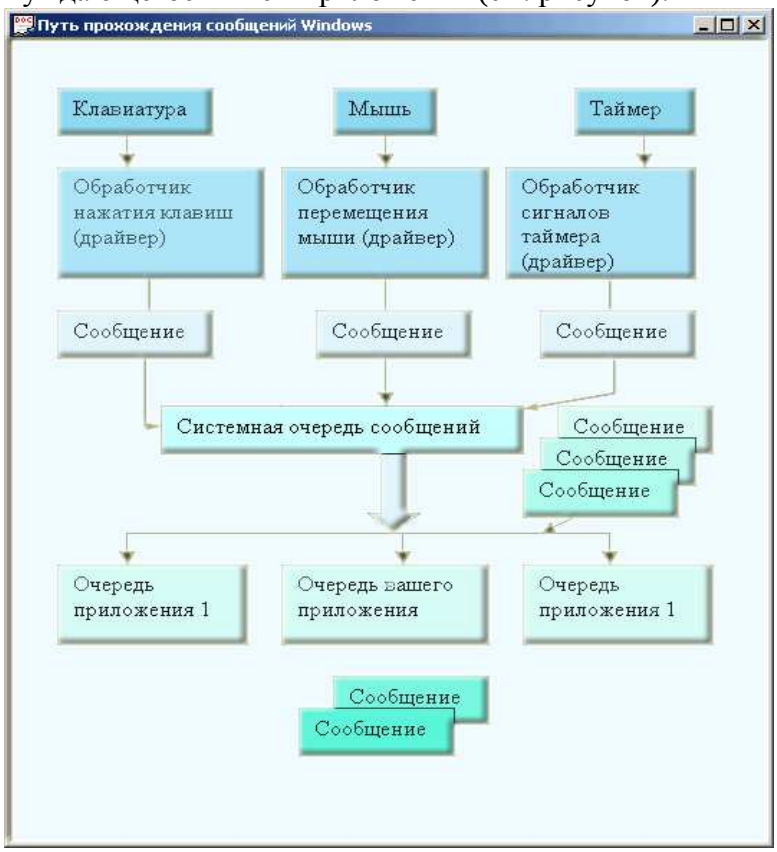
```
{
    static void Main()
    {
        // Declare an object:
        CoOrds coords1;

        // Initialize:
        coords1.x = 10;
        coords1.y = 20;

        // Display results:
        Console.Write("CoOrds 1: ");
        Console.WriteLine("x = {0}, y = {1}", coords1.x, coords1.y);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: CoOrds 1: x = 10, y = 20
```

Программная генерация событий.

Публикация и подписка. Паттерн “Издатель – подписчик”.

Событие представляет собой сигнал о том, что в программе произошло нечто, достойное внимания оператора. Примерами событий могут служить нажатия на клавиши, срабатывание таймера, окончание работы принтера.

Физически возникновение события фиксируется в виде возникновения сообщения, которое поступает в системную очередь сообщений, в которой определяется, для кого оно представляет интерес – и в соответствии с этим интересом оно поступает в очередь нуждающегося в нём приложения (см. рисунок).



Объект, способный генерировать события, называется **издателем** и является источником событий.

Генерирование событий иногда называют возбуждением событий.

Один или несколько объектов могут подписаться (или зарегистрироваться) на уведомление об определённом событии, происшедшем в другом объекте. Их называют подписчиками или абонентами.

Каждый подписчик должен нести в себе метод для обработки события подписки. Он называется обработчиком события *(event handler).*

Когда соответствующее событие возникает, один за другим выполняются все обработчики.

Подписчик может зарегистрироваться на несколько разных типов событий и содержать в себе несколько различных обработчиков.

При обмене событиями классу отправителя событий (издателю) не известен объект или метод, который будет получать (обрабатывать) вызванные отправителем события.

В языке C# любой объект может *опубликовать* набор событий, а другие объекты могут на них *подписаться*. Когда публикующий класс вызывает событие, все подписавшиеся классы уведомляются об этом.

Подобное проектное решение является реализацией паттерна «издатель/подписчик», описанного в книге «Гамма, Хелм, Джонсон, Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования.» - Пер. с англ. - СПб:Питер, 2000 г.».

В книге формулируется назначение данного паттерна следующим образом: «Определить между объектами зависимость «один-ко-многим», чтобы при смене состояния одного объекта все зависимые объекты уведомлялись об этом и автоматически меняли свои состояния».

Например, кнопка уведомляет заинтересованных наблюдателей, когда пользователь щелкнет по ней. Кнопка называется *издателем,* потому что публикует событие Click, а другие классы называются *подписчиками,* потому что подписываются на это событие.

События в C# обрабатываются с помощью делегатов. Класс-издатель определяет, какой делегат должен быть реализован классами-подписчиками при обработке данного события.

Когда возникает событие, методы подписчиков вызываются через этот делегат.

Существует соглашение, по которому обработчики событий в Framework возвращают void и принимают два параметра. Первый - это *источник* события, то есть имя объекта-издателя. Вторым параметром передается объект, производный от класса EventArgs.

EventArgs является классом, базовым для всех классов событий. Он содержит открытое статическое поле empty, представляющее событие без состояния (чтобы обеспечить эффективную обработку таких событий). Классы, производные от EventArgs, содержат информацию о возникшем событии.

Общая схема реализации паттерна «Издатель –Подписчик».

1. Создаётся класс аргументов события, производный от EventArgs.
2. Объявляется делегат.
3. Объявляется класс издателя события (sender, writer, source,…), в котором
   - Объявляется событие
   - Проверяются условия для возникновения события (например, наличие хотя бы одного обработчика этого события в программе)
   - Зажигается событие
4. Создаются классы подписчиков (ресиверы), каждый из которых содержит свой метод для обработки этого события

5. В классе «Исполнитель» или «Демонстратор» содержится основная программа, в которой и происходит обработка события всеми подписчиками.
   Каждое событие характеризуется
   - наличием необходимой для его реализации информации (назовём эту информацию аргументами);
   - именем издателя события (т.е. класса, в котором оно зажигается);
   - списком обработчиков события;
   - условиями возникновения события.

Зажигание события происходит, как обращение к обычному методу, имеющему имя события, с той лишь разницей, что метод, зажигающий событие, в момент его объявления не определён – в нём отсутствует тело. Его определение отсрочено, оно произойдёт позднее, с помощью делегата.

Пример 1.

```
using System;
//объявление делегата
 delegate void MyEventHandler();

// объвление класса - издателя события
 class MyEvent {
 public event MyEventHandler SomeEvent;   //объявление события SomeEvent
 public void FireSomeEvent() {
 if(SomeEvent != null)         //проверка, можно ли зажигать событие


   SomeEvent();                 //метод для зажигания события SomeEvent
}
}
//конец класса - издателя

// объявление класса - ресивера
 class Resiver {
 public static void handler()   {
    Console.WriteLine("Событие произошло");
                         }
}
// конец класса - ресивера

// создание класса - демонстратора
 class EventDemo   {
 public static void Main()   {
 // создание экземпляра класса - издателя
 MyEvent evt = new MyEvent();
// добавление обработчика события в общий список
 evt.SomeEvent += new MyEventHandler(Resiver.handler);
// зажигание события
evt.FireSomeEvent();
}    // конец определения метода Main
}   // конец класса - демонстратора
```

Задание 1 (создание события).

Напишите консольную программу, в которой программа просит ввести с клавиатуры любое число в диапазоне от 0 до 9.

Если введённое число не равно 0, вырабатывается событие, при обработке которого выводится сообщение «Введено правильное число» и повторяется запрос на ввод нового числа.

При вводе 0 программа должна молча завершиться.

Задание 2 (создание двух событий).

Напишите консольную программу, в которой программа просит ввести с клавиатуры любое число в диапазоне от 0 до 9.

Если введено чётное число, вырабатывается событие, по которому выводится само число и дата.

Если введено нечётное число, вырабатывается событие, по которому выводится само число и строка из какого-нибудь стихотворения.

Задание 3 (одно событие обрабатывается по-разному в разных ресиверах).

Напишите консольную программу, в которой программа просит ввести с клавиатуры любое число в диапазоне от 0 до 9.

Если введено чётное число, ресивер 1 выводит текущую дату и время, а ресивер 2 – пустую строку и затем – строку из какого-нибудь стихотворения. После этого программа завершается.

Если введено нечётное число, запрос числа повторяется.

Использование встроенного делегата EventHandler

Для многих событий параметр типа EventArgs не используется. Для упрощения процесса создания кода в таких ситуациях среда .NET Framework включает встроенный тип делегата, именуемый EventHandler. Его можно использовать для объявления обработчиков событий, которым не требуется дополнительная информация. Рассмотрим пример использования типа EventHandler.

```
// Использование встроенного делегата EventHandler.
using System;
// Объявляем класс события:
class MyEvent {
public event EventHandler SomeEvent;
// Здесь объявление использует делегат EventHandler.
// Следующий метод вызывается для генерирования SomeEvent-события:
public void OnSomeEvent() {
if(SomeEvent != null)
SomeEvent(this, EventArgs.Empty);
}
}
class EventDemo {
    static void handler(object source, EventArgs arg)
    {
        Console.WriteLine("Событие произошло.");
        Console.WriteLine("Источником является класс " + source + " . ");
    }
public static void Main() {
MyEvent evt = new MyEvent();
// Добавляем обработчик handler() в список событий:
evt.SomeEvent += new EventHandler(handler);
// Генерируем событие:
evt.OnSomeEvent();
}
}
```

В данном случае параметр типа EventArgs не используется и вместо него передается объект-заполнитель EventArgs.Empty. Результаты выполнения этой программы весьма лаконичны:

Пример события keypress.

Проанализируйте программу. Чем она отличается от предыдущих?

```
  using System;

// Создаём свой класс для хранения кода клавиши:
class KeyEventArgs : EventArgs {
  public char ch;
}

// Объявление делегата:
delegate void KeyHandler(object source, KeyEventArgs arg);

// Объявление класса для события keypress:
class KeyEvent {
  public event KeyHandler KeyPress;

  // Метод вызывается при нажатии клавиши:
  public void OnKeyPress(char key) {
    KeyEventArgs k = new KeyEventArgs();

    if(KeyPress != null) {
      k.ch = key;
      KeyPress(this, k);
    }
  }
}

// Ресивер - класс, принимающий уведомление о событии:
class ProcessKey {
  public void keyhandler(object source, KeyEventArgs arg) {
    Console.WriteLine("Received keystroke: " + arg.ch);
  }
}

// Другой ресивер:
class CountKeys {
  public int count = 0;

  public void keyhandler(object source, KeyEventArgs arg) {
    count++;
  }
}

// Демонстратор работы событий:
class KeyEventDemo {
  public static void Main() {
    KeyEvent kevt = new KeyEvent();
    ProcessKey pk = new ProcessKey();
    CountKeys ck = new CountKeys();
    char ch;

    kevt.KeyPress += new KeyHandler(pk.keyhandler);
    kevt.KeyPress += new KeyHandler(ck.keyhandler);
```

```
    Console.WriteLine("Enter some characters. " +
                      "Enter a period to stop.");
    do {
      ch = (char) Console.Read();
      kevt.OnKeyPress(ch);
    } while(ch != '.');
    Console.WriteLine(ck.count + " keys pressed.");
  }
}
```

Задание 4. Откомментируйте программу:

Объясните, зачем выделены некоторые операторы.

```
1. using System;
2. using System.ComponentModel;
3. using System.Windows.Forms;
4. using System.Drawing;
5. public class MyForm : Form
6. {
7. private TextBox box;
8. private Button button;

9. public MyForm() : base()
10.  {
11.  box = new TextBox();
12.  box.BackColor = System.Drawing.Color.Cyan;
13.  box.Size = new Size(100,100);
14.  box.Location = new Point(50,50);
15.  box.Text = "Hello";

16.  button = new Button();
17.  button.Location = new Point(50,100);
18.  button.Text = "Click Me";

19.  // To wire the event, create
20.  // a delegate instance and add it to the Click event.
21.  button.Click += new EventHandler(this.Button_Clicked);
22.  Controls.Add(box);
23.  Controls.Add(button);
24.  }
25.  // The event handler.
26.  private void Button_Clicked(object sender, EventArgs e)
27.  {
28.  box.BackColor = System.Drawing.Color.Green;
29.  }
```

```
30.  // STAThreadAttribute indicates that Windows Forms uses the
31.  // single-threaded apartment model.
32.  [STAThreadAttribute]
33.  public static void Main(string[] args)
34.  {
35.  Application.Run(new MyForm());
36.  }
37.  }
```

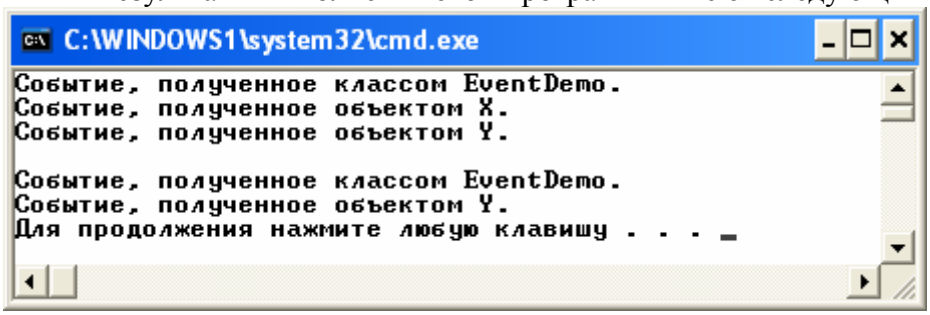
Приложение. Примеры программ для работы с событиями.

Пример события для многоадресатной передачи

Подобно делегатам события могут предназначаться для многоадресатной передачи. В этом случае на одно уведомление о событии может отвечать несколько объектов, рассмотрим пример.

```
// Демонстрация использования события, предназначенного
// для многоадресатной передачи.
using System;
// Объявляем делегат для события:
delegate void MyEventHandler();
// Объявляем класс события:
class MyEvent {
public event MyEventHandler SomeEvent;
// Этот метод вызывается для генерирования события:
public void OnSomeEvent() {
if(SomeEvent != null)
SomeEvent();
}
}
class X {
public void Xhandler() {
Console.WriteLine("Событие, полученное объектом X.");
}
}
class Y {
public void Yhandler() {
Console.WriteLine("Событие, полученное объектом Y.");
}
}
class EventDemo {
static void handler() {
Console.WriteLine("Событие, полученное классом EventDemo.");}
public static void Main() {
MyEvent evt = new MyEvent();
X xOb = new X();
Y yOb = new Y () ;
// Добавляем обработчики в список события:
evt.SomeEvent += new MyEventHandler(handler);
evt.SomeEvent += new MyEventHandler(xOb.Xhandler);
evt.SomeEvent += new MyEventHandler(yOb.Yhandler);
// Генерируем событие:
evt.OnSomeEvent();
Console.WriteLine();
// Удаляем один обработчик:
evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
evt.OnSomeEvent();
}
}
```

Результаты выполнения этой программы имеют следующий вид:



В этом примере создается два дополнительных класса X и Y, в которых также определяются обработчики событий, совместимые с сигнатурой делегата MyEventHandler.

Следовательно, эти обработчики могут стать частью цепочки событийных вызовов.

Обработчики в классах X и Y не являются статическими.

Это значит, что сначала должны быть созданы объекты каждого класса, после чего в цепочку событийных вызовов должен быть добавлен обработчик, связанный с каждым экземпляром класса. Различие между статическими обработчиками и обработчиками экземпляров классов рассматривается в следующем разделе.

Сравнение методов экземпляров классов со статическими методами, используемыми в качестве обработчиков событий

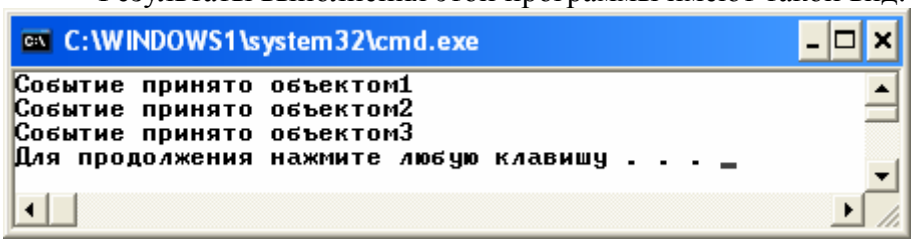
Несмотря на то что и методы экземпляров классов, и статические методы могут служить обработчиками событий, в их использовании в этом качестве есть существенные различия. Если в качестве обработчика используется статический метод, уведомление о событии применяется к классу (и неявно ко всем объектам этого класса). Если же в качестве обработчика событий используется метод экземпляра класса, события посылаются к конкретным экземплярам этого класса. Следовательно, каждый объект класса, который должен получать уведомление о событии, необходимо регистрировать в отдельности. На практике в большинстве случаев "роль" обработчиков событий "играют" методы экземпляров классов, но, безусловно, все зависит от конкретной ситуации.

В следующей программе создается класс X, в котором в качестве обработчика событий определен метод экземпляра. Это значит, что для получения информации о событиях каждый объект класса X необходимо регистрировать отдельно. Для демонстрации этого факта программа готовит уведомление о событии для многоадресатной передачи трем обьектам типа X.

```
/* При использовании в качестве обработчиков событий
методов экземпляров уведомление о событиях принимают
отдельные объекты. */
using System;
// Объявляем делегат для события:
delegate void MyEventHandler();
// Объявляем класс события:
class MyEvent {
public event MyEventHandler SomeEvent;
// Этот метод вызывается для генерирования события:
public void OnSomeEvent() {
if(SomeEvent != null)
SomeEvent();
}
}
class X {
int id;
public X(int x) { id = x; }
// Метод экземпляра, используемый в качестве обработчика событий:
    public void Xhandler()
```

```
    {
        Console.WriteLine("Событие принято объектом" + id);
    }
}
class EventDemo {
public static void Main() {
MyEvent evt = new MyEvent();
X o1 = new X(1) ;
X o2 = new X(2) ;
X o3 = new X(3);
evt.SomeEvent += new MyEventHandler(o1.Xhandler);
evt.SomeEvent += new MyEventHandler(o2.Xhandler);
evt.SomeEvent += new MyEventHandler(o3.Xhandler);
// Генерируем событие,
evt.OnSomeEvent() ;
}
}
```

Результаты выполнения этой программы имеют такой вид:

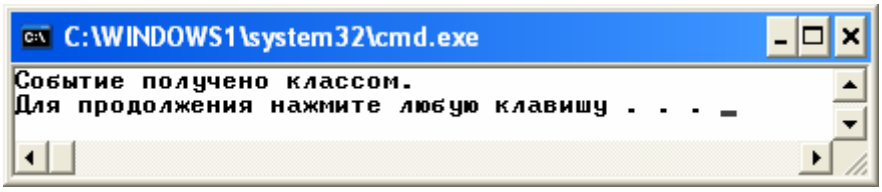


Как подтверждают эти результаты, каждый объект заявляет о своей заинтересованности в событии и получает о нем отдельное уведомление.

Если же в качестве обработчика событий используется статический метод, то, как показано в следующей программе, события обрабатываются независимо от объекта.

```
/* При использовании в качестве обработчиков событий
статического метода уведомление о событиях получает
класс. */
using System;
// Объявляем делегат для события:.
delegate void MyEventHandler();
// Объявляем класс события:
class MyEvent {
public event MyEventHandler SomeEvent;
// Этот метод вызывается для генерирования события:
public void OnSomeEvent() {
if(SomeEvent != null)
SomeEvent();
}
}
class X {
/* Это статический метод, используемый в качестве обработчика события: */
public static void Xhandler() {
Console.WriteLine("Событие получено классом.");
}
}
class EventDemo
{
    public static void Main()
    {
        MyEvent evt = new MyEvent();
        evt.SomeEvent += new MyEventHandler(X.Xhandler);
        // Генерируем событие,
        evt.OnSomeEvent();
    }
}
```

Вот как выглядят результаты выполнения программы:



Обратите внимание на то, что в программе не создается ни одного объекта типа x.

Но поскольку handler () — статический метод класса X, его можно связать с событием SomeEvent и обеспечить его выполнение при вызове метода OnSomeEvent ().

Смешанные средства обработки событий

События можно определять в интерфейсах.

"Поставкой" событий должны заниматься соответствующие классы.

События можно определять как абстрактные. Обеспечить реализацию такого события должен производный класс. Однако события, реализованные с использованием средств доступа add и remove, абстрактными быть не могут.

Любое событие можно определить с помощью ключевого слова sealed.

Событие может быть виртуальным, т.е. его можно переопределить в производном классе.

Рекомендации по обработке событий в среде .NET Framework

C# позволяет програмисту создавать события любого типа. Однако в целях компонентной совместимости со средой .NET Framework необходимо следовать рекомендациям, подготовленным Microsoft специально для этих целей. Центральное место в этих рекомендациях занимает требование того, чтобы обработчики событий имели два параметра. Первый должен быть ссылкой на объект, который будет генерировать событие. Второй должен иметь тип EventArgs и содержать остальную информацию, необходимую обработчику. Таким образом, .NET-совместимые обработчики событий должны иметь следующую общую форму записи:

void handler(object *source*, EventArgs *arg) {*

Обычно параметр source передается вызывающим кодом. Параметр типа EventArgs содержит дополнительную информацию, которую в случае ненадобности можно проигнорировать.

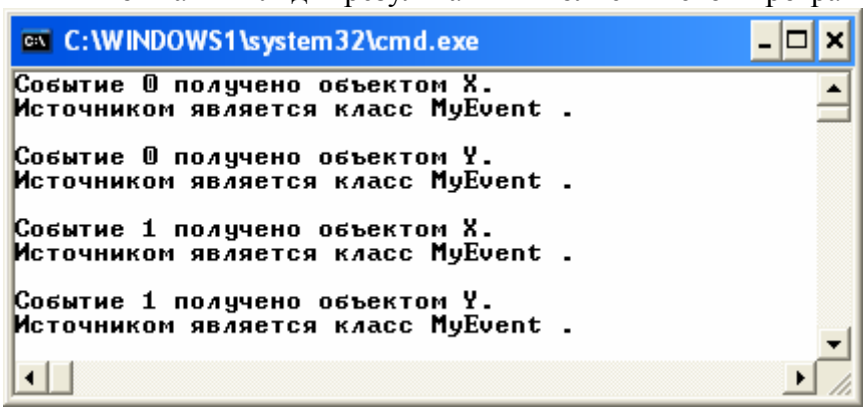
Класс EventArgs не содержит полей, которые используются при передаче дополнительных данных обработчику; он используется в качестве базового класса, из которого можно выводить класс, содержащий необходимые поля. Но поскольку многие обработчики обходятся без дополнительных данных, в класс EventArgs включено статическое поле Empty, которое задает объект, не содержащий никаких данных.

Ниже приведен пример, в котором создается .NET-совместимое событие.

```
// .NET-совместимое событие.
using System;
// Создаем класс, производный от класса EventArgs:
class MyEventArgs : EventArgs {
public int eventnum;
}
// Объявляем делегат для события:
delegate void MyEventHandler(object source,MyEventArgs arg);
// Объявляем класс события:
class MyEvent {
static int count = 0;
public event MyEventHandler SomeEvent;
// Этот метод генерирует SomeEvent-событие:
public void OnSomeEvent() {
MyEventArgs arg = new MyEventArgs();
```

```
if(SomeEvent != null) {
arg.eventnum = count++;
SomeEvent(this, arg);
}
}
class X {
public void handler(object source, MyEventArgs arg) {
Console.WriteLine("Событие " + arg.eventnum + " получено объектом X.");
Console.WriteLine("Источником является класс " + source + " . " ) ;
Console.WriteLine();
}
}
class Y {
public void handler(object source, MyEventArgs arg) {
Console.WriteLine("Событие " + arg.eventnum +
" получено объектом Y.");
Console.WriteLine("Источником является класс " + source + " . " ) ;
Console.WriteLine();
}
}
class EventDerno {
public static void Main() {
X obi = new X() ;
Y ob2 = new Y();
MyEvent evt = new MyEvent();
// Добавляем обработчик handler() в список событий:
evt. SomeEvent += new MyEventHandler (obi.handler) ;
evt.SomeEvent += new MyEventHandler(ob2.handler);
// Генерируем событие:
evt.OnSomeEvent();
evt.OnSomeEvent();
}
}
}
```

Вот как выглядят результаты выполнения этой программы:



В этом примере класс MyEventArgs выводится из класса EventArgs. В классе MyEventArgs добавлено только одно "собственное" поле — eventnum. В соответствии с требованиями .NET Framework делегат для обработчика событий MyEventHandler теперь принимает два параметра. Первый из них представляет собой объектную ссылку на генератор событий, а второй — ссылку на класс EventArgs или производный от класса EventArgs. В данном случае здесь используется ссылка на объект типа MyEventArgs.

Ответы на заданные программы.

Задача 1.

```
using System;
```

```
namespace Program1Events
{
    delegate void MyEventHandler();
    class MyEvent
    {
        public event MyEventHandler SomeEvent;
        public void FireSomeEvent()
        {
            if (SomeEvent != null)
                SomeEvent();
        }
    }

    class Reciver
    {
        static void handler()
        {
            Console.WriteLine("Введено верное число");
        }
    class Program
    {
       public static void Main()
        {
           MyEvent evt = new MyEvent();
           evt.SomeEvent +=new MyEventHandler(handler);
            int cifra = 0;
            do
            {
            Console.WriteLine("Введите число от 0 до 9:");
                cifra = int.Parse(Console.ReadLine());
            if (cifra != 0)
           evt.FireSomeEvent();
            } while(cifra != 0);
          }
        }
    }
}

/*
Результат:

C:\15\SomeEvent1\Events>kiri1
Введите число от 0 до 9:
6
Введено верное число
Введите число от 0 до 9:
6
Введено верное число
Введите число от 0 до 9:
0

C:\15\SomeEvent1\Events>
*/
```

Задача 2.

```
using System;

namespace Program1Events
{
    delegate void MyEventHandler1();
    delegate void MyEventHandler2();
    class Event1
```

```
    {
        public event MyEventHandler1 myEvent1;
        public void FiremyEvent1()
        {
            if (myEvent1 != null)
                myEvent1();
        }
    }

    class Event2
    {
        public event MyEventHandler2 myEvent2;
        public void FiremyEvent2()
        {
            if (myEvent2 != null)
                myEvent2();
        }
    }

    class Reciver1
    {
       public static void handler1()
        {
            DateTime d = DateTime.Now;
            Console.WriteLine(d);
        }
    }

    class Reciver2
    {
       public static void handler2()
        {
            //Console.WriteLine("Введено число  " + cifra);
            Console.WriteLine("Белеет парус одинокий");
        }
    }
    class Program
    {
        public static void Main()
        {
           Event1 evt1 = new Event1();
           evt1.myEvent1 += new MyEventHandler1(Reciver1.handler1);
           Event2 evt2 = new Event2();
           evt2.myEvent2 += new MyEventHandler2(Reciver2.handler2);

            int  cifra = 0;
            Console.WriteLine("Введите число от 0 до 9:");
                cifra = int.Parse(Console.ReadLine());

                if ((cifra == 0) | (cifra == 2) | (cifra == 4) | (cifra == 6)
| (cifra == 8))
                {
                    Console.WriteLine("Введено число  " + cifra);
                    evt1.FiremyEvent1();
                }
                else
                {
                    Console.WriteLine("Введено число  " + cifra);
                    evt2.FiremyEvent2();
                }

         }
    }
   }
```

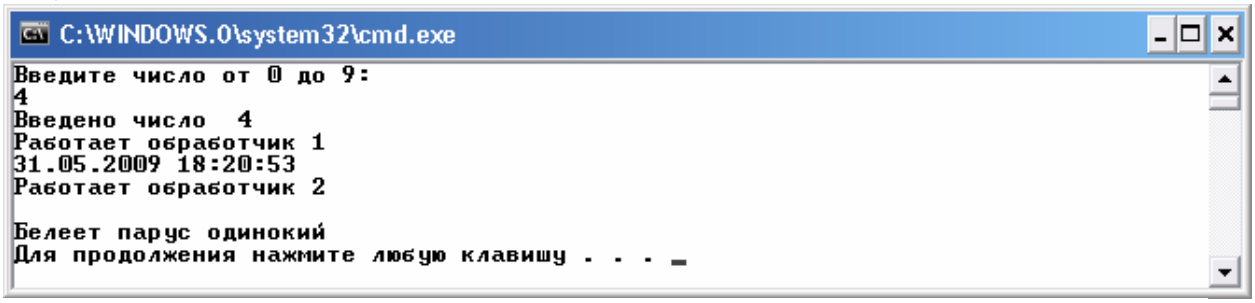
Результат:

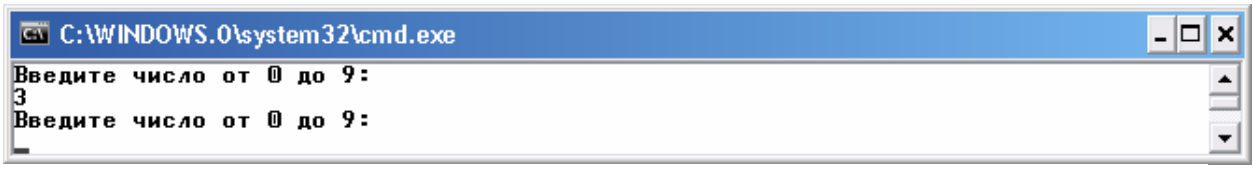
Задача 3.

```
using System;

namespace Program1Events
{
    delegate void MyEventHandler1();
    class Event1
    {
        public event MyEventHandler1 myEvent1;
        public void FiremyEvent1()
        {
            if (myEvent1 != null)
                myEvent1();
        }
    }


    class Reciver1
    {
       public static void handler1()
        {
           Console.WriteLine("Работает обработчик 1");
           DateTime d = DateTime.Now;
            Console.WriteLine(d);
        }
    }

    class Reciver2
    {
       public static void handler2()
        {
           Console.WriteLine("Работает обработчик 2");
           Console.WriteLine("");
            Console.WriteLine("Белеет парус одинокий");
        }
    }
    class Program
    {
        public static void Main()
        {
           Event1 evt1 = new Event1();
           evt1.myEvent1 += new MyEventHandler1(Reciver1.handler1);
           evt1.myEvent1 += new MyEventHandler1(Reciver2.handler2);

            int  cifra = 0;
            A:
            Console.WriteLine("Введите число от 0 до 9:");
                cifra = int.Parse(Console.ReadLine());
```

```
                if ((cifra == 0) | (cifra == 2) | (cifra == 4) | (cifra == 6)
| (cifra == 8))
                {
                    Console.WriteLine("Введено число  " + cifra);
                    evt1.FiremyEvent1();
                }
                else
                    goto A;
            }
        }
    }
```

Результат:





Принципы объектно-ориентированного программирования.

Инкапсуляция

Характерные признаки Инкапсуляции.

В основе инкапсуляции лежат следующие характерные признаки: абстракция, соединение данных и методов их обработки, общедоступный интерфейс, и сокрытие реализации (Защита полей и методов).

1. Абстракция — это процесс упрощения сложной задачи. Намереваясь решить определенную задачу, вы не стремитесь учесть все детали, а выбираете только те, которые нужны для решения.

Допустим, вам нужно составить модель дорожного движения. Очевидно, что вы создадите классы светофоров, машин, шоссе, двухсторонних и односторонних улиц, погодных условий и так далее. Каждый из этих элементов влияет на движение транспорта. Однако вы не будете создавать модели насекомых или птиц, хотя они тоже могут появиться на дороге. Более того, вы не будете выделять марки машин.

Вы упрощаете реальный мир и используете только основные элементы.

У абстракции есть два преимущества. Во-первых, она упрощает решение задачи. И, что еще более важно, во-вторых, благодаря абстракции компоненты программного обеспечения можно использовать повторно.

Компоненты программы часто чрезмерно специализированы. Специализация уменьшается, компоненты становятся более универсальными, если абстрагироваться от ненужных деталей. Универсализация компонентов позволяет облегчить повторное использование фрагмента программы в каком-либо другом месте. По мере возможности старайтесь создавать объекты, которые будут решать целый ряд задач.

Абстракция позволяет использовать решение одной задачи для решения других задач из данной предметной области.

2. Соединение данных и методов их обработки происходит в процессе формирования нового объекта.

Инкапсуляция — это способ связывания атрибутов и методов для формирования объектов. В результате объединения взаимосвязанных на концептуальном уровне элементов получается КЛАСС.

На основе класса может быть создано множество ОБЪЕКТОВ – ЭКЗЕМПЛЯРОВ КЛАССА. Каждый объект имеет свои собственные ДАННЫЕ. Для их обработки используются МЕТОДЫ, которые объект получил от своего класса. При этом объект остается независимым от других объектов. Объект сам решает, когда используются его методы.

Инкапсуляция позволяет разбить программу на множество мелких **независимых** элементов, а не рассматривать ее как некую монолитную вещь. Каждый элемент представляет собой модуль, выполняющий свои функции независимо от других элементов.

Инкапсуляция произвела революцию в способах написания программ и стала краеугольным камнем объектно-ориентированного проектирования.

Как известно метод — это определение поведения объекта. Однако в мире *процедурного* программирования методы и атрибуты не объединены вместе и не ассоциированы с объектами.

Объектно-ориентированные языки программирования позволяют программистам *инкапсулировать атрибуты и методы и связывать их с объектами, что соответствует реальному миру*. Это значительно уменьшает возможность неправильного использования атрибутов и методов.

3. Одним из основных преимуществ объектно-ориентированного подхода является возможность скрывать некоторые атрибуты и поведение одного объекта от других объектов.

Интерфейс и реализация — это две противоположности. Для повышения надёжности программы нужно защитить её поля и методы, закрыть их, спрятать. Тогда никто не сможет их «подсмотреть» или откорректировать.

Но всё спрятать нельзя -тогда с программой невозможно будет работать. Поэтому и приходится в каждой программе выделять две составляющих — интерфейс, и реализацию. Интерфейс должен быть доступным, а реализация должна быть спрятана от всех.

В хорошо разработанной программе внешние по отношению к объекту структуры имеют доступ только к необходимому для взаимодействия с ним **интерфейсу**. Доступ к остальным элементам объекта, не относящимся к его использованию, для этих структур закрыт.

Единственный способ получить доступ к атрибутам и методам объекта состоит в создании экземпляра объекта.

Объекты в программе создаются с помощью определения класса.

Инкапсуляция *позволяет* программисту:

1) после создания в классе атрибутов и методов,

2) определить правила доступа к ним.

Благодаря инкапсуляции повышается степень независимости программ, поскольку внутренние детали, или реализация, скрываются за интерфейсом.

Новый термин Инкапсуляция — это объектно-ориентированная характеристика модульности. С помощью инкапсуляции можно разделить программное обеспечение на модули, выполняющие определенные функции, детали реализации которых скрыты от внешнего мира.

По существу термин инкапсуляция означает "герметизированная; защищенная от внешних воздействий часть программы".

Защита с использованием модификаторов доступа

Программисты контролируют доступ к атрибутам и методам класса с помощью спецификаторов доступа в определении класса. Спецификатор доступа (access specifier) — это ключевое слово языка программирования, которое говорит компьютеру, какая часть программы может получить доступ к атрибутам и методам, являющимся членами класса.

Рассмотрим следующие спецификаторы доступа — public, private и protected. Спецификатор доступа public определяет атрибуты и методы, которые доступны при использовании экземпляра класса.

Спецификатор доступа private определяет атрибуты и методы, которые доступны только методам, определенным в классе.

Спецификатор доступа protected определяет атрибуты и методы, которые могут быть наследованы другими классами.

В языке C# есть еще два спецификатора доступа: internal и protected internal.

Спецификатор доступа public

Когда объявляется экземпляр класса можно использовать этот экземпляр для доступа к атрибутам и методам, определенным в части класса со спецификатором доступа public. Вы определяете часть класса со спецификатором доступа public с помощью ключевого слова public, как показано в следующем примере.

Метод Display() объявлен с помощью спецификатора доступа public. Это означает, что он может быть вызван напрямую из любой части программы с помощью объявления экземпляра класса Student. Вот пример:

```
 class Student
 {
 public void Display()
 {
 // Поместите здесь операторы
 }
… ….
```

Можно напрямую обращаться к атрибутам и методам, определенным с ключевым словом public, используя в программе имя экземпляра, оператор «точка» и название атрибута или метода, к которым надо получить доступ.

Следующая программа иллюстрирует использование спецификатора доступа public для отображения информации о студенте.

```
//  Спецификатор доступа public
1 using System;
2
3 namespace ConsAppl
4 {
5 class Student
6 {
7 public void Display()
8 {
9 Console.WriteLine("{0}", "Данные о студенте отображаются здесь.");
10 }
11 }
```

```
12 class StudentTest
13 {
14 public static void Main()
15 { // myStudent – объект (экземпляр) класса Student
16 Student myStudent = new Student();
17 myStudent.Display(); // для объекта myStudent вызывается метод
18 Console.ReadLine();
19 }
20 }
21 }
```

Первый оператор в методе Main() объявляет экземпляр класса Student (см. строку 16). Второй оператор вызывает метод Display() класса Student (строка 17).

Спецификатор доступа private и использование конструкции «метод-член»

Спецификатор доступа private ограничивает доступ к атрибутам и методам теми методами, которые являются членами того же самого класса.

Следующий пример иллюстрирует, как это делается. Цель — предотвратить прямой доступ к идентификатору студента, его имени и статусу окончания обучения при использовании экземпляра класса Student. Это осуществляется с помощью спецификатора доступа private.

Спецификатор доступа private не запрещает методу Display() обращаться к этим атрибутам (строки 6, 7), так как метод Display() (строки 9…12) является членом класса Student (строки 4…13).

Для использования других членов класса (атрибуты и методы) из методов этого же класса создавать экземпляр класса не нужно:

```
// Листинг 4. Спецификатор доступа private
1 using System;
2 namespace ConsAppl
3 {
4 class Student
5 {
6 int m_ID = 1, m_Graduation = 5; // здесь не нужны спецификаторы доступа
7 String m_First = "Ivan" , m_Last = "Petrov"; // - “ - “- “ -
8
9 public void Display() // сделать эксперимент: заменить public на private:
возникнет ошибка
10 {
11 Console.WriteLine("Идентификатор студента - " + m_ID + "; Имя - " +
m_First + ";
Фамилия - " + m_Last + ". Специалистов - " + m_Graduation);
12 }
13 }
14 class StudentTest
15 {
16 public static void Main()
17 {
18 Student myStudent = new Student();
19 myStudent.Display();
20 Console.ReadLine();
21 }
22 }
23 }
```

Техника, показанная в этом примере, — краеугольный камень в объектно-ориентированном программировании, потому что она требует использовать метод-член (это метод Display()) для доступа к атрибутам класса. Это позволяет программисту,

который создал класс, закодировать в методе-члене правила, управляющие доступом к атрибутам.

Представьте, что нужно отобразить информацию о студенте. Но доступа напрямую к информации о студенте нет. Для получения доступа необходимо вызвать метод-член класса, отображающий информацию о студенте. Это дает программисту, который создал класс, полный контроль над тем, к каким атрибутам можно получить доступ и как они будут отображены.

Спецификатор доступа protected

Спецификатор доступа protected определяет атрибуты и методы, которые могут быть использованы:

1. только методами, являющимися членами класса,
2. а также методами, которые являются членами производных классов.

Класс, от которого производится наследование, называется *базовым*, а класс, который наследуется от базового класса, *производным* классом.

Наследование рассмотрим на следующей лекции, здесь же рассмотрим его поверхностно, чтобы можно было понять, как работает спецификатор доступа protected. Предположим, что у нас есть два класса. Один класс называется Student, а другой — GradStudent.

Класс Student содержит атрибуты и варианты поведения, которые характерны для всех студентов. Класс GradStudent содержит атрибуты и варианты поведения, которые уникальны для *аспирантов*, а также включает все атрибуты и варианты поведения *студентов*. Аспирант является студентом.

Вместо того чтобы дублировать атрибуты и варианты поведения класса Student в классе GradStudent, можно сделать так, чтобы класс GradStudent наследовал все или некоторые атрибуты и варианты поведения класса Student.

Атрибуты и варианты поведения, определенные с помощью спецификаторов доступа public и protected, можно напрямую использовать в классе GradStudent.

Следущая программа (листинг 5) на C# показывает, как использовать спецификатор доступа protected.

В ней объявлены три класса. Первые два определения класса такие же, как и в примере со спецификатором доступа private. Третье определение класса — новое. Это определение аспиранта, называется этот класс GradStudent. Класс GradStudent наследует атрибуты и методы-члены класса Student, что заложено в строке 18 кода (см. ниже).

Листинг 5. Использование спецификатора доступа protected в C#

```
1 using System; // Спецификатор доступа protected в действии
2 namespace ConsAppl
3 {
4 class Student // базовый класс Student
5 {
6 // значения следующих переменных поступают из класса GradStudent, так как
//их спецификатор доступности protected;
7 protected int m_ID, m_Graduation;
8 // Сделать эксперимент: если protected заменить на private, то возникнет
//ошибка
9 protected String m_First;
10 protected String m_Last;
11
12 public void Display()
13 { // 4. Печать 1-й строки (строка 14)
14 Console.WriteLine(" Студент: " + m_ID + "; " + m_First + " " + m_Last + ";
Дипломирован: " + m_Graduation);
15 }
```

```
16 } ////////////////// конец базового класса Student //////////////////
17
18 class GradStudent : Student //производный класс GradStudent
19 {
20 private int YearGraduated;
21 private String m_UndergradSchool;
22 private String m_Major;
23 // Конструктор GradStudent() класса GradStudent
24 public GradStudent(int ID, int Grad, String Fname, String Lname, int
yrGrad, String unSch, String major)
25 {
26 m_ID = ID; // идентификатор
27 m_Graduation = Grad; // индикатор окончания вуза
28 YearGraduated = yrGrad; // год окончания вуза
29 m_First = Fname; // Имя
30 m_Last = Lname; // Фамилия
41 m_UndergradSchool = unSch; // название вуза
42 m_Major = major; // специализация
43 }
44
45 public void GradDisplay()
46 {
47
48 base.Display(); // 3. Обращение к м-ду Display() базового класса Student
//(строки 12…15)
49 Console.WriteLine(" " + m_UndergradSchool + ", " + m_Major + ", " +
YearGraduated);
// 5. Печать 2-й строки результата (строка 49)
50 }
51 } /////////// конец производного класса GradStudent //////////////////
52
53 class StudentTest /////////// класс StudentTest ////////////////////
54 {
55 public static void Main(String [ ] args)
56 { // 1. Обращение к конструктору GradStudent() и инициализация (строки
//24…43)
57 GradStudent myStudent = new GradStudent(10, 1, "Иван", "Петров", 2004,
"ГУ-ВШЭ", "Computer Science");
58 myStudent.GradDisplay(); // 2. Обращение к м-ду GradDisplay() класса
//GradStudent (строки 45…50)
59 Console.ReadLine();
60 }
61 } ////////////// конец класса StudentTest ////////////////////
62 }
```

Из строки 18

```
18: class GradStudent : Student
```

следует, что класс GradStudent является производным от базового класса Student. На это указывает следущая конструкция в этой строке – “ : Student “. Таким образом, класс GradStudent наследует атрибуты (строки 7…10) и метод-член Display() (строки 12…15) класса Student.

Для обращения к методу Display() в строке 48 использовано ключевое слово “ base ” и оператор уточнения “ . ” .

```
48: base.Display();
```

Определение класса Student практически идентично определению этого класса в примере со спецификатором доступа private (см. в листинге 4 строки 6, 7), за одним исключением. Обратите внимание, что перед названиями атрибутов стоит ключевое слово protected (см. в листинге 5 строки 7…10), которое сообщает компьютеру, что к этим атрибутам можно обращаться из методов-членов класса GradStudent.

Экземпляр myStudent класса GradStudent объявляется в методе Main() класса StudentTest (см. строку 57), а затем используется для присвоения значений атрибутам

класса GradStudent и атрибутам, унаследованным от класса Student (см. строки 24…43). Затем эти значения выводятся на экран с помощью вызова метода GradDisplay() класса GradStudent (см. строку 58).

Обратите внимание, что метод GradDisplay() класса GradStudent (см. строки 45…50) немного отличается от метода Display() класса Student (см. строки 12…15) . Рассмотрим это подробнее и начнем с метода Display() класса Student.

Метод Display() класса Student отображает значения атрибутов, определенных в классе Student.

Метод GradDisplay() класса GradStudent расширяет возможности метода Display() класса Student — он отображает атрибуты и класса Student, и класса GradStudent.

Вот как это делается: вспомните, что класс GradStudent может наследовать не только public-, но и protected-члены класса Student. Это означает, что метод GradDisplay() класса GradStudent может вызвать метод Display() класса Student (см. строку 48). Класс, от которого производится наследование, называется базовым, для доступа к его атрибутам и методам-членам используется ключевое слово base.

В этой программе оператор base.Display() сообщает компьютеру, что необходимо вызвать метод Display() класса Student, который выводит атрибуты класса Student на экран (1-я строка вывода на экран). Следующий оператор (см. строку 49) отображает атрибуты класса GradStudent на 2-й строке вывода.

Интерфейс является основным средством связи между объектами.

Каждый класс определяет интерфейс, позволяющий создавать объекты этого класса и обеспечивать их правильное функционирование. Любое выполняемое объектом действие должно вызываться сообщением, использующим один из предоставленных интерфейсов. Интерфейс должен полностью описывать правила взаимодействия класса с его пользователями. В языке C# для объявления методов, которые являются частью интерфейса, используется ключевое слово public.

Скрытые (затенённые) данные

Согласно принципам объектно-ориентированного проектирования все атрибуты объекта должны быть объявлены с использованием модификатора доступа private. Таким образом, атрибуты не являются частью интерфейса. В интерфейс класса входят только методы, объявленные как public.

Объявление атрибутов как public противоречит концепции затенения данных.

Обычно интерфейс класса не включает в себя атрибуты (то есть данные) объекта - только его методы.

Если пользователю необходимо получить доступ к некоторому атрибуту, в *рамках объекта* должен быть создан метод, позволяющий возвращать значение этого атрибута (метод возврата значения).

Когда пользователю понадобится соответствующая информация, то вызванный метод вернет требуемое значение.

Таким образом, доступ к атрибуту контролируется содержащим его объектом. Контроль над доступом к атрибутам объекта чрезвычайно важен, особенно с точки зрения тестирования и последующего сопровождения программы.

Если вы контролируете доступ к атрибуту, то в случае возникновения проблемы вам не придется отслеживать каждый фрагмент кода, который мог бы изменить значение этого атрибута, поскольку существует только один способ его изменения (с использованием метода присвоения значения).

Интерфейсы классов

Помимо интерфейсов классов существуют также интерфейсы методов.
Интерфейс класса – это public-методы, доступ к ним открыт для объектов

Интерфейсы методов

Интерфейсы методов описывают способ вызова последних. Другими словами, интерфейс метода – это его заголовок (сигнатура - первая строка)

Реализация

Интерфейс составляют только те атрибуты и методы, доступ к которым открыт (public) для других объектов. Пользователь не должен иметь доступ к реализации, взаимодействуя с классом исключительно через его интерфейс.

Пример применения концепции отделения интерфейса от реализации в реальном мире

Диаграмма класса IntSquare соответствует следующему программному коду:

```
1 using System; // Листинг 1. Инкапсуляция: релизация и интерфейс
2
3 namespace ConsAppl_Weisf_c32_IntSquare
4 {
5 public class IntSquare //////// класс IntSquare ////////////////////
6 {
7 private int squareValue; // скрытый атрибут
8 // интерфейс класса, состоящий из одного метода, открытого для пользователя
9 public int getSquare(int value)
10 {
11 squareValue = calculateSquare(value);
12 return squareValue;
13 }
14
15 private int calculateSquare(int value) // реализация, скрытая от
//пользователя
16 { /* другая реализация метода - Math.Exp(2 * Math.Log(value)); эта
реализация с существенно большими возможностями: любое число возводится в
любую степень. Необходимо везде заменить int на double */
17 return value * value;
18 }
19 }  //--------------------------------
20 public class IntSquareTest
21 {
22 public static void Main(String [ ] args)
23 {
24 IntSquare intSquare = new IntSquare();
25 Console.WriteLine("Квадрат числа:" + intSquare.getSquare(25));
26 Console.ReadLine();
27 }
28 }  //================================
29 }
```

Единственной частью класса, к которой пользователь имеет доступ, является метод getSquare (получитьКвадрат, строки 9…12), который и представляет собой интерфейс.

Реализация алгоритма вычисления квадрата содержится в скрытом методе calculateSquare (вычислитьКвадрат, строки 15…18).

Атрибут squareValue (квадратЧисла, строка 7) также является скрытым, поскольку пользователям необязательно знать о его существовании.

Таким образом, мы сделали раздел реализации недоступным для других объектов. Объект позволяет пользователю получать доступ только к необходимому для взаимодействия с ним интерфейсу, в то время как не имеющие отношения к использованию объекта подробности скрыты от других объектов.

Если вам понадобится внести изменения в раздел реализации - например, вы решите использовать встроенную в язык C# функцию вычисления квадрата числа, - вам не придется менять интерфейс класса.

Пользователь сможет получать результат прежним способом, но реализация вычислений будет другой. Такой подход очень важен при разработке кода, имеющего дело с данными; например, вы можете переносить информацию из файла в базу данных, не заставляя при этом пользователя менять код какого-либо приложения.

Следование принципу инкапсуляции позволяет защитить внутренние данные класса от неумышленного повреждения. Для этого достаточно все внутренние данные сделать закрытыми (объявив внутренние переменные с использованием ключевых слов private или protected).

Работа с внутренними данными.

Для обращения к внутренним данным можно использовать один из трёх способов:

- создать традиционную пару методов :
  - *а)* для получения информации (accessor),
  - *б)* для внесения изменений (mutator);
- определить именованное свойство.
- использовать ключевое слово readonly.

Однако какой бы способ не был выбран, общий принцип остается тем же самым — *инкапсулированный класс должен прятать детали своей реализации от внешнего мира*. Такой подход часто называется «программированием по методу черного ящика».

Еще одно преимущество такого подхода заключается в том, что можно как угодно совершенствовать внутреннюю реализацию класса, полностью изменяя его содержимое.

Единственное, о чем вам придется позаботиться, — чтобы в новой реализации остались методы с той же сигнатурой и функциональностью, что и в предыдущих версиях. В этом случае не придется менять ни строчки существующего кода за пределами данного класса.

Первый способ инкапсуляции: при помощи традиционных методов доступа и изменения

Рассмотрим класс Employee (Служащий). Если необходимо, чтобы внешний мир смог работать с внутренними данными private этого класса (пусть это будет только одна переменная fullName, для которой используется тип данных string), то традиционный подход рекомендует создание метода доступа (accessor, или get method) и метода изменения (mutator, или set method).

Набор таких методов может выглядеть следующим образом:

```
01 /*Определение традиционных методов доступа и изменения для закрытой
переменной private
(см. ConsAppl_Troel_c151_инкапсл) */
02 class Employee
03 {
04 private string fullName; // внутренняя (private) переменная
05 // Метод доступа (accessor)
06 public string GetFullName()
07 {
08 return fullName;
09 }
```

```
10 // Метод изменения (mutator)
11 public void SetFullName(string n)
12 {
13 // Логика для удаления неположенных символов (!. @. #, $. % и прочих
14 // Логика для проверки максимальной длины и прочего
15 fullName = n;
16 }
17 }
```

Такой подход требует наличия двух методов для взаимодействия с каждой из переменных.

Вызов этих методов может выглядеть следующим образом:

```
18 /* Применение методов доступа и изменения (См.
ConsAppl_Troel_c151_инкапсл)  */
19 public static void Main(string [ ] args)
20 {
21 Employee p = new Employee();
22 p.SetFullName("Алексей");
23 Console.WriteLine("Имя служащего: " + p.GetFullName());
24
25 // Ошибка! К закрытым данным нельзя обращаться напрямую через экземпляр
объекта!
26 // p.FullName; // в этой строке ошибка: сделать эксперимент - убрать «//»
и запуск программы
27 }
```

Второй способ инкапсуляции: применение свойств класса

Помимо традиционных методов доступа и изменения для обращения к закрытым private членам класса можно также использовать свойства (properties).

Свойства позволяют имитировать доступ к внутренним (private) данным класса: при получении информации или внесении изменений через свойство синтаксис выглядит так же, как при обращении к обычной открытой (public) переменной. Но на самом деле любое свойство состоит из двух скрытых внутренних методов.

Преимущество свойств заключается в том, что вместо того, чтобы использовать два отдельных метода, пользователь класса может использовать единственное свойство, работая с ним так же, как и с открытой переменной-членом данного класса:

```
01 // Обращение к имени сотрудника через свойство
02 public static void Main(string [ ] args)
03 {
04 Employee p = new Employee();
05 p.EmpID = 81; // Устанавливаем значение (set)
06
07 Console.WriteLine("Идентификатор сотрудника: {0}", p.EmpID); // Получаем
значение (get)
08 Console.ReadLine();
09 }
```

Если заглянуть внутрь определения класса, то свойства всегда отображаются в «реальные» методы доступа и изменения. А уже в определении этих методов вы можете реализовать любую логику (например, для устранения лишних символов, проверки допустимости вводимых числовых значений и прочего).

Ниже представлен синтаксис класса **Employee** с определением свойства EmpID:

```
01 // Пользовательское свойство EmpID для доступа к переменной empID
02 public class Employee ////////// начало класса Employee //////////
03 {
04 private int empID; // empID - закрытая (private) переменная
```

```
05 // Свойство для empID
06 public int EmpID ////////////////// начало свойства EmpID ///////////////
07 {
08 get {return empID;}
09 set
10 {
11 // Здесь вы можете реализовать логику для проверки вводимых
12 // значений и выполнения других действий
13 empID = value;
14 }
15 } ////////////////////// конец свойства EmpID // /////////////////
15 } ////////////////////// конец класса Employee ////////////////////
```

Свойство C# состоит из двух блоков — блока доступа (get block) и блока изменения (set block).

Ключевое слово value представляет правую часть выражения при присвоении значения посредством свойства.

Как и все в C#, то, что представлено словом value — это также объект. Совместимость того, что передается свойству как value, с самим свойством, зависит от определения свойства.

Например, свойство EmpID предназначено (согласно своему определению в классе) для работы с закрытым целочисленным empID, поэтому число 81 вполне его устроит:

// В данном случае типом данных, используемым для value, будет
int p.EmpID = 81;

Показать дополнительные возможности применения ключевого слова value можно на таком при мере (GetType() и ToString():

```
01 public int EmpID // Свойство для empID
02 { //(см. Troelsen_c153_инкапсул)
03 get { return empID; }
04 set
05 {
06 // Как еще можно использовать value
07 Console.WriteLine("value - указывает тип: {0}", value.GetType());
08 Console.WriteLine("value - строка: {0}", value.ToString());
09 empID = value;
10 }
11 }
```

Результат работы данной программы представлен на рис. 5.

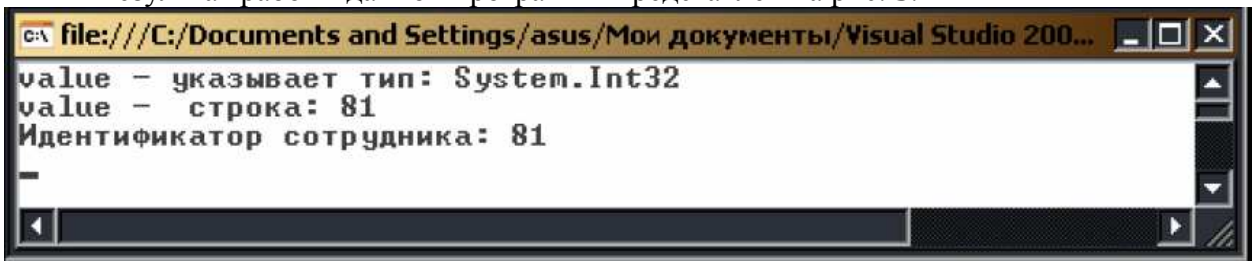


Рис. 5. Значение «value» при EmpID = 81

Необходимо отметить, что обращаться к объекту **value** можно только в пределах программного блока **set** внутри определения свойства. Попытка обратиться к этому объекту из любого другого места приведет к ошибке компилятора.

Внутреннее представление свойств C#

Многие программисты стараются использовать для имен традиционных методов доступа и изменения соответственно приставки get_ и set_ (например, get_Name() и set_Name()). Само по себе это не представляет проблемы. Проблему представляет другое: C# для внутреннего представления свойства использует методы с теми же самыми префиксами. Поэтому подобное определение класса вызовет ошибку компилятора.

Свойства: только для чтения, только для записи и статические

Свойство EmpID было создано как свойство, доступное:

1) и для чтения,

2) и для записи.

Однако при создании пользовательских свойств класса часто возникает необходимость создать свойство, которое будет доступно только для чтения.

Делается это очень просто: необходимо в определении свойства пропустить блок set.

В результате свойство станет доступным только для чтения:

```
1 public class Employee
2 /* Будем считать, что исходное значение переменной ssn
присваивается с помощью конструктора класса */
3 private string ssn; // закрытая переменая ssn
4 // А вот так выглядит свойство Ssn только для чтения значения ssn
5 public string Ssn { get { return ssn; } }
… ….
}
```

C# также поддерживает статические свойства.

Статические переменные предназначены для хранения информации на уровне всего класса, а не его отдельных объектов.

Если объявлены статические данные (то есть те же переменные), то обращаться к ним и устанавливать значения должны статические свойства.

Предположим, что в классе Employee необходимо, помимо всего прочего, хранить еще и информацию об имени организации, в которой работают все сотрудники — объекты класса Employee. Для этого будет использована специальная статическая переменная (см. строку 6).

Соответствующее статическое свойство для работы с этой переменной представлено в листинге 9 (см. строки 8…12).

Обращение к статическим свойствам производится так же, как и к статическим методам. В обращении указываются: имя класса, оператор «точка», имя свойства (см. строки 20, 21).

```
1 using System; /* Со статическими данными должны работать статические
свойства   */
2 namespace ConsAppl  // листинг 9
3 {
4 public class Employee
5 {
6 private static string compName; // Статическая переменная compName
7
8 public static string Company // Статическое свойство Company
9 {
10 get { return compName; }
11 set { compName = value; }
12 }
13 // ...
14 }
15 public class TestEmployee
16 {
17 // Задаем и получаем информацию об имени компании
18 public static void Main(string [ ] args)

19 { /* Два обращения к свойству Company: объект (экземпляр) класса Employee
для обращения к статической переменной создавать не нужно, т.к. в обращении
указывается имя класса: Employee.Company*/
20 Employee.Company = "Intertech.Inc"; // Задаем информацию на уровне класса (set)
21 // Получаем информацию на уровне класса (get)
Console.WriteLine("Эти люди работают в {0}", Employee.Company);
22 //...
```

```
23 Console.ReadLine();
24 }
25 }
26 }
```

Статические конструкторы

Само словосочетание «статический конструктор» звучит несколько странно — ведь конструкторы нужны для создания объектов, а все, что имеет определение «статический», работает на уровне классов, а не отдельных объектов. Однако в C# такие конструкторы вполне имеют право на существование. Единственное их назначение — присваивать исходные значения статическим переменным.

С точки зрения синтаксиса статические конструкторы — это достаточно причудливые образования. Например, для них *нельзя* использовать модификаторы области видимости, однако слово static должно присутствовать обязательно (см. строки 9…12).

Вот пример ситуации, в которой может пригодиться статический конструктор: предположим, что необходимо, чтобы статической переменной compName (см. строку 7) всегда при создании присваивалось значение Intertech.Inc.

При использовании свойства Company (см. строки 13…16), присваивать ему исходное значение не придется — статический конструктор сделает это автоматически (см. строку 11).

Это можно осуществляется следующим образом (листинг 10):

```
1 using System; /* Статические конструкторы используются ТОЛЬКО
для инициализации статических переменных*/
2 namespace ConsAppl  //(листинг 10)
3 {
4 public class Employee
5 {
6 //...
7 private static string compName; // статическая переменная
8
9 static Employee() // статический конструктор срабатывает АВТОМАТИЧЕСКИ
10 {
11 compName = "Intertech.Inc";
12 }
13 public static string Company // Статическое свойство Company только для
чтения
14 {
15 get { return compName; }
16 }
23
17
18 //...
19 }
20 public class TestEmployee
21 { /* Значение ("Intertech.Inc") свойства Company будет АВТОМАТИЧЕСКИ
установлено через статический конструктор*/
22 public static void Main(string [ ] args)
23 {
24 // Обращение к статическому свойству Company
25 Console.WriteLine(" Эти люди работают в {0}", Employee.Company);
26 Console.ReadLine();
27 }
28 }
29 }
```

Подводя итоги этого раздела, можно отметить, что свойства классов C# используются для тех же самых целей, что и традиционные методы доступа и изменения значений.

Главное преимущество свойств заключается в том, что пользователь может работать через них со внутренними (private) данными, используя *единственное* имя (вместо двух разных имен методов).

Псевдоинкапсуляция: создание полей «только для чтения»

Помимо свойств только для чтения, в C# также предусмотрены поля, значения которых *изменять нельзя*.

Поля (fields) — это открытые (public) данные класса.

Обычно применение полей в рабочем приложении — не самая лучшая идея, поскольку поля беззащитны — им легко присвоить ошибочное значение и тем самым испортить внутреннее состояние объекта.

Однако в C# предусмотрена возможность запретить любую возможность изменять значение поля, объявив его с ключевым словом readonly:

```
1 public class Employee
2 {
3 ….
4 // Поле только для чтения (его значение устанавливается конструктором)
5 public readonly string ssnField;
6 }
```

Любая попытка изменить значение такого поля приведет к ошибке компилятора.

Статические поля «только для чтения»

Статические поля, определенные как «только для чтения», также вполне имеют право на существование. Обычно они используются в тех ситуациях, когда вы хотите создать некоторое количество постоянных значений, связанных с определенным классом.

Очень похожие задачи выполняют обычные константы, которые можно назвать родственниками статических полей только для чтения.

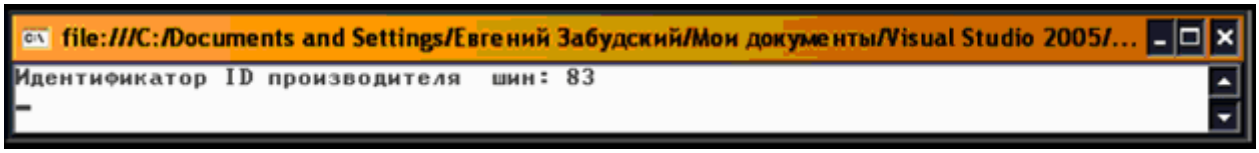
Однако между такими полями и константами есть существенное различие: константа заменяется на свое значение уже в процессе компиляции, в то время как значения статических полей только для чтения вычисляются лишь в процессе выполнения программы.

Например, предположим, что есть объект Car (автомобиль), который в процессе выполнения должен создавать объект Tire (шины).

Для этой цели можно применить класс Tire, использовав в нем набор статических полей только для чтения (листинг 11):

```
1 using System; // В классе Tire определен набор статических полей только для
чтения
2 //(листинг 11):
3 namespace ConsAppl
4 {
5 public class Tire
6 {
7 public static readonly Tire GoodStone = new Tire(90); //статические поля
только для чтения
8 public static readonly Tire FireYear = new Tire(100); // -"-
9 public static readonly Tire ReadyLine = new Tire(43); // -"-
10 public static readonly Tire Blimpy = new Tire(83); // -"-
11
12 private int manufacturelD; // закрытая переменная
13
14 public int MakelD // Свойство MakelD для доступа к закрытой переменной
```

```
15 {
16 get { return manufacturelD; }
17 }
18
19 public Tire(int ID) // Конструктор (!) Tire(int ID) класса Tire
20 {
21 manufacturelD = ID;
22 }
23 } ////////////////// конец класса Tire ////////////////////////////
24 // Так можно использовать динамически создаваемые поля только для чтения
25 public class Car
26 {
27 // Какая у меня марка шин?
28 public Tire tireType = Tire.Blimpy; // Возвращает новый объект Tire
29 } ////////////////// конец класса Car ////////////////////////////
30 public class CarApp
31 {
32 public static void Main(string[ ] args)
33 {
34 Car c = new Car();
35 //Выводим на консоль идентификатор производителя шин (в нашем случае - 83)
36 Console.WriteLine("Идентификатор ID производителя шин: {0}",
c.tireType.MakelD);
37 Console.ReadLine();
38 }
39 } ////////////// конец класса CarApp ////////////////////////////
40 }
```



Управление видимостью элементов типа с помощью модификаторов

Классы языка C# содержат такие элементы, как индексаторы, методы и свойства, которые помогают инкапсулировать внутреннее состояние типа.

Видимость элементов типа управляется с помощью модификаторов, приведенных в таблице 1.

Эти модификаторы управляют открытым интерфейсом типа для внешних типов.

Таблица 1. Модификаторы видимости

Модификатор видимости		**Кто может видеть**
public	(открытый)	Все другие типы
private	(закрытый)	Только внутри этого типа
protected	(защищенный)	Производные типы
internal	(внутренний)	Типы внутри той же программной сборки
protected internal	(внутренний защищенный)	Типы внутри той же программной сборки и производные типы

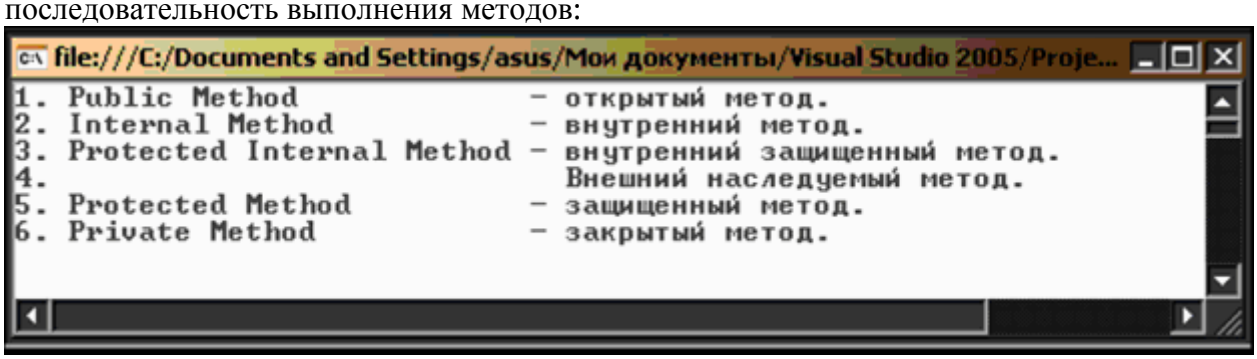
В листингах 12 (клиент) и 13 (компонент) приведены примеры использования модификаторов видимости для управления открытым интерфейсом класса.

```
01 using System; /* Вызывающий класс для демонстрации модификаторов
видимости. Используется dll-файл ClassLib_Mayo.dll – см. листинг 13 */
```

```
02 using ClassLib_Mayo;
// присоединение пространства имен dll-файла ClassLib_Mayo.dll
03 namespace ConsAppl_Mayo // Листинг 12
04 {
05 internal class ExternalDerived : PublicClass /// базовый класс
ExternalDerived
06 {
07 internal void ExternalDerivedMethod()
08 {
09 Console.WriteLine("4. Внешний наследуемый метод");
10 ProtectedMethod();
11 }
12 }
13
14 class Visibility /////////////// класс Visibility ///////////////
15 {
16 static void Main()
17 {
18 PublicClass myPublicClass = new PublicClass();
19 myPublicClass.PublicMethod();
20
21 ExternalDerived myExternalDerived = new ExternalDerived();
22 myExternalDerived.ExternalDerivedMethod();
23
24 Console.ReadLine();
25 }
26 } ///////////////// окончание класса Visibility ////////////
27 }
01 using System; /* Вызываемый класс используется для демонстрации
модификаторов видимости. Из этого кода сформирован dll-файл –
ClassLib_Mayo.dll */
02 namespace ClassLib_Mayo // Листинг 13
03 {
04 internal class BaseInternal //////// базовый класс BaseInternal//////
05 {
06 public void BaseInternalMethod()
07 {
08 Console.WriteLine("2. Internal Method - внутренний метод.");
09 }
10
11 protected internal void ProtectedInternalMethod()
12 {
13 Console.WriteLine("3. Protected Internal Method - внутренний защищенный
метод.");
14 }
15 } ////////////// конец базового класса BaseInternal ///////////////
16
17 internal class DerivedInternal : BaseInternal ///// производный класс
DerivedInternal
18 {
19 internal void DerivedInternalMethod()
20 {
21 ProtectedInternalMethod();
22 }
23 } ////////////////// конец производного класса DerivedInternal ///////
24
25 public class PublicClass ////////// класс PublicClass //////////////
26 {
27 public void PublicMethod()
28 {
29 Console.WriteLine("1. Public Method - открытый метод.");
30
27
31 BaseInternal myBaseInternal = new BaseInternal();
```

```
32 myBaseInternal.BaseInternalMethod();
33
34 DerivedInternal myDerivedInternal = new DerivedInternal();
35 myDerivedInternal.DerivedInternalMethod();
36 }
37
38 protected void ProtectedMethod()
39 {
40 Console.WriteLine("5. Protected Method - защищенный метод.");
41 PrivateMethod();
42 }
43
44 private void PrivateMethod()
45 {
46 Console.WriteLine("6. Private Method - закрытый метод.");
47 }
48 } ////////////////// конец класса PublicClass ////////////////////////
49 }
```

Ниже приводятся выходные данные работы этой программы, демонстрирующие последовательность выполнения методов:



Посмотрите внимательно каждый модификатор в листингах 12 и 13, перед тем как читать дальнейшие объяснения.

Убедитесь в том, что листинги 12 и 13 относятся к разным проектам.

Листинг 12 будет консольным приложением (cs-файл), а листинг 13 — библиотекой классов (dll-файл).

Консольное приложение создается в VS .NET известным образом.

Наследование.

Классификация и агрегация.

Различается два типа отношений между объектами: классификация и агрегация.

Классификация – это отношение типа «является, или есть_некоторый», оно связывает объекты в иерархическую зависимость. Например, «стол – это мебель», «треугольник – это геометрическая фигура». В этих определениях понятия принадлежат к разным уровням иерархии. Такие взаимоотношения в C# называются “Is-A”, является.

Агрегация – это отношение типа «есть_часть, или входит_в_состав». Противоположной этому отношению является связь: «состоит из». Например, «двигатель – есть_часть автомобиля», «автомобиль состоит_из…». В этих определениях одни понятия являюся составными частями других. Такие взаимоотношения в C# называются “Has-A”, содержит.

При классификационной связи переход с одного уровня иерархии на другой связан с понятием наследования.

Наследование — это механизм, который дает возможность создавать новый (производный) класс на основе определения уже существующего (базового) класса. C

помощью механизма наследования производный класс наследует все свойства и поведение, представленные в базовом классе.

Все методы и свойства интерфейса базового класса автоматически появляются в интерфейсе производного класса.

Рассмотрим следующий класс: Листинг 1.

```
5 public class Employee /* Листинг 1. */
6 { /* Класс Employee - базовый. Объекты этого класса
могут появиться в системе платежной ведомости базы данных */
7 // этот класс соответствует любому конкретному служащему – объекту класса
8 private String first_name;
9 private String last_name;
10 private double wage;
11
12 public Employee(String first_name, String last_name, double wage)
13 { /* конст-р с тремя параметрами базового класса,
к нему обращание из производного класса CommissionedEmployee */
14 this.first_name = first_name;
15 this.last_name = last_name;
16 this.wage = wage;
17 }
18
19 public double Wage // свойство Wage – возвращает размер Базовой зарплаты,
это закрытая переменная
20 {
21 get
22 {
23 return wage;
24 }
25 }
26
27 public String First_Name // свойство First_Name - возвращает имя, это
закрытая переменная
28 {
29 get
30 {
31 return first_name;
32 }
33 }
34
35 public String Last_Name // свойство Last_Name - возвращает фамилию, это
закрытая переменная
36 {
37 get
38 {
39 return last_name;
40 }
41 }
42 } ///////// Конец класса Employee ///////////////
```

Экземпляры класса, подобно Employee, могут понадобиться в прикладной системе с базой данных для расчета платежной ведомости. Представим себе, что нам необходимо смоделировать служащего, получающего ставку комиссионного вознаграждения. Такой служащий получает основной оклад плюс небольшую комиссионную надбавку за каждую продажу. Кроме этого простого требования, класс CommissionedEmployee точно такой же, как и Employee.

Непосредственно используя инкапсуляцию, можно написать новый класс CommissionedEmployee. Для этого нужно переписать программу Employee, добавив коды, необходимые для подсчета комиссионных и суммарного жалования. В результате придется сопровождать два отдельных, но похожих в основе кода.

Когда надо будет найти ошибку, искать придется всюду. То есть это не лучший способ создания класса CommissionedEmployee.

С помощью наследования можно рациональнее создать лучший класс CommissionedEmployee.

```
44 public class CommissionedEmployee : Employee // Листинг 2
45 { /*Производный класс CommissionedEmployee ДЕМОНСТРИРУЕТ
Наследство, унаследованное от базового класса Employee*/
46
47 private double commission; // - в $ за единицу проданного товара
48 private int units; // - следит за количеством проданных устройств
49 // Обращение к конст-ру Employee() (базового класса) с тремя параметрами
50 public CommissionedEmployee(String first_name, String last_name, double
wage,double commission) : base(first_name, last_name, wage)
51 { // ключевое слово base дает доступ к реализации базового класса
52 this.commission = commission;
53 }
54
55 public double CalculatePay() /* метод:
расчет итоговой зарплаты с учетом комиссионных; обращение к свойству Wage*/
56 {
57 return Wage + ( commission * units );
58 }
59
60 public int AddSales(int units) // метод: вычисление количества проданных
единиц товара
61 {
62 return this.units += units;
63 }
64 public int ResetSales() /* метод: сброс количества проданных единиц товара
в ноль, units - закрытая переменная*/
65 {
66 return units = 0;
67 }
68
69 } /////////////// Конец класса CommissionedEmployee //////////////
```

В определении класса CommissionedEmployee используется уже существующий (базовый) класс Employee (см. строку 44). Из-за того, что класс CommissionedEmployee является наследником класса Employee, частью определения класса CommissionedEmployee стали свойства First_Name, Last_Name и Wage, и закрытые переменные first_name, last_name и wage (см. код класса Employee).

Так как общедоступный интерфейс класса Employee стал частью интерфейса класса CommissionedEmployee, классу CommissionedEmployee можно посылать все те сообщения, которые можно посылать и классу Employee.

Ниже это продемонстрировано в методе Main().

```
75 public class EmployeeExample // Листинг 3
76 {
77 public static void Main(String[] args)
78 { // За единицу проданного товара Иван Петров получает комиссионные,
commission = $1.00
79 CommissionedEmployee c = new CommissionedEmployee("Иван", "Петров", 5.50,
1.00);
80 c.AddSales(4); // см. строку 60
81 Console.WriteLine( "\nИмя: " + c.First_Name );
82 Console.WriteLine("\nФамилия: " + c.Last_Name);
83 Console.WriteLine("\nБазовая зарплата: $" + c.Wage);
84 Console.WriteLine("\n\nКоличество проданных единиц товара: " +
c.AddSales(0));
85 Console.WriteLine("\nИтоговая зарплата: $" + c.CalculatePay());
```

```
86 c.ResetSales(); // см. строку 64
87 Console.WriteLine("\n\nСброс количества проданных единиц товара: " +
c.ResetSales());
88 c.AddSales(2); // см. строку 60
89 Console.WriteLine("\n\nКоличество проданных единиц товара: " +
c.AddSales(0)); // строка 55
90 Console.WriteLine("\nИтоговая зарплата: $" + c.CalculatePay());
91 Console.ReadLine();
92 } /////////////// конец метода Main() /////////////////
93 } ////////////// Конец класса EmployeeExample ////////////////////
```

На рис. 1 показано, что должно получиться после выполнения этого кода.

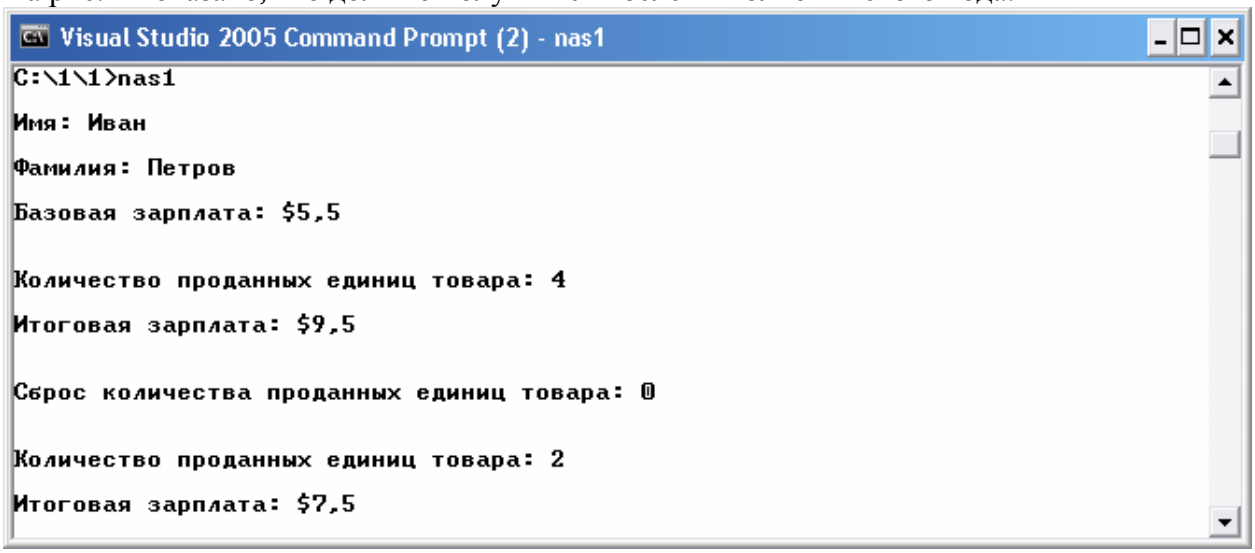


Рис. 1. Результат, выданный классом CommissionedEmployee

Наследование позволяет в классе-наследнике (производном классе) переопределять любое поведение (то есть метод), если оно чем-то вам не подходит. Такая полезная особенность позволяет адаптировать программное обеспечение к изменению требований.

Если необходимо внести изменения:

1) напишите новый класс (см. выше строки 44…69), наследующий функции старого (см. выше строки 5…42);

2) переопределите методы, которые надо изменить;

3) или добавьте недостающие методы (см. выше строки 55…67).

Ценность такой подмены выражается в том, что она позволяет изменить работу объекта, не изменяя определение первоначального (базового) класса!

Ведь в этом случае тщательно протестированные, проверенные на правильность основные (базовые) классы можно оставить нетронутыми.

Подмена работает даже тогда, когда у вас нет исходного текста класса.

Наследование реализует функцию «Классификация». Кроме наследования существует ещё «включение» или «агрегация». Агрегат образуется, когда в объект одного класса включаются объекты других классов, являющиеся составными частями основного объекта.

Применение наследования позволяет производным классам наследовать реализацию базовых классов.

Наследственная иерархия — это древовидное отображение отношений, которые устанавливаются между классами в результате наследования.

На рис. 2 показана реальная иерархия, существующая в языке программирования Java.

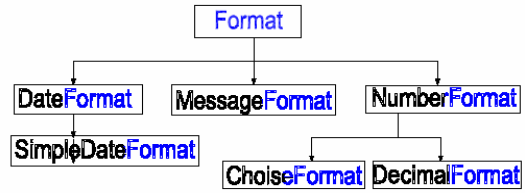


Рис. 2. Образец иерархии

Наследование определяет новый класс — дочерний по отношению к старому, который в этой ситуации называется родителем. Такие потомственно-родительские отношения являются простейшими наследственными отношениями. Ведь и в самом деле каждая наследственная иерархия начинается с родителя и потомка

Потомственный класс, или потомок — это класс, созданный в результате наследования; он называется в языке C# производным классом

Родительский класс — это класс, непосредственным наследником которого является потомственный класс. Родительский класс называется в языке C# базовым классом На рис. 3 показаны потомственно-родительские отношения. NumberFormat является родителем двух потомков — ChoiseFormat и DecimalFormat.

Уточним определение наследования.

Наследование — это механизм, который позволяет установить отношение "Is-a" ("является") между классами. Это отношение также позволяет производным классам наследовать свойства и поведения базового класса

Примечание

Когда производный класс наследует у базового класса свойства и поведения, он получает также все те свойства и поведения, которые базовый класс, возможно, унаследовал от какого-то другого класса

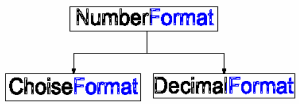


Рис. 3. Родитель с несколькими потомками

Для того чтобы наследственная иерархия имела смысл, должна быть возможность производить над потомками те же действия, что производились над родителями.

Проверка этого осуществляется с помощью теста "Is-a". Производному классу разрешается только: 1) расширять методы 2) и добавлять новые. Но ему не разрешается удалять методы.

Как и настоящие родители и дети, потомственный и родительский классы похожи между собой - классы используют информацию о типе (эта информация для каждого из поддерживаемых объектом интерфейсов включает:

1) список методов

2) и свойств интерфейса,

3) а также описание параметров этих методов).

В языке C++ классы могут иметь более одного родителя. Этот вид наследования называется множественным наследованием.

В других языках, например в C# и Java, реализацию можно наследовать только у одного родителя, но вместе с тем в этих языках предусмотрен аналог механизма множественного наследования для интерфейсов.

Таким образом, единичное наследование — форма наследования функций и свойств классами, при которой производный класс может иметь единственный базовый класс, а множественное наследование — это форма наследования функций и свойств классами, при которой производный класс может иметь любое число базовых.

Производные классы могут прибавлять себе новые поведения и свойства. Когда вы хотите прибавить к классу новое поведение:

1) прибавьте новый метод

2) или переопределите старый.

Технология наследования

Когда один класс наследует другой, он наследует:

1) поведения

2) и свойства.

Это значит, что все методы и свойства, имеющиеся в распоряжении родительского интерфейса, будут переданы в интерфейс потомка. Класс, построенный с помощью наследования, может иметь три важных вида методов и свойств:

.. **переопределенные**: новый класс не просто наследует метод или свойство родительского класса, но и дает этому методу новое определение;

.. **новые**: новый класс прибавляет совершенно новый метод или свойство;

.. **рекурсивные**: новый класс просто наследует метод или свойство родительского класса.

Сначала рассмотрим пример (Листинг 4).

Листинг 4:

```
1 using System; // Наследование - переопределение метода: изменяется
реализация, но интерфейс метода неизменен.
2 // вызов конструктора базового класса из производного класса
3 namespace ConsAppl_OOP21_c97_104 Листинг 4
4 {
5 public class TwoDimensionalPoint // это базовый класс
6 {
7 // Класс TwoDimensionalPoint МОДЕЛИРУЕТ двумерную точку. Точка имеет x и y
координаты
8
9 private double x_coord; // сделайте эксперимент: замените private на
protected
10 private double y_coord;
```

ЕСЛИ оказалось, что из производного класса необходимо удалить метод, это служит указанием на то, что в наследственной иерархии производный класс должен был бы предшествовать базовому.

В отличие от биологических детей, производный класс в языке C# может иметь только один базовый класс.

```
11
12 public TwoDimensionalPoint( double x, double y ) // конст-тор базового
класса с двумя параметрами
13 {
14 setXCoordinate( x );
15 setYCoordinate( y );
16 }
17
18 public double getXCoordinate() // значение x_coord присваивается имени
метода getXCoordinate()
19 { // метод getXCoordinate() – пример рекурсивного метода, см. с. 13 и 14
20 return x_coord;
21 }
22
23 public void setXCoordinate( double x ) // в этом классе set../get.. методы
24 {
25 x_coord = x;
26 }
27
```

```
28 public double getYCoordinate()
29 {
30 return y_coord;
31 }
32
33 public void setYCoordinate( double y )
34 {
35 y_coord = y;
36 }
37 // объявление virtual обеспечивает возм-ть переопред-ния метода в
производном классе, Прак. Зан. # 6
38 public virtual String toString() // переопределяЕМЫЙ (virtual) метод
toString() баз-го класса, см. стр. 67
39 {
40 return "Это двумерная точка.\n" +
41 "x-координата (абсцисса): " + getXCoordinate() + "\n" +
42 "y-координата (ордината): " + getYCoordinate();
43 }
44 } ////////////// конец класса TwoDimensionalPoint ///////////////
45
46 public class ThreeDimensionalPoint : TwoDimensionalPoint // производный
класс
47 {
48 /* Класс ThreeDimensionalPoint (производный) демонстрирует наследство
унаследованное у класса
TwoDimensionalPoint. Наследство - это переопределенный метод toString()*/
49
50 private double z_coord;
51 // ключевое слово base дает доступ к реализации базового класса
52 public ThreeDimensionalPoint( double x, double y, double z ) : base (x, y
)
53 { // вызывается конструктор TwoDimensionalPoint( double x, double y )
базового класса
54 setZCoordinate( z );
55 }
56
57 public double getZCoordinate() // новый метод производного класса,
программирование отличий
58 {
59 return z_coord;
60 }
61
62 public void setZCoordinate( double z )
63 {
64 z_coord = z;
65 }
66 /*объявление override допускает переопред-ние одноименного м-да базового
класса в производном классе, см. выше строку 38. Прак. Зан. # 6 */
67 public override String toString() // переопределяЮЩИЙ (override) м-д
toString() произв-го класса (другая реал-ция метода)
68 {
69 return "\nЭто трехмерная точка.\n" +
70 "x-координата (абсцисса) : " + getXCoordinate() + "\n" +
71 "y-координата (ордината) : " + getYCoordinate() + "\n" +
72 "z-координата (аппликата): " + getZCoordinate();
73 }
74 } /////////// конец класса ThreeDimensionalPoint ////////////
75
76 public class PointExample
77 {
78 // класс PointExample иллюстрирует переопределение метода toString()
79
80 public static void Main(String[] args)
81 {
```

```
82 TwoDimensionalPoint two = new TwoDimensionalPoint(1,2);
83 ThreeDimensionalPoint three = new ThreeDimensionalPoint(1,2,3);
84
85 Console.WriteLine(two.toString());
86 Console.WriteLine(three.toString());
87 Console.ReadLine();
88 } /////////////// конец метода Main ////////////////
89 } /////////// конец класса PointExample //////////////
90 }
```

В листинге 4 есть два класса точек, представляющих геометрические точки. Точки могут использоваться сервисной программой для вычерчивания графиков, диаграмм или кривых, программой визуального моделирования с использованием видеоданных или для составления плана полетов.

Точки используются в разных целях.

Класс TwoDimensionalPoint вмещает координаты x и y. Он определяет методы для получения и установки координат, а также создает строковое представление экземпляра точки (см. выше строки 5…45).

Класс ThreeDimensionalPoint (см. выше строки 46…74) — наследник класса TwoDimensionalPoint. Он прибавляет координату z, а также метод для возвращения и установки ее величины. Класс также имеет метод для получения строкового представления экземпляра. Изза того, что ThreeDimensionalPoint является наследником класса TwoDimensionalPoint, он имеет все методы, содержащиеся внутри класса TwoDimensionalPoint.

В данном примере демонстрируются методы всех типов.

Переопределенные методы

Наследование позволяет переопределить уже существующий метод. Переопределение метода позволяет изменить поведение объекта при вызове данного метода.

Переопределенный метод или свойство присутствует и в базовом, и в производном классе. Например, класс ThreeDimensionalPoint переопределяет метод toString(), который имеется в классе TwoDimensionalPoint:

```
// фрагмент класса TwoDimensionalPoint
38 public virtual String toString()
39 { // переопределяЕМЫЙ (virtual) метод toString() базового класса
40 return "Это двумерная точка.\n" +
41 "x-координата (абсцисса): " + getXCoordinate() + "\n" +
42 "y-координата (ордината): " + getYCoordinate();
43 }
```

В классе TwoDimensionalPoint определен метод toString(), отождествляющий экземпляр с точкой на плоскости и выводящий две ее координаты.

Класс ThreeDimensionalPoint так переопределяет метод toString(), что он отождествляет экземпляр с точкой в пространстве и выводит три ее координаты:

```
// фрагмент класса ThreeDimensionalPoint
67 public override String toString()
68 { // переопределяЮЩИЙ (override) одноименный м-д toString() произв-го
класса (другая реал-ция метода)
69 return "\nЭто трехмерная точка.\n" +
70 "x-координата (абсцисса) : " + getXCoordinate() + "\n" +
71 "y-координата (ордината) : " + getYCoordinate() + "\n" +
72 "z-координата (аппликата): " + getZCoordinate();
73 }
```

На рис. 4 показано, что должно получиться после исполнения метода Main().

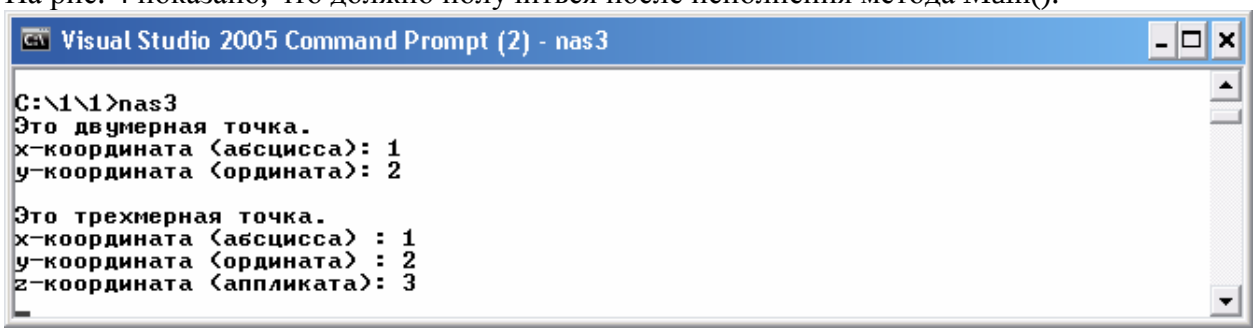


Рис. 4. Проверка переопределенного метода toString()

Как можно увидеть на рис. 4, метод ThreeDimensionalPoint возвращает свое переопределенное строковое представление.

Производный класс обеспечивает собственный вариант реализации метода с помощью переопределения метода. Новая реализация обеспечивает новое поведение метода. ThreeDimensionalPoint так переопределяет поведение метода toString(), что он переводит результат в строковый тип (String).

Переопределение — это способ обработки, с помощью которого производный класс как бы переписывает метод, существующий в базовом классе, с целью изменения поведения метода.

Как объект знает, какое именно определение метода нужно использовать?

Большинство объектно-ориентированных систем сначала ищут определение в том объекте, которому передается сообщение. Если там определения найти не удается, то поиск поднимается по иерархии, пока не найдет какое-то определение. Важно понимать, что именно так происходит управление сообщением и именно благодаря этому и работает процесс переопределения. Именно определение, найденное в производном классе как раз и будет вызвано из-за того, что оно найдено первым.

Новым методом или свойством называется метод или свойство, который появляется в производном классе, но не существует в базовом.

Производный класс прибавляет в свой интерфейс новый метод или свойство. Новый метод вы видели в примере класса ThreeDimensionalPoint (см. выше строки 46…74 на с.10, 11). Класс ThreeDimensionalPoint прибавляет новые методы getZCoordinate() и setZCoordinate() (см. строки 57…65 на с.10).

В интерфейс производного класса можно добавить новую функцию путем прибавления новых методов и свойств.

Рекурсивные методы и свойства

Рекурсивные методы и свойства определяются в базовом классе, но не определяются в производном классе. Чтобы получить доступ к нужному методу или свойству, сообщение поднимается по наследственной иерархии до тех пор, пока не найдет определение метода. Здесь используется тот же механизм, что и при переопределении методов и свойств.

Рекурсивные методы можно увидеть в классах TwoDimensionalPoint hreeDimensionalPoint.

Метод getXCoordinate() — пример рекурсивного метода (см. строки 18…21). Он определяется в классе TwoDimensionalPoint и не определяется в классе ThreeDimensionalPoint.

Переопределенные методы также могут стать рекурсивными. Несмотря на то, что переопределенный метод появляется в производном классе, в объектно-ориентированном языке программирования C# предусмотрен механизм, который позволяет переопределенным методам вызывать версию метода базового класса.

Эта возможность позволяет поднять версию базового класса во время определения нового поведения производного класса.

В языке программирования C# ключевое слово base дает доступ к реализации базового класса (см. выше строку 50 в листинге 2 строку 52 в листинге 4).

Типы наследования

Вообще говоря, наследование применяют в трех главных случаях:

a) для многократного использования реализации;

b) для отличия;

c) для замены типов.

Наследование для многократного использования реализации

Наследование позволяет новому классу многократно использовать реализацию старого класса.

Вместо вырезания и вставки кода или создания экземпляра и использования компонента с помощью формирования, наследование делает код автоматически доступным, т.е. доступ к нему осуществляется так, как если бы он был частью нового класса. Магия наследования состоит в том, что новый класс рождается вместе с функциями.

Иерархия Employee представляет многократное использование реализации. Потомки многократно используют поведения их родителей (листинги 1, 2, 3).

Применяя наследование, вы будете связаны с унаследованной реализацией.

Класс, правильно определенный с точки зрения наследственности, способен интенсивно использовать защищенные (protected) методы более низкого уровня. Производный класс (наследник) может переопределить защищенные методы с целью изменения реализации. Переопределение может уменьшить влияние: 1) плохой 2) или несоответствующей реализации.

Наследование для отличия

Вы видели применение наследования для отличия на примере классов ThreeDimensionalPoint (см. в листинге 4 строки 57…60) и TwoDimensionalPoint. Вы также видели его на примере класса Employee см. в листинге 2 строки 55…67).

Программирование отличий позволяет специфицировать только отличия между производным классом и его базовым классом.

Под программированием отличий подразумевается, что к классу наследнику добавляются только коды, которые делают новый класс отличным от наследуемого класса

В случае ThreeDimensionalPoint прибавляется координата z, которая и отличает его от родительского класса. Чтобы поддерживать координату z, ThreeDimensionalPoint прибавляет два новых метода для установки и считывания свойств (см. выше строки 57…65 на с.10, листинг 4). Кроме того, ThreeDimensionalPoint переопределяет метод toString() (см. выше строки 67…73, листинг 4).

Небольшой объем кодирования и повышенная управляемость кода облегчают разработку проекта. А поскольку вам приходится писать меньше строчек кода, чем при других подходах к программированию, то в соответствии с теорией, уменьшается количество добавляемых ошибок. Поэтому такое пошаговое программирование позволяет написать намного более качественный код, притом быстрее, чем обычно. Подобно наследованию реализации, пошаговые изменения можно делать без изменения ранее написанного кода (то есть используется то, что есть в базовом классе, или то, что есть /добавлено/ в производном классе).

Наследование предоставляет два способа программирования отличий:

1) добавление новых поведений и свойств,

2) переопределение старых поведений и свойств.

Оба эти метода называются специализацией. Давайте присмотримся к специализации внимательнее.

Специализация — это способ обработки, с помощью которого производный класс определяет себя, указывая свои отличия от базового класса. Таким образом, после завершения специализации производный класс содержит в себе только те элементы, которые отличают его от базового класса

Производный класс является специализацией базового класса посредством:

1) добавления новых методов и свойств к интерфейсу

2) или переопределения уже существующих свойств и методов.

Добавление новых методов или переопределение уже существующих позволяет производному классу вести себя иначе, чем базовый класс.

Специализация позволяет только прибавлять или переопределять поведения и свойства, унаследованные от базового класса.

Специализация, несмотря на свое название, не позволяет удалять у производного класса унаследованные поведения и свойства.

Стало быть, в классе не может реализоваться выборочное наследование (то есть все то, что есть в базовом классе, наследуется в производном)

Проследим, например, что происходит при специализации класса TwoDimensionalPoint.

Специализация в этом примере, в общем-то, ограничивает множество (а также его дополнение) тех объектов, которые могут считаться точкой в трехмерном пространстве. ThreeDimensionalPoint может всегда рассматриваться как TwoDimensionalPoint, но вот рассматривать TwoDimensionalPoint в качестве ThreeDimensionalPoint возможно далеко не всегда. Иными словами, ThreeDimensionalPoint — это специализация TwoDimensionalPoint, а TwoDimensionalPoint — ЭТО обобщение ThreeDimensionalPoint.

На рис. 6 представлена разница между обобщением и специализацией. При спуске по лестнице иерархии происходит специализация, а при подъеме вверх — обобщение. Чем выше степень обобщения, тем больше классов могут считаться специализацией данного класса. А чем выше степень специализации, тем меньше классов удовлетворяют всем критериям, которым должны удовлетворять классы, которые могут быть отнесены к данному уровню специализации.

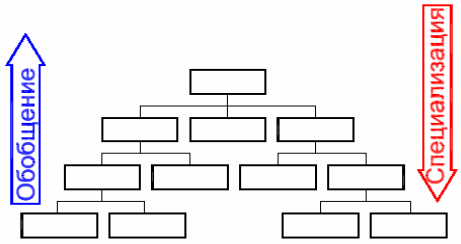


Рис. 6. При подъеме по лестнице иерархии происходит обобщение, а при спуске — специализация

Специализация — это не ограничение функциональности, это ограничение категорий типов.

Фактически нет необходимости начинать с TwoDimensionalPoint. Глубина наследования может быть любой.

Поэтому наследование применяется для формирования сложной иерархической структуры классов.

Идея наследования приводит нас к двум новым понятиям: предка и потомка.

> Не следует стремиться к необоснованному углублению иерархии. Наоборот, желательно минимизировать глубину иерархии. Ведь чем глубже иерархия, тем труднее ее поддерживать

Возьмем для примера некоторый класс, тогда все классы, стоящие после этого класса в иерархии наследования, называются потомками, наследниками или производными классами.

На рис. 7 изображен класс DecimalFormat, который является потомком класса Format

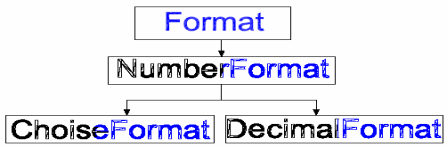


Рис. 7. DecimalFormat — потомок класса Format

На рис. 8 представлена возможная иерархия наследования. OneDimensionalPoint — родитель TwoDimensionalPoint и предок ThreeDimensionalPoint и FourDimensionalPoint.
То есть, TwoDimensionalPoint, ThreeDimensionalPoint, FourDimensionalPoint — потомки OneDimensionalPoint.

Все потомки совместно используют свойства и методы своих предков.

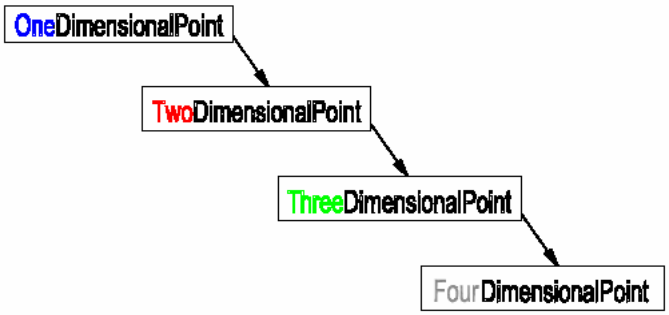


Рис. 8. Иерархия классов, представляющих точки

Вот еще одно высказывание, относящееся к рассматриваемой иерархии классов: OneDimensionalPoint — корень, а FourDimensionalPoint — лист.

Корневой класс — класс, который стоит на самом верху иерархии наследования. На рис. 8 OneDimensionalPoint — корневой класс

Лист — класс без потомков. На рис. 8 FourDimensionalPoint — лист

Очень важно помнить, что потомок отражает изменения, происходящие в предке. Если ошибка обнаружилась в TwoDimensionalPoint, и вы ее исправите, то все нижестоящие классы ThreeDimensionalPoint, FourDimensionalPoint также будут исправлены. Поэтому при исправлении ошибки или повышении эффективности реализации в выигрыше окажутся все потомки.

Множественное наследование

Во всех предыдущих примерах применялось единичное наследование.

В некоторых реализациях наследования один объект может быть непосредственным (прямым) наследником более одного класса. Эта реализация наследования называется множественным наследованием.

Польза множественного наследования для ООП многими специалистами оспаривается. Одни утверждают, что пользы от него нет никакой, что оно лишь усложняет понимание программ, их разработку и сопровождение. Другие же клянутся, что без него язык программирования выглядит незавершенным.

При правильном и корректном применении множественное наследование может принести определенную пользу. Но с его применением связано много проблем.

Наследование для замены типов

Последний вид наследования — наследование для замены типов. Замена типов позволяет описывать заменяемость отношений. Что же это такое — заменяемость отношений?

Разберем класс Line (см. ниже строки 60…95):

```
1 using System; // Наследование для замены типов
2
3 namespace ConsAppl
4 {
5 public class TwoDimensionalPoint
6 { //класс TwoDimensionalPoint моделирует двумерную точку. Точка имеет x и y
координаты.
7
8 private double x_coord;
9 private double y_coord;
10
11 public TwoDimensionalPoint( double x, double y ) // конструктор
12 {
13 setXCoordinate( x );
14 setYCoordinate( y );
15 }
16
17 public double getXCoordinate() // возврат значения
18 {
19 return x_coord;
20 }
21
22 public void setXCoordinate( double x ) // задать значение
23 {
24 x_coord = x;
25 }
26
27 public double getYCoordinate() // возврат значения
28 {
29 return y_coord;
30 }
31
32 public void setYCoordinate( double y ) // задать значение
33 {
34 y_coord = y;
35 }
36 } /////////конец класса TwoDimensionalPoint ////////////
37
38 // ThreeDimensionalPoint демонстрирует наследство полученное из
TwoDimensionalPoint
39 public class ThreeDimensionalPoint : TwoDimensionalPoint
40 {
41
42 private double z_coord;
// ключевое слово base дает доступ к реализации базового класса
44 public ThreeDimensionalPoint( double x, double y, double z ) : base( x, y
)
45 { // вызывается конст-р TwoDimensionalPoint() класса TwoDimensionalPoint
46 setZCoordinate( z );
47 }
48
49 public double getZCoordinate()
50 {
51 return z_coord;
52 }
53
54 public void setZCoordinate( double z )
55 {
56 z_coord = z;
57 }
58 } ////////////конец класса ThreeDimensionalPoint ////////////
59
```

```
60 public class Line /* Класс Line моделирует 2-мерную линию. Цель –
// продемонстрировать заменяемость типов
61
62 private TwoDimensionalPoint p1;
63 private TwoDimensionalPoint p2;
64
65 public Line(TwoDimensionalPoint p1, TwoDimensionalPoint p2) // конструктор
66 {
67 this.p1 = p1;
68 this.p2 = p2;
69 }
70
71 public TwoDimensionalPoint getEndpoint1() // возврат значения – 1-я точка
линии
72 {
73 return p1;
74 }
75
76 public TwoDimensionalPoint getEndpoint2() // возврат значения– 2-я точка
линии
77 {
78 return p2;
79 }
80
81 public double getDistance() // метод: определение расстояния
82 { // обращаемся к public-методам класса TwoDimensionalPoint
83 double x = Math.Pow((p2.getXCoordinate() - p1.getXCoordinate()), 2);
84 double y = Math.Pow((p2.getYCoordinate() - p1.getYCoordinate()), 2);
85 double distance = Math.Sqrt(x + y);
86 return distance;
87 }
88
89 public TwoDimensionalPoint getMidpoint() // метод: определение коор-т
средней точки
90 {
91 double new_x = (p1.getXCoordinate() + p2.getXCoordinate()) / 2;
92 double new_y = (p1.getYCoordinate() + p2.getYCoordinate()) / 2;
93 return new TwoDimensionalPoint(new_x, new_y); // обращение к конст-ру кл-
са TwoDimensionalPoint (?)
94 }
95 } ////////////// конец класса Line //////////////
96
97 public class LineExample // класс LineExample демонстрирует заменяемость
типов
98 {
99 public static void Main(String[ ] args)
100 {
101 ThreeDimensionalPoint p1 = new ThreeDimensionalPoint(12, 12, 2); //
создан экз-р 3-мерной точки
102 TwoDimensionalPoint p2 = new TwoDimensionalPoint(16, 16); // создан экз-р
2-мерной точки
103
104 Line l = new Line( p1, p2 ); // создан экземпляр линии
105
106 TwoDimensionalPoint mid = l.getMidpoint(); // создан экз-р средней
двумерной точки линии
107
108 Console.WriteLine( "Средняя точка: (" + mid.getXCoordinate() + " ," +
mid.getYCoordinate() + ")" );
109 Console.WriteLine( "Расстояние: " + l.getDistance()); // вызов метода,
см. строку 81…87
110 Console.ReadLine();
111 } /////////////// конец метода Main() ///////////
112 } /////////// класса LineExample ///////////////
```

```
113 }
```

Класс Line принимает два аргумента типа TwoDimensionalPoint (см. строки 62, 63), и предоставляет несколько методов выдачи значений (см. строки 71…79), метод расчета расстояния между точками (см. строки 81…87) и метод нахождения средней точки (см. строки 89…94).

Заменяемость отношений означает, что конструктору Line можно передавать любой объект, который является наследником TwoDimensionalPoint.

Вспомните, что при наследовании потомками родителей потомок находится в отношении является ("Is-a") с родителем. Вот почему ThreeDimensionalPoint находится в отношении является ("Is-a") с TwoDimensionalPoint. А потому ThreeDimensionalPoint можно передать конструктору (см. строку 101).

Разберем метод Main() (см. строки 99…111)

Заметьте, что Main передает объекты TwoDimensionalPoint и ThreeDimensionalPoint (см. строки 101…102) в конструктор линии (Line) (см. строку 104 и строки 65…69). На рис. 9 изображено то, что получится в результате выполнения метода Main(). На рис. 10 представлена диаграмма классов Листинга 5

> Примечание: Какую же пользу можно извлечь из заменяемости отношений. В разобранном выше примере с классом линии (Line) заменяемость отношений позволяет быстро переключаться от трехмерного представления к двухмерному в графическом пользовательском интерфейсе

Возможность замены — одно из важных понятий в ООП. Поскольку классу-потомку (производный класс) можно посылать те же сообщения, что и классу-родителю (базовый класс), то с классом-потомком можно обращаться так, как если бы он заменял класс-родитель. Именно потому нельзя удалять поведения при создании класса-потомка.

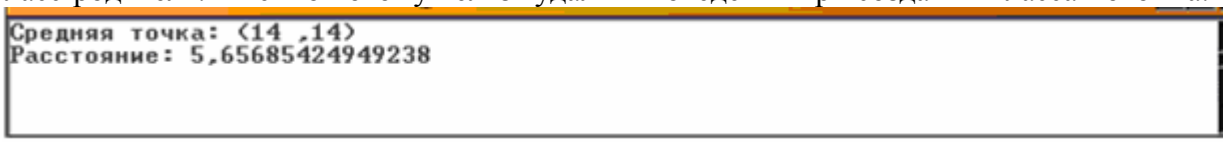


Рис. 9. Проверка заменяемости отношений

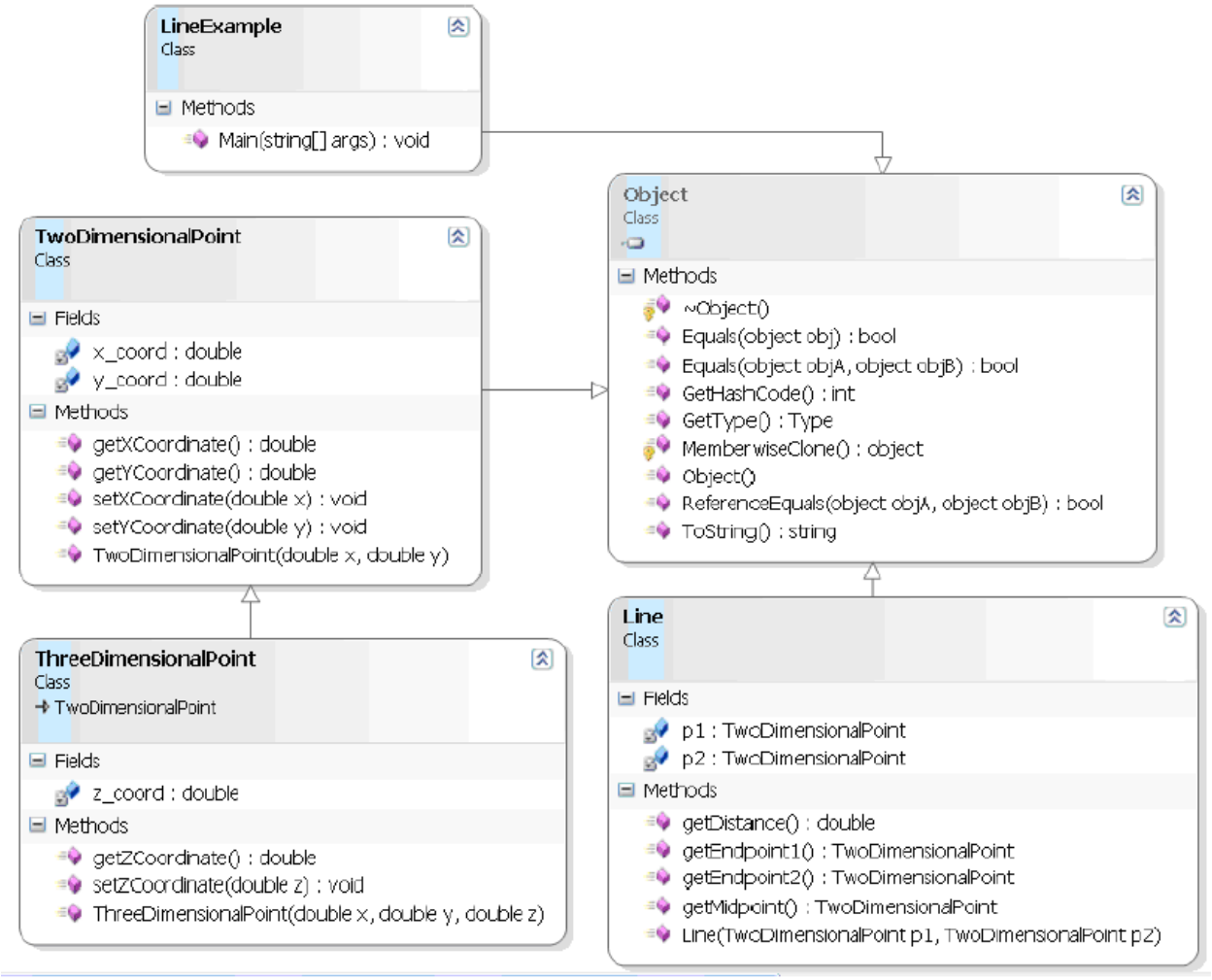


Рис. 10. Диаграмма классов листинга 5

Применяя возможность замены, к программе можно в любое время прибавлять любые подтипы (подтип — это расширение другого типа с помощью наследственности).

Если в программе используется предок (базовый класс), то она будет знать, как использовать новые объекты. Для программы не играет существенную роль тип существующего объекта. Ведь благодаря возможности замены отношений с тем типом, который ожидается, программа может использовать объект нового типа.

> Заменяемость отношений позволяет только подниматься по иерархии наследования. Из этого вытекает следствие: пусть, например, вы программируете объект, который должен принимать объект определенного типа. Тогда программируемому объекту нельзя передать родителя того класса, которого ожидает программируемый объект. Зато ему можно передать любого наследника.

Возьмем в качестве примера конструктор Line (см. выше строки 65…69 на с. 18):

```
65 public Line(TwoDimensionalPoint p1, TwoDimensionalPoint p2)
```

Вы можете передать конструктор экземпляру TwoDimensionalPoint или какому-нибудь другому наследнику TwoDimensionalPoint. Но этот конструктор нельзя передать экземпляру OneDimensionalPoint, потому что в иерархии наследования OneDimensionalPoint находится выше, чем TwoDimensionalPoint (см. рис. 8).

Возможность замены облегчает многократное использование кода. Предположим, например, что у нас есть контейнер для хранения TwoDimensionalPoint. Благодаря возможности замены этот же контейнер подойдет и для любого наследника TwoDimensionalPoint.

Возможность замены также позволяет писать родовой код. Вместо того, чтобы иметь множество операторов выбора или проверок if / else для определения типа точки, можно просто предполагать, что объект имеет тип TwoDimensionalPoint.

Полиморфизм.

Формы полиморфизма

Термин полиморфизм означает "много форм".

Говоря на языке программирования, с помощью полиморфизма одно имя класса или метода может представлять различный, выбранный автоматическим механизмом код.

Таким образом, одно и то же имя может принимать много форм и, так как оно может представлять различный код, одно и то же имя может обозначать различное поведение.

Разговор о многочисленных поведениях может показаться слишком абстрактным.

Слово “открыть” можно использовать в различных жизненных ситуациях, оно многозначно (открыть книгу; открыть счёт в банке, …)

Каждый объект, с которым используется это слово, придает ему особое значение.

Однако во всех случаях для описания действия используется одно и то же слово – открыть.

Язык C#, в котором предусмотрено использование полиморфизма, называется полиморфным языком.

Рассмотрим три формы полиморфизма:

a) полиморфизм включения;

b) переопределение;

c) перегрузку.

Полиморфизм включения иногда еще называют чистым полиморфизмом.

Применяя такую форму полиморфизма, родственные объекты можно использовать обобщенно:

В листинге 1 содержатся следующие классы:

- PersonalityObject,
- PessimisticObject,
- OptimisticObject,
- IntrovertedObject,
- ExtrovertedObject

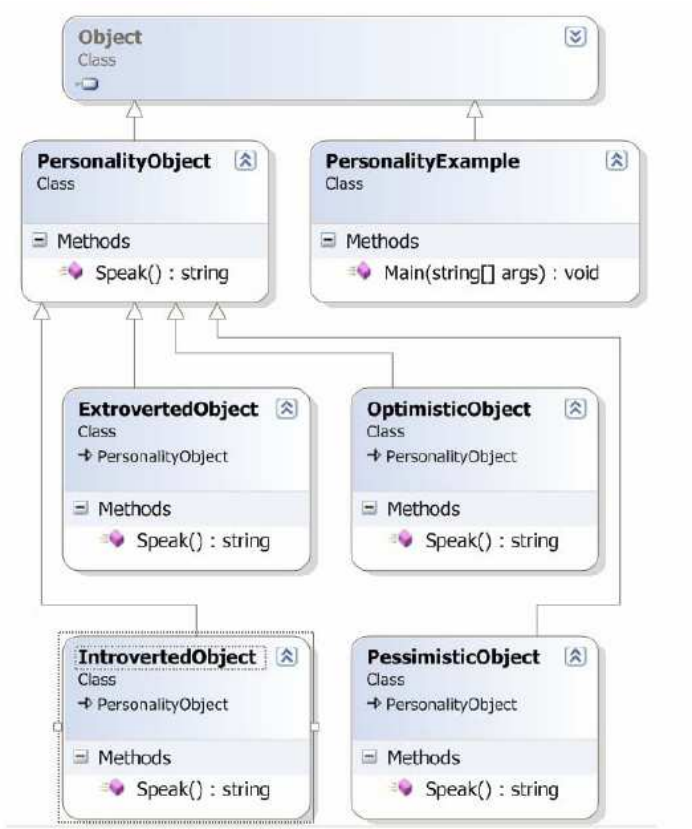
Они образуют довольно простую иерархию наследования.

Базовый класс PersonalityObject описывает один метод: Speak() (строка 8).

Каждый производный класс переопределяет Speak() и возвращает свое собственное сообщение (стр. 17, 26, 35, 44), которое отражает его индивидуальные особенности.

Иерархия формирует отношения замещения между производными классами и базовым классом.

Диаграмма классов программы, содержащейся на листинге 1:



Рассмотрим метод Main() (стр. 53):

Первые две трети Main() не представляют ничего нового (стр. 55...68).

Однако следующая часть примера довольно интересна (стр. 71, 72) .

Судя по этим данным, создается впечатление, что метод Speak() объекта PersonalityObject может выражать различные поведения.

Хотя в PersonalityObject метод Speak() определен как вывод "Я объект", PersonalityObject демонстрирует более одного поведения.

Несмотря на то, что массив содержит экземпляры PersonalityObject (строки 64…68), каждый элемент массива ведет себя по-разному, когда метод Main вызывает метод Speak() (строка 72).

В этом и состоит суть полиморфного поведения:
кажется, что одно имя PersonalityObject представляет различные стили поведения:

```
C:\3>21
PersonalityObject[0] говорит:  Я - объект.
PersonalityObject[1] говорит:  Пессимист: Стакан полупуст.
PersonalityObject[2] говорит:  Пессимист: Стакан наполовину полон.
PersonalityObject[3] говорит:  Интроверт: Эй ...
PersonalityObject[4] говорит:  Экстраверт: Привет! Вы знаете это ...

C:\3>_
```

personalities — это пример полиморфной переменной.

Полиморфная переменная имеет тип базового класса, поэтому она может хранить объекты всех производных классов (строки 63…68 и 72).

Листинг 1

```
1   using System; //Демонстрация ПОЛИМОРФИЗМа
2
3   namespace ConsAppl
4   {
5   // класс PersonalityObject - базовый класс для всех индивидуальностей - объектов
```

```
6    public class PersonalityObject
7    {
8    public virtual String Speak()      // см. строки 17, 26, 35, 44
9    {
10   return " Я - объект.";
11   }
12   }
13
14   //   класс PessimisticObject демонстрирует полиморфизм
//     (индивидуальность - пессимист).
15   public class PessimisticObject : PersonalityObject
16   {
17   public override String Speak() // см. строку 8
18   {
19   return " Пессимист: Стакан полупуст.";
20   }
21   }
22
23   //  класс OptimisticObject демонстрирует полиморфизм
//       (индивидуальность - оптимист).
24   public class OptimisticObject : PersonalityObject
25   {
26   public override String Speak()       // см. строку 8
27   {
28   return " Пессимист: Стакан наполовину полон.";
29   }
30   }
31
32   //      класс IntrovertedObject демонстрирует полиморфизм
//              (индивидуальность - интроверт)
33   public class IntrovertedObject : PersonalityObject
34   {
35   public override String Speak()      // см. строку 8
36   {
37   return " Интроверт: Эй ...";
38   }
39   }
40
41   //   класс ExtrovertedObject демонстрирует полиморфизм
//        (индивидуальность - экстраверт).
42   public class ExtrovertedObject : PersonalityObject
43   {
44   public override String Speak()      // см. строку 8
45   {
46   return " Экстраверт: Привет! Вы знаете это ...";
47   }
48   }
49
50
     /* В классе PersonalityExample создается множество [5] различных PersonalityObject
      (индивидуальностей - объектов. Их (поведение) реализуется полиморфно */
51   public class PersonalityExample
52   {
53   public static void Main(String[ ] args)
54   {
55   PersonalityObject personality = new PersonalityObject();
56   PessimisticObject pessimistic = new PessimisticObject();
```

```
57   OptimisticObject optimistic = new OptimisticObject();
58   IntrovertedObject introverted = new IntrovertedObject();
59   ExtrovertedObject extroverted = new ExtrovertedObject();
60
61          /*personalities - это полиморфная переменная базового класса.
      Полиморфная переменная может хранить объекты производных классов*/
62   // Заменяемость позволяет сделать следующее
63   PersonalityObject [ ] personalities = new PersonalityObject[5];
64   personalities[0] = personality;
65   personalities[1] = pessimistic;
66   personalities[2] = optimistic;
67   personalities[3] = introverted;
68   personalities[4] = extroverted;
69
70   /* Благодаря полиморфизму кажется, что PersonalityObject имеет много различных
поведений. */
71   for (int i = 0; i < personalities.Length; i++)
72   Console.WriteLine("PersonalityObject[" + i + "] говорит: " + personalities[i].Speak());
73   Console.ReadLine();
74   }
75   }
76   }
```

Полиморфизм посредством переопределения методов.

Переопределение — это важный тип полиморфизма (см. Листинг 2)

В базовом классе определяется абстрактный метод, не содержащий тела. Тело абстрактному методу сообщается (присваивается) в каждом производном классе своё.

Рассмотрим более подробно определения классов MoodyObject (см. ниже стр. 5,сл.) и HappyObject (см. ниже стр. 18,сл.).

Листинг 2.

```
5  public abstract class MoodyObject    // базовый абстрактный класс
6  {
7  // MoodyObject  - базовый абстрактный класс для создания объектов с разным настроем
8  //   возврат настроения (см. строки 21, 34)
9  protected abstract String getMood();   //  ПЕРЕопредеЛЯЕМЫЙ абстрактный метод
10
11                //  спросите у объекта, как он себя чувствует
12   public void quireMood()
13   {
14  Console.Writeline(“Я чувствую себя сегодня” + getMood() + “!”)
15   }
16   }   // конец класса MoodyObject
```

Абстрактные методы часто относят к отложенным методам, так как определение производных классов откладывается. Однако, как и в случае с любым другим методом, класс, в котором определен абстрактный метод, может вызвать этот метод.

Можно заметить, что HappyObject (строки 21…24) переопределяет метод getMood (строка 9) объекта MoodyObject.

Интересно то, что внутри определения queryMood() (строки 12…15) в MoodyObject имеется вызов getMood() (строка 14).

Обратим внимание, что HappyObject не переопределяет метод queryMood().

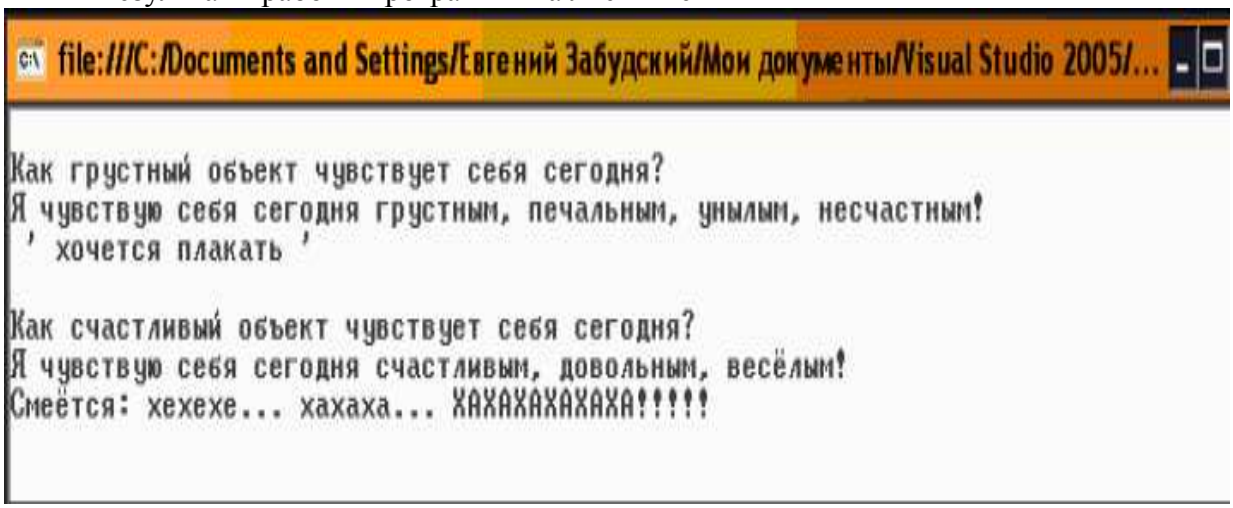
Вместо этого, HappyObject просто наследует данный метод как рекурсивный метод от MoodyObject.

Когда queryMood() вызывается к HappyObject, полиморфизм экземпляра (то есть объекта) обеспечивает скрытый вызов переопределенной версии getMood(), принадлежащей HappyObject.

Теперь полиморфизм заботится о том, какой метод должен быть вызван. Таким образом, программисту уже не нужно переопределять queryMood(), чтобы была вызвана правильная версия getMood().

Метод getMood() сделан абстрактным в базовом классе (см. строку 9) с помощью ключевого слова abstract.

Результаты работы программы на листинге 2



Полиморфизм, посредством перегрузки методов (листинг 3)

Определение метода **определяет** поведение объекта.

Например, метод **Display()** отображает информацию о студенте.
Студент — это объект.
Вы **определяете** метод этого объекта, указывая:

- **1)** его название,
- **2)** список аргументов (если таковые имеются),
- **3)** тело метода
- **4)** возвращаемое значение (если оно есть).

Название метода используется для вызова метода из оператора программы, а список аргументов содержит данные, необходимые для осуществления методом его поведения.

Вместе название метода и его аргументы называются **сигнатурой** метода.
Тело метода содержит один или более операторов, которые выполняются, когда вызывается метод.
Вот где на самом деле осуществляется поведение!

Возвращаемое значение — это значение, которое возвращается программе после того, как метод закончит выполняться.
Некоторым методам не требуется список аргументов или возвращаемое значение.

Перегрузка – это изменение тела и сигнатуры метода при одном и том же названии.
Перегрузка является частным случаем полиморфизма.

С помощью перегрузки одно и тоже имя может обозначать разные методы.

Причем методы различаются *только телом, а так же - количеством и типом параметров.*
Рассмотрим следующие методы, определенные в классе Math:

// большее из двух 16-битовых целых чисел со знаком
- public static int Math.Max(int a, int b);

// максимальный
- public static long Math.Max(long a, long b);

// максимальный с плавающей точкой
- public static float Math.Max(float a, float b);

// максимальный с плавающей точкой двойной точности
- public static double Math.Max(double a, double b);

Все методы Math.Max() являются примерами перегрузки.
Можно заметить, что эти методы различаются только типом параметров.

Перегрузка (overloading) — это один из терминов, который вы можете услышать вместе с полиморфизмом.

Перегрузка означает, что два или более метода имеют одно название, но разные списки аргументов.
Перегрузка методов предоставляет нам способ реализовать похожее поведение для разных типов данных с помощью написания своей версии метода для каждого используемого типа данных.

Любая вариация списка аргументов делает метод отличным от других методов с таким же названием.

То есть списки аргументов с разным :
- 1) количеством аргументов,
- 2) типами данных аргументов

считаются различными.
Перегрузку методов иллюстрирует следующий пример (листинг 3).
В этой программе объявляются классы Student и GradStudent.
Класс GradStudent наследуется от класса Student.
Каждый класс имеет метод Display() и метод Write().
Метод-член Write() присваивает значения переменным каждого класса.
Метод Display() отображает значения переменных.
Каждый из этих классов спроектирован с полиморфизмом.
Каждый класс содержит метод Display() (и метод Write()), которые выполняют похожие задачи, но делают это по-разному.
Это и есть полиморфизм в действии.
Оба определения класса определяют методы-члены Write() (см. стр. 18 и 49) и Display() (см. стр. 26 и 57).
Метод-член Write() присваивает значения аргументов из своего списка аргументов атрибутам класса.
Метод-член Display() выводит эти атрибуты на экран.
Класс Student содержит атрибуты, которые являются общими для всех студентов:
идентификационный номер студента **m_ID**;
имя студента **m_First, m_Last;** и
признак окончания обучения **m_Graduation (см. стр. 7…10)**.

Класс GradStudent имеет атрибуты, которые принадлежат студенту, окончившему обучение. Эти атрибуты — название учебного заведения m_UndergradSchool; год окончания m_UndergradGraduation; и профилирующий предмет m_Major (см. стр. 39…41).

Класс GradStudent также может обращаться к protected-атрибутам класса Student (см. стр. 7…10), так как класс GradStudent наследуется от класса Student (см. стр. 37).

Метод Display() класса GradStudent вызывает метод Display() базового класса Student для повторного использования кода (см. стр. 57, 59 и 26). К тому же сигнатуры методов Display() классов Student и GradStudent идентичны.

В методе Main() первые два оператора объявляют экземпляры классов Student и GradStudent (см. стр. 74 и 75).

Следующие два оператора используют эти экземпляры для вызова метода Write() каждого из экземпляров, передавая им информацию о студентах (см. стр. 77 и 78).

Последние два оператора вызывают функцию Display() соответствующих классов (см. стр. 81 и 83).

Вот что выводит программа, приведенная в листинге 3:

```
file:///C:/Documents and Settings/Евгений Забудский/Мои документы/Visual Studio 2005/...
1-я  печать результатов в соотв-вии с м-дом Display() в базовом классе

Идентиф. #: 100
Имя:        Иван
Фмилия:     Петров
Окончание учебы в университете: 2004

2-я  печать результатов в соотв-вии с м-дом Display() в базовом классе

Идентиф. #: 101
Имя:        Екатерина
Фмилия:     Маслова
Окончание учебы в университете: 2004

3-я  печать результатов в соотв-вии с м-дом Display() в производном классе

Специальность:         Computer Science
Undergrad school:      ГУ-ВШЭ
Окончание аспирантуры: 2007
```

Диаграмма классов программы на листинге 3

Связывание

- это передача адреса метода при его вызове (в то место, откуда вызывается метод)

Каждый раз, когда вы вызываете метод в своем приложении, вызов метода (его имя) должен быть ассоциирован с определением метода (его тело). Программисты называют этот процесс связыванием (binding).

Связывание, то есть передача адреса, происходит:

- **1)** либо в процессе компиляции,
- **2)** либо во время выполнения программы.

Связывание во время компиляции называется ранним связыванием (early binding) и используется, если вся информация, необходимая для вызова метода (то есть адрес метода), известна на момент компиляции приложения.

Связывание во время выполнения называется поздним связыванием (late binding) или динамическим связыванием и используется, если какая-то информация отсутствует во время компиляции и становится известной только во время выполнения приложения.

Раннее связывание используется для вызова обычных методов. При выполнении программы не происходит потери времени, так как связывание завершается при создании исполняемого кода программы. Это дает преимущество по сравнению с поздним связыванием.

Позднее связывание реализуется с помощью **виртуальных** (virtual) методов.

Виртуальный метод получает базовую ссылку для указания на тип объекта, используемого методом.

Во многих ситуациях ссылка на объект **неизвестна** до времени выполнения. Поэтому связывание нельзя осуществить во время компиляции, и необходимо ждать, пока программа не запустится, чтобы **связать** вызов метода с его телом.

Позднее связывание способно замедлить выполнение приложения, однако оно позволяет программе реагировать на события, которые происходят во время выполнения.

Нет необходимости писать код для обработки непредвиденных ситуаций, которые могут возникнуть в процессе выполнения, что является важным преимуществом позднего связывания.

Полиморфизм времени выполнения

Полиморфизм времени выполнения — это способ, с помощью которого программисты могут использовать преимущества, как полиморфизма, так и позднего связывания.

Полиморфизм времени выполнения использует виртуальные методы для создания стандартных интерфейсов и вызова методов, лежащих в их основе. Эти определения методов связываются с вызовами методов во время выполнения.

Термин виртуальный метод (virtual method) — это основа полиморфизма времени выполнения.

Слово виртуальный означает что-то, что кажется реальным, но таковым не является.

В случае виртуального метода компьютер «обманывается» определением метода, но на самом деле метод на тот момент не определен.

Виртуальный метод выступает как «заполнитель» реального метода.

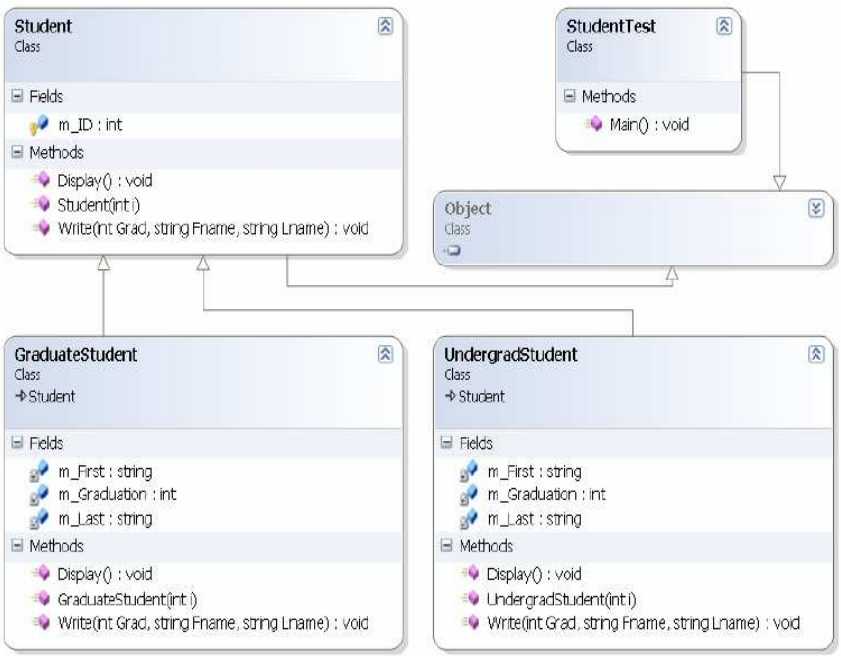
Реальный метод определяется во время выполнения программы.

Следующий пример (Листинг 4) очень похож на предыдущий (Листинг 3) . Обе программы записывают и отображают информацию о студенте.

В листинге 3 приведена программа, которая использует **перегруженные методы** для реализации полиморфизма.

В листинге 4 приведена программа, которая для реализации полиморфизма использует **виртуальные методы**.

Диаграмма классов программы на листинге 4



В этом примере определены три класса: Student (стр. 5), UndergradStudent (стр. 23) и GraduateStudent (стр. 46).

Класс Student — это базовый класс, который наследуется другими классами программы. Базовый класс (base class) — это класс, наследуемый другими классами, которые называются производными классами (derived classes).

Класс Student определяет атрибут с названием m_ID (стр. 7), который используется для хранения идентификационного номера студента.

Он также определяет конструктор, который в качестве аргумента получает идентификационный номер студента, и присваивает его значение атрибуту m_ID (стр. 9).

Конструктор вызывается всегда, когда объявляется экземпляр класса.

Последние два оператора в определении класса Student определяют два виртуальных метода: Display() (стр. 14) и Write() (стр. 18).

Объявление виртуального метода включает:

- 1) ключевое слово virtual,
- 2) сигнатуру метода (имя и список аргументов)
- 3) возвращаемое значение.

Виртуальные методы могут быть «настоящими» методами или просто заполнителями для реальных методов, которые должны быть реализованы в производных классах.

Если определяется виртуальный метод без тела, это означает, что оно должно быть реализовано в производном классе (выбора нет, иначе программа не откомпилируется).

Классы с такими методами называются *абстрактными классами*, потому что это не законченные классы, а, скорее, указание для создания реальных классов. (Например, абстрактный класс может заявлять: «Вы должны создать метод Display()».)

В C# можно создать виртуальный метод без тела, добавив “ ; “ после его сигнатуры (такие методы называются чисто виртуальными). В C# для создания виртуальных методов без тела используется ключевое слово abstract.

Классы UndergradStudent и GraduateStudent практически одинаковы, за исключением метода Display(), который при выводе информации на экран идентифицирует студента:

1) как обучающегося (см. стр. 40) или

- 2) уже окончившего обучение (см. стр. 64).

Оба класса определяют методы Write() и Display().

Метод Write() копирует информацию о студенте, полученную в списке аргументов, в атрибуты класса (см. стр. 18, 33, 57).

Метод Display() отображает содержимое этих атрибутов и значение идентификационного номера студента класса Student (см. стр. 14, 40, 64).

Все действия происходят в методе Main() (см. стр. 72).

Первые два оператора объявляют экземпляры классов UndergradStudent и GraduateStudent.

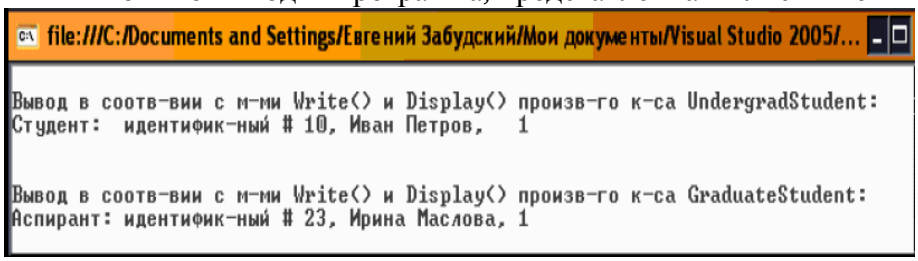
Идентификационный номер студента передается в конструктор каждого экземпляра (см. стр. 29, 53).

В каждом конструкторе производного класса вызывается конструктор базового класса Student (см. стр. 9), который присваивает значение идентификационного номера студента атрибуту m_ID.

Полиморфизм времени выполнения реализован в следующих двух операторах (см. стр. 76, 77), вызывающих сначала метод Write(), а затем метод Display() для отображения атрибутов обучающегося студента.

Затем вызываются методы Write() и Display() (см. стр. 78, 79) для отображения атрибутов студента закончившего обучение в вузе.

Вот что выводит программа, представленная в листинге 4



Таким образом:

Полиморфизм — это такое состояние объекта, при котором он имеет много форм.

Полиморфизм — это механизм, позволяющий одному имени представлять различный код.

Так как одно имя может представлять различный код, это имя может выражать различное поведение.

С помощью полиморфизма можно легко написать многоликий код, т.е. код, который может демонстрировать различное поведение.

Рассмотрены три различных вида полиморфизма:

- полиморфизм включения;
- переопределение;
- перегрузка.

Абстрактные классы

Абстрактные классы и абстрактные методы заставляют производные классы (в принудительном порядке) переопределить методы, которые в базовом классе не имеют никакого смысла.

Абстрактный метод создается с помощью спецификатора типа abstract.

Абстрактный метод не содержит тела. Поэтому производный класс должен его переопределить, поскольку он не может использовать версию, предложенную в базовом классе.

Абстрактный метод автоматически является виртуальным, поэтому нельзя использовать модификатор virtual.

Формат объявления абстрактного метода:

abstract *тип_возврата имя* (*список_параметров*) **;**

Спецификатор abstract нельзя использовать применительно к static-методам.

Класс, содержащий один или несколько абстрактных методов, также должен быть объявлен как абстрактный класс с помощью спецификатора abstract.

Поскольку абстрактный класс нереализуем в полном объеме, невозможно создать его объекты.

Свойства и индексаторы также могут быть абстрактными.

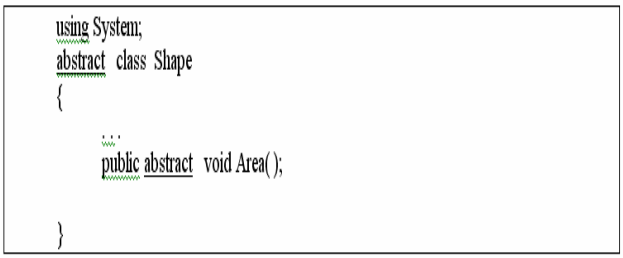
Абстрактный класс в отличие от интерфейса может содержать наряду с абстрактными членами обычные поля и методы, которые будут унаследованы потомками.

Если производный класс выводится из абстрактного, он должен реализовать все абстрактные методы базового класса.

В противном случае такой производный класс также должен быть определен как абстрактный.

Таким образом, спецификатор abstract наследуется до тех пор, пока реализация абстрактных методов класса не будет полностью достигнута.

Пример абстрактного класса и абстрактного метода:

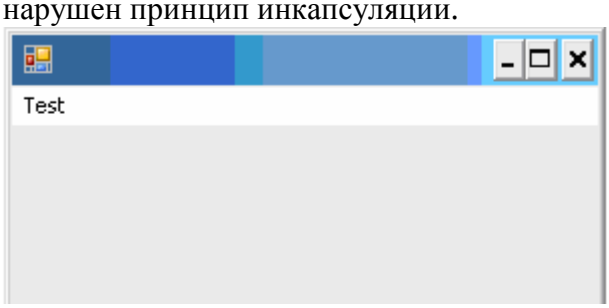


Ограничения на использование Visual Studio.

Имеем исходную программу, взятую с сайта Firststeps.ru (шаг 21):

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

class AppForm : Form
{
      public  AppForm()
      {
            MainMenu mnuFileMenu = new MainMenu();
            this.Menu = mnuFileMenu;
            mnuFileMenu.MenuItems.Add("Test");
      }

      protected override void OnMouseDown(MouseEventArgs e)
      {
            MessageBox.Show("You clicked on Form ","First Step Site");
      }
}

class MyForm : AppForm
{
      public static void Main()
      {
            Application.Run(new MyForm());
      }
}
```

В этой программе обработчик прерывания, возникающего при щелчке по форме, находится в классе, основное назначение которого – визуализация информации. Если при щелчке по форме надо будет выполнить несвойственное классу Form действие, будет нарушен принцип инкапсуляции.



Решение проблемы.

Вариант 1. Перенос обработчика в другой класс.

Просто переносим обработчик в другой класс, поближе к Main().

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

class AppForm : Form
{
      public  AppForm()
      {
            MainMenu mnuFileMenu = new MainMenu();
            this.Menu = mnuFileMenu;
            mnuFileMenu.MenuItems.Add("Test");
      }

}

class MyForm : AppForm
{
      public static void Main()
      {
            Application.Run(new MyForm());
      }
      protected override void OnMouseDown(MouseEventArgs e)
      {
            MessageBox.Show("You clicked on Form ","First Step Site");
      }

}
```

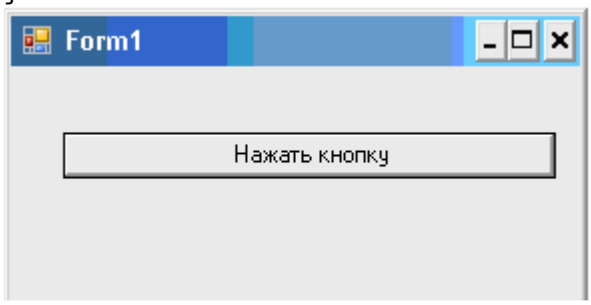
Всё работает так же. Перенос обработчика в класс, содержащий метод Main, освободил класс формы от несвойственной этому классу информации. Но классы здесь связаны наследованием.

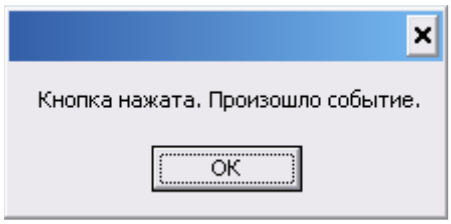
Вариант 2. Перенос обработчика для несвязанных классов.

В этом примере реализовано 2 класса: Form1 и Program. Класс Form1 предназначен для визуального отображения элементов управления, нанесённых на форму (в данном случае – одной кнопки). Класс Program содержит метод Main и обработчик события «Кнопка нажата». Принцип инкапсуляции по составу содержащейся в классах информации не нарушен. Программа содержит классы, объекты, методы. Объекты создаются и используются в классе, содержащем метод Main. По своей структуре программа может быть отнесена к асинхронным объектно-ориентированным программам.

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

namespace WindowsFormsApplication
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            // button1
            //
            this.button1.Location = new System.Drawing.Point(26, 32);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(246, 23);
            this.button1.Text = "Нажать кнопку";
            //
            // Form1
            //
            this.ClientSize = new System.Drawing.Size(284, 262);
            this.Controls.Add(this.button1);
            this.Name = "Form1";
            this.Text = "Form1";

            this.ResumeLayout(false);
        }
            public System.Windows.Forms.Button button1;
        }
    static class Program
    {
        static Form1 f1;

        [STAThread]
        static void Main()
        {
```

```
            f1 = new Form1();
            f1.button1.Click += new EventHandler(button1_Click);
            Application.Run(f1);
        }
        static void button1_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Кнопка нажата. Произошло событие. ");
        }
    }
}
```





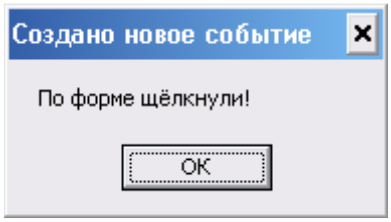
Вариант 3. Зажигание нового события.

Можно оставить обработчик события OnMouseDown (для программы с сайта «Feststeps.ru») в классе AppForm, но поручить ему зажечь новое событие, например – «по форме щёлкнули!», которое будет обрабатываться в других классах. Принцип инкапсуляции для класса AppForm не будет нарушен, в этот класс чуждая ему информация не попадёт.

```
using System;
using System.Windows.Forms;
using System.Drawing;
    //Определяем делегат
    delegate void MyEventHandler();
    class MyEvent
    {
      //В классе MyEvent объявляем событие Create
       public event MyEventHandler Create;
        //Создаём метод EventOn_Create для зажигания нового события
        public void EventOn_Create()
        {
            if (Create != null)
              {
               Create();
              }
        }
        //Создаём обработчик события handler
        public static void handler()
        {
          MessageBox.Show("По форме щёлкнули!","Создано новое событие");
        }
}

class AppForm : Form
```

```
{
     //Создаём событийный объект evt
        MyEvent evt = new MyEvent();

      public  AppForm()
      {
            MainMenu mnuFileMenu = new MainMenu();
            this.Menu = mnuFileMenu;
            mnuFileMenu.MenuItems.Add("Test");
      }
      //В обработчике события OnMouseDown запускаем генерацию нового события
      protected override void OnMouseDown(MouseEventArgs e)
      {
            evt.Create += new MyEventHandler(MyEvent.handler);
              evt.EventOn_Create();

      }
}

class MyForm : AppForm
{
      public static void Main()
      {
            Application.Run(new MyForm());
      }
}
```

В результате обработчик события выведен из класса MyForm, а при исполнении этой программы после щелчка по форме получаем:



Технология объектно-ориентированного программирования.

В отличии от визуальной модели программирования при объектно-ориентированном программировании необходимо учитывать дополнительные условия, предъявляющие к структуре программы специфические требования.

Характерные особенности объектно-ориентированных программ:

1. Объектная ориентация связана с использованием особой конструкции программы – объектов.

2. Объекты с похожими свойствами, то есть с одинаковыми наборами переменных состояния и методов, образуют класс.

3. Класс **(class)** – это шаблон, чертёж, схема объекта. Он определяет лишь типовые черты объектов.

4. **Объект (object)**- это конкретная реализация, экземпляр класса.

5. Для классов характерно, что их методы и свойства взаимосвязаны, класс должен определять только одну логическую сущность. Обычно класс содержит Абстрактный тип данных – группу тесно связанных между собой данных и методов (функций), которые могут осуществлять операции над этими данными. Смешивать данные и методы их обработки для разных типов объектов в одном классе недопустимо.

6. В конечном итоге программа может содержать большое количество классов, как универсальных, так и специализированных, как правило, слабо связанных между собой, или даже совсем не связанных.

7. Кроме этого в объектно-ориентированной программе должна существовать какая-то объединяющая конструкция, в которой из этих классов создаются объекты и в которой эти объекты живут, взаимодействуют, развиваются.

8. В языке C# такая конструкция создаётся в виде отдельного класса, отличительной чертой которого является находящийся в нём метод Main().

9. С метода Main() начинается исполнение любой самой сложной программы. Имя этого метода является зарезервированным. Главное его назначение – он является ведущим методом программного проекта, именно в нём должны создаваться, жить, развиваться и взаимодействовать все объекты.

9. Класс, в котором находится метод Main(), не имеет какого-либо специального имени, оно может быть любым, выбирает его пользователь. Кроме метода Main() в этом классе могут содержаться и другие методы и свойства, необходимые для работы программы и не имеющие отношения к другим классам проекта.

10. Для визуального отображения элементов управления создаётся специальный класс Windows.Forms.

11. Обычно Visual Studio размещает метод Main() в файле Program.cs, создавая для этого класс «static class Program».

Пример асинхронной объектно-ориентированной программы: Общая схема реализации паттерна «Издатель –Подписчик».

```
Using System;
//-------------------------------------------------------------
//объявление делегата
 delegate void MyEventHandler();
//-------------------------------------------------------------
// объвление класса - издателя события
 class MyEvent {
 public event MyEventHandler SomeEvent;   //объявление события SomeEvent
 public void FireSomeEvent() {
 if(SomeEvent != null)         //проверка, можно ли зажигать событие
   SomeEvent();                 //метод для зажигания события SomeEvent
}
}
//конец класса - издателя
//-------------------------------------------------------------
// объявление класса - ресивера
 class Resiver {
 public static void handler()   {
    Console.WriteLine(«Событие произошло»);
                        }
}
// конец класса - ресивера
//-------------------------------------------------------------
// создание класса - демонстратора
 class EventDemo
  {
 public static void Main()   {
 // создание экземпляра класса - издателя
 MyEvent evt = new MyEvent();

// добавление обработчика события в общий список
 evt.SomeEvent += new MyEventHandler(Resiver.handler);

// зажигание события
evt.FireSomeEvent();
```

```
  }    // конец определения метода Main

}   // конец класса - демонстратора
```

В этом примере видна ещё одна отличительная особенность объектно-ориентированной программы: объекты (т.е. экземпляры классов) создаются и взаимодействуют в классе, содержащем метод Main.

В книге Г.Шилдта содержится учебный проект, демонстрирующий синхронную объектно - ориентированную программу:

> «Стек — это классический пример объектно-ориентированного программирования, в котором сочетаются как средства хранения информации, так и методы получения доступа к этой информации».

```
//Учебный проект Шилдта:
using System;

class Stack {
  //  Эти члены закрытые.
  Char[] stck; // Массив для хранения данных стека.
  Int tos;     // Индекс вершины стека.

  // Создаем пустой класс Stack заданного размера.
  Public Stack(int size) {
    stck = new char[size]; // Выделяем память для стека.
    Tos = 0;
  }

  // Помещаем символы в стек.
  Public void push(char ch) {
    if(tos==stck.Length) {
      Console.WriteLine(" - Стек заполнен.");
      return;
    }

    stck[tos] = ch;
    tos++;
  }

  // Извлекаем символ из стека.
  Public char pop() {
    if(tos==0) {
      Console.WriteLine(" - Стек пуст.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }

  // Метод возвращает значение true, если стек полон.
  Public bool full() {
    return tos==stck.Length;
  }

  // Метод возвращает значение true, если стек пуст.
  Public bool empty() {
    return tos==0;
  }

  // Возвращает общий объем стека.
  Public int capacity() {
```

```
    return stck.Length;
  }

  // Возвращает текущее количество объектов в стеке.
  Public int getNum() {
    return tos;
  }
}

// Демонстрация использование класса Stack.


Class StackDemo {
  public static void Main() {
    Stack stk1 = new Stack(10);
    Stack stk2 = new Stack(10);
    Stack stk3 = new Stack(10);
    char ch;
    int I;

    // Помещаем ряд символов в стек stkl.
    Console.WriteLine(«Помещаем символы от A до Z в стек stkl.»);
    for(i=0; !stk1.full(); i++)
      stk1.push((char) (‘A’ + i));

    if(stk1.full()) Console.WriteLine(“Стек stkl полон.”);

    // Отображаем содержимое стека stkl.
    Console.Write(«Содержимое стека stkl: «);
    while( !stk1.empty() ) {
      ch = stk1.pop();
      Console.Write(ch);
    }

    Console.WriteLine();

    if(stk1.empty()) Console.WriteLine(“Стек stkl пуст.\n”);

    // Помещаем еще символы в стек stk1
    Console.WriteLine(«Снова помещаем символы от A до Z в стек stk1.»);
    for(i=0; !stk1.full(); i++)
      stk1.push((char) (‘A’ + i));

    /* Теперь извлекаем элементы из стека stkl и помещаем их в стек stk2.
В результате элементы стека stk2 должны быть расположены в обратном
порядке.*/
    Console.WriteLine(«Теперь извлекаем элементы из стека stkl» +
                      «и помещаем их в стек stk2.»);
    while( !stk1.empty() ) {
      ch = stk1.pop();
      stk2.push(ch);
    }

    Console.Write(“Содержимое стека stk2: “);
    while( !stk2.empty() ) {
      ch = stk2.pop();
      Console.Write(ch);
    }

    Console.WriteLine(“\n”);

    // Помещаем 5 символов в стек
    Console.WriteLine(«Помещаем 5 символов в стек stk3.»);
    for(i=0; I < 5; i++)
```

```
      stk3.push((char) ('A' + i));

    Console.WriteLine("Вместимость стека stk3: " + stk3.capacity());
    Console.WriteLine(«Количество объектов в стеке stk3: « +
                      stk3.getNum());

  }
}
/*                Результат работы программы:
Помещаем символы от A до Z в стек stkl.
Стек stkl полон.
Содержимое стека stkl: JIHGFEDCBA
Стек stkl пуст.

Снова помещаем символы от A до Z в стек stk1.
Теперь извлекаем элементы из стека stkl и помещаем их в стек stk2.
Содержимое стека stk2: ABCDEFGHIJ

Помещаем 5 символов в стек stk3.
Вместимость стека stk3: 10
Количество объектов в стеке stk3: 5
*/
```

Проект содержит два класса: Stek и StekDemo. Класс Stek описывает типовую структуру стека и определяет методы, характеризующие возможные поведения стека. А в классе StekDemo на основе класса Stek создаётся три объекта и проводится работа по размещению и перемещению символов в объектах, проверка стека на заполненность, и др.

Из анализа этих двух примеров и требований, предъявляемых к объектно-ориентированным программам видно, что архитектура объектно-ориентированной программы значительно сложнее всех рассмотренных выше моделей программирования.

Основные принципы ООП.

В различных источниках делается акцент на те или иные особенности внедрения и применения ООП, но 3 основных (базовых) понятия ООП остаются неизменными.

К ним относятся:

- Инкапсуляция (**Encapsulation**)
- Наследование (**Inheritance**)
- Полиморфизм (**Polymorphism**)

Эти понятия, как три кита, лежат в основе мира ООП.

Кроме того, есть и другие особенности, свойственные ООП:

1. ООП позволяет разложить проблему на связанные между собой задачи.

Каждая проблема становится самостоятельным объектом, содержащим свои собственные коды и данные, которые относятся к этому объекту.

В этом случае исходная задача в целом упрощается, и программист получает возможность оперировать с большими по объему программами.

В этом определении ООП отражается известный подход к решению сложных задач, когда мы разбиваем задачу на подзадачи и решаем эти подзадачи по отдельности.

С точки зрения программирования подобный подход значительно упрощает разработку и отладку программ.

2. И наоборот, ООП позволяет укрупнить задачу, абстрагироваться от того, каким способом нужно выполнять каждую её часть.

В этом случае объектно-ориентированный подход оставляет за объектом право решать как отреагировать и что сделать в ответ на поступившее в него задание. А Вы можете даже не знать как объект выполнил задание. В отличие от ООП, при процедурном подходе необходимо описать каждый шаг, для достижения конечного результата.

Суть этого принципа ООП: "Объект отвечает за все действия, которые он производит в ответ на запрос клиента".

Наш мир состоит из объектов – любых вещественных предметов.

Каждый **объект** - это осязаемая сущность, которая четко проявляет свое поведение.

Объект характеризуется тремя понятиями:

- имя объекта;
- состояние (переменные состояния);
- методы (операции).

Имя необходимо для идентификации конкретного объекта.

Относительно двух других понятий можно сказать, что *это - совокупность переменных состояния и связанных с ними методов(операций).*

Методы определяют, как объект взаимодействует с окружающим миром.

Возможность управлять состояниями объекта посредством вызова методов в итоге и определяет поведение объекта.

Объекты с одинаковыми свойствами, то есть с одинаковыми наборами переменных состояния и методов, образуют класс.

Класс (class) - это группа данных и методов (функций) для работы с этими данными.

В сущности, класс - шаблон, чертёж, схема объекта.

Объект (object)- это конкретная реализация, экземпляр класса.

Обычно, если объекты соответствуют конкретным сущностям реального мира, то классы являются некими абстракциями, выступающими в роли понятий.

Для формирования какого-либо реального объекта необходимо иметь шаблон, на основании которого и строится создаваемый объект.

При рассмотрении основ ООП часто смешиваются понятия объекта и класса.

Класс - это некоторое абстрактное понятие, а объект - это конкретное понятие, физическая реализация класса (шаблона).

Например, класс, как абстракция, может содержать такое свойство, как цвет объекта, и даже может в качестве значения «по умолчанию» указывать конкретное значение цвета. Тогда, как каждый объект может это свойство переопределять по-своему. И истинное значение цвета объекта будет находиться именно в объекте, а не в классе.

Методы (methods)- это функции(процедуры), принадлежащие классу.

Инкапсуляция является одним из ключевых понятий ООП.

Формальное определение этого понятия:

Инкапсуляция - это механизм, который объединяет данные и методы, манипулирующие этими данными, и защищает их от внешнего вмешательства или неправильного использования.

Можно сказать, что инкапсуляция подразумевает скрытие данных (**data hiding**), что позволяет защитить эти данные. При инкапсуляции переменные, характеризующие состояния объекта скрыты от внешнего мира.

Изменение состояния объекта (его переменных) возможно ТОЛЬКО с помощью его методов (операций).

Это существенно ограничивает возможность *введения объекта в недопустимое состояние и/или несанкционированное разрушение этого объекта*.

Применение этого принципа ведет к снижению эффективности доступа к элементам объекта. Это обусловлено необходимостью вызова методов для изменения внутренних элементов (переменных) объекта.

Однако, при современном уровне развития вычислительной техники, эти потери в эффективности не играют существенной роли.

Это понятие (инкапсуляция) наиболее ярко реализовано в такой структуре, как абстрактные типы данных (**ABSTRACT DATA TYPES**). *Абстрактный тип данных - это*

группа тесно связанных между собой данных и методов (функций), которые могут осуществлять операции над этими данными.

Поскольку подразумевается, что эта структура защищена от внешнего влияния, то она считается инкапсулированной структурой.

Важным же отличием от других аналогичных структур является то, что данные заключенные в этой структуре, тесно связаны и активно взаимодействуют между собой внутри структуры и имеют слабые связи с внешним миром (связи, которые осуществляются посредством ограниченного интерфейса).

Наследование является одним из фундаментальных понятий ООП.

Приведем его определение:

Наследование - это процесс, посредством которого, один класс может наследовать свойства другого.

Смысл и универсальность наследования заключается в том, что при создании нового объекта (в том числе и такого, как класс) не надо каждый раз заново (с нуля) описывать его состояния и методы, а можно указать родителя (базовый класс) и описать отличительные особенности нового класса.

В результате, новый класс будет обладать всеми свойствами родительского класса плюс свои собственные отличительные особенности.

Приведем пример.

Можно создать какой-то базовый класс "транспортное средство", который универсален для всех средств передвижения, к примеру, на 4-х колесах. Этот класс "знает" как двигаются колеса, как они поворачивают, тормозят и т.д.

А затем на основе этого класса создадим класс "легковой автомобиль", который унаследуем из класса "транспортное средство".

Поскольку мы новый класс унаследовали из класса "транспортное средство", то мы и унаследовали все особенности этого класса и нам не надо в очередной раз описывать как двигаются колеса и т.д. Мы просто добавим те черты, особенности поведения, которые характерны для легковых автомобилей.

В то же время мы можем взять за основу этот же класс "транспортное средство" и построить класс "грузовые автомобили". Описав отличительные особенности грузовых автомобилей, мы получим уже новый класс "грузовые автомобили".

А, к примеру, на основании класса "грузовой автомобиль" уже можно описать определенный подкласс грузовиков и т.д. Таким образом, нам не надо каждый раз описывать все "с нуля".

В этом и заключается главное преимущество использования механизма наследования. Мы как бы сначала формируем простой шаблон, а затем все усложняя и конкретизируя, поэтапно создаем все более сложный шаблон.

В данном случае был приведен пример простого наследования, когда наследование производится только из одного класса.

В некоторых объектно-ориентированных языках программирования определены механизмы наследования, позволяющие наследовать от одного и более класса. Однако реализация подобных механизмов зависит от самого применяемого языка ООП.

Кроме того, следует отметить, что особенности реализации даже простого наследования могут различаться от языка к языку.

В описаниях языков ООП (в том числе – и в C#) принято класс, из которого наследуют называть родительским классом (parent class) или основной класса(base class).

Класс, который получается в результате наследования, называется порожденным классом (derived or child class).

Родительский класс всегда считается более общим и развернутым. Порожденный же класс всегда более строгий и конкретный, что делает его более удобным в применении при конкретной реализации.

Благодаря наследованию можно считать, что ООП - это процесс построения иерархии классов, возможность наследовать характеристики из более простых, общих типов.

Слово **полиморфизм** имеет греческое происхождение и переводится как "имеющий много форм".

Общее определение:

Полиморфизм *- это свойство, которое позволяет одно и тоже имя использовать для решения нескольких технически разных задач*.

В общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий.

Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного интерфейса для единого класса действий. Выбор конкретного действия, в зависимости от ситуации, возлагается на компилятор.

Применительно к ООП, целью полиморфизма, является использование одного имени для задания **общих для класса** действий.

На практике это означает способность **объектов** выбирать внутреннюю процедуру (метод) исходя из сложившихся в этом объекте особенностей.

К примеру, у нас есть класс "автомобиль", в котором описано как должен передвигаться автомобиль, как он поворачивает, как подает сигнал и т.д. Там же описан метод "переключение передачи". Допустим, что в этом методе класса "автомобиль" мы описали автоматическую коробку передач.

А теперь нам необходимо описать класс "спортивный автомобиль", у которого механическое (ручное) переключение скоростей.

Конечно, можно было бы описать заново все методы для класса "спортивный автомобиль". Но зачем, если у нас уже практически все описано и отлажено? Для этого и существует механизм наследования.

Мы указываем, что класс "спортивный автомобиль" наследован из класса "автомобиль", а следовательно он обладает всеми свойствами и методами, описанными для класса-родителя. Единственное, что нам надо сделать - это переписать метод "переключение передач" для механической коробки передач. В результате, при вызове метода "переключение передач" будет выполняться метод не родительского класса, а самого класса "спортивный автомобиль".

Схема объектноориентированного программирования.

В Интернет предлагается такая схема «Допустим, что мы делаем стратегическую игрушку по мотивам Великой Отечественной войны.

У нас есть:

- игровое поле с разбросанными по нему неподвижными объектами,
- какое-то количество движущихся, умирающих и отстреливающихся объектов (юнитов),
- некий модуль искусственного интеллекта, заставляющий юниты охотиться друг на друга и не ломиться сквозь стену, выполняя приказ игрока, а аккуратно объезжать препятствия.
- некий программный код - физический движок, обеспечивающий
  - покачивания танка на ухабах,
  - плавный разгон и торможение,
  - появление повреждений и прочие "детали", придающие сцене естественность.
- И, конечно же, у нас есть графический движок, который отвечает за отображение творящегося на экране.

Словом, есть все необходимое, и нам остается только собрать все это в единую программу.

Сборка заключается в следующем:

пишется некоторый кусочек кода (назовем его GameTick), который последовательно перебирает все имеющиеся в игре объекты, "вычисляя" события, происходящие с ними в данный момент времени.

Скажем,

один **объект** - солдат в окопе - "поразмыслил" своим **модулем** AI, принял решение бросить гранату и бросил ее, **сгенерировав новый игровой объект** - летящую гранату.

Другой объект - брошенный другим солдатом секунду назад "коктейль Молотова" - в результате вычислений физического движка изменил свое положение в пространстве, прошел проверку на соприкосновение с броней танка и прекратил существование.

Танк, угодивший под бутылку с зажигательной смесью, **перешел из состояния** "танк обыкновенный" в состояние "танк горящий".

Другой танк повернул башню еще на пять градусов влево.

На этом игровые объекты, требующие вычислений, закончились, и сцена с игровыми объектами ушла на обработку к графическому движку, который конвертировал абстрактных солдат, танков и игровое поле во вполне осязаемые полигоны и текстуры, понятные видеокарте.

При этом на экране появилась описанная выше картинка, сменившая предшествовавший кадр.

А меж тем наша программа опросила клавиатуру и мышь (не решил ли пользователь как-то повлиять на ситуацию?) и перешла на уже известный нам участок кода делать очередной GameTick.

Добавим сюда музыкальное сопровождение по вкусу - и игра "заработала"

- танки ездят,
- снаряды летают,
- геймер отчаянно дает бойцам указания мышкой,
- в звуковых колонках "бумкает" все, что положено…

остается лишь записать свежесотворенный шедевр на DVD и топать к издателю».

Реализация проектов ООП.

Объектно-ориентированное проектирование предполагает эффективную реализацию проектов в первую очередь на традиционных объектно-ориентированных языках программирования "с классами" типа C++,. Object Pascal, C#, VB .NET и др.

Сторонники объектно-ориентированного подхода рассматривают разработку программного комплекса автоматизации процесса решения тех или иных задач с точки зрения *вовлекаемых в этот процесс объектов*. Что это дает?

Это открывает, по крайней мере, следующие возможности:

Вместо того чтобы применять или искать новые подходящие аналитические или синтетические методы структурирования программной системы,

достаточно использовать непосредственное соответствие частей (модулей) системы объектам предметной области той задачи, для решения которой предназначена программная система.

Учитывая, что объекты конкретных предметных областей достаточно *стабильны* и *инградиентны* относительно решаемых в этих областях задач,

а также имеют тенденцию к повторению в других предметных областях,

появляется реальная возможность многократного (повторного) использования готовых объектно-ориентированных модулей (классов) в качестве конструктивных (сборочных) элементов различных программных систем.

Объектно-ориентированный подход к разработке программных комплексов предусматривает наличие следующих составляющих:

- объектно-ориентированный анализ;
- объектно-ориентированное проектирование;
- объектно-ориентированное программирование (реализация).

Объектно-ориентированный анализ - это методология, при которой требования к программе формулируются с точки зрения объектов предметной области, вовлекаемых в решение задачи, соответствующей предназначению программы.

Объектно-ориентированное проектирование - это методология, соединяющая в себе

- объектную декомпозицию цели разработки и
- методы представления объектно-ориентированных моделей проектируемой программы (комплекса, системы).

Объектно-ориентированное программирование - это методология программирования, основанная на таких методах реализации объектно-ориентированных моделей программы, которые обеспечивают представлении программы в виде совокупности классов объектов, а классы образуют иерархию наследования свойств.

Появление новой методологии объектно-ориентированного анализа и проектирования вызвано сложностью моделирования предметной области и разработки объемных (например, корпоративных) информационных систем.

На практике, как правило, применяют оба подхода:

- для проектирования сложной системы в целом до уровня классов и их методов используют ООАП, а
- для реализации отдельных методов или решения несложной (разовой) задачи используют процедурно-ориентированное проектирование.

Таким образом, оба рассмотренных подхода к проектированию остаются актуальными.

Независимо от выбранного подхода к проектированию процесс разработки программного обеспечения начинают с *этапа выработки и анализа требований.*

Идея анализа предметной области - выделить те объекты, операции и связи, которые эксперты данной области считают наиболее важными.

Выделение исходных компонентов предметной области, требуемых для решения той или иной задачи, представляет, в общем случае, нетривиальную проблему.

Результатом анализа предметной области, и весьма желательным, может быть вывод о возможности использования или адаптации существующей разработки.

Основное требование к модели программной системы состоит в том, что она должна быть понятна заказчику и всем специалистам проектной группы, включая бизнес-аналитиков и программистов.

Последовательность разработки объектно-ориентированной программы.

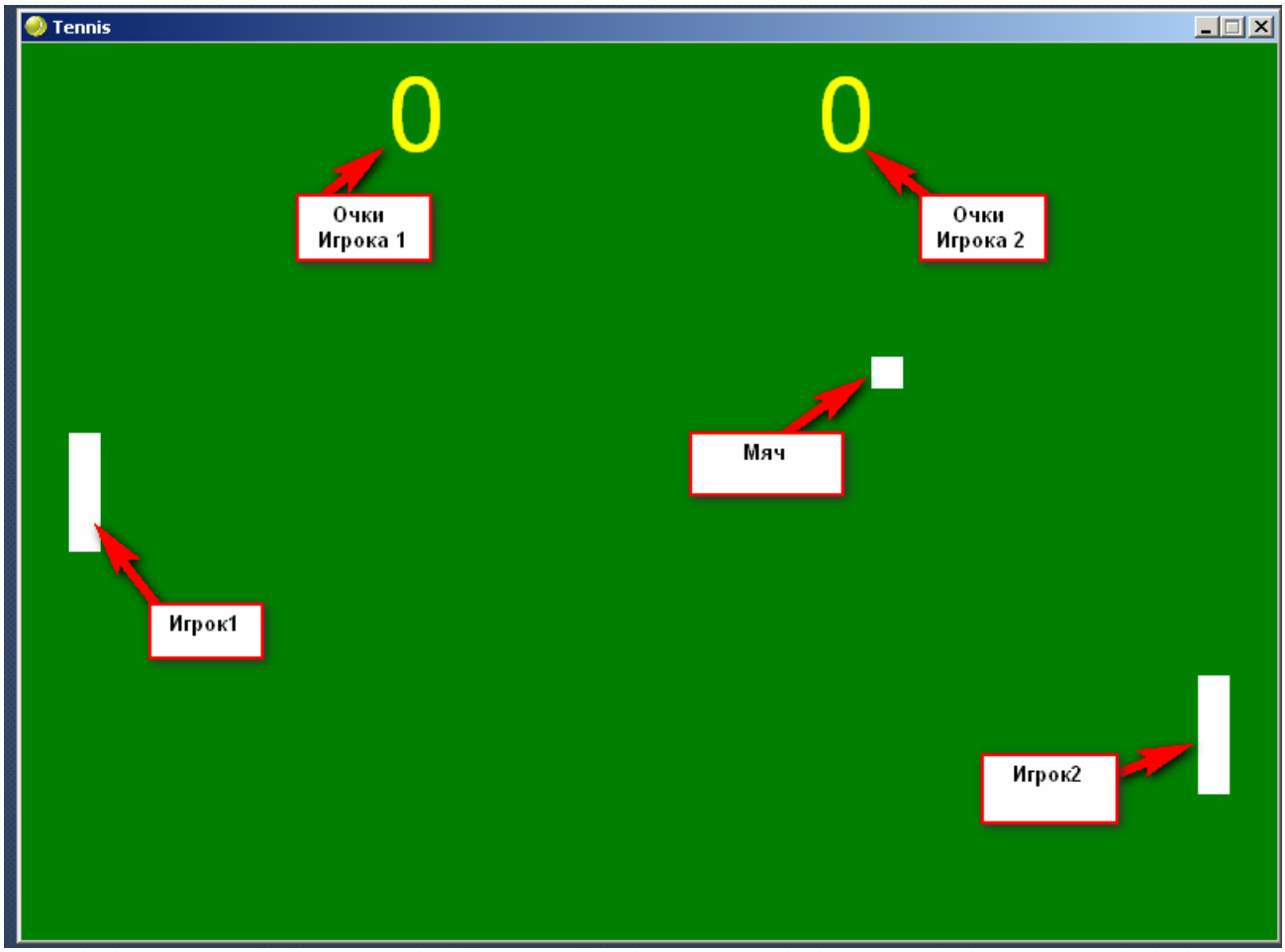
Начинать составление объектно-ориентированной программы надо с объектной декомпозиции задания. Сначала необходимо определить состав объектов, которые будут действовать в программе.

После определения состава объектов по такому заданию необходимо для каждого из объектов разработать «чертёж», т.е. класс, составить описание каждого класса, после чего можно начинать писать программу, например, с помощью Visual Studio.

Пример разработки программы “Тенис”.

Рассмотрим работу по реализации задачи ООП на примере игры “Тенис”.
Модель игры “Тенис”:

- Поле для игры
- 2 Игрока, представляющие собой прямоугольники, перемещающиеся вдоль фиксированной вертикальной прямой
- Игрок 1 – управляется клавишами
- Игрок 2 – управляется компьютером
- Мяч, представляющий собой прямоугольник автоматически перемещающийся по линейным траекториям, меняя траекторию после соприкосновения.
- Табло, отображающее количество очков, набранной Игроками



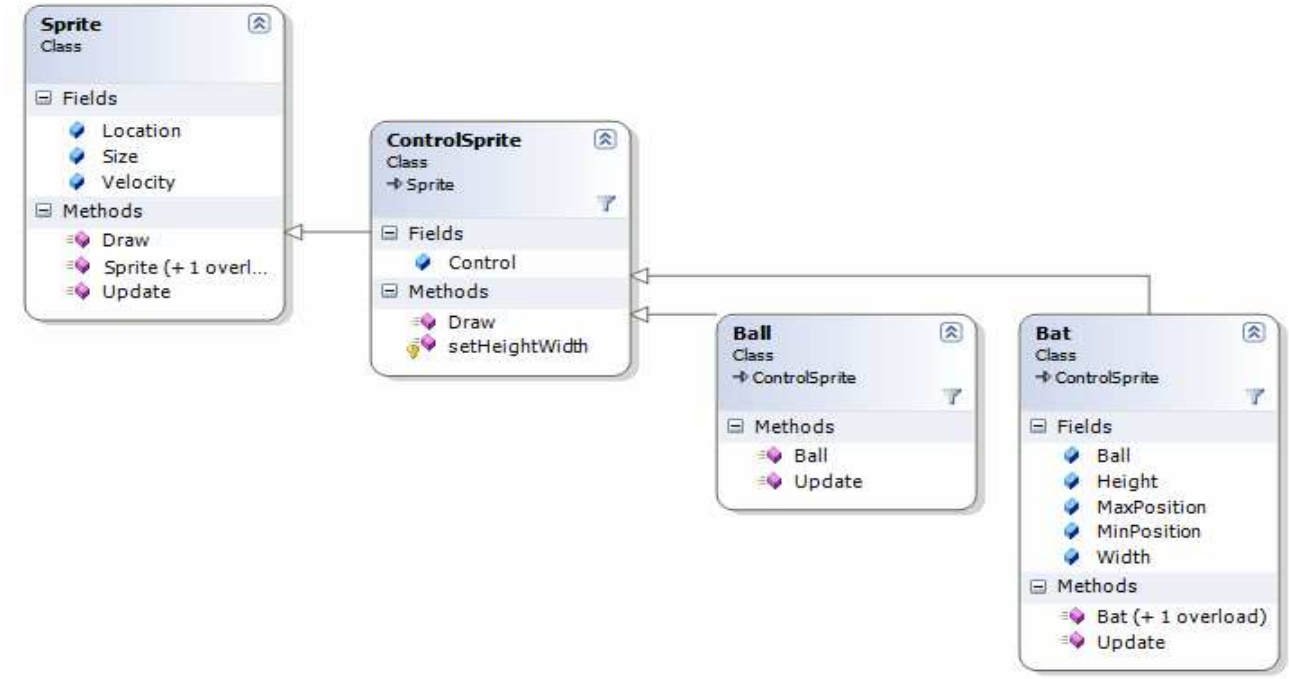
Инструкция для пользователя:

3. Пользователь запускает проект Tennis.exe
4. Игра начинается автоматически
5. Управление Игроком 1 – клавишами “Вверх”|”Вниз” на клавиатуре компьютера

Игрок 2 перемещается под управлением компьютера.
Мяч изменяет траекторию после касания с Игроком.
Очко защитывается Игроку 1, в случае если мяч касается правой границы игрового поля. Игроку 2 – если мяч касается левой границы поля.

Остановить Игру можно кнопкой  в правом верхнем углу экрана игры.

Иерархия классов программы.



Нейросеть «Перцептрон HSE-5»

Постановка задачи.

Разработать объектно-ориентированную программу, демонстрирующую распознавание графических образов с помощью нейронной сети.

Объектная ориентация.

Для нейросети характерно создание, развитие и взаимодействие следующих объектов:

Нейросеть. Чем характеризуется нейросеть:

- тип нейросети (перцептрон),

- количество слоёв,
- тип нейронов,
- количество нейронов,
- передаточная функция (функция активации) нейронов,
- Learning function – функция обучения, отвечающая за обновление весов и смещений сети в процессе обучения

Алгоритм обучения:

Метод обучения,

«Параметры обучения» (Training parameters):

- количество эпох (epochs) – определяет число эпох (интервал времени), по прошествии которых обучение будет прекращено (эпохой называют однократное представление всех обучающих входных данных на входы сети);
- достижение цели или попадание (goal) – здесь задаётся абсолютная величина функции ошибки, при которой цель будет считаться достигнутой;
- период обновления (show) – период обновления графика кривой обучения, выраженный числом эпох;
- время обучения (time) – по истечении указанного временного интервала, выраженного в секундах, обучение прекращается.
- порог достижения цели: ошибка рассчитывается как разница между целью и полученным выходом.
- инициализация сети, т. е. задание значений весов и смещений определённым образом.

входной набор данных,
тестирующий набор данных,
экзаменационный набор данных.

В программе используется 8 классов:

Класс Form1() – работает с формами и элементами управления. Унаследован от класса Form.

Класс Program() – содержит метод Main(), в котором создается и запускается форма и объект Seti().

Класс Seti() – основной класс программы.

Методы класса Seti()

```
public Seti(Form1 frm)  // создает поле для прорисовки символа
private void EventClass()  // метод с делегатами


// методы обработки событий
private void pbField_Paint(object sender, PaintEventArgs e)
private void Form1_Paint(object sender, PaintEventArgs e)
private void Form1_Shown(object sender, EventArgs e)
private void Form1_Validated(object sender, EventArgs e)
private void Form1_Validating(object sender, CancelEventArgs e)
private void Form1_MouseMove(object sender, MouseEventArgs e)

private void pbField_MouseClick(object sender, MouseEventArgs e) // при нажатии мышки
на клетку в PictrureBox, устанавливает занчение клетки true или false
private void pbField_MouseMove(object sender, MouseEventArgs e) // при движении мышки
с зажатой кнопкой по клеткам в PictrureBox, устанавливает значение клетки true или
false
private void SetInputs()  // на вход нейросети задает двоичные сигналы в соответствии
со значениями клеток(1 или 0)
private void button1_Click(object sender, EventArgs e)  // метод, который обучает
нейросеть
```

```
private void button2_Click(object sender, EventArgs e)  // метод, который угадывает
символ
private void btnOrient_Click(object sender, EventArgs e)  // метод, который определяет
ориентацию символа
private void button3_Click(object sender, EventArgs e)  // метод для очистки поля
ввода
private void btnCleatNet_Click(object sender, EventArgs e)  // метод для очистки
нейросети
private void btnSave_Click(object sender, EventArgs e)  // метод для сохранения
нейросети. Сериализация в двоичный формат с помощью BinaryFormatter
private void btnLoad_Click(object sender, EventArgs e)  // метода для загрузки
нейросети


// методы прорисовки символа
private void DrawNet()  // разбиение PictureBox на клетки и прорисовка сетки
private void RedrawNet()  // перерисовывает рисунок
private void DrawCell(int i, int j)  // прорисовка закрашенного прямоугольника
(клетки)
private void InitCells()  // создание и индексация клеток со значениями false
private void SetCell(int i, int j)  // установка значения клетки false
private void UnsetCell(int i, int j)    // установка значения клетки true
```

Класс NeuronNet() – класс, где задается структура нейросети.

Методы класса NeuronNet()

```
public NeuronNet(int nInputs) // метод для создания нейросети
public void TeachFromInput(string name)  // метод для обучения нейросети
public void AddNewPerc(string name)  // метод для добавления нового персептрона
```

Класс Perc() - класс, где задается структура персептрона.

```
public Perc(string name)  // метод для создания объетка персептрон
private bool StepFunction(float Sum)  // сравнение с пороговым значением
public float Sum  // метод сумматор
public int Out  // определения выхоного сигнала
```

Класс Input() – работает с входными сигналами.

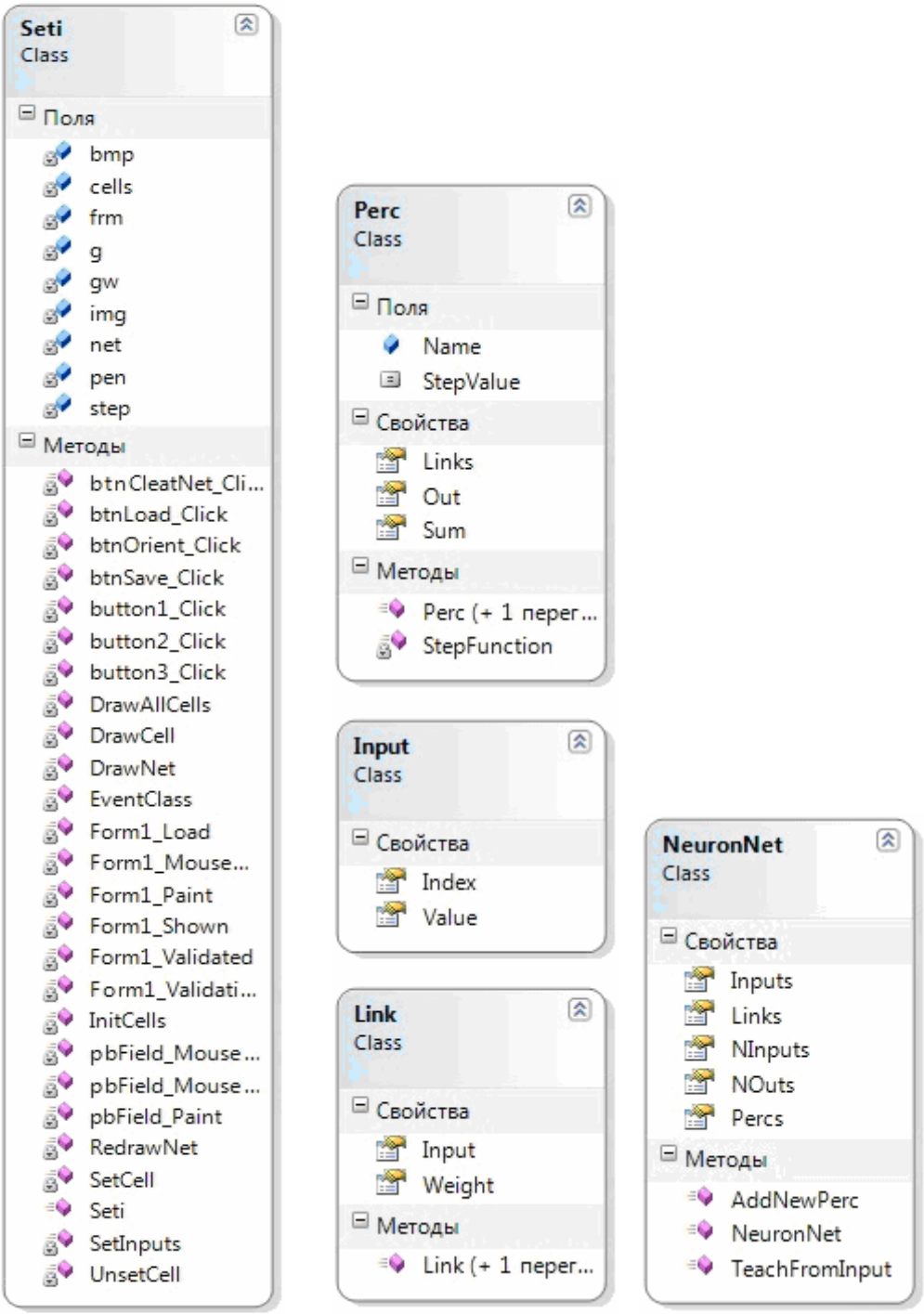
Сохранение обученной нейросети и загрузка уже обученной нейросети были реализованы с помощью механизма сериализации в двоичном формате (BinnaryFormatter). Сериализация представляет собой процесс преобразования объекта в поток байтов для хранения объекта или передачи его в память, базу данных или файл. Ее основное назначение – сохранить состояние объекта для того, чтобы иметь возможность воссоздать его при необходимости. Обратный процесс называется десериализацией.

Чтобы сделать объект сериализируемым нужно снабдить каждый связанный с ним класс или структуру аттрибутом [Serializable].

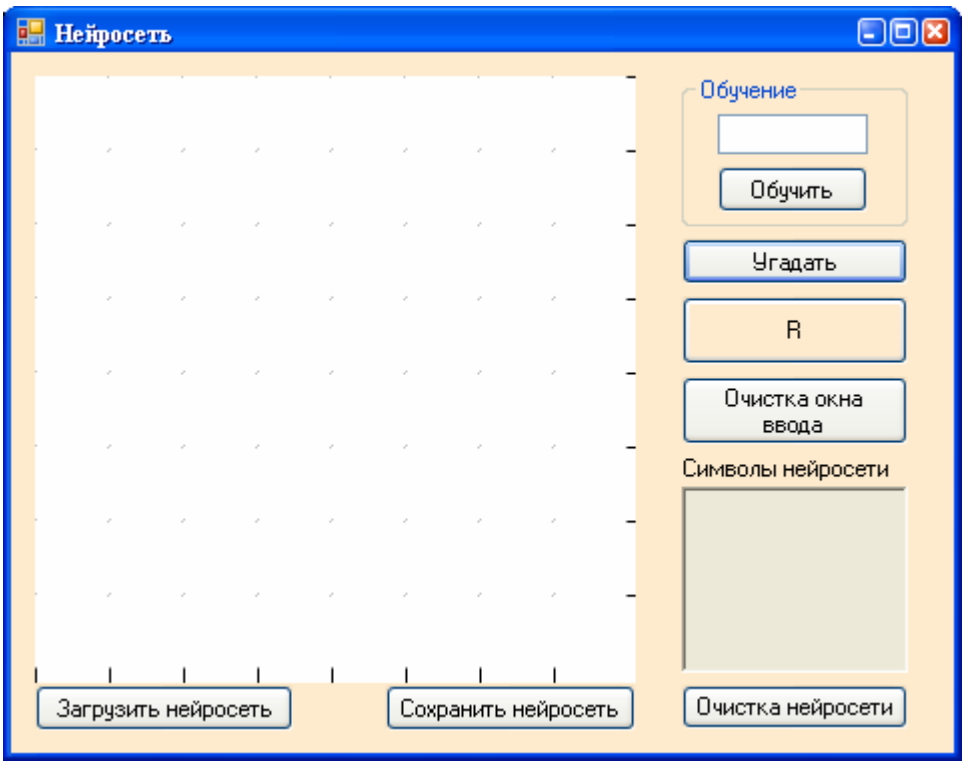
Если есть поля, которые по какой-то причине нужно исключить из сериализации, их необходимо пометить атрибутом [NonSerialized]. Классы Input, Link, NeuronNet, Perc были сериализованы.

```
[Serializable]
    public class Input

[Serializable]
    public class Link

[Serializable]
    public class NeuronNet

[Serializable]
    public class Perc
```

Ниже приведена диаграмма основных классов:



При запуске программы появляется основное окно:



Исходный текст программы:

```
//File 1File.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.Drawing.Drawing2D;
using System.Linq;
using System.Runtime.Serialization.Formatters.Binary;

//Form1.cs
namespace Perseptron
{
    public partial class Form1 : Form
    {
        public Form1()  // метод для создания формы
        {
            InitializeComponent();  // метод для добавления элементов
управления на форму
        }

//Form1.Designer.cs
        /// <summary>
        /// Требуется переменная конструктора.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Освободить все используемые ресурсы.
```

```
        /// </summary>
        /// <param name="disposing">истинно, если управляемый ресурс должен
быть удален; иначе ложно.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Код, автоматически созданный конструктором форм Windows

        /// <summary>
        /// Обязательный метод для поддержки конструктора - не изменяйте
        /// содержимое данного метода при помощи редактора кода.
        /// </summary>

        private void InitializeComponent()
        {
            this.pbField = new System.Windows.Forms.PictureBox();
            this.btnTeach = new System.Windows.Forms.Button();
            this.textBox1 = new System.Windows.Forms.TextBox();
            this.btnGuess = new System.Windows.Forms.Button();
            this.btnClear = new System.Windows.Forms.Button();
            this.Teach = new System.Windows.Forms.GroupBox();
            this.richTextBox1 = new System.Windows.Forms.RichTextBox();
            this.btnCleatNet = new System.Windows.Forms.Button();
            this.label1 = new System.Windows.Forms.Label();
            this.btnSave = new System.Windows.Forms.Button();
            this.btnLoad = new System.Windows.Forms.Button();
            this.btnOrient = new System.Windows.Forms.Button();

((System.ComponentModel.ISupportInitialize)(this.pbField)).BeginInit();
            this.Teach.SuspendLayout();
            this.SuspendLayout();
            //
            // pbField
            //
            this.pbField.Location = new System.Drawing.Point(12, 12);
            this.pbField.Name = "pbField";
            this.pbField.Size = new System.Drawing.Size(300, 303);
            this.pbField.TabIndex = 0;
            this.pbField.TabStop = false;
            //
            // btnTeach
            //
            this.btnTeach.Location = new System.Drawing.Point(18, 45);
            this.btnTeach.Name = "btnTeach";
            this.btnTeach.Size = new System.Drawing.Size(75, 23);
            this.btnTeach.TabIndex = 1;
            this.btnTeach.Text = "Обучить";
            this.btnTeach.UseVisualStyleBackColor = true;
            //
            // textBox1
            //
            this.textBox1.Location = new System.Drawing.Point(18, 19);
            this.textBox1.Name = "textBox1";
            this.textBox1.Size = new System.Drawing.Size(75, 20);
            this.textBox1.TabIndex = 2;
            //
            // btnGuess
            //
```

```
this.btnGuess.Location = new System.Drawing.Point(335, 93);
this.btnGuess.Name = "btnGuess";
this.btnGuess.Size = new System.Drawing.Size(113, 23);
this.btnGuess.TabIndex = 3;
this.btnGuess.Text = "Угадать";
this.btnGuess.UseVisualStyleBackColor = true;
//
// btnClear
//
this.btnClear.Location = new System.Drawing.Point(335, 162);
this.btnClear.Name = "btnClear";
this.btnClear.Size = new System.Drawing.Size(113, 34);
this.btnClear.TabIndex = 4;
this.btnClear.Text = "Очистка окна ввода";
this.btnClear.UseVisualStyleBackColor = true;
//
// Teach
//
this.Teach.Controls.Add(this.btnTeach);
this.Teach.Controls.Add(this.textBox1);
this.Teach.Location = new System.Drawing.Point(335, 12);
this.Teach.Name = "Teach";
this.Teach.Size = new System.Drawing.Size(113, 75);
this.Teach.TabIndex = 5;
this.Teach.TabStop = false;
this.Teach.Text = "Обучение";
//
// richTextBox1
//
this.richTextBox1.Location = new System.Drawing.Point(335, 217);
this.richTextBox1.Name = "richTextBox1";
this.richTextBox1.ReadOnly = true;
this.richTextBox1.Size = new System.Drawing.Size(113, 93);
this.richTextBox1.TabIndex = 6;
this.richTextBox1.Text = "";
//
// btnCleatNet
//
this.btnCleatNet.Location = new System.Drawing.Point(335, 316);
this.btnCleatNet.Name = "btnCleatNet";
this.btnCleatNet.Size = new System.Drawing.Size(113, 22);
this.btnCleatNet.TabIndex = 9;
this.btnCleatNet.Text = "Очистка нейросети";
this.btnCleatNet.UseVisualStyleBackColor = true;
//
// label1
//
this.label1.AutoSize = true;
this.label1.Location = new System.Drawing.Point(332, 201);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(110, 13);
this.label1.TabIndex = 10;
this.label1.Text = "Символы нейросети";
//
// btnSave
//
this.btnSave.Location = new System.Drawing.Point(187, 316);
this.btnSave.Name = "btnSave";
this.btnSave.Size = new System.Drawing.Size(125, 23);
this.btnSave.TabIndex = 11;
this.btnSave.Text = "Сохранить нейросеть";
this.btnSave.UseVisualStyleBackColor = true;
//
// btnLoad
```

```
            //
            this.btnLoad.Location = new System.Drawing.Point(12, 316);
            this.btnLoad.Name = "btnLoad";
            this.btnLoad.Size = new System.Drawing.Size(129, 23);
            this.btnLoad.TabIndex = 12;
            this.btnLoad.Text = "Загрузить нейросеть";
            this.btnLoad.UseVisualStyleBackColor = true;
            //
            // btnOrient
            //
            this.btnOrient.Location = new System.Drawing.Point(335, 122);
            this.btnOrient.Name = "btnOrient";
            this.btnOrient.Size = new System.Drawing.Size(113, 34);
            this.btnOrient.TabIndex = 13;
            this.btnOrient.Text = "R";
            this.btnOrient.UseVisualStyleBackColor =false;
            //
            // Form1
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.BackColor = System.Drawing.Color.BlanchedAlmond;
            this.ClientSize = new System.Drawing.Size(471, 350);
            this.Controls.Add(this.btnOrient);
            this.Controls.Add(this.label1);
            this.Controls.Add(this.btnCleatNet);
            this.Controls.Add(this.richTextBox1);
            this.Controls.Add(this.Teach);
            this.Controls.Add(this.btnClear);
            this.Controls.Add(this.btnGuess);
            this.Controls.Add(this.pbField);
            this.Controls.Add(this.btnSave);
            this.Controls.Add(this.btnLoad);
            this.Name = "Form1";
            this.StartPosition =
System.Windows.Forms.FormStartPosition.CenterScreen;
            this.Text = "Нейросеть";

((System.ComponentModel.ISupportInitialize)(this.pbField)).EndInit();
            this.Teach.ResumeLayout(false);
            this.Teach.PerformLayout();
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion

        public System.Windows.Forms.PictureBox pbField;
        public System.Windows.Forms.Button btnTeach;
        public System.Windows.Forms.TextBox textBox1;
        public System.Windows.Forms.Button btnGuess;
        public System.Windows.Forms.Button btnClear;
        public System.Windows.Forms.GroupBox Teach;
        public System.Windows.Forms.RichTextBox richTextBox1;
        public System.Windows.Forms.Button btnCleatNet;
        private System.Windows.Forms.Label label1;
        public System.Windows.Forms.Button btnSave;
        public System.Windows.Forms.Button btnLoad;
        public System.Windows.Forms.Button btnOrient;
    }
```

// Perseptron.cs

```
    public static class Constants
    {
        public const int Count = 8;  // задаем константу
    }

    public class Seti
    {
        private Bitmap bmp;
        private Graphics g;
        private Graphics gw;
        private Image img;
        private Form1 frm;
        Pen pen = new Pen(Color.Black, 1);

        private int step = 0;

        private bool[,] cells = new bool[Constants.Count, Constants.Count];

        private NeuronNet net = new NeuronNet((Constants.Count *
Constants.Count)); // создаем новую нейросеть

        public Seti(Form1 frm)  // создает поле для прорисовки символа
        {
            this.frm = frm;
            InitCells();

            bmp = new Bitmap(frm.pbField.Width, frm.pbField.Height);  //
Инкапсулирует точечный рисунок GDI+, состоящий из данных пикселей
графического изображения
            img = Image.FromHbitmap(bmp.GetHbitmap());  //Создает объект img
точечного рисунка GDI из данного изображения bmp
            g = Graphics.FromImage(img);  // Создает новый объект Graphics из
указанного рисунка Image.
            g.SmoothingMode = SmoothingMode.HighQuality;
            gw = frm.pbField.CreateGraphics();  // Задает объект Graphics для
элемента управления.

            DrawNet();  // прорисовка сети
            RedrawNet();  // прорисовка сети
            EventClass();  // метод с делегатми
        }

        private void EventClass()  // метод с делегатми
        {
            frm.pbField.Paint += new
System.Windows.Forms.PaintEventHandler(this.pbField_Paint);
            frm.pbField.MouseClick += new
System.Windows.Forms.MouseEventHandler(this.pbField_MouseClick);
            frm.pbField.MouseMove += new
System.Windows.Forms.MouseEventHandler(this.pbField_MouseMove);
            frm.MouseMove += new
System.Windows.Forms.MouseEventHandler(this.Form1_MouseMove);
            frm.Validating += new
System.ComponentModel.CancelEventHandler(this.Form1_Validating);
            frm.Validated += new System.EventHandler(this.Form1_Validated);
            frm.Load += new System.EventHandler(this.Form1_Load);
            frm.Shown += new System.EventHandler(this.Form1_Shown);
            frm.Paint += new
System.Windows.Forms.PaintEventHandler(this.Form1_Paint);
            frm.btnClear.Click += new
System.EventHandler(this.button3_Click);
            frm.btnGuess.Click += new
System.EventHandler(this.button2_Click);
```

```
            frm.btnTeach.Click += new
System.EventHandler(this.button1_Click);
            frm.btnCleatNet.Click += new
System.EventHandler(this.btnCleatNet_Click);
            frm.btnSave.Click += new System.EventHandler(this.btnSave_Click);
            frm.btnLoad.Click += new System.EventHandler(this.btnLoad_Click);
//            frm.btnOrient.Click +=new EventHandler(btnOrient_Click);
        }

        private void DrawNet()  // разбиение PictureBox на клетки и
прорисовка сетки
        {
            int currPosX = 0;
            int currPosY = 0;

            step = (frm.pbField.Height - 2) / Constants.Count;  // определяем
шаг для разбиения PictureBox на клетки

            g.Clear(Color.White);
            for (int i = 0; i < Constants.Count; i++, currPosY += step)
            {
                currPosX = 0;
                for (int j = 0; j < Constants.Count; j++, currPosX += step)
                {
                    if (j == Constants.Count)
                    {
                        pen.Color = Color.Black;
                        pen.Width = 1;
                    }
                    else
                        g.DrawLine(pen, currPosX, 0, currPosX, frm.Width -
2);  // прорисовка двух линий по X
                }
                g.DrawLine(pen, 0, currPosY, frm.Width - 2, currPosY);  //
прорисовка двух линий по У
            }
        }

        private void RedrawNet()  // перерисовывает рисунок
        {
            DrawNet();
            DrawAllCells();
            gw.DrawImage(img, new PointF(0, 0));  // отображение рисунка в
PicturBox
        }
        private void DrawAllCells()  // прорисовка всех клеток
        {
            for (int i = 0; i < Constants.Count; i++)
            {
                for (int j = 0; j < Constants.Count; j++)
                {
                    DrawCell(i, j);  // закрашивание клетки
                }
            }
        }
        private void DrawCell(int i, int j)  // прорисовка закрашеных
прямоугольников (клеток)
        {
            int y = (i * step);
            int x = (j * step);

            Brush brush = cells[i, j] ? new SolidBrush(Color.Coral) : new
SolidBrush(Color.White); // определения цвета клетки
```

```
            g.FillRectangle(brush, x - 1, y - 1, step, step);  // прорисоовка
прямоугольника (клетки)
        }

        private void InitCells()  // создание и индексация клеток со
значениями false
        {
            for (int i = 0; i < Constants.Count; i++)
            {
                for (int j = 0; j < Constants.Count; j++)
                {
                    cells[i, j] = false;
                }
            }
        }

        private void SetCell(int i, int j)  // установка значения клетки
false
        {
            cells[i, j] = true;
        }

        private void UnsetCell(int i, int j)    // установка значения клетки
true
        {
            cells[i, j] = false;
        }

        // методы для реализации событий - перерисовать PictureBox
        private void pbField_Paint(object sender, PaintEventArgs e)
        {
            RedrawNet();
        }

        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            RedrawNet();
        }

        private void Form1_Shown(object sender, EventArgs e)
        {
            RedrawNet();
        }

        private void Form1_Validated(object sender, EventArgs e)
        {
            RedrawNet();
        }

        private void Form1_Validating(object sender, CancelEventArgs e)
        {
            RedrawNet();
        }

        private void Form1_MouseMove(object sender, MouseEventArgs e)
        {
            RedrawNet();
        }


        private void pbField_MouseClick(object sender, MouseEventArgs e) //
при нажатии мышки на клетку в PictrureBox, устанавливает занчение клетки true
или false
        {
```

```
            int j = (int)Math.Floor((float)((e.X) / step));  // определяем
индекс клетки по X
            int i = (int)Math.Floor((float)((e.Y) / step));  // определяем
индекс клетки по У

            if (i >= Constants.Count || j >= Constants.Count || i < 0 || j <
0)  //проверка на корректность
                return;

            if (e.Button == MouseButtons.Left)  // если нажата левая кнопка
мыши, то значение клетки становится true
                SetCell(i, j);

            else
                if (e.Button == MouseButtons.Right)  // если нажата правая,
то значение клетки становится false
                {
                    UnsetCell(i, j);
                }
            RedrawNet();
        }

        private void pbField_MouseMove(object sender, MouseEventArgs e) //
при движеии мышки с зажатой кнопкой по клеткам в PictrureBox, устанавливает
значение клетки true или false
        {
            int j = (int)Math.Floor((float)((e.X) / step));
            int i = (int)Math.Floor((float)((e.Y) / step));

            if (i >= Constants.Count || j >= Constants.Count || i < 0 || j <
0)
                return;

            if (e.Button == MouseButtons.Left)
                SetCell(i, j);
            else if (e.Button == MouseButtons.Right)
            {
                UnsetCell(i, j);
            }
            RedrawNet();
        }

        private void SetInputs()  // на вход нейросети задает двоичные
сигналы в соответсвии со значениями клеток(1 или 0)
        {
            int k = 0;
            for (int i = 0; i < Constants.Count; i++)
            {
                for (int j = 0; j < Constants.Count; j++)
                {
                    net.Inputs[k].Value = cells[i, j] ? 1 : 0;
                    k++;
                }
            }
        }
        private void button1_Click(object sender, EventArgs e)  // метод,
который обучает нейросеть
        {

            var name = frm.textBox1.Text;  // присваиваем переменной name
текст из textBox1

            if (!net.Percs.Any(x => x.Name == name))  // проверка на наличие
персептрона с таким же именем
```

```
            {
                net.AddNewPerc(name);  // если нет, то добавляется новый
персептрон с именем значения name
                frm.richTextBox1.AppendText(name + "\n");  // добавляем имя
персептрона в richTextBox1
            }

            SetInputs();  // подаем на вход двоичные сигналы
            net.TeachFromInput(name);  // обучаем нейронную сеть
        }

        private void button2_Click(object sender, EventArgs e)  // метод,
который угадывает символ
        {

            SetInputs();

            string str1 = "";

            for (int i = 0; i < net.NOuts; i++)  // перебор всех персептронов
            {
                if (net.Percs[i].Out == 1)  // если выход равен 1, то имя
персептрона выводится на Messagebox
                {
                    str1 = "Символ: ";
                    str1 += net.Percs[i].Name + " ";
                }
            }
            MessageBox.Show(str1);
        }
/*
private void btnOrient_Click(object sender, EventArgs e)  // метод, который
определяет ориентацию символа
        {
            double igor = 0;
            double iver = 0;
            double c = 0.4;  // коэффициент точности определения ориентации
            string str1 = "";
            for (int i = 0; i < Constants.Count; i++)
            {
                int gor = 0, ver = 0;
                for (int j = 0; j < Constants.Count; j++)
                {
                    if (cells[i, j] == true) { gor += 1; }  // проверка
клеток по горизонтали
                    if (cells[j, i] == true) { ver += 1; }  // проверка
клеток по вертикали
                }
                // подсчет суммы квадратов нарисованных клеток по
горизонтали, по вертикали
                igor += Math.Pow(gor, 2); iver += Math.Pow(ver, 2);
            }
            // если отношение сумм квадратов больше порогового, то ориентация
вертикальна либо горизонтальна
            if (iver / igor > 1 + c) { str1 += "Ориентир вертикальный"; }
            else if (iver / igor < 1 - c) { str1 += "Ориентир
горизонтальный"; }
            else { str1 += "Ориентир равномерный"; }
            MessageBox.Show(str1);
        }
*/
        private void button3_Click(object sender, EventArgs e)  // метод для
очистки поля ввода
        {
```

```
            InitCells();
            //DrawAllCells();
            RedrawNet();
        }


        private void btnCleatNet_Click(object sender, EventArgs e)  // метод
для очистки нейросети
        {
            net = new NeuronNet(Constants.Count * Constants.Count);
            frm.richTextBox1.Clear();
            InitCells();
            RedrawNet();
        }

        private void btnSave_Click(object sender, EventArgs e)  // метод для
сохранения нейросети. Сериализация в двоичный формат с помощью
BinaryFormatter
        {
            BinaryFormatter formatter = new BinaryFormatter();  //Сохраняем
состояние объекта net в двоичном формате с помощью
            using (var fstream = new FileStream("./NeuronNet.dat",
FileMode.Create, FileAccess.Write, FileShare.None))  // в файле NeuronNet.dat
            {
                formatter.Serialize(fstream, net);
            }
        }

        private void btnLoad_Click(object sender, EventArgs e)  // метода для
загрузки нейросети
        {
            BinaryFormatter formatter = new BinaryFormatter();  //
восстановим состояние объекта из файла NeuronNet.dat
            using (var fstream = File.OpenRead("./NeuronNet.dat"))
            {
                net = (NeuronNet)formatter.Deserialize(fstream);
            }
            frm.richTextBox1.Clear();
            foreach (Perc pr in net.Percs)  // выводим имена всех сохраненных
персептронов
            {
                frm.richTextBox1.AppendText(pr.Name + "\n");
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
        }
    }

    [Serializable]  // метка что класс сериализуется
    public class Input  // входной сигнал
    {
        public int Value { get; set; }
        public int Index { get; set; }
    }
    [Serializable]
    public class Link  // связь с входным сигналом и его весами
    {
        public Input Input { get; set; }
        public float Weight { get; set; }

        public Link(Input input, float weight)
        {
```

```
            Input = input;
            Weight = weight;
        }

        private Link()
        {
        }
    }
    [Serializable]
    public class NeuronNet  // класс где задается структура нейросети
    {

        public List<Input> Inputs { get; set; }

        public List<Link> Links { get; set; }


        public List<Perc> Percs { get; set; }

        public int NInputs { get; set; }
        public int NOuts { get; set; }

        public NeuronNet(int nInputs) // метод для создания нейросети
        {


            NInputs = nInputs;
            NOuts = 0;


            Percs = new List<Perc>();
            Inputs = new List<Input>();

            Links = new List<Link>();



            for (int i = 0; i < nInputs; i++)
            {
                Inputs.Add(new Input { Index = i });  // индексация входных
сигналов
            }

        }


        public void TeachFromInput(string name)  // метод для обучения
нейросети
        {

            int index = Percs.IndexOf(Percs.First(x => x.Name == name));
            //коэффициент быстроты обучения сети
            float etha = 0.4f;


            int[] X = new int[NInputs];

            int[] D = new int[NOuts];

            int[] Y = new int[NOuts];

            int[] E = new int[NOuts];
```

```
            for (int i = 0; i < NInputs; i++)
            {
                if (Inputs[i].Index < Constants.Count * Constants.Count)
                {

                    X[i] = Inputs[i].Value;
                }
            }

            D[index] = 1;


            for (int i = 0; i < NOuts; i++)
            {

                Y[i] = Percs[i].Out;

                E[i] = D[i] - Y[i];
            }


            for (int i = 0; i < NOuts; i++)
            {
                for (int j = 0; j < Percs[i].Links.Count; j++)
                {
                    Percs[i].Links[j].Weight += etha * E[i] * X[j];  //
увеличение весов
                }
            }
        }

        public void AddNewPerc(string name)  // метод для добавления нового
персептрона
        {
            NOuts++;

            var newPerc = new Perc(name);  // создание объекта персептрон


            for (int j = 0; j < NInputs; j++)
            {
                Link link = new Link(Inputs[j], 0);


                Links.Add(link);


                newPerc.Links.Add(link);

            }

            Percs.Add(newPerc);
        }



    }
    [Serializable]
    public class Perc  // класс, где задается структура персептрона
    {
        public const double StepValue = 1;  //пороговое значение
        public readonly string Name;
        public List<Link> Links { get; set; }
```

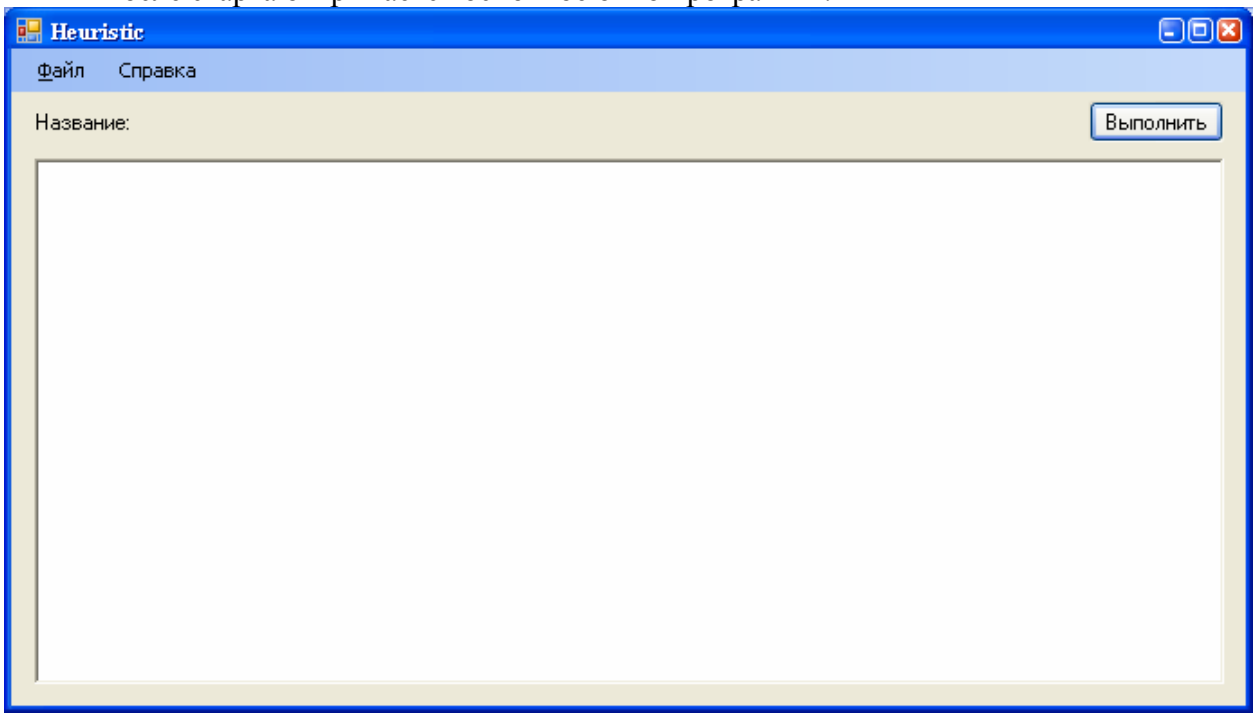
```
        public Perc(string name)  // метод для создания объетка персептрон
        {
            Links = new List<Link>();
            Name = name;
        }

        private Perc()
        {
        }
        public float Sum  // метод сумматор
        {
            get
            {
                float _sum = 0;

                foreach (var link in Links)
                {
                    _sum += link.Input.Value * link.Weight; // взвешанные
веса
                }
                return _sum;
            }
        }

        private bool StepFunction(float Sum)  // сравнение с пороговым
значением
        {
            return Sum >= StepValue;
        }

        public int Out  // определения выхоного сигнала
        {
            get { return StepFunction(Sum) ? 1 : 0; }  //задаем выход
        }
    }

// Program.cs
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Form1 frm1 = new Form1(); // создание формы
            Seti frm = new Seti(frm1);  // создание объекта Сети
            Application.Run(frm1);  // запуск формы frm1
        }
    }

}
```

Эвристическая машина.

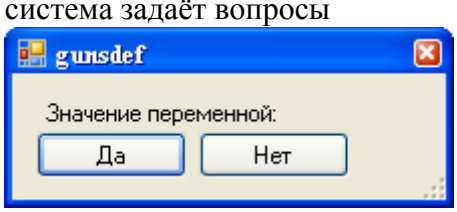
Программа представляет собой экспертную систему. Исходный текст написан на языке C# с использованием графического интерфейса пользователя (Windows Forms),

имеет дружественный интерфейс. Управляют программой эвристические модели, записанные в виде текстовых файлов.

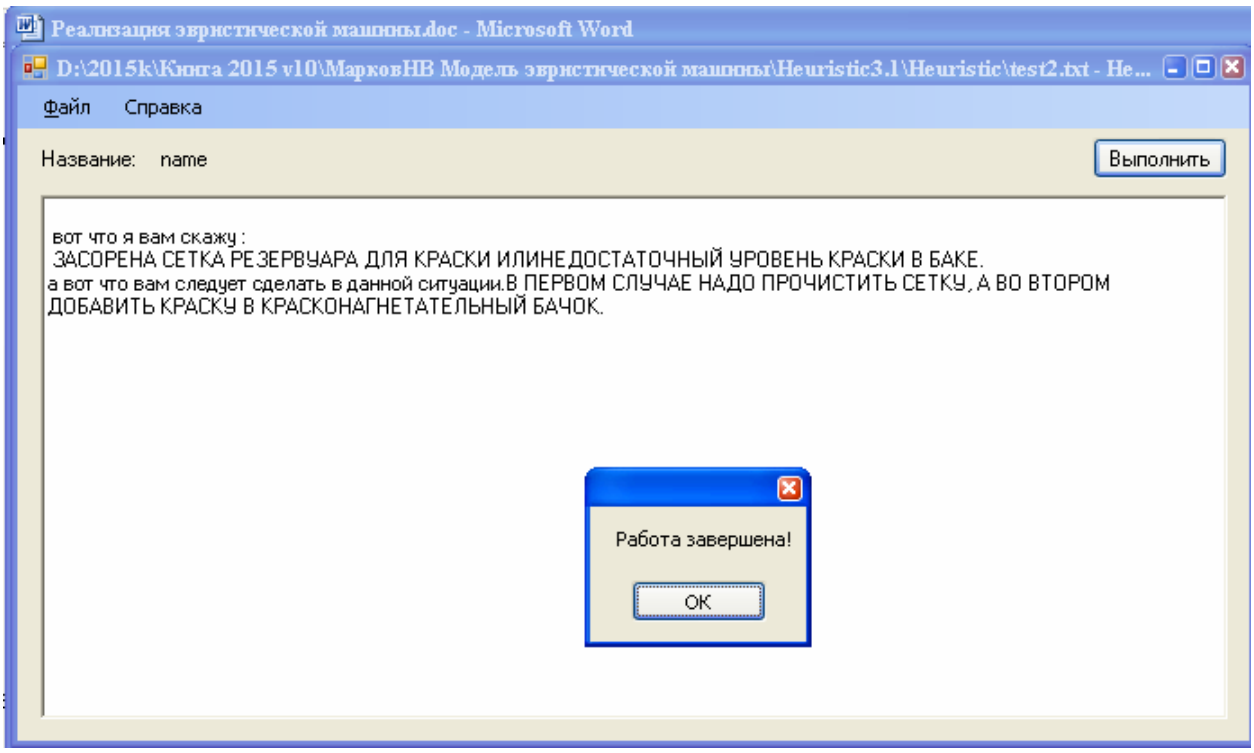
После старта открывается основное окно программы:



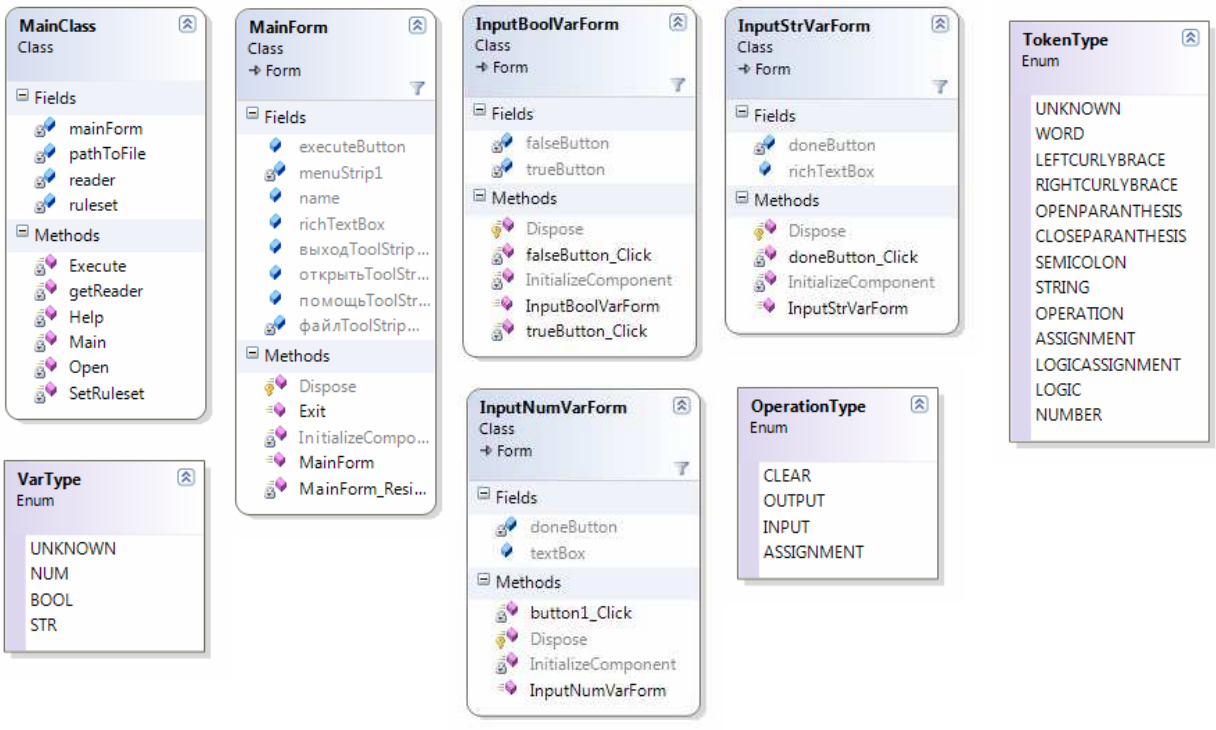
Для работы программы необходимо загрузить эвристическую модель из текстового файла. Для примера в приложении 1 приведена демонстрационная модель. После загрузки эвристической модели нажатие кнопки Пуск начинает диалог, во время которого пользователь сообщает исходную информацию для решаемой задачи, а экспертная система задаёт вопросы

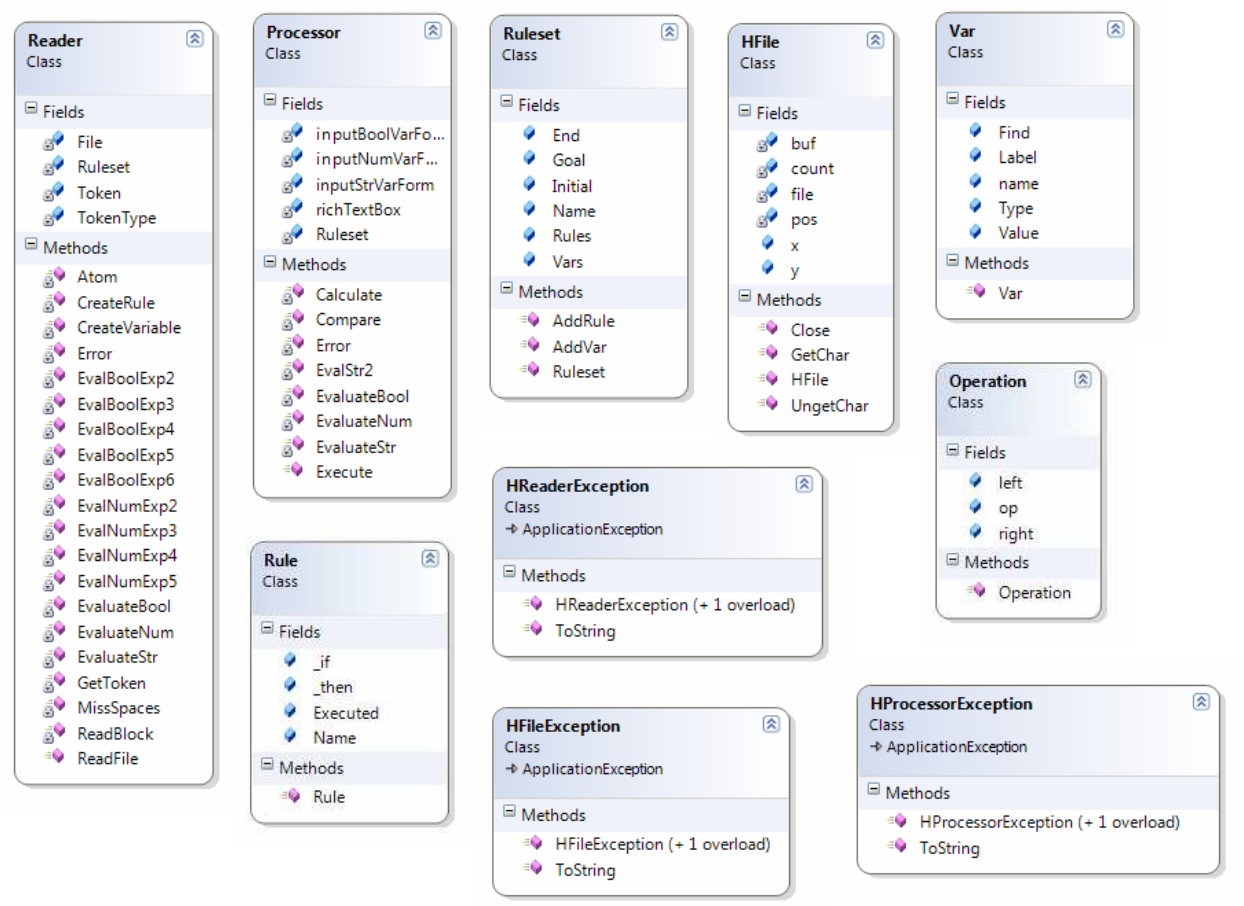


и выводит на экран результат решения в виде рекомендаций эксперта.



Структура экспертной системы может быть представлена в виде диаграммы классов:





Назначение классов:

- Класс Ruleset переводит файл с набором команд в понятный для машины вид. Класс Processor выполняет команды, содержащиеся в объекте класса Ruleset.
- Основной класс и класс основного окна программы: MainClass и MainForm Class.
- Классы-формы для ввода данных (их названия начинаются со слова Input)
- Класс Reader, осуществляющий чтение файла и формирование объекта класса Ruleset.
- Класс HFile осуществляет буферизацию символов потока и при необходимости возвращает символы в поток.
- Элементы классов Var и Rule реализуют одноименные поля в файлах.
- Перечисление VarType используется для определения типа переменной.
- TokenType служит для определения типа токена (токен (лексема) - последовательность символов имеющих значение для интерпретатора).
- Operation – класс реализующий вершины-операции в дереве разбора.
- OperationType – тип операции (сложение, вычитание и т.д.).

- Классы-исключения создаваемые при нахождении программой ошибок времени исполнения:

Приложение 1. Исходный текст эвристической машины.

```
using System;
using System.Windows.Forms;
using System.Collections;
using System.Linq;
using System.IO;
using System.Drawing;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Text;

namespace Heuristic
{
    class MainClass
    {
        static Ruleset ruleset;
        static Reader reader;
        static MainForm mainForm;
        static string pathToFile;

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);

            mainForm = new MainForm();
            //mainForm.выходToolStripMenuItem.Click += new
EventHandler(Exit);
            mainForm.открытьToolStripMenuItem.Click += new
EventHandler(MainClass.Open);
            mainForm.executeButton.Click += new
EventHandler(MainClass.Execute);

            Application.Run(mainForm);
        }

        static void Execute(object sender, EventArgs e)
        {
            if (ruleset != null)
            {
                Processor proc = new Processor();
                proc.Execute(ruleset, mainForm.richTextBox);
            }
            else
            {
                MessageBox.Show("Не загружен ни один набор правил!");
            }
        }

        static void SetRuleset(string nameOfFile)
        {
            mainForm.name.Text = ruleset.Name;
            mainForm.Text = nameOfFile + " - Heuristic";
```

```
            return;
        }

        // Открывает файл и считывает из него набор правил
        static void Open(object sender, EventArgs e)
        {
            OpenFileDialog file = new OpenFileDialog();
            if (file.ShowDialog() == DialogResult.OK)
            {
                pathToFile = file.FileName; // сохраняем имя файла для
дальнейшего использования
                //reader = new Reader(realPath);
                ruleset = getReader().ReadFile(pathToFile);
                SetRuleset(pathToFile);
            }
        }

        /* создает новый экземпляр класса Reader если ещё не создан
         * либо возвращает уже созданный экземпляр */
        static Reader getReader()
        {
            if (reader == null)
                return reader = new Reader();
            else
                return reader;
        }
    }
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
        }

        private void MainForm_Resize(object sender, EventArgs e)
        {
            this.executeButton.Location = new Point(this.Width - 103,
this.executeButton.Location.Y);
            this.richTextBox.Size = new Size(this.Width - 40, this.Height -
108);
        }

        /// <summary>
        /// Завершение работы программы
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        public void Exit(object sender, EventArgs e)
        {
            Application.Exit();
            //Environment.Exit(0);
        }
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be
disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
```

```
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.menuStrip1 = new System.Windows.Forms.MenuStrip();
            this.файлToolStripMenuItem = new
System.Windows.Forms.ToolStripMenuItem();
            this.открытьToolStripMenuItem = new
System.Windows.Forms.ToolStripMenuItem();
            this.toolStripSeparator1 = new
System.Windows.Forms.ToolStripSeparator();
            this.выходToolStripMenuItem = new
System.Windows.Forms.ToolStripMenuItem();
            this.помощьToolStripMenuItem = new
System.Windows.Forms.ToolStripMenuItem();
            this.nameLabel = new System.Windows.Forms.Label();
            this.name = new System.Windows.Forms.Label();
            this.executeButton = new System.Windows.Forms.Button();
            this.richTextBox = new System.Windows.Forms.RichTextBox();
            this.menuStrip1.SuspendLayout();
            this.SuspendLayout();
            //
            // menuStrip1
            //
            this.menuStrip1.Items.AddRange(new
System.Windows.Forms.ToolStripItem[] {
            this.файлToolStripMenuItem,
            this.помощьToolStripMenuItem});
            this.menuStrip1.Location = new System.Drawing.Point(0, 0);
            this.menuStrip1.Name = "menuStrip1";
            this.menuStrip1.Size = new System.Drawing.Size(684, 24);
            this.menuStrip1.TabIndex = 0;
            this.menuStrip1.Text = "menuStrip1";
            //
            // файлToolStripMenuItem
            //
            this.файлToolStripMenuItem.DropDownItems.AddRange(new
System.Windows.Forms.ToolStripItem[] {
            this.открытьToolStripMenuItem,
            this.toolStripSeparator1,
            this.выходToolStripMenuItem});
            this.файлToolStripMenuItem.Name = "файлToolStripMenuItem";
            this.файлToolStripMenuItem.Size = new System.Drawing.Size(48,
20);
            this.файлToolStripMenuItem.Text = "&Файл";
            //
            // открытьToolStripMenuItem
            //
            this.открытьToolStripMenuItem.Name = "открытьToolStripMenuItem";
            this.открытьToolStripMenuItem.Size = new System.Drawing.Size(152,
22);
            this.открытьToolStripMenuItem.Text = "Открыть...";
            //
```

```
            // toolStripSeparator1
            //
            this.toolStripSeparator1.Name = "toolStripSeparator1";
            this.toolStripSeparator1.Size = new System.Drawing.Size(149, 6);
            //
            // выходToolStripMenuItem
            //
            this.выходToolStripMenuItem.Name = "выходToolStripMenuItem";
            this.выходToolStripMenuItem.Size = new System.Drawing.Size(152,
22);
            this.выходToolStripMenuItem.Text = "Выход";
            this.выходToolStripMenuItem.Click += new
System.EventHandler(this.Exit);
            //
            // помощьToolStripMenuItem
            //
            this.помощьToolStripMenuItem.Name = "помощьToolStripMenuItem";
            this.помощьToolStripMenuItem.Size = new System.Drawing.Size(65,
20);
            this.помощьToolStripMenuItem.Text = "Справка";
            //
            // nameLabel
            //
            this.nameLabel.AutoSize = true;
            this.nameLabel.Location = new System.Drawing.Point(10, 34);
            this.nameLabel.Name = "nameLabel";
            this.nameLabel.Size = new System.Drawing.Size(60, 13);
            this.nameLabel.TabIndex = 1;
            this.nameLabel.Text = "Название:";
            //
            // name
            //
            this.name.AutoSize = true;
            this.name.Location = new System.Drawing.Point(76, 34);
            this.name.Name = "name";
            this.name.Size = new System.Drawing.Size(0, 13);
            this.name.TabIndex = 2;
            //
            // executeButton
            //
            this.executeButton.Location = new System.Drawing.Point(597, 29);
            this.executeButton.Name = "executeButton";
            this.executeButton.Size = new System.Drawing.Size(75, 23);
            this.executeButton.TabIndex = 8;
            this.executeButton.Text = "Выполнить";
            this.executeButton.UseVisualStyleBackColor = true;
            //
            // richTextBox
            //
            this.richTextBox.Location = new System.Drawing.Point(13, 61);
            this.richTextBox.Name = "richTextBox";
            this.richTextBox.Size = new System.Drawing.Size(659, 291);
            this.richTextBox.TabIndex = 9;
            this.richTextBox.Text = "";
            //
            // MainForm
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(684, 364);
            this.Controls.Add(this.richTextBox);
            this.Controls.Add(this.executeButton);
            this.Controls.Add(this.nameLabel);
            this.Controls.Add(this.menuStrip1);
```

```
            this.Controls.Add(this.name);
            this.MainMenuStrip = this.menuStrip1;
            this.Name = "MainForm";
            this.Text = "Heuristic";
            this.Resize += new System.EventHandler(this.MainForm_Resize);
            this.menuStrip1.ResumeLayout(false);
            this.menuStrip1.PerformLayout();
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion

        private System.Windows.Forms.MenuStrip menuStrip1;
        private System.Windows.Forms.ToolStripSeparator toolStripSeparator1;
        private System.Windows.Forms.ToolStripMenuItem
помощьToolStripMenuItem;
        public System.Windows.Forms.ToolStripMenuItem
открытьToolStripMenuItem;
        public System.Windows.Forms.ToolStripMenuItem выходToolStripMenuItem;
        private System.Windows.Forms.ToolStripMenuItem файлToolStripMenuItem;
        private System.Windows.Forms.Label nameLabel;
        public System.Windows.Forms.Label name;
        public System.Windows.Forms.Button executeButton;
        public System.Windows.Forms.RichTextBox richTextBox;
    }
    class HProcessorException : ApplicationException
    {
        public HProcessorException() : base() { }
        public HProcessorException(string s) : base(s) { }
        public override string ToString()
        {
            return Message;
        }
    }

    class Processor
    {
        private Ruleset Ruleset;
        private InputNumVarForm inputNumVarForm; // форма для ввода значения
переменной типа num
        private InputStrVarForm inputStrVarForm; // форма для ввода значения
переменной типа str (string)
        private InputBoolVarForm inputBoolVarForm; // форма для ввода
значения переменной типа bool
        private RichTextBox richTextBox;

        public void Execute(Ruleset Ruleset, RichTextBox richTextBox)
        {
            this.richTextBox = richTextBox;
            this.Ruleset = Ruleset;
            int i;

            // Восстанавливаем набор правил в первоначальное состояние
            for (i = 0; i < Ruleset.Rules.Count; i++)
                ((Rule)Ruleset.Rules[i]).Executed = false;
            Ruleset.Vars[Ruleset.Goal].Value = null;
            string[] keys = new string[Ruleset.Vars.Count];
            Ruleset.Vars.Keys.CopyTo(keys, 0);
            for (i = 0; i < keys.Length; i++)
                ((Var)Ruleset.Vars[keys[i]]).Value = null;
            // -----
```

```
            //try
            //{
                for (i = 0; i < Ruleset.Initial.Count; i++)
                {
                    Calculate((object[])Ruleset.Initial[i]);
                }

                for (i = 0; i < Ruleset.Rules.Count; )
                {
                    if (!((Rule)Ruleset.Rules[i]).Executed)
                    {
                        object result =
EvaluateBool(((Rule)Ruleset.Rules[i])._if);
                        if (result == null)
                            result = false;
                        if ((bool)result)
                        {
                            for (int j = 0; j <
((Rule)Ruleset.Rules[i])._then.Count; j++)
                            {

Calculate((object[])(((Rule)Ruleset.Rules[i])._then[j]));
                            }
                            if (Ruleset.Vars[Ruleset.Goal].Value != null)
break;
                            ((Rule)(Ruleset.Rules[i])).Executed = true;
                            //Ruleset.Rules.RemoveAt(i);
                            i = 0; // начинаем просматривать все правила
сначала
                        }
                        else
                        {
                            i++; // переходим к след. правилу
                            continue;
                        }
                    }
                    else i++;
                }

                for (i = 0; i < Ruleset.End.Count; i++)
                {
                    Calculate((object[])Ruleset.End[i]);
                }

            MessageBox.Show("Работа завершена!");
        }

        private void Calculate(object[] expr)
        {
            if (expr[0] is OperationType)
            {
                switch ((OperationType)expr[0])
                {
                    case OperationType.CLEAR:
                        richTextBox.Text = "";
                        break;
                    case OperationType.INPUT:
                        switch (((Var)expr[1]).Type)
                        {
                            case VarType.NUM:
                                double number;
                                while (true)
                                {
                                    inputNumVarForm = new InputNumVarForm();
```

```
                                    inputNumVarForm.Text =
((Var)expr[1]).name;
                                    inputNumVarForm.ShowDialog();
                                    try
                                    {
                                        number =
double.Parse(inputNumVarForm.textBox.Text);
                                        break;
                                    }
                                    catch
                                    {
                                        MessageBox.Show("Не получилось
считать число. Попробуйте снова", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error);
                                        continue;
                                    }
                                }
                                ((Var)expr[1]).Value = number;
                                break;
                            case VarType.BOOL:
                                while (true)
                                {
                                    inputBoolVarForm = new
InputBoolVarForm();
                                    inputBoolVarForm.Text =
((Var)expr[1]).name;
                                    inputBoolVarForm.ShowDialog();
                                    switch (inputBoolVarForm.DialogResult)
                                    {
                                        case DialogResult.Yes:
                                            ((Var)expr[1]).Value = true;
                                            break;
                                        case DialogResult.No:
                                            ((Var)expr[1]).Value = false;
                                            break;
                                        default:
                                            continue;
                                    }
                                    break;
                                }
                                Console.WriteLine();
                                break;
                            case VarType.STR:
                                inputStrVarForm = new InputStrVarForm();
                                inputStrVarForm.Text = ((Var)expr[1]).name;
                                inputStrVarForm.ShowDialog();
                                ((Var)expr[1]).Value =
inputStrVarForm.richTextBox.Text;
                                break;
                            /*default:
                                break;*/
                        }
                        break;
                    case OperationType.OUTPUT:
                        if (expr[1] is Var)
                        {
                            switch (((Var)expr[1]).Type)
                            {
                                case VarType.BOOL:
                                    switch ((bool)(((Var)expr[1]).Value))
                                    {
                                        case true:
                                            richTextBox.Text += "true";
                                            break;
```

```
                                        case false:
                                            richTextBox.Text += "false";
                                            break;
                                        /*default:
                                            break;*/
                                    }
                                    break;
                                case VarType.NUM:
                                    richTextBox.Text +=
(double)(((Var)expr[1]).Value);
                                    break;
                                case VarType.STR:
                                    if ((((Var)expr[1]).Value) == null)
                                    {
                                        if (((Var)expr[1]).Find != null)
                                        {
                                            for (int i = 0; i <
((Var)expr[1]).Find.Count; i++)
                                            {

Calculate((object[])((Var)expr[1]).Find[i]);
                                            }
                                            if (((Var)expr[1]).Value != null)
                                                richTextBox.Text +=
(string)(((Var)expr[1]).Value);
                                            break;
                                        }
                                        break;
                                    }
                                    richTextBox.Text +=
(string)(((Var)expr[1]).Value);
                                    break;
                                /*case VarType.UNKNOWN:
                                    break;
                                default:
                                    break;*/
                            }
                        }
                        else
                        {
                            richTextBox.Text += (string)expr[1];
                        }
                        break;
                    /*default:
                        break;*/
                }
            }
            else if (expr[0] is string) // т.е. присваивание
            {
                switch (((Var)expr[1]).Type)
                {
                    case VarType.NUM:
                        switch ((string)expr[0])
                        {
                            case "=":
                                ((Var)expr[1]).Value =
(double)EvaluateNum(expr[2]);
                                break;
                            case "+=":
                                ((Var)expr[1]).Value =
(double)(((Var)expr[1]).Value) + (double)EvaluateNum(expr[2]);
                                break;
                            case "-=":
```

```
                                ((Var)expr[1]).Value =
(double)(((Var)expr[1]).Value) - (double)EvaluateNum(expr[2]);
                                break;
                            case "*=":
                                ((Var)expr[1]).Value =
(double)(((Var)expr[1]).Value) * (double)EvaluateNum(expr[2]);
                                break;
                            case "/=":
                                ((Var)expr[1]).Value =
(double)(((Var)expr[1]).Value) / (double)EvaluateNum(expr[2]);
                                break;
                            /*default:
                                break;*/
                        }
                        break;
                    case VarType.STR:
                        switch ((string)expr[0])
                        {
                            case "=":
                                ((Var)expr[1]).Value = EvaluateStr(expr[2]);
                                break;
                            case "+=":
                                ((Var)expr[1]).Value =
(string)(((Var)expr[1]).Value) + EvaluateStr(expr[2]);
                                break;
                        }
                        break;
                    case VarType.BOOL:
                        switch ((string)expr[0])
                        {
                            case "=":
                                ((Var)expr[1]).Value = EvaluateBool(expr[2]);
                                break;
                            default:
                                Error("Неизвестная операция для применения к
типу bool (ф-ия Calculate())!");
                                break;
                        }
                        break;
                    /*default:
                        break;*/
                }
            }
            else
            {
                Error("Ошибка в функции Calculate()!");
            }
        }


        // Надо ли возвращать null?
        private object EvaluateBool(object p)
        {
            object result;
            object right;

            if (p is Var)
            {
                if ((((Var)p).Value) == null)
                {
                    if (((Var)p).Find != null)
                    {
                        for (int i = 0; i < ((Var)p).Find.Count; i++)
                        {
```

```
                    Calculate((object[])((Var)p).Find[i]);
                }
                if ((((Var)p).Value) == null)
                    return null;
                return ((bool)(((Var)p).Value));
            }
            return null;
        }
        else return (bool)(((Var)p).Value);
    }
    else if (p is Operation)
    {
        switch (((Operation)p).op)
        {
            case "!":
                result = EvaluateBool(((Operation)p).left);
                if (result == null)
                    return false;
                else
                    return !(bool)result;
            case "&":
                result = EvaluateBool(((Operation)p).left);
                if (result == null)
                    return false;
                else
                    right = EvaluateBool(((Operation)p).right);
                if (right == null)
                    return false;
                else return (bool)result & (bool)right;
            case "|":
                result = EvaluateBool(((Operation)p).left);
                if (result == null)
                    return false;
                else
                    right = (bool)EvaluateBool(((Operation)p).right);
                if (right == null)
                    return false;
                else return (bool)result | (bool)right;
            default:
                return Compare(p);
        }
    }
    else if (p is bool)
    {
        return (bool)p;
    }
    else
    {
        Error("Неизвестный тип узла в дереве разбора!");
        return false;
    }
}

private bool Compare(object p)
{
    object left = EvaluateNum(((Operation)p).left);
    if (left == null)
        return false;
    object right = EvaluateNum(((Operation)p).right);
    if (right == null)
        return false;
    switch (((Operation)p).op)
    {
            case ">":
```

```
            if ((double)left > (double)right)
                    return true;
                return false;
            case "<":
                if ((double)left < (double)right)
                    return true;
                return false;
            case ">=":
                if ((double)left >= (double)right)
                    return true;
                return false;
            case "<=":
                if ((double)left <= (double)right)
                    return true;
                return false;
            case "==":
                if ((double)left == (double)right)
                    return true;
                return false;
            case "!=":
                if ((double)left != (double)right)
                    return true;
                return false;
    }
    return false;
}

private string EvaluateStr(object p)
{
    if (p is string)
        return (string)p;
    else if (p is Operation)
        return EvalStr2((Operation)p);
    else
    {
        Error("Неизвестный тип в дереве разбора!");
        return null;
    }
}

private string EvalStr2(Operation p)
{
    string left = "";
    if (p.left is string)
        left = (string)p.left;
    else if (p.left is Var)
        left = (string)(((Var)p.left).Value);

    string right;
    if (((Operation)p).right is Operation)
    {
        right = EvalStr2((Operation)(((Operation)p).right));
        return left + right;
    }
    else if (((Operation)p).right is string)
    {
        return left + (string)(((Operation)p).right);
    }
    else if (((Operation)p).right is Var)
    {
        return left + (string)(((Var)(((Operation)p).right)).Value);
    }
    else
    {
```

```
                Error("Error in EvalStr2!");
                return null;
            }
        }

        // вычисляет арифметическое выражение
        private object EvaluateNum(object root)
        {
            if (root is Var)
            {
                if ((((Var)root).Value) == null)
                {
                    if (((Var)root).Find != null)
                    {
                        for (int i = 0; i < ((Var)root).Find.Count; i++)
                        {
                            Calculate((object[])((Var)root).Find[i]);
                        }
                        if ((((Var)root).Value) == null)
                            return null;
                        return ((double)(((Var)root).Value));
                    }
                    return null;
                }
                return ((double)(((Var)root).Value));
            }
            else if (root is double)
                return (double)root;
            else if (root is Operation)
            {
                if (((Operation)root).right == null) //  унарный "-" и "+"
                {
                    switch (((Operation)root).op)
                    {
                        case "-":
                            return -
(double)EvaluateNum(((Operation)root).left);
                        case "+":
                            return
(double)EvaluateNum(((Operation)root).left);
                    }
                }
                else
                {
                    double left =
(double)EvaluateNum(((Operation)root).left);
                    if ((object)left == null)
                        return null;
                    double right =
(double)EvaluateNum(((Operation)root).right);
                    if ((object)right == null)
                        return null;
                    //---
                    switch (((Operation)root).op)
                    {
                        case "+":
                            return left + right;
                        case "-":
                            return left - right;
                        case "*":
                            return left * right;
                        case "/":
                            return left / right;
                        /*default:
```

```
                            break;*/
                    }
                }
            }
            else
            {
                Error("Вычисление арифметического выражения потерпело
крах!");
                return 0.0;
            }
            return 0.0;
        }

        private void Error(string error)
        {
            throw new HProcessorException(error);
        }
    }
    public enum OperationType : byte
    {
        CLEAR,
        OUTPUT,
        INPUT,
        ASSIGNMENT/*,
        ASSIGNMENT_PLUS,
        ASSIGNMENT_MINUS,
        ASSIGNMENT_DIVISION,
        ASSIGNMENT_MULTI,
        ASSIGNMENT_AND,
        ASSIGNMENT_OR,
        ASSIGNMENT_NOT*/
    }

    public enum TokenType : byte // Тип токена
    {
        UNKNOWN,            // Неизвестный тип
        WORD,               // Слово
        LEFTCURLYBRACE,     // Открытая фигурная скобка
        RIGHTCURLYBRACE,    // Закрытая фигурная скобка
        OPENPARANTHESIS,    // Открытая круглая скобка
        CLOSEPARANTHESIS,   // Открытая круглая скобка
        SEMICOLON,          // Точка с запятой
        STRING,             // Строка
        OPERATION,          // Операция
        ASSIGNMENT,         // Обычное и арифметическое присваивание
        LOGICASSIGNMENT,    // Логическое присваивание
        LOGIC,              // Логическая операция
        NUMBER              // Число
    }

    public enum VarType : byte // Тип переменной
    {
        UNKNOWN,    // Неизвестный тип
        NUM,        // Число
        BOOL,       // Логичекая переменная
        STR         // Строка
    }

    class HReaderException : ApplicationException
    {
        public HReaderException() : base() { }
        public HReaderException(string s) : base(s) { }
        public override string ToString()
        {
```

```
        return Message;
    }
}

class Reader
{
    private Ruleset Ruleset;        // Создаваемый набор правил
    private HFile File;             // Файл для обработки
    private string Token;           // Прочитаный токен
    private TokenType TokenType;    // Тип токена

    /*public Reader(string pathToFile)
    {
        File = new HFile(pathToFile);
    }*/

    public Ruleset ReadFile(string pathToFile)
    {
        File = new HFile(pathToFile);
        try
        {
            GetToken();
            if (Token != "ruleset")
                Error("Ожидается \"ruleset\"");
            GetToken();
            if (TokenType != TokenType.WORD)
                Error("Недопустимое имя для набора правил!");
            Ruleset = new Ruleset(Token);
            GetToken();
            if (TokenType != TokenType.LEFTCURLYBRACE)
                Error("Ожидается {");

            // разбор раздела "vars"
            GetToken();
            if (Token != "vars")
                Error("Ожидается раздел \"vars\"");
            GetToken();
            if (TokenType != TokenType.LEFTCURLYBRACE)
                Error("Ожидается {");
            GetToken();
            while (TokenType != TokenType.RIGHTCURLYBRACE)
            {
                if (TokenType != TokenType.WORD)
                    Error("Ожидается токен типа WORD");
                if (Token == "num")
                    CreateVariable(VarType.NUM);
                else if (Token == "bool")
                    CreateVariable(VarType.BOOL);
                else if (Token == "str")
                    CreateVariable(VarType.STR);
                else
                    Error("Неизвестный тип в объявлении переменной!");
                GetToken();
            }

            // Чтение цели
            GetToken();
            if (Token != "goal")
                Error("Ожидается раздел \"goal\"");
            GetToken();
            if (TokenType != TokenType.WORD)
                Error("Недопустимое имя для цели!");
            Ruleset.Goal = Token;
            GetToken();
```

```
                if (TokenType != TokenType.SEMICOLON)
                    Error("Ожидается ;");

                // Чтение раздела "initial"
                GetToken();
                if (Token != "initial")
                    Error("Ожидается раздел \"initial\"");
                Ruleset.Initial = ReadBlock();

                // Чтение раздела "end"
                GetToken();
                if (Token != "end")
                    Error("Ожидается раздел \"end\"");
                Ruleset.End = ReadBlock();

                // чтение правил
                GetToken();
                while (TokenType != TokenType.RIGHTCURLYBRACE)
                {
                    if (Token != "rule")
                        Error("Ожидается токен типа WORD");
                    CreateRule();
                    GetToken();
                }
            }
            catch (HReaderException e)
            {
                //Console.WriteLine("Токен: " + Token);
                //Console.WriteLine("Тип токена: " + TokenType);
                MessageBox.Show(e.Message, "Ошибка при чтении файла:",
MessageBoxButtons.OK, MessageBoxIcon.Error);
                Environment.Exit(1);
            }
            catch (Exception e)
            {
                MessageBox.Show("Неизвестная ошибка: " + e.Message +
"\nСтрока: " + File.x + "\nСимвол: " + File.y, "Ошибка при чтении файла:",
MessageBoxButtons.OK, MessageBoxIcon.Error);
                //Console.ReadKey(true);
                Environment.Exit(1);
            }

            //Console.WriteLine("Файл успешно прочитан...");
            //Console.ReadKey(true);
            File.Close();
                  return Ruleset;
        }

        // Создание правила
            private void CreateRule()
            {
            GetToken();
            if (TokenType != TokenType.WORD)
                Error("Для имени правила тип токена должен быть WORD.");
            string name = Token;
            Rule rule = new Rule(name);
            GetToken();
            if (TokenType != TokenType.LEFTCURLYBRACE)
                Error("Ожидается {");
            GetToken();
            if (Token != "if")
                Error("Тут должно быть \"if\".");
            //GetToken();
            //while (tokentype != TokenType.SEMICOLON)
```

```
            //{
                rule._if = EvaluateBool();
                //GetToken();
            //}
            rule._then = ReadBlock();
            Ruleset.Rules.Add(rule);
            GetToken();
            if (TokenType != TokenType.RIGHTCURLYBRACE)
                Error("Правило должно заканчиваться символом }.");
            }


        private void GetToken()
        {
            Token = "";
            TokenType = TokenType.UNKNOWN;

            MissSpaces();
            char ch = File.GetChar();

            if (char.IsLetter(ch)) // Это слово?
            {
                TokenType = TokenType.WORD;
                Token += ch;
                ch = File.GetChar();
                while (char.IsLetter(ch) || char.IsNumber(ch) || ch == '_')
                {
                    Token += ch;
                    ch = File.GetChar();
                }
                File.UngetChar(ch);
            }
            else if (char.IsDigit(ch)) // Это число?
            {
                //bool b = false; // Была ли найдена запятая в числе?
                TokenType = TokenType.NUMBER;
                Token += ch;/*
                ch = file.GetChar();
                while (true)
                {
                    if (char.IsDigit(ch))
                    {
                        token += ch;
                    }
                    else if (ch == '.')
                    {
                        if (!b)
                            token += ch;
                        else
                            throw new HReaderException("В числе не может быть
две точки.");
                    }
                    else break;
                    ch = file.GetChar();
                }
                file.UngetChar(ch);*/
                // --- new version ---
                while (char.IsDigit((ch = File.GetChar())))
                    Token += ch;
                if (ch == ',')
                {
                    bool c = false; // была ли найдена хотябы одна цифра
после запятой?
                    Token += ch;
```

```
                    while (char.IsDigit(ch = File.GetChar()))
                    {
                        if (!c)
                            c = true;
                        Token += ch;
                    }
                    if (!c)
                        Error("После запятой должна быть хотябы одна
цифра!");
                }
                if (" +-/*;)".IndexOf(ch) != -1)
                    File.UngetChar(ch);
                else
                    Error("После числа идет неизвестный символ!");
            }
            else if ('"'.Equals(ch)) // Это строка?
            {
                TokenType = TokenType.STRING;
                //Token += ch;
                ch = File.GetChar();
                while (ch != '"')
                {
                    if (ch == '\n')
                    {
                        Error("Строковая переменная не может быть разорвана
между строками текста.");
                    }
                    else if (ch == '\t')
                    {
                        Token += "\t";

                    }
                    else if (ch == '\\')
                    {
                        ch = File.GetChar();
                        switch (ch)
                        {
                            case 'n':
                                Token += '\n';
                                ch = File.GetChar();
                                continue;
                            case 't':
                                Token += '\t';
                                ch = File.GetChar();
                                continue;
                            case '\\':
                                Token += '\\';
                                ch = File.GetChar();
                                continue;
                            default:
                                Error("Доступные управляющие
последовательности: \\t, \\n и \\\\.");
                                break;
                        }
                    }
                    Token += ch;
                    ch = File.GetChar();
                }
                //Token += ch;
            }
            else if ('{'.Equals(ch))
            {
                Token += ch;
                TokenType = TokenType.LEFTCURLYBRACE;
```

```
}
else if ('}'.Equals(ch))
{
    Token += ch;
    TokenType = TokenType.RIGHTCURLYBRACE;
}
else if ('('.Equals(ch))
{
    Token += ch;
    TokenType = TokenType.OPENPARANTHESIS;
}
else if (')'.Equals(ch))
{
    Token += ch;
    TokenType = TokenType.CLOSEPARANTHESIS;
}
else if (';'.Equals(ch))
{
    Token += ch;
    TokenType = TokenType.SEMICOLON;
}
else if (ch == '<' || ch == '>')
{
    Token += ch;
    TokenType = TokenType.LOGIC;
    ch = File.GetChar();
    if (ch == '=')
        Token += ch;
    else
        File.UngetChar(ch);
}
else if (ch == '=')
{
    Token += ch;
    ch = File.GetChar();
    if (ch == '=')
    {
        Token += ch;
        TokenType = TokenType.LOGIC;
    }
    else
    {
        TokenType = TokenType.ASSIGNMENT;
        File.UngetChar(ch);
    }
}
else if ("+-*/".IndexOf(ch) != -1)
{
    Token += ch;
    ch = File.GetChar();
    if (ch == '=')
    {
        Token += ch;
        TokenType = TokenType.ASSIGNMENT;
    }
    else
    {
        TokenType = TokenType.OPERATION;
        File.UngetChar(ch);
    }
}
else if (ch == '&' || ch == '|')
{
    Token += ch;
```

```
                TokenType = TokenType.LOGIC;
            }
            else if (ch == '!')
            {
                Token += ch;
                TokenType = TokenType.LOGIC;
                ch = File.GetChar();
                if (ch == '=')
                    Token += ch;
                else
                    File.UngetChar(ch);
            }
            else if (ch == '\0')
                Error("Неожиданный конец файла.");
            else
            {
                Token += ch;
                TokenType = TokenType.UNKNOWN;
                Error("Неизвестный токен " + Token);
            }
        }

        // Читает блок выражений между { и }
            private ArrayList ReadBlock()
            {
                  GetToken();
                  if (TokenType != TokenType.LEFTCURLYBRACE)
                Error("Ожидается {");
            ArrayList block = new ArrayList(1);
                  GetToken();
                  while (TokenType != TokenType.RIGHTCURLYBRACE)
                  {
                if (Token == "clear")
                {
                    object[] expr = new object[1];
                    expr[0] = OperationType.CLEAR;
                    block.Add(expr);
                    GetToken();
                    if (TokenType != TokenType.SEMICOLON)
                        Error("Ожидается ;");
                }
                else if (Token == "output")
                {
                    object[] expr = new object[2];
                    expr[0] = OperationType.OUTPUT;
                    GetToken();
                    if (TokenType == TokenType.STRING)
                    {
                        //Token = Token.Remove(0, 1); // Удаление начальных
кавычек
                        //Token = Token.Remove(Token.Length - 1, 1); //
Удаление завершающих кавычек
                        expr[1] = Token;
                        block.Add(expr);
                        GetToken();
                        if (TokenType != TokenType.SEMICOLON)
                            Error("Ожидается ;");
                    }
                    else if (TokenType == TokenType.WORD)
                    {
                        if (!Ruleset.Vars.ContainsKey(Token))
                            Error("Переменная " + Token + " не была
объявлена!");
                        expr[1] = Ruleset.Vars[Token];
```

```
                        block.Add(expr);
                        GetToken();
                        if (TokenType != TokenType.SEMICOLON)
                            Error("Ожидается ;");
                    }
                    else if (TokenType == TokenType.SEMICOLON)
                    {
                        expr[1] = "\n";
                        block.Add(expr);
                    }
                    else
                        Error("Недопустимый параметр для команды output!");
                }
                else if (Token == "input")
                {
                    object[] expr = new object[2];
                    expr[0] = OperationType.INPUT;
                    GetToken();
                    if (TokenType != TokenType.WORD)
                        Error("Ожидается токен типа word!");
                    if (!Ruleset.Vars.ContainsKey(Token))
                        Error("Переменная " + Token + " не была объявлена!");
                    expr[1] = Ruleset.Vars[Token];
                    block.Add(expr);
                    GetToken();
                    if (TokenType != TokenType.SEMICOLON)
                        Error("Ожидается ;");
                }
                else if (TokenType == TokenType.WORD) // найдено имя обычной
переменной
                {
                    if (!Ruleset.Vars.ContainsKey(Token))
                        Error("Переменная " + Token + " не была объявлена!");
                    object[] expr = new object[3];
                    expr[1] = Ruleset.Vars[Token];
                    GetToken();
                    //if (tokentype != TokenType.OPERATION)
                    //    Error("Ожидается операция");
                    expr[0] = Token;
                    switch (((Var)expr[1]).Type)
                    {
                        case VarType.NUM:
                            if (TokenType != TokenType.ASSIGNMENT)
                                Error("Ожидается операция присваивания!");
                            expr[2] = EvaluateNum();
                            break;
                        case VarType.BOOL:
                            if (Token != "=")
                                Error("Ожидается операция присваивания!");
                            expr[2] = EvaluateBool();
                            break;
                        case VarType.STR:
                            if (Token != "=" && Token != "+=")
                                Error("Для переменных строкового типа
допустимы операции присваивания '=' и '+='!");
                            expr[2] = EvaluateStr();
                            break;
                        default:
                            Error("Неизвестный тип переменной!");
                            break;
                    }
                    block.Add(expr);
                }
                else // Ошибка
```

```
        {
            Error("Неизвестный символ!");
        }
        GetToken();
          }
          return block;
    }

// строит дерево разбора для логического выражения
private object EvaluateStr()
{
    object result;
    Operation Operation;

    GetToken();
    result = Atom(VarType.STR);

    while (Token == "+")
    {
        //GetToken();
        Operation = new Operation(Token);
        Operation.left = result;
        Operation.right = EvaluateStr();
        result = Operation;
    }

    if (TokenType != TokenType.SEMICOLON)
        Error("Неизвестный символ! Ожидается ;");

    return result;
}

// строит дерево разборя для логического выражения
private object EvaluateBool()
{
    object result;

    GetToken();
    EvalBoolExp2(out result);

    if (TokenType != TokenType.SEMICOLON)
        Error("Неизвестный символ! Ожидается ;");

    return result;
}

// логическое ИЛИ
private void EvalBoolExp2(out object result)
{
    string op;
    object partialResult;
    Operation Operation;

    EvalBoolExp3(out result);
    while ((op = Token) == "|")
    {
        GetToken();
        EvalBoolExp3(out partialResult);
        Operation = new Operation(op);
        Operation.left = result;
        Operation.right = partialResult;
        result = Operation;
    }
}
```

```
// логическое И
private void EvalBoolExp3(out object result)
{
    string op;
    object partialResult;
    Operation Operation;

    EvalBoolExp4(out result);
    while ((op = Token) == "&")
    {
        GetToken();
        EvalBoolExp4(out partialResult);
        Operation = new Operation(op);
        Operation.left = result;
        Operation.right = partialResult;
        result = Operation;
    }
}

// логическое И
private void EvalBoolExp4(out object result)
{
    string op;
    object partialResult;
    Operation Operation;

    EvalBoolExp5(out result);
    if ((op = Token) == "<"
        || op == "<="
        || op == ">"
        || op == ">="
        || op == "=="
        || op == "!="
    )
    {
        GetToken();
        EvalNumExp2(out partialResult);
        Operation = new Operation(op);
        Operation.left = result;
        Operation.right = partialResult;
        result = Operation;
    }
}

// Выполнение операции "отрицание"
private void EvalBoolExp5(out object result)
{
    string op = "";
    Operation Operation;

    if (Token == "!")
    {
        op = Token;
        GetToken();
    }
    EvalBoolExp6(out result);
    if (op == "!")
    {
        Operation = new Operation(op);
        Operation.left = result;
        Operation.right = null;
        result = Operation;
    }
```

```
}

// обработка выражения в круглых скобках
private void EvalBoolExp6(out object result)
{
    if (Token == "(")
    {
        GetToken();
        EvalBoolExp2(out result);
        if (Token != ")")
            Error("Ожидается закрывающая скобка в выражении!");
        GetToken();
    }
    else
    {
        object atom = Atom(VarType.BOOL);
        result = atom;
    }
}

// строит дерево разбора для арифметического выражения
private object EvaluateNum()
{
    object o;

    GetToken();
    EvalNumExp2(out o);

    if (TokenType != TokenType.SEMICOLON)
        Error("Неизвестный символ! Ожидается ;");
    return o;
}

// сложение или вычитание
private void EvalNumExp2(out object result)
{
    string op;
    object partialResult;
    Operation Operation;

    EvalNumExp3(out result);
    while ((op = Token) == "+" || op == "-")
    {
        GetToken();
        EvalNumExp3(out partialResult);
        Operation = new Operation(op);
        Operation.left = result;
        Operation.right = partialResult;
        result = Operation;
    }
}

// умножение или деление
private void EvalNumExp3(out object result)
{
    string op;
    object partialResult;
    Operation Operation;

    EvalNumExp4(out result);
    while ((op = Token) == "*" || op == "/")
    {
        GetToken();
        EvalNumExp4(out partialResult);
```

```
        if (op == "/" && partialResult is double)
            if ((double)partialResult == 0.0)
                Error("Деление на ноль!");
        Operation = new Operation(op);
        Operation.left = result;
        Operation.right = partialResult;
        result = Operation;
    }
}

// выполнение операции унарного "+" или "-"
private void EvalNumExp4(out object result)
{
    string op = "";
    Operation Operation;

    if (Token == "+" || Token == "-")
    {
        op = Token;
        GetToken();
    }
    EvalNumExp5(out result);
    if (op == "-")
    {
        Operation = new Operation(op);
        Operation.left = result;
        Operation.right = null;
        result = Operation;
    }
}

// обработка выражения в круглых скобках
private void EvalNumExp5(out object result)
{
    if (Token == "(")
    {
        GetToken();
        EvalNumExp2(out result);
        if (Token != ")")
            Error("Ожидается закрывающая скобка в выражении!");
        GetToken();
    }
    else
    {
        object atom = Atom(VarType.NUM);
        result = atom;
    }
}

// возвращает число, строку, false, true или переменную
private object Atom(VarType vartype)
{
    object atom = null;
    switch (vartype)
    {
        case VarType.BOOL:
            if (TokenType == TokenType.WORD)
            {
                if (Token == "false")
                    atom = false;
                else if (Token == "true")
                    atom = true;
                else
                {
```

```
                            if (!Ruleset.Vars.ContainsKey(Token))
                                Error("Переменная " + Token + " не была
объявлена!");
                            if (Ruleset.Vars[Token].Type != VarType.BOOL &&
Ruleset.Vars[Token].Type != VarType.NUM)
                                Error("Необходима переменная типа bool или
num");
                            //if (Ruleset.Vars[Token].Value == null)
                              //  atom = false;
                            else
                                atom = Ruleset.Vars[Token];
                        }
                    }
                    else if (TokenType == TokenType.NUMBER)
                    {
                        try
                        {
                            atom = Double.Parse(Token);
                        }
                        catch (FormatException)
                        {
                            Error("Не удалось прочитать число!");
                        }
                    }
                    else
                        Error("Необходимо число или переменная типа bool или
num");
                    break;
                case VarType.NUM:
                    if (TokenType == TokenType.NUMBER)
                    {
                        try
                        {
                            atom = Double.Parse(Token);
                        }
                        catch (FormatException)
                        {
                            Error("Не удалось прочитать число!");
                        }
                    }
                    else if (TokenType == TokenType.WORD)
                    {
                        if (!Ruleset.Vars.ContainsKey(Token))
                            Error("Переменная " + Token + " не была
объявлена!");
                        if (Ruleset.Vars[Token].Type != VarType.NUM)
                            Error("Невозможно привести переменную к типу
num");
                        atom = Ruleset.Vars[Token];
                    }
                    else
                        Error("Необходимо число или имя переменной!");
                    break;
                case VarType.STR:
                    if (TokenType == TokenType.STRING)
                    {
                        atom = Token;
                    }/*
                    else if (TokenType == TokenType.NUMBER)
                    {
                        try
                        {
                            atom = Double.Parse(Token);
                        }
```

```
                        catch (FormatException)
                        {
                            Error("Не удалось прочитать число!");
                        }
                    }*/
                    else if (TokenType == TokenType.WORD)
                    {/*
                        if (Token == "false")
                            atom = false;
                        else if (Token == "true")
                            atom = true;
                        else
                        {*/
                        if (!Ruleset.Vars.ContainsKey(Token))
                            Error("Переменная " + Token + " не была
объявлена!");
                        if (Ruleset.Vars[Token].Type != VarType.STR)
                            Error("Переменная должна иметь тип str!");
                        atom = Ruleset.Vars[Token];
                        //}
                    }
                    else return null;
                    break;
                default:
                    atom = 0.0;
                    Error("Не удалось получить значение числа, имени
переменной или строку символов!");
                    break;
            }
            GetToken();
            return atom;
        }

        private void CreateVariable(VarType Type)
        {
            bool labelfound = false; // Было ли найдено поле "label". В
переменной поля не могут повторятся несколько раз
            bool findfound = false;  // Было ли найдено поле "find"
            GetToken(); // Получаем имя переменной
            if ((TokenType != TokenType.WORD) || Token == "false" || Token ==
"true")
                Error("Недопустимое имя переменной!");
            string name = Token; // Сохраняет имя переменной для дальнейшего
использования
            Var var = new Var(Type, name); // Создает новую переменную
            Ruleset.AddVar(var, name);
            GetToken();
            if (TokenType == TokenType.LEFTCURLYBRACE)
            {
                GetToken();
                while (TokenType != TokenType.RIGHTCURLYBRACE)
                {
                    if (TokenType != TokenType.WORD)
                        Error("Ожидается токен типа WORD.");
                    if (Token == "label")
                    {
                        if (labelfound == false)
                        {
                            labelfound = true;
                            GetToken();
                            if (TokenType != TokenType.STRING)
                                Error("Ожидается токен типа STRING.");
                            //Token = Token.Remove(0, 1); // Удаление
начальных кавычек
```

```
                            //Token = Token.Remove(Token.Length - 1, 1); //
Удаление завершающих кавычек
                            var.Label = Token;
                            GetToken();
                            if (TokenType != TokenType.SEMICOLON)
                                Error("Ожидается токен типа SEMICOLON.");
                        }
                        else
                        {
                            Error("Слово \"label\" должно появляться в
описании переменной только один раз.");
                        }
                    }
                    else if (Token == "find")
                    {
                        if (findfound == false)
                        {
                            findfound = true;
                            Ruleset.Vars[name].Find = ReadBlock();
                        }
                        else
                        {
                            Error("Слово \"find\" должно появляться в
описании переменной только один раз.");
                        }
                    }
                    else
                    {
                        Error("Неизвестный идентификатор.");
                    }
                    GetToken();
                }
            }
            else if (TokenType != TokenType.SEMICOLON)
                Error("Ожидается точка с запятой или открытая фигурная
скобка.");
        }

        private void MissSpaces() //FIXME: Добавить проверку на неожиданный
конец файла
        {
            char ch;
            while (true)
            {
                while (char.IsWhiteSpace(ch = File.GetChar())) // Удаление
пробелов, табов, переносов строки и т.д.
                    ;

                if (ch == '/') // Удаление комментариев
                {
                    ch = File.GetChar();
                    if (ch == '/') // Комментарий в стиле C++
                    {
                        while ((ch = File.GetChar()) != '\n')
                            ; // Пропуск пустых символов до конца строки
                    }
                    else if (ch == '*') // Комментарий в стиле C
                    {
                                    while (true)
                                    {
                            while ((ch = File.GetChar()) != '*')
                                ; // Найти первую "звездочку"
                                        ch = File.GetChar();
                                        if (ch == '/')
```

```
                                                    break;  // Найден конец
многострочного комментария
                                                // Выйти из цикла и перейти к новой
итерации
                                continue;   // Иначе продолжить поиск конца
многострочного комментария
                                        }
                                        continue; // Начать сначала цикл поиска
"пустых" символов и комментариев
                    }
                    else
                    {
                        File.UngetChar(ch);
                        File.UngetChar('/');
                        break;
                    }
                }
                else // Возврат последнего символа если он не является
"пустым" или не слэш
                {
                    File.UngetChar(ch);
                    break;
                }

                /*
                if (ch == '\0')
                                                        throw new
HReaderException("Неожиданный конец файла.");
                */
            }
        }

        private void Error(string error)
        {
            throw new HReaderException("Строка: " + File.x + "\nСимвол: " +
File.y + "\nОшибка: " + error);
        }
    }
    public class Rule
    {
        public string Name;
        public object _if;
        public ArrayList _then;

        // true если правило уже выполнялось, иначе false
        public bool Executed;

        public Rule(string Name)
        {
            this.Name = Name;
        }
    }
    class HRulesetException : ApplicationException
    {
        public HRulesetException() : base() { }
        public HRulesetException(string s) : base(s) { }
        public override string ToString()
        {
            return Message;
        }
    }

    public class Ruleset
    {
```

```
        public string Name;   // Имя набора правил
        public ArrayList Initial;
        public ArrayList End;
        public string Goal;   // Имя цели
        public Dictionary<string, Var> Vars;
        //public Dictionary<string, Rule> Rules;
        public ArrayList Rules;

        public Ruleset(string name)
        {
            Name = name;
            Vars = new Dictionary<string, Var>();
            Rules = new ArrayList();
            //Rules = new Dictionary<string, Rule>();
#if _DEBUG_
            Console.WriteLine("Экземпляр класса для набора правил создан.");
#endif
        }

        public void AddVar(Var var, string Name)
        {
            Vars.Add(Name, var);
        }

        public void AddRule(Rule rule)
        {
            Rules.Add(rule);
        }
    }
    public class Var
    {
        public object Value; // значение переменной
        public VarType Type; // тип переменной
        public string name;
        public string Label; // описание переменной
        public ArrayList Find;  //TODO:

        public Var(VarType Type, string name)
        {
            this.Type = Type;
            Value = null;
            /*switch (this.Type)
            {
                case VarType.BOOL: Value = false; break;
                case VarType.NUM:  Value = 0;     break;
                case VarType.STR:  Value = "";    break;
                default: break;
            }*/
            this.name = name;
        }
    }
    class Operation
    {
        public Operation(string op)
        {
            this.op = op;
        }

        public string op;
        public object right;
        public object left;
    }
    class HFileException : ApplicationException
    {
```

```
        public HFileException() : base() { }
        public HFileException(string s) : base(s) { }
        public override string ToString()
        {
            return Message;
        }
    }

    class HFile
    {
        private StreamReader file;
        private char[] buf; // буфер
        private int pos; // указатель на позицию в буфере
        private int count; // кол-во считаных символов
        public int x, y; // номер строки и символа в файле

        public HFile(string PathToFile)
        {
            file = new StreamReader(PathToFile,
System.Text.Encoding.Default);
            //Console.WriteLine("Файл открыт!...");
                  //Console.ReadKey(true);
            buf = new char[5];
            pos = count = 0;
            y = x = 1;
        }

        public char GetChar()
        {
            if (count == 0)
            {
                count = file.Read(buf, 2, 3);
                if (count == 0) // конец файла
                    return '\0';
                pos = 2;
            }
            count--;
            if (buf[pos] == '\n')
            {
                x++; y = 1;
            }
            else
                y++;
            pos++;
            return buf[pos-1]; //TODO: return buf[pos++];
        }

        public void UngetChar(char ch)
        {
            if (count == 5)
                throw new HFileException("В буфере нет места для возвращения
символа!");
            if (ch == '\n')
            {
                x--; y = 1;
            }
            else
            {
                y--;
            }
            buf[--pos] = ch;
            count++;
        }
```

```
        public void Close()
        {
            file.Close();
        }
    }
    public partial class InputBoolVarForm : Form
    {
        public InputBoolVarForm()
        {
            InitializeComponent();
        }

        private void trueButton_Click(object sender, EventArgs e)
        {
            DialogResult = DialogResult.Yes;
            this.Hide();
        }

        private void falseButton_Click(object sender, EventArgs e)
        {
            DialogResult = DialogResult.No;
            this.Hide();
        }
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be
disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.trueButton = new System.Windows.Forms.Button();
            this.falseButton = new System.Windows.Forms.Button();
            this.label1 = new System.Windows.Forms.Label();
            this.SuspendLayout();
            //
            // trueButton
            //
            this.trueButton.Location = new System.Drawing.Point(12, 29);
            this.trueButton.Name = "trueButton";
            this.trueButton.Size = new System.Drawing.Size(75, 23);
            this.trueButton.TabIndex = 0;
            this.trueButton.Text = "Да";
            this.trueButton.UseVisualStyleBackColor = true;
```

```
            this.trueButton.Click += new
System.EventHandler(this.trueButton_Click);
            //
            // falseButton
            //
            this.falseButton.DialogResult =
System.Windows.Forms.DialogResult.Cancel;
            this.falseButton.Location = new System.Drawing.Point(93, 29);
            this.falseButton.Name = "falseButton";
            this.falseButton.Size = new System.Drawing.Size(75, 23);
            this.falseButton.TabIndex = 1;
            this.falseButton.Text = "Нет";
            this.falseButton.UseVisualStyleBackColor = true;
            this.falseButton.Click += new
System.EventHandler(this.falseButton_Click);
            //
            // label1
            //
            this.label1.AutoSize = true;
            this.label1.Location = new System.Drawing.Point(13, 13);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(123, 13);
            this.label1.TabIndex = 2;
            this.label1.Text = "Значение переменной:";
            //
            // InputBoolVarForm
            //
            this.AcceptButton = this.trueButton;
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.CancelButton = this.falseButton;
            this.ClientSize = new System.Drawing.Size(216, 63);
            this.Controls.Add(this.label1);
            this.Controls.Add(this.falseButton);
            this.Controls.Add(this.trueButton);
            this.MaximizeBox = false;
            this.MinimizeBox = false;
            this.Name = "InputBoolVarForm";
            this.Text = "InputBoolVarForm";
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion

        private System.Windows.Forms.Button trueButton;
        private System.Windows.Forms.Button falseButton;
        private System.Windows.Forms.Label label1;
    }
    public partial class InputNumVarForm : Form
    {
        public InputNumVarForm()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            this.Hide();
        }
        /// <summary>
        /// Required designer variable.
        /// </summary>
```

```
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be
disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.textBox = new System.Windows.Forms.TextBox();
            this.nameLabel = new System.Windows.Forms.Label();
            this.doneButton = new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            // textBox
            //
            this.textBox.Location = new System.Drawing.Point(16, 29);
            this.textBox.Name = "textBox";
            this.textBox.Size = new System.Drawing.Size(238, 20);
            this.textBox.TabIndex = 0;
            //
            // nameLabel
            //
            this.nameLabel.AutoSize = true;
            this.nameLabel.Location = new System.Drawing.Point(13, 13);
            this.nameLabel.Name = "nameLabel";
            this.nameLabel.Size = new System.Drawing.Size(123, 13);
            this.nameLabel.TabIndex = 1;
            this.nameLabel.Text = "Значение переменной:";
            //
            // doneButton
            //
            this.doneButton.Location = new System.Drawing.Point(16, 55);
            this.doneButton.Name = "doneButton";
            this.doneButton.Size = new System.Drawing.Size(75, 23);
            this.doneButton.TabIndex = 2;
            this.doneButton.Text = "Готово";
            this.doneButton.UseVisualStyleBackColor = true;
            this.doneButton.Click += new
System.EventHandler(this.button1_Click);
            //
            // InputNumVarForm
            //
            this.AcceptButton = this.doneButton;
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(270, 89);
            this.Controls.Add(this.doneButton);
            this.Controls.Add(this.nameLabel);
```

```
            this.Controls.Add(this.textBox);
            this.MaximizeBox = false;
            this.MinimizeBox = false;
            this.Name = "InputNumVarForm";
            this.Text = "InputNumVarForm";
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion

        private System.Windows.Forms.Label nameLabel;
        private System.Windows.Forms.Button doneButton;
        public System.Windows.Forms.TextBox textBox;
    }
    public partial class InputStrVarForm : Form
    {
        public InputStrVarForm()
        {
            InitializeComponent();
        }

        private void doneButton_Click(object sender, EventArgs e)
        {
            this.Hide();
        }
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be
disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.richTextBox = new System.Windows.Forms.RichTextBox();
            this.label1 = new System.Windows.Forms.Label();
            this.doneButton = new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            // richTextBox
            //
            this.richTextBox.Location = new System.Drawing.Point(15, 25);
            this.richTextBox.Name = "richTextBox";
            this.richTextBox.Size = new System.Drawing.Size(265, 142);
```

```
            this.richTextBox.TabIndex = 0;
            this.richTextBox.Text = "";
            //
            // label1
            //
            this.label1.AutoSize = true;
            this.label1.Location = new System.Drawing.Point(12, 9);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(105, 13);
            this.label1.TabIndex = 1;
            this.label1.Text = "Текст переменной:";
            //
            // doneButton
            //
            this.doneButton.Location = new System.Drawing.Point(13, 174);
            this.doneButton.Name = "doneButton";
            this.doneButton.Size = new System.Drawing.Size(75, 23);
            this.doneButton.TabIndex = 2;
            this.doneButton.Text = "Готово";
            this.doneButton.UseVisualStyleBackColor = true;
            this.doneButton.Click += new
System.EventHandler(this.doneButton_Click);
            //
            // InputStrVarForm
            //
            this.AcceptButton = this.doneButton;
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(292, 209);
            this.Controls.Add(this.doneButton);
            this.Controls.Add(this.label1);
            this.Controls.Add(this.richTextBox);
            this.MaximizeBox = false;
            this.MinimizeBox = false;
            this.Name = "InputStrVarForm";
            this.Text = "InputStrVarForm";
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion

        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Button doneButton;
        public System.Windows.Forms.RichTextBox richTextBox;
    }
}
```

Приложение 2. Пример эвристической модели.

Структура файла с набором правил:

Весь файл состоит из блоков. Блоки выделяются фигурными скобками. Сначала идет блок ruleset. После слова ruleset должно обязательно идти слово, которое будет именем данного набора правил, и открытая фигурная скобка. Например так:

```
ruleset testing
{
```

Дальше должен идти блок vars. Внутри него идет объявление всех используемых переменных.

Определение переменной состоит из указания типа переменной (возможные значения: num, bool, str), имени. Дальше идет либо точка с запятой, либо открывающая фигурная скобка, внутри которой можно прописать поля label, в котором находится пояснение к переменной, и/или find – блок команд с помощью которых машина сможет получить значение переменной, если она встретиться в программе и будет неопределена. Пример объявления переменных:

```
vars
{
	num x;
	bool b;
	str s
	{
		label “Просто строка!”
		find
		{
			output “Введите значение переменной s”;
			input s;
		}
	}

}
```

Дальше идет слово goal и имя переменной-цели, которой будет стремиться достичь машина.

```
goal x;
```

Дальше идет блок initial, состоящий из команд, которые необходимо выполнить перед тем как машина начнет просматривать правила.

```
initial
{
	...
}
```

Дальше идет блок end, также состоящий из набора команд, но которые необходимо выполнить при завершении работы.

```
end
{
	...
}
```

Дальше следует какое угодно количество правил. Правила строятся по следующим правилам.
Сначала идет слово rule, свидетельствующие о том, что сейчас будет описано правило. Дальше имя правила и открывающая фигурная скобка, внутри находятся два поля.
Первое поле – if. В поле if прописывается условие выполнения данного правила. if заканчивается точкой с запятой.
Второе поле – поле с набором команд, которые будут выполнены, если условие окажется истинным.
Пример:

```
rule R1
{
```

```
        if b;
        {
              x = 100;
              output “100!”;
        }
}
```

Программа заканчивается закрывающей фигурной скобкой, которая соответствует скобке в начале программы

```
}
```

Типы переменных:

num – число с плавающей точкой. Соответствует типу double в C#.
bool – тип для логической переменной. Соответствует типу bool в C#.
str – строка. Соответствует типу string в C#. В строках допускается использование escase-последовательностей \n, \t и \\. Константы типа str не могут быть разорваны между строками в исходном файле.

Строение команд:

Команды бывают четырех типов

1) Команда output [имя_переменной или константа];
   Выводит на экран значение переменной или константы.
   Пример: output “Hello, World!\n”;
2) Команда input имя_переменной;
   Выводит на экран окно для ввода значения переменной.
   Пример: input x;
3) Команда clear;
   Очищает окно сообщений.
4) Присваивание
   Присваивает значение стоящее справа от знака равно (=, +=, -=, *=, /=) переменной стоящей слева от знака. При несовпадении типов, ещё на этапе чтения файла, выдается ошибка.
   Для переменных типа num доступны все типы присваивания.
   Для строковых переменных только: ‘=’ и ‘+=’
   Для логических только: ‘=’

Операторы языка:

Арифметические операторы:
+ - * /
Логические операторы:
| & !
Операторы сравнения:
== != < > <= >=
Также допустимо использование скобок для явного указания приоритета.
Арифметические и логические операторы имеют тот же приоритет, что и в других языках, в частности C#.

Исходный текст демонстрационного примера.

```
ruleset name
{
	vars
	{
		str reasons;
		/*{
			find
			{
				reasons = "Не известная причина!";
			}
		}*/

		bool gunsdef
		{
			label "наличие дефектов пистолета-краскопульта";
		}

		bool drops;

		bool harddrops
		{
			label "";
			find
			{
				output " наблюдаются резкие толчки и пульсация ";
				output " струи (y/n)? ";
				input harddrops;
			}
		}

		bool notkrask
		{
			label "подача краски в пистолет.";
			find
			{
				output " краска совсем не подается в пистолет (y/n)?";
				input notkrask;
			}
		}

		bool badklap
		{
			label "негоден клапан пистолета.";
			find
			{
				output "kpacka проходит при закрытом клапане(y/n)?";
				input badklap;
			}
		}

		str ways
		{
			label "способ устранения неполадок.";
			find
			{
				reasons="причина появления этих неполадок мне " ;
				reasons=reasons+"неизвестна.";
				ways="попробуйте обратиться в мастерскую.";
			}
		}

		bool lowfluct;
```

```
      bool highfluct;
}

goal ways;

initial
{
      clear;
      output "День \t\tдобрый, \tмистер (миссис).\\\n";
      output "вы решились окрасить какую-то поверхность при помощи";
      output "пистолета-краскопульта. а ваш пистолетик что-то";
      output "ненормально работает. мы постараемся дать вам совет, ";
      output "как устранить некоторые неисправности и сообщим вам ";
      output "причины их появления.";
      output "но для этого вы должны предоставить мне всю";
      output "информацию.";
      output "итак, поехали...";
      gunsdef=true;
      output "при работе пистолета появились какие-то дефекты (y/n)?";
      input gunsdef;
}

end
{
      clear;
      output;
      output " вот что я вам скажу :";
      output;
      output reasons;
      output;
      output "а вот что вам следует сделать в данной ситуации.";
      output ways;
}

rule r1
{
      if gunsdef;
      {
            output " что-то неладное со струей краски (y/n)? ";
            input drops;
      }
}

rule r2
{
      if !gunsdef;
      {
            output;
            reasons="нет дефектов, значит нечего волноваться зря.";
            ways="пойдите лучше отдохните, ведь у вас все в ";
            ways=ways+"порядке.";
      }
}

rule r3
{
      if notkrask;
      {
            reasons=" ЗАСОРЕНА СЕТКА РЕЗЕРВУАРА ДЛЯ КРАСКИ ИЛИ";
            reasons=reasons+"НЕДОСТАТОЧНЫЙ УРОВЕНЬ КРАСКИ В ";
            reasons=reasons+"БАКЕ. ";
            ways="В ПЕРВОМ СЛУЧАЕ НАДО ПРОЧИСТИТЬ СЕТКУ, А ВО ";
            ways=ways+"ВТОРОМ ДОБАВИТЬ КРАСКУ В КРАСКОНАГНЕТА";
            ways=ways+"ТЕЛЬНЫЙ БАЧОК.";
```

```
	}
}

rule r4
{
	if drops;
	{
		output;
		output "слабый распыл струи?";
		input lowfluct;
	}
}

rule r5
{
	if drops & !lowfluct;
	{
		output;
		output " слишком разбросанная струя, сильное ";
		output "туманообразование?";
		input highfluct;
	}
}

rule r6
{
	if drops & harddrops;
	{
		reasons=" такой эффект возможен только в трех ";
		reasons=reasons+"случаях: слабое давление на " ;
		reasons=reasons+"краску, слишком густая краска или ";
		reasons=reasons+"неплотно прижат воздухопровод." ;
		ways=" соответственно необходимо: отрегулировать ";
		ways=ways+"давленйе, разбавить краску растворителем,";
		ways=ways+"прижать и закрепить воздухопровод.";
	}
}

rule r7
{
	if drops & lowfluct;
	{
		reasons="такой эффект возможен только в двух случаях:";
		reasons=reasons+"мало воздуха или загрязнено отверстие" ;
		reasons=reasons+" сопла.";
		ways=" соответственно необходимо : проверить и устра" ;
		ways=ways+"нить места утечки воздуха, прочистить сопло." ;
	}
}

rule r8
{
	if drops & highfluct;
	{
		reasons="takoh эффект возможен только в трех случаях:" ;
		reasons=reasons+" загрязнен кончик иглы, сработалось" ;
		reasons=reasons+" сальниковое уплотнение иглы " ;
		reasons=reasons+"излишняя подача воздуха или недос" ;
		reasons=reasons+"tatочная подача краски." ;
		ways=" соответственно необходимо: очистить кончик " ;
		ways=ways+"иглы, подтянуть гайку сальникового " ;
		ways=ways+"уплотнения, отрегулировать подачу воздуха" ;
		ways=ways+"и краски.";
	}
```

```
        }

        rule r9
        {
              if !drops & badklap;
              {
                    reasons=" такой эффект возможен только в двух " ;
                    reasons=reasons+"случаях: сработалась игла сопла," ;
                    reasons=reasons+"игла he закрывает отверстия сопла.";
                    ways=" соответственно необходимо: притереть или " ;
                    ways=ways+"сменить иглу, выдвинуть иглу путем " ;
                    ways=ways+"отвинчивания регулировочных гаек." ;
              }
        }
}
```

Список литературы.

1. Павловская Т.А. C#. Программирование на языке высокого уровня. Изд.Питер, С-Пб, 2008.
2. Фролов А.В. Язык C#. Самоучитель. М. Диалог-МИФИ, 2005.
3. Г.Шилдт. Полный справочник по C#. М., С-Петербург. Киев. Издательский дом “Вильямс”, 2004.
4. Кириченко А.А. «Комплексирование программных средств: использование в C#-программах системы команд и программ ОС Windows. М. ООО «ИПЦ Маска», 2010. ISBN 978-5-91146-546-9
5. Кириченко А.А. «Архитектурные особенности различных видов программирования», М. ООО «ИПЦ Маска», 2012. ISBN 978-5-91146-764-7
6. Д.Прайс, М.Гандерлой. Visual C#.NET. Полное руководство. М. Энтроп, С-Петербург. КОРОНАпринт, Киев. ВЕК, 2004.
7. В.Фаронов Создание приложений с помощью C#. Руководство программиста. Изд. ЭКСМО, М., 2008.
8. Кириченко А.А. «Нейропакеты - современный интеллектуальный инструмент исследователя», сетевое электронное учебное пособие, ISBN 978-5-9904911-1-3 http://www.hse.ru/pubs/share/direct/document/91940629
9. Кириченко А.А. «О непреднамеренном нарушении принципа инкапсуляции при разработке объектно-ориентированных программ». Материалы международной научно-практической конференции «Инновационные информационные технологии в образовании I^2T», Прага, 2013. ISSN 2303-9728.
10. Кириченко А. А. К вопросу об архитектуре нейросетевых пакетов // В кн.: Инновации на основе информационных и коммуникационных технологий: Материалы международной научно-практической конференции (2013) / Отв. ред.: И. А. Иванов; под общ. ред.: С. У. Увайсов; науч. ред.:А. Н. Тихонов. М. : МИЭМ НИУ ВШЭ, 2013. С. 298-300.
11. Кириченко А.А. «Администрирование компьютерных сетей», сетевое электронное учебное пособие, 2014, Ил.207, табл. 58. ISBN 978-5-9904911-2-0
12. Кириченко А.А. «Технологии экстремального программирования», 2014. Сетевое электронное издание монографии. 161 страница, формат PDF. ISBN 978-5-9904911-3-7
13. А.П.Пятибратов, Л.П.Гудыно, А.А.Кириченко Вычислительные системы, сети и телекоммуникацию изд Финансы и статистика, М., 2008. С. 268-274 и 351-365.
14. http://ai-center.botik.ru/planning/index.php?ptl=materials/gps-overview.htm
15. http://www.ershov.ras.ru/archive/eaimage.asp?lang=1&did=8230&fileid=103548
16. Методы добычи данных. Internet-ресурс – http://www.statsoft.ru .

Сетевое электронное издание учебного пособия.

ISBN 978-5-9904911-4-4

Кириченко А.А.
профессор департамента «Программная инженерия» факультета компьютерных наук Федерального государственного автономного образовательного учреждения высшего профессионального образования “Национальный исследовательский университет "Высшая школа экономики" при правительстве РФ”.

«Объектно-ориентированное программирование на алгоритмическом языке C#»

Рецензент: к.т.н., профессор Фомин Г.П.
(кафедра математических методов в экономике РЭУ им. Г.В.Плеханова)

Редактор Альбицкая Н.Б.

Издательство "Высшая школа экономики".