БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра технологии программирования

Ж. В. Василенко

ОСНОВЫ ЯЗЫКА C#

Учебно-методическое пособие
для студентов специальностей
1-31 03 03 «Прикладная математика (по направлениям)»,
1-31 03 04 «Информатика»,
1-31 03 05 «Актуарная математика»,
1-31 03 06 «Экономическая кибернетика (по направлениям)»

МИНСК
2008

УДК 004.43 (075.8)
ББК 32.973.26 – 018.1я73
В19

Утверждено на заседании
кафедры технологий программирования БГУ
20 ноября 2008 г., протокол № 4

Василенко, Ж. В.

В19 Основы языка C#: пособие для студентов спец. 1-31 03 03 «Прикладная математика (по направлениям)», 1-31 03 04 «Информатика», 1-31 03 05 «Актуарная математика», 1-31 03 06 «Экономическая кибернетика (по направлениям)» / Ж. В. Василенко. – Минск : БГУ, 2008. – 50 с.

В пособии рассматривается среда программирования, платформа .NET, распределение и управление памятью, динамическое определение типов, основы синтаксиса и конструкций языка C#.

Пособие предназначено для студентов БГУ, обучающихся по специальностям «Прикладная математика», «Информатика», «Актуарная математика», «Экономическая кибернетика».

УДК 004.43 (075.8)
ББК 32.973.26-018.1я73

ВВЕДЕНИЕ

В середине 2000 года корпорация *Microsoft* представила новую модель для создания приложений, основой которой является платформа .NET[1]. Платформа .NET образует каркас, который включает **технологии разработки** *Windows*-приложений, *Web*-приложений и *Web*-сервисов, **технологии доступа к данным** и **межпрограммного взаимодействия**, **технологии межъязыкового взаимодействия** (*cross-language interoperability*) программных и аппаратных изделий разных поставщиков, или **многоязыковое программирование** (*mixed-language programming*).

В состав платформы входит обширная библиотека классов. Основным инструментом для разработки является интегрированная среда *MS Visual Studio*.

Платформа .NET позволяет с легкостью создавать и интегрировать приложения, написанные на различных языках программирования. Специально для .NET был разработан язык программирования C#. Этот язык сочетает простой синтаксис, похожий на синтаксис языков C++ и *Java*, и полную поддержку всех современных объектно-ориентированных концепций и подходов. В качестве ориентира при разработке языка было выбрано безопасное программирование, нацеленное на создание надежного, простого в сопровождении кода.

Цель данного курса – рассмотреть программирование для платформы .NET с использованием языка программирования C#.

Пособие содержит фрагменты кода и небольшие программы, иллюстрирующие теоретический материал. Примеры могут служить основой при написании лабораторных работ, связанных с объектно-ориентированным программированием с использованием C#.

[1] Произносится как «дот-нэт».

ГЛАВА 1. ПЛАТФОРМА .NET

1.1. ПЛАТФОРМА .NET – ОБЗОР АРХИТЕКТУРЫ

1.1.1. Общая структура *.NET Framework*

Задача ***платформы .NET*** (*.NET Framework*) – предоставить программистам более эффективную и гибкую среду разработки традиционных и Web-приложений. Одна из наиболее важных особенностей *.NET Framework* – способность обеспечить совместную работу кода, написанного на различных языках программирования.

Базой платформы является **общеязыковая среда исполнения** (*Common Language Runtime*, *CLR*). CLR является «прослойкой» между операционной системой и кодом приложений для .NET Framework. Такой код называется *управляемым* (*managed code*). Среда выполнения программ CLR реализует управление памятью, типами данных, межъязыковым взаимодействием, разворачиванием (*deployment*) приложений.

В состав платформы .NET входит библиотека классов *Framework Class Library* (*FCL*). Элементом этой библиотеки является базовый набор классов *Base Class Library* (*BCL*). В BCL входят классы для работы со строками, коллекциями данных, поддержки многопоточности и множество других классов. Частью FCL являются компоненты, поддерживающие различные технологии обработки данных и организации взаимодействия с пользователем. Это классы для работы с XML, базами данных (ADO.NET), создания Windows-приложений и Web-приложений (ASP.NET).

Кроме упомянутых элементов, выделяют еще 2 части платформы .NET:

- *Common Language Specification* (CLS) – набор правил для языка программирования. Соблюдение этих правил обеспечивает создание на разных языках программ, легко взаимодействующих между собой. CLS–спецификация включает в себя подмножество системы типов данных *Common Type System* (CTS).
- CTS-система определяет правила в отношении типов данных. CTS–спецификация содержит базовые, не зависящие от языка программирования (ЯП) примитивные типы, которыми может манипулировать CLR (общеязыковая среда исполнения *Common Language Runtime*).

C# поддерживает как CLS, так и CTS-спецификации.

В стандартную поставку .NET Framework включены компиляторы для платформы. Это компиляторы языков *C#, Visual Basic.NET, J#.* Благодаря открытым спецификациям компиляторы для .NET предлагаются различными сторонними производителями. На данный момент количество компиляторов измеряется десятками.

1.1.2. *CLR Common Language Runtime* (Общеязыковая среда исполнения)

Рассмотрим подробнее компоненты и роль CLR.

Рис. 1. Схема компиляции в CLR

В ходе выполнения процедуры трансляции исходный текст программы (написанный на C#, Visual Basic, C++ или любом из множества других языков программирования, который поддерживается .NET) преобразуется компилятором в так называемую сборку (*assembly*) и сохраняется в виде файла динамически присоединяемой библиотеки (*Dynamically Linked Library, DLL*) или исполняемого файла (*Executable, EXE*).

В итоге программный проект формируется в виде сборки – самодостаточного компонента для разворачивания, тиражирования и повторного использования.

Сборка состоит из следующих частей:

1. *Манифест* (*manifest*) – описание сборки: версия, ограничения безопасности, список внешних сборок и файлов, необходимых для работы данной сборки.

2. *Метаданные* (*metadata*) – специальное описание всех пользовательских типов данных, размещенных в сборке.

3. *Код на промежуточном языке Microsoft Intermediate Language* (*MSIL* или просто *IL*). Данный код является независимым от операционной системы и типа процессора, на котором будет выполняться приложение. В процессе работы приложения он компилируется в машинно-зависимый код специальным компилятором (*Just-in-Time compiler*, *JIT compiler (точно к нужному моменту)*). Естественно, что для каждого

компилятора (будь то компилятор языка C#, csc.exe или Visual Basic, vbc.exe) средой времени выполнения производится необходимое отображение используемых типов в типы CTS, а программного кода – в код «абстрактной машины» .NET – MSIL (*Microsoft Intermediate Language*).

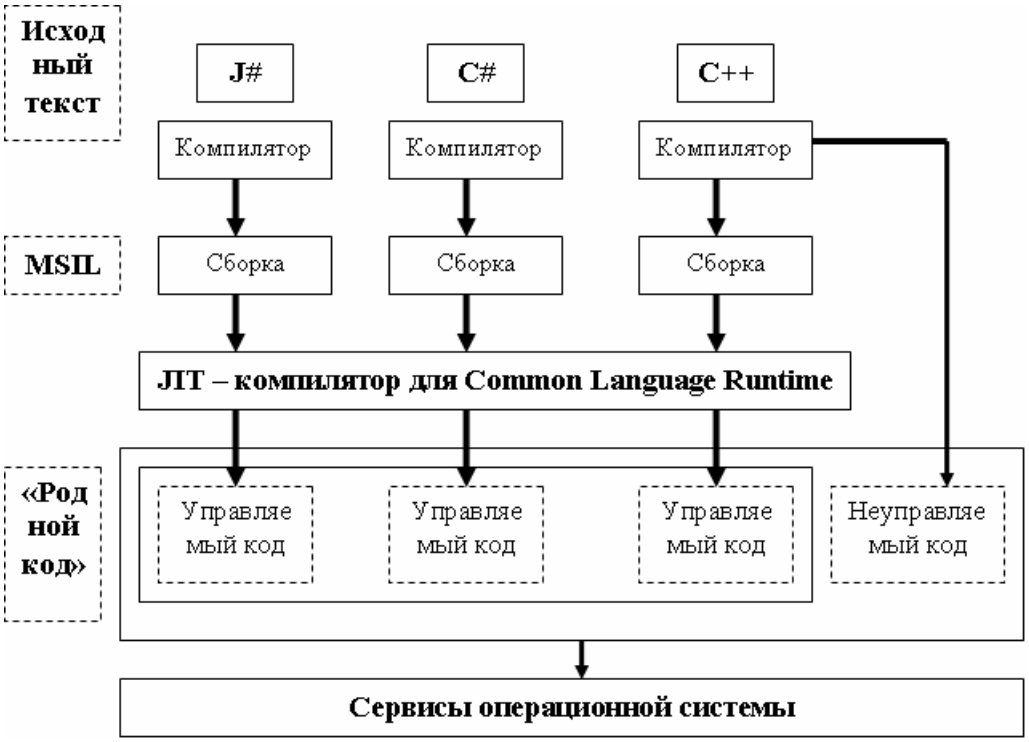


Рис. 2. Схема выполнения CLR

Предположим, что компоненты проекта написаны на трех языках программирования: C#, J#, а также C++, который характеризуется возможностью написания потенциально небезопасного кода (в частности, динамического распределения памяти).

Исходные тексты компонент проекта транслируются соответственно компиляторами с языков C#, J# и C++ в унифицированный MSIL-код и сохраняются в файлах в виде сборок.

В ходе компоновки и выполнения программного проекта *Just-In-Time (JIT)* компилятор среды CLR производит выполнение проекта с ленивым (по мере необходимости) присоединением оттранслированного промежуточного кода сборок.

Откомпилированные из IL платформенно-зависимые инструкции JIT помещает в кэш-памяти, что очень сильно ускоряет работу приложения. При первом вызове метода JIT откомпилирует относящийся к этому методу код IL в платформенно-зависимые инструкции. При ПОВТОР-

НЫХ вызовах этого метода JIT уже не будет заниматься компиляцией, а просто возьмет готовый откомпилированный код из кэша в оперативной памяти.

Существенно, что потенциально небезопасный код на языке C++ принципиально невыполним собственно JIT-компилятором, но исполняется посредством сервисов операционной системы. Ответственность за работоспособность программы и безопасность кода в этом случае лежит уже не на среде проектирования и разработки программного обеспечения .NET, а на программисте-разработчике.

Итак, основная задача CLR – это манипулирование сборками: загрузка сборок, трансляция кода IL в машинно-зависимый код, создание окружения для выполнения сборок. Важной функцией CLR является управление размещением памяти при работе приложения и выполнение *автоматической сборки мусора*, то есть фонового освобождения неиспользуемой памяти. Кроме этого, CLR реализует в приложениях для .NET верификацию типов, управление политиками безопасности при доступе к коду и некоторые другие функции.

1.2. ПРИМЕРЫ

Пример 1.1. Программа «Hello, World» традиционно используется для первого знакомства с языком программирования. Вот пример этой программы на языке C#.

```
// Вывести на экран строку “Hello, world”
using System;
// Объявляем, что мы работаем в пространстве имен Ex1_1
namespace Ex1_1
{
   class Program
   {
      public static void Main(string[] args)
      {
         Console.WriteLine("Hello, world");
      }
   }
}
```

Дадим некоторые пояснения.

Программа представляет собой описание пользовательского типа – класса Program.

Любая исполняемая программа на C# должна иметь специальную *точку входа*, с которой начинается выполнение приложения. Такой точкой входа является статический метод Main(), объявленный в некотором классе программы (в данном случае – в классе Program).

Метод Main() был определен как public и как static. Ключевое слово public в определении метода означает, что этот метод будет доступен извне, а ключевое слово static говорит о том, что этот метод позиционируется на уровне класса, а не отдельного объекта и будет доступен даже тогда, когда еще не создано ни одного экземпляра объекта данного класса.

Метод Main() принимает единственный параметр, который должен быть набором символов (string[] args). Его можно использовать для приема параметров командной строки при запуске приложения.

Метод Main() содержит вызов метода WriteLine() класса Console из пространства имен System. Ключевое слово using служит для подключения пространства имен System, содержащего базовые классы. Использование using позволяет вместо полного имени класса System.Console записать короткое имя Console.

Если программа содержится в файле hello.cs, то после компиляции будет получена сборка hello.exe.

В примерах приложений для вывода информации используются методы WriteLine() и Write() класса Console. Ввод данных осуществляется функцией Console.ReadLine(). Функция возвращает введенную строку, которая обычно преобразуется в значение требуемого типа.

Пример 1.2.

```
// Сложить 2 целых числа
namespace Ex1_2
{
   using System;
   // В классе Calc определен метод Add
   class Calc
   {
      public int Add(int x, int y)
      {
         return x + y;
      }
```

```
    }
    // В классе Program определена точка входа
    // приложения - метод Main()
    class Program
    {
        public static void Main(string[] args)
        {
            // Создаем объект Calc, складываем два числа, выводим результат
            Calc c = new Calc();
            int result = c.Add(10, 84);
            Console.WriteLine("Сумма = {0} ", result);
        }
    }
}
```

Пример 1.3.

```
// Описать структуру, состоящую из 3-х полей: имя студента, оценка за
// контрольную работу, экзаменационная оценка студента.
// Необходимо поставить экзаменационную оценку студенту
// в зависимости от его оценки за контрольную работу
namespace Ex1_3
{
    using System;
    // Определяем структуру C#
    struct Student
    {
        // В структуре могут быть поля:
        int mark;
        int markExam;
        string nameOfStudent;
        // В структуре можно определить конструкторы, но только с
        // параметрами. Все поля должны быть проинициализированы
        public Student(string nameOfStudent, int mark)
        {
            this.nameOfStudent = nameOfStudent;
            this.mark = mark;
            this.markExam = 0;
        }
```

```
        // В структурах могут быть определены методы:
        public string GetNameOfStudent()
        {
            return nameOfStudent;
        }
        public int GetMarkExam()
        {
            return markExam;
        }

        public void SetMarkExam()
        {
            markExam = mark;
        }
    }

    class Program
    {
        public static void Main(string[] args)
        {
            Student student = new Student("Victor", 9);
            student.SetMarkExam();
            Console.WriteLine("Name = {0} , markExam = {1} ",
                                student.GetNameOfStudent(),
                                student.GetMarkExam());
            student = new Student("Petr", 8);
            student.SetMarkExam();
            Console.WriteLine("Name = {0} , markExam = {1} ",
                                student.GetNameOfStudent(),
                                student.GetMarkExam());
        }
    }
}
```

Пример 1.4.

```
// Работа с перечислением в C#
namespace Ex1_4
{
    // Перечисление C#:
```

```
    enum CRTColors
    {
        Red = 100, // По умолчанию принимаются значения, начиная с нуля
        Green = 101,
        Blue = 102
    }
    class Program
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine(CRTColors.Red);      //Red

            CRTColors variable = CRTColors.Green;
            System.Console.WriteLine(variable);           //Green

            CRTColors []array = new CRTColors[3];

            array[0] = CRTColors.Red;
            array[1] = CRTColors.Green;
            array[2] = CRTColors.Blue;

            for (int i = 0; i < array.Length; i++)
                System.Console.WriteLine((int)array[i]);  //100 101 102
            for (int i = 0; i < array.Length; i++)
                System.Console.WriteLine(array[i]);       //Red Green Blue
        }
    }
}
```

Пример 1.5.

```
// Сгенерировать графическое окно сообщения (message box).
using System;
using System.Windows.Forms;
namespace Ex1_5
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            MessageBox.Show("Hello... ");
        }
    }
}
```

Результат работы примера:

Пример 1.6.

```
// Работа с несколькими модулями программы.
// Создадим дополнительный исходный файл HeLLoMsg.cs с классом
// HelloMessage
using System;
using System.Windows.Forms;
namespace Ex1_6
{
    public class HelloMessage
    {
        public void Speak()
        {
            MessageBox.Show("Hello, world...");
        }
    }
}
// Файл Program.cs изменим
using System;
using System.Windows.Forms;
namespace Ex1_6
{
    class Program
    {
        static void Main(string[] args)
        {
            MessageBox.Show("Hello... ");
            HelloMessage hello = new HelloMessage();
            hello.Speak();
```

```
        }
    }
}
```

Пример 1.7.

```
// Работа с коллекцией
using System;
using System.Collections.Generic;
namespace Ex1_7
{
    class Program
    {
        static void Main(string[] args)
        {
            int n = 5;
            List<int> list = new List<int>(n);
            for (int i = 0; i < 6; i++)
                if (i < n)
                {
                    list.Add(i);
                    Console.Write("Add - {0} \n", i);
                }
                else
                {
                    list.Remove(0);
                    Console.Write("Remove\n");
                    list.Add(i);
                    Console.Write("Add - {0} \n",i);
                }
        }
    }
}
// Число 0 будет удалено, а число 5 – добавлено.
```

Пример 1.8.

```
// Ввод информации с клавиатуры
using System;
namespace Ex1_8
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            int number1, number2;

            Console.Write("Enter the first number: ");
            number1 = int.Parse(Console.ReadLine());

            Console.Write("Enter the second number: ");
            number2 = int.Parse(Console.ReadLine());

            // Вместо {0} , {1}  и {2}  подставляются переменные number1,
            // number2 и number1 + number2
            Console.WriteLine("{0} +{1} ={2} ",
                                    number1, number2, number1 + number2);
        }
    }
}
```

Пример 1.9.

```
// Работа с файлами
using System;
using System.IO;

namespace Ex1_9
{
    class Program
    {
        static void Main(string[] args)
        {
            try    // Это на тот случай, если файла data.txt на месте не окажется
            {
                // Открываем файл data.txt
                StreamReader strReader = File.OpenText("data.txt");
                // Считываем каждую строку и выводим ее на консоль
                string strLine;
                while((strLine = strReader.ReadLine())!=null)
                {
```

```
                Console.WriteLine(strLine);
            }
            // Файл необходимо закрыть
            strReader.Close();
        }
        catch(FileNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }
    }
  }
}
```

Пример 1.10.

```
// Работа со свойствами
using System;
// Объявляем, что мы работаем в пространстве имен Ex1_10
namespace Ex1_10
{
  // Класс Client содержит информацию о клиенте - его имя и баланс
  // счета
  class Client
  {
    // Эта переменная хранит имя клиента
    private string name;
    // Эта переменная хранит баланс счета клиента
    private int balance;
    // Это свойство используется для доступа к имени клиента.
    // Принято все переменные класса закрывать для доступа извне
    // и предоставлять доступ к ним через свойства, чтобы
    // контролировать доступ.
    public string Name
    {
      // Получение значения свойства
      get
      {
        return name;
      }
```

```
            // Присвоение свойству нового значения
            // Параметр value представляет новое значения свойства
            set
            {
                name = value;
            }
        }
        // Это свойство служит для доступа к счету клиента. Доступ на
        // запись для него закрыт, чтобы нельзя было извне класса
        // в обход его методов изменить сумму на счету клиента
        public int Balance
        {
            get
            {
                return balance;
            }
        }
        // Определяем констуктор - метод, используемый для создания
        // нового объекта данного класса.
        // Параметр name задает имя для нового клиента
        // Параметр balance задает начальную сумму, которая будет на счету
        public Client(string name, int balance)
        {
            // Задаем имя клиента через свойство
            Name = name;
            // Задаем начальное значение счета через непосредственный
            // доступ к переменной
            this.balance = balance;
        }
        // Этот метод перечисляет деньги на счет клиента
        // Параметр задает сумму, которая будет перечислена
        public void AddMoney(int amount)
        {
            // Перечисляем деньги
            balance += amount;
        }
    }
    // Класс приложения. В нем определена точка входа
```

```
class Program
{
    // Точка входа в программу
    static void Main(string[] args)
    {
        // Создаем нового клиента по имени Иван Петров и 10 тысячами
        // на счету
        Client client = new Client("Иван Петров", 10000);
        // Выводим начальное состояние клиента
        Console.WriteLine("Начальное состояние клиента {0}  - {1} ",
                                client.Name, client.Balance);
        client.AddMoney(10000);
        client.AddMoney(100);
        Console.WriteLine("Конечное состояние клиента {0}  - {1} ",
                                client.Name, client.Balance);
    }
}
}
```

ГЛАВА 2. ОСНОВЫ ЯЗЫКА C#

2.1. ОБЩИЕ КОНЦЕПЦИИ СИНТАКСИСА

Ключевыми структурными понятиями в языке C# являются *сборки*, *программы*, *пространства имен*, *типы и элементы типов*.

2.1.1. Структура программы на языке C#

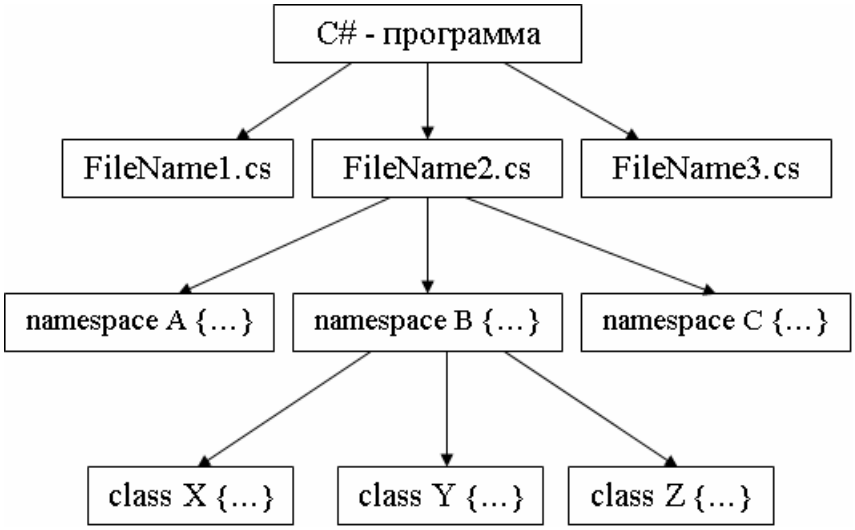


Рис. 3. Структура программы на языке C#

Заметим, что программа на C# может состоять как из одного, так и из нескольких файлов, содержащих исходный текст на языке программирования C#. Каждый такой файл имеет расширение .cs (в нашем примере файлы названы FileName1.cs, FileName2.cs и FileName1.cs).

Любой файл с исходным текстом на языке программирования C# может как содержать пространства имен, так и не содержать их (в нашем примере файл FileName2.cs содержит три пространства имен (A, B и C), а FileName1.cs и FileName3.cs не содержат пространств имен).

Наконец, каждое пространство имен может как содержать описание (одного или нескольких) классов, так и не содержать их (в нашем примере пространство имен B содержит три описания трех классов (X, Y и Z), а пространство имен A и C не содержат ни одного описания классов).

Итак, программа на языке C# размещается в одном или нескольких текстовых файлах, стандартное расширение которых – .cs. В программе объявляются пользовательские типы, которые состоят из элементов. Примерами пользовательских типов являются классы и структуры, а

примером элемента типа может служить метод класса. Типы могут быть логически сгруппированы в пространства имен. При компиляции программы получается сборка, представляющая собой файл с расширением .exe или .dll.

Исходный текст программы на языке C# содержит *операторы* и *комментарии*. Основными видами операторов в C# являются следующие:

- *Оператор-выражение*. Под выражением может пониматься вызов метода, присваивание, а также допустимые комбинации операндов и операций. Оператор-выражение завершается символом “;” (точка с запятой).
- *Операторы управления* ходом выполнения программы, такие как оператор условного перехода или операторы циклов.
- *Блок операторов*. Блок – это набор операторов, обрамленных фигурными скобками – ‘{‘ и ‘}’. Блоки необходимо использовать там, где синтаксис языка требует выполнение более одного оператора.
- *Операторы объявлений* пользовательских типов, элементов типов и локальных переменных и констант.

Программа может содержать комментарии, игнорируемые при компиляции. Различают следующие виды комментариев:

1. *Строчный комментарий* – это комментарий, начинающийся с последовательности символов // и продолжающийся до конца строки.
2. *Блочный комментарий* – все символы, заключенные между /* и */.
3. *Комментарии для документации* – напоминают строчные комментарии, но начинаются с последовательности символов /// и могут содержать специальные XML-тэги.

В языке C# различаются строчные и прописные символы при записи идентификаторов и ключевых слов.

Количество пробелов в начале строки, в конце строки и между элементами строки значения не имеет. Это позволяет улучшить структуру исходного текста программы.

Язык C# требует, чтобы вся программная логика была заключена в определения типов (под типом подразумеваются классы, интерфейсы, структуры и аналогичные компоненты языка). В отличие от языков C и

C++ глобальные функции и глобальные переменные в чистом виде в языке C# использовать нельзя.

2.1.2. Обработка параметров командной строки

Пример 2.1

```
using System;
namespace Ex2_1
{
   class Program
   {
      static void Main(string[] args)
      {
         // Выводим параметры на консоль
         for(int x=0; x < args.Length; x++)
         {
            Console.WriteLine("Arg: {0} ", args[x]);
         }
      }
   }
}
```

Цикл для последовательной обработки всех параметров командной строки можно организовать, используя конструкцию foreach:

Пример 2.2

```
using System;
namespace Ex2_2
{
   class Program
   {
      static void Main(string[] args)
      {
         // Выводим параметры на консоль
         foreach (string s in args)
            Console.WriteLine("Arg: {0} ", s);
      }
   }
}
```

2.1.3. Создание объектов: конструкторы

Во всех объектно-ориентированных языках четко различаются понятия «класс» и «объект». Класс — это определяемый пользователем тип данных (*user-defined type, UDT*). Объектом называется конкретный экземпляр определенного класса, с помощью которого обычно и производятся определенные действия в программе.

Единственный способ создания нового объекта в C# — использовать ключевое слово *new*.

Ключевое слово *new* означает, что среде выполнения следует выделить необходимое количество оперативной памяти под экземпляр создаваемого объекта. **Выделение памяти производится из «кучи»**, находящейся в распоряжении среды выполнения .NET (*managed heap* — **управляемой кучи**).

В C# переменные класса — это ссылки на объект в памяти, но не сами объекты.

C# автоматически снабжает конструктором по умолчанию любой класс. Как и в C++, конструктор класса по умолчанию — это конструктор, который не принимает никаких параметров. Однако в C# имеется существенное отличие от C++ в отношении конструктора по умолчанию. В C++ при создании объекта при помощи конструктора по умолчанию переменные остаются неинициализированными (то есть в них могут быть любые случайные значения). **В C# конструктор по умолчанию** (как и все другие конструкторы), **если не указано иное, присваивает всем данным состояния** (например, переменным-членам) **значения по умолчанию**.

В большинстве случаев класс помимо конструктора по умолчанию снабжается и другими конструкторами — принимающими параметры. Использование такого конструктора — самый простой способ инициализировать состояние объекта в момент его создания, установив нужные значения.

Пример 2.3

```
using System;
namespace Ex2_3
{
   class HelloClass
   {
      // Объявляем переменные-члены класса
```

```
        private int intX;
        private int intY;

        // Конструктор по умолчанию присвоит переменнын-членам
        // значения по умопчанию
        public HelloClass()
        {
            Console.WriteLine("Default");
        }
        // Наш собственный конструктор присвоит переменным-
        // членам специальные значения
        public HelloClass(int x, int y)
        {
            Console.WriteLine("Custom");
            intX = x;
            intY = y;
        }
        // Точка входа для программы
        static void Main(string[] args)
        {
            // Применяем конструктор по умолчанию
            HelloClass c1 = new HelloClass();
            Console.WriteLine("c1.intX = {0} \nc1.intY = {1} \n",
                                    c1.intX, c1.intY);
            // Применяем конструктор с параметрами
            HelloClass c2 = new HelloClass(100, 200);
            Console.WriteLine("c2.intX = {0} \nc2.intY = {1} \n",
                                    c2.intX,c2.intY);
        }
    }
}
```

Результат работы примера:

```
Default
c1.intX = 0
c1.intY = 0
Custom
c2.intX = 100
c2.intY = 200
```

2.1.4. Утечка памяти

В C# программистам не надо думать о явном удалении объектов и освобождении памяти — сборщик мусора .NET освободит память автоматически. Поэтому **в C# нет даже зарезервированного слова delete**.

2.1.5. Инициализация членов класса

В C# можно инициализировать переменные, находящиеся в классе, прямо в момент их объявления.

2.1.6. Ввод и вывод с использованием класса Console

Класс System.Console — один из многих классов, определенных внутри пространства имен System. Этот класс предназначен для работы с вводом и выводом (в том числе сообщений об ошибках) с использованием системной консоли. Этот класс чаще всего используется в консольных приложениях .NET.

Главные методы класса Console — это методы ReadLine() и WriteLine() (оба этих метода определены как статические).

Метод ReadLine() позволяет считать информацию с системной консоли до ближайшего символа перехода на новую строку, метод Read() считывает с системной консоли единственный символ. Функция ReadLine()возвращает введенную строку, которая обычно преобразуется в значение требуемого типа.

Метод WriteLine() выводит символьную строку (дополняя ее в конце символами перехода на новую строку и возврата каретки) на системную консоль. Метод Write() делает то же самое, но уже без дополнения символами перехода на новую строку.

Первый параметр, передаваемый методу WriteLine(), представляет собой строку форматирования с подстановочными выражениями {0}, {1}, {2} и т.п. Остальные параметры WriteLine() — это как раз те значения, которые последовательно подставляются в места, обозначенные подстановочными выражениями. Создатели библиотеки базовых типов позаботились о перегрузке метода WriteLine() таким образом, что в качестве второго параметра этого метода можно передавать массив объектов. При этом подстановочные выражения будут указывать на элементы этого массива.

Например:

```
object[] obj = {"Hello", 20.9, 1, "Hi", "83", 99.99933};
```

```
Console.WriteLine("Objects: {0} , {1} , {2} , {3} , {4} , {5}", obj);
```

Результат:

```
Objects: Hello, 20.9, 1, Hi, 83, 99.99933
```

2.1.7. Средства форматирования строк в C#

В C# предусмотрены новые средства форматирования символьных строк.

Пример 2.4

```
using System;
namespace Ex2_4
{
  class BasicIO
  {
    static void Main(string[] args)
    {
      int theInt = 90;
      float theFloat = 99.99f;
      // Комбинируем символьную строку:
      Console.WriteLine("Int is: {0} \nFloat is: {1} \n", theInt, theFloat);
    }
  }
}
```

Результат выполнения программы:

```
Int is: 90
Float is: 99.99
```

В каждом подстановочном выражении можно использовать параметры форматирования.

Таблица 1. **Параметры форматирования C#**

Параметр	Значение
C или c	Используется для вывода значений в денежном формате
D или d	Используется для вывода десятичных значений. После этих символов можно указать количество выводимых символов после запятой
E или e	Для вывода значений в экспоненциальном формате
F или f	Вывод значений с фиксированной точкой

G или g	Общий формат. Применяется для вывода значений с фиксированной точностью или в экспоненциальном формате
N или n	Стандартное числовое форматирование с использованием разделителей между разрядами.
X или x	Вывод значений в шестнадцатеричном формате. При использовании прописной буквы X символы в шестнадцатеричном формате тоже будут прописными

Символы форматирования следуют в подстановочных выражениях сразу же за номером подставляемого параметра через двоеточие: {0:C}, {l:d}, {2:X} и т.п.

Пример 2.5

```
using System;
namespace Ex2_5
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("C format: {0:C}", 99989.987);
         Console.WriteLine("D9 format: {0:D9}", 99999);
         Console.WriteLine("E format: {0:E}", 99999.76543);
         Console.WriteLine("F3 format: { 0:F3} ", 99999.9999);
         Console.WriteLine("N format: {0:N}", 99999);
         Console.WriteLine("X format: {0:X}", 99999);
         Console.WriteLine("x format: {0:x}", 99999);
      }
   }
}
```

Результаты выполнения этой программы:

```
C format: 99 989,99р.
D9 format: 000099999
E format: 9,999977E+004
F3 format: 100000,000
N format: 99 999,00
X format: 1869F
x format: 1869f
```

2.2. СИСТЕМА ТИПОВ ЯЗЫКА C#

2.2.1. Классификация систем типизации в языках программирования

Существуют следующие системы типизации: *строгая*, *сильная*, *слабая*, *полиморфная*.

1. **Строгая типизация**. Исторически наиболее распространенной для языков программирования является строгая типизация. При таком формировании системы типов в языке в любой момент существования любого языкового объекта существует однозначное соответствие между объектом и его типом. Другими словами, можно запрограммировать функцию, определяющую тип объекта. Строго типизированными являются классические языки программирования Pascal, FORTRAN, PL/1 и др. Отметим, что классический вариант языка программирования C не является строго типизированным.
2. **Сильная типизация** необходима для обеспечения корректности связывания переменных со значениями **до выполнения программы**. Примером языка программирования с сильной типизацией может служить SML. Язык программирования, не имеющий сильной системы типизации, может быть назван языком со **слабой типизацией**.
3. **Полиморфная типизация**. При таком подходе допустимы выражения переменного типа (скажем, функция упорядочения списка с неопределенным типом элементов).

2.2.2. Универсальная система типизации .NET

Существенным позитивным отличием Microsoft .NET от существующих аналогов на современном рынке программного обеспечения является универсальная система типизации. Проведем ее классификацию.

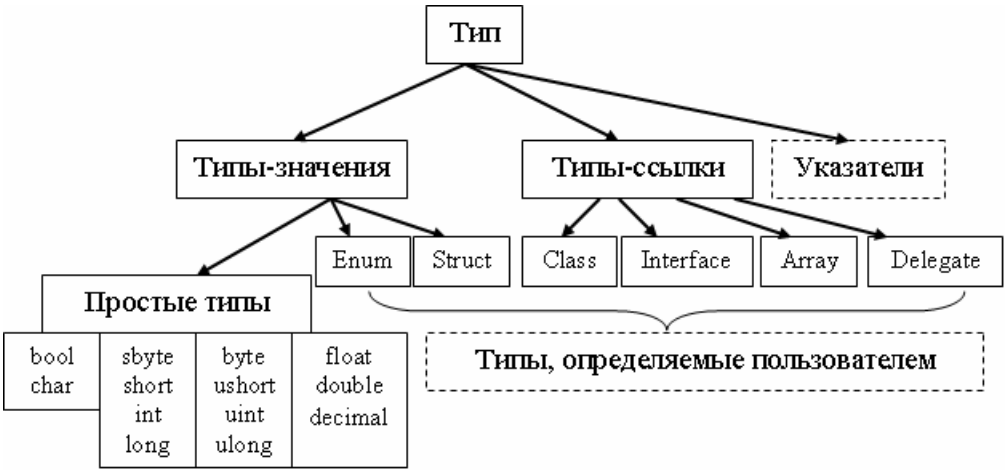


Рис. 4. Универсальная система типизации

2.2.3. Структурные и ссылочные типы

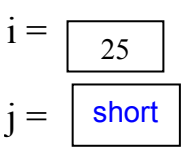
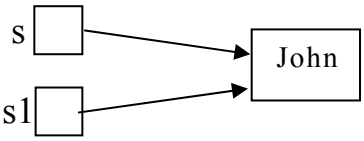
Таким образом, система типов Microsoft .NET образует иерархию с возрастанием общности снизу вверх (см. Рис. 4), в которой явно выделяются две большие группы типов, а именно, **структурные типы** (*value-based*) и **ссылочные типы** (*reference-based*).

К *структурным* типам относятся все числовые типы данных (int, float и пр.), а также перечисления и структуры. К *ссылочным* типам – классы и интерфейсы.

Таблица 2. **Сравнение структурных и ссылочных типов**

	Структурные типы	**Ссылочные типы**
Переменная содержит	значение	ссылку на значение
Выделение памяти	в стеке	в куче (упр. дин. памяти)
Значение по умолчанию	0, false, '\0'	null
Оператор присваивания	копирует значение	копирует ссылку
Передача параметров	как значений (вызывающей функции передаются только локальные копии значений переменных)	как ссылок (вызывающей функции передается ссылка на адрес данного объекта в оперативной памяти)

Можно ли определить конструкторы для этого типа	Да, но конструктор по умолчанию зарезервирован (т.е. все конструкторы программиста должны принимать значения)	Конечно
Освобождение памяти, занятой переменной	При выходе за область видимости	Во время процесса сборки мусора (garbage collection) в управляемой куче

Пример	int i = 25;
	int j = i;
	 
		string s = “John";
		string s1 = s;
		 

В соответствии с названиями, переменная в случае использования типов-значений содержит собственно значение, а при использовании ссылочных типов – не само значение, а лишь ссылку (указатель) на него.

Местом хранения переменной, определенной как тип-значение, является стек, а определенной как ссылочный тип – «куча» (последнее необходимо для динамического выделения и освобождения памяти для хранения переменной произвольным образом).

Значением, которым переменная инициализируется по умолчанию (необходимость выполнения этого требования диктуется идеологией безопасности Microsoft .NET) в случае определения посредством типа-значения является 0 (для целого или вещественного типа данных), false (для логического типа данных), '\0' (для строкового типа данных), а в случае определения посредством ссылочного типа – значение пустой ссылки null.

При выполнении оператора присваивания в случае переменной-значения копируется значение, а в случае переменной-ссылки – ссылка.

2.2.4. Класс System.Object

В C# все типы данных (как структурные, так и ссылочные) производятся от единого общего предка – класса System.Object. Можно явно указывать, что класс производится от System.Object:

```
class Program:System.Object
```

Как и в любом другом классе C#, в классе System.Object существует свой набор членов. Некоторые члены определены как виртуальные и должны быть замещены в определении производного класса.

```
//Самый верхний класс в иерархии классов .NET: System.Object
namespace System
{
   public class Object
   {
      public Object();
      public virtual Boolean Equals(Object obj);
      public virtual Int32 GetHashCode();
      public Type GetType();
      public virtual String ToString();
      protected virtual void Finalize();
      protected Object MemberwiseClone();
   }
}
```

Таблица 3. **Главные методы объекта System.Object**

Метод	Назначение
Equals()	По умолчанию этот метод возвращает true, когда сравниваемые сущности указывают на одну и ту же область в оперативной памяти. Поэтому этот метод в его исходной реализации предназначен только для сравнения объектов ссылочных типов. Для работы с объектами структурных типов этот метод необходимо заместить. При замещении этого метода необходимо заместить метод GetHashCode()
GetHashCode()	Возвращает целочисленное значение, идентифицирующее конкретный экземпляр объекта данного типа

GetType()	Метод возвращает объект Type(), полностью описывающий тот объект, из которого метод был вызван
ToString()	Возвращает символьное представление объекта в формате <имя пространства имен>.<имя класса>. Если тип определен вне пространства имен, возвращается только имя класса. Метод может быть замещен для представления информации о внутреннем состоянии объекта
Finalize()	Основное назначение метода – освободить все ресурсы, занятые объектом данного класса, перед удалением этого объекта
MemberwiseClone()	Метод предназначен для создания еще одной ссылки на область, занимаемую объектом данного типа в оперативной памяти. Метод не может быть замещен.

Пример 2.6

```
// Создаем объекты и знакомимся с методами, унаследованными от
// System.Object
using System;
namespace Ex2_6
{
   class Program
   {
      static void Main(string[] args)
      {
         // Создаем экземпляр класса Program
         Program c1 = new Program();  // Память выделена в куче
         // Выводим информацию на консоль
         Console.WriteLine("ToString: {0}", c1.ToString());
         Console.WriteLine("HashCode: {0}", c1.GetHashCode());
         Console.WriteLine("Type: {0}", c1.GetType().ToString());
         // Создаем еще одну ссылку на объект c1
         Program c2 = c1; // Не создается новый объект, а создана ссылка
         object obj = c2;   // Создается еще одна ссылка
         // Действительно ли все три экземпляра указывают на одну
         // и ту же область в оперативной памяти?
         if(obj.Equals(c1) && c2.Equals(obj))        // Сравниваются ссылки
```

```
            Console.WriteLine("Same instance!");   //Да
        }
    }
}
```

Обратите внимание, что реализация по умолчанию метода ToString() возвращает только имя того типа, из которого данный метод был вызван (Program).

Исходный вариант Equals() использует для сравнения ссылочную, а не структурную семантику. В начале был создан новый объект типа Program с именем c1. В результате для этого объекта в области управляемой кучи была выделена оперативная память. Объект c2 также относится к типу Program. Однако при его создании мы не создавали нового экземпляра объекта. Вместо этого была создана еще одна ссылка на ту же область оперативной памяти, которую занимает объект c1. Точно таким же способом была создана и третья ссылка на ту же область памяти — объект obj. Поскольку и c1, и c2, и obj — это ссылки на одну и ту же область оперативной памяти, операция сравнения с использованием метода Equals() вернула положительный результат.

2.2.4.1. Замещение методов System.Object

Пример 2.7

```
// Сравнение ссылок на объекты
using System;
namespace Ex2_7
{
   class Person
   {
      public Person(string firstName, string lastName, string SSN, byte age)
      {
         this.firstName = firstName;
         this.lastName = lastName;
         this.SSN = SSN;
         this.age = age;
      }
      public Person(){ }  // Всем переменным-членам будут присвоены
                          // значения по умолчанию
      // Данные о человеке
      public string firstName;  // имя
```

```
        public string lastName;   // фамилия
        public string SSN;        // номер социального страхования
        public byte age;          // возраст
    }
}
```

В качестве замещаемого метода выберем Object.ToString(). Мы хотим, чтобы этот метод возвращал не имя типа, а информацию о внутреннем состоянии объекта.

```
    public override string ToString()
    {
        string sb = "FirstName = " + this.firstName;
        sb = sb + " LastName = " + this.lastName;
        sb = sb + " SSN = " + this.SSN;
        sb = sb + " Age = " + this.age";
        return sb;
      }
```

Следующий метод для замещения — метод Equals(). Этот метод возвращает истинно только в тех ситуациях, когда сравниваются ссылки на один и тот же объект в оперативной памяти. Во многих ситуациях имеет смысл заместить этот метод таким образом, чтобы он был полезен не только при работе со ссылочными типами, но и со структурными. Да и в отношении ссылочных типов иногда может потребоваться, чтобы этот метод вел себя не так, как его исходный вариант. Пусть наш метод Equals() возвращает истинно тогда, когда у сравниваемых объектов типа Person одинаковые внутренние состояния (то есть значения переменных firstName, lastName, SSN и age совпадают):

```
    public override bool Equals (object obj)
    {
        // Совпадают ли у объекта, принимаемого в качестве параметра,
        // значения переменных с текущими?
        Person temp = (Person)obj;     // или Person temp = obj as Person;
        if (temp.firstName = = this.firstName &&
            temp.lastName = = this.lastName &&
            temp.SSN = = this.SSN &&
            temp.age = = this.age)
                return true;
        else
```

```
            return false;
        }
```

Следует обратить внимание еще на один момент. Если замещается метод Equals(), то необходимо также заместить метод GetHashCode(). В случае, если это сделано не будет, компилятор выдаст предупреждение. Метод GetHashCode() возвращает числовое значение, идентифицирующее объект в оперативной памяти.

Существует несколько способов сгенерировать уникальный хэш-код для объекта. Для наших целей вполне подойдет значение, возвращаемое тем же самым методом GetHashCode(), но для переменной SSN (ведь тип данных, который для нее использован, также является производным от System.Object, и поэтому этот метод применим):

```
// Возвращаем хэш-код, основанный на номере социального страхования
// (переменной SSN)
    public override int GetHashCode()
    {
      return SSN.GetHashCode();
    }
```

После этого наш класс Person готов к использованию. Его применение может выглядеть так:

```
    static void Main(string[] args)
    {
      // Создаем несколько объектов и производим их сравнение на
      // идентичность.
      // Присвоим одинаковые значения переменным-членам для
      // целей тестирования метода Equals()
      Person p1 = new Person("Иван", "Иванов", "222", 98);
      Person p2 = new Person("Иван", "Иванов", "222", 98);
      // Используем замещенный для сравнения внутреннего состояния
      // метод Equals()
      if(p1.Equals(p2) && p1.GetHashCode() = = p2.GetHashCode())
        Console.WriteLine ("P1 and P2 have same state\n");   // P1=P2
      else
        Console.WriteLine ("P1 and P2 are DIFFERENT\n");
      // Теперь меняем внутреннее состояние p2
        p2.age = 2;
      // Производим сравнение заново
```

```
        if(p1.Equals(p2) && p1.GetHashCode() = = p2.GetHashCode())
            Console.WriteLine ("P1 and P2 have same state\n");
        else
            Console.WriteLine ("P1 and P2 are DIFFERENT\n");  // P1!=P2
        // Теперь используем замещенный метод ToString()
        Console. WriteLine(p1.ToString());
        Console. WriteLine(p2); // Метод WriteLine() вызывает метод
                                // ToString() автоматически
    }
```

Результат выполнения этой программы:

```
P1 and P2 have same state
P1 and P2 are DIFFERENT
FirstName = Иван LastName = Иванов SSN = 222 Age = 98
FirstName = Иван LastName = Иванов SSN = 222 Age = 2
```

2.2.4.2. Статические члены System.Object

В System.Object имеются два статических члена, которые также являются исключительно полезными при сравнении объектов, особенно когда нужно отдельно производить сравнение внутреннего состояния объектов и сравнение адресов ссылок — указывают ли ссылки на одну и ту же область в оперативной памяти или нет. Рассмотрим следующий код:

```
// Статические члены класса System.Object
Person p3 = new Person("Иван", "Иванов", "123", 4);
Person p4 = new Person("Иван", "Иванов", "123", 4);
// Одинаково ли внутреннее состояние p3 и p4? Да!
Console.WriteLine("P3 and P4 have same state: {0} ",
                  object.Equals (p3, p4));
// Представляют ли они один и тот же обьект в памяти? Нет!
Console.WriteLine("P3 and P4 are pointing to same object: {0}",
                  object.ReferenceEquals(p3, p4));
```

Таким образом, используя эти варианты методов Equals() и ReferenceEquals(), можно просто передавать два объекта любого типа классу System.Object (эти методы — статические) — все остальные операции по сравнению будут произведены автоматически.

2.2.5. Системные типы языка C#

Рассмотрим основные типы, входящие в состав Common Type System (CTS) Microsoft .NET.

Числовые типы составляют подмножество примитивных типов. Информация об элементарных типах содержится в табл. 2.4.

Таблица 2.4. **Элементарные типы языка C#**

Категория	**Размер (бит)**	**CTS Системный тип данных**	**Имя Типа Псевдоним C#**	**Диапазон/Точность**
Знаковые целые	8	System.SByte	sbyte	–128...127
	16	System.Int16	short	–32 768...32 767
	32	System.Int32	int	–2 147 483 648...2 147 483 647
	64	System.Int64	long	-9 223 372 036 854 775 808...9 223 372 036 854 775 807
Беззнаковые целые	8	System.Byte	byte	0...255
	16	System.UInt16	ushort	0...65535
	32	System.UInt32	uint	0...4294967295
	64	System.UInt64	ulong	0...18446744073709551615
Вещественные	32	System.Single	float	1.5×10^{-45} … 3.4×10^{38}, точность 7 цифр
	64	System.Double	double	5.0×10^{-324} … 1.7×10^{308}, точность 15 цифр
	128	System.Decimal	decimal	1.0×10^{-28} … 7.9×10^{28}, точность 28 цифр
Логический	8	System.Boolean	bool	true, false
Символьный	16	System.Char	char	16-битовый символ (в коде Unicode)

Строковый	Ограничено только системной памятью	System.String	string	Последовательность Unicode-символов. Хотя тип – примитивный, но переменная этого типа хранит адрес строки в динамической памяти
Object		System.Object	object	Общий предок всех типов

Замечание. Типы sbyte, ushort, uint, ulong не соответствуют Common Language Specification (CLS). Это означает, что данные типы не следует использовать в интерфейсах многоязыковых приложений и библиотек.

Примечание. Хотя тип string относится к примитивным типам, переменная этого типа хранит адрес строки в динамической памяти.

Имя примитивного типа в языке C# является синонимом (псевдонимом) соответствующего типа Framework Class Library. Например, типу int в C# соответствует тип System.Int32, типу float – тип System.Single и т. д.

У каждого системного типа данных (к примеру, Int32, Char, Boolean и т.п.) есть схожий набор членов, которые могут оказаться весьма полезными. Например, свойства MaxValue и MinValue. Первое позволяет получить информацию о максимальном значении, для хранения которого можно использовать данный тип, а второе, соответственно — о минимальном. Например, UInt16.MaxValue = 65 535.

2.2.6. Упаковка и распаковка

В C# допускается рассмотрение значений структурных типов как переменных типа object. Преобразование в объект называется *операцией упаковки* (*boxing*), обратное преобразование – *операцией распаковки* (*unboxing*).

Упаковка – это процесс явного преобразования структурного типа в ссылочный. При упаковке в динамической памяти (управляемой куче) создается объект, содержащий значение структурного типа из стека. При распаковке проверяется фактический тип объекта, и значение из динамической памяти переписывается в соответствующую переменную в стеке. Операция распаковки требует явного указания целевого типа.

```
int i = 123;
object obj = i;              // Упаковка
```

Перед выполнением распаковки среда выполнения производит проверку на совместимость между типом объекта в оперативной памяти и тем структурным типом, в который будет производиться распаковка.

```
int j = (int)obj;                // Распаковка
```

Если же среда выполнения обнаружит, что происходит попытка произвести распаковку в неподходящий тип данных, будет сгенерировано исключение InvalidCastException:

```
// Неверная распаковка
try
{
   // Мы пытаемся распаковать в тип данных string объект,
   // исходный тип данных для которого - int
   string str = (string)obj;
}
catch(InvalidCastException e)
{
   Console.WriteLine("Ошибка!\n{0}", e.ToString());
}
```

Результат выполнения этого фрагмента:

```
Ошибка!
System.InvalidCastException: Unable to cast object of type 'System.Int16' to type 'System.String'.
```

В большинстве случаев операции по упаковке и распаковке выполняются компилятором C# полностью автоматически. Например, если происходит передача структурного типа методу, который принимает в качестве параметра объект, упаковка будет произведена без нашего участия:

```
// Предположим, что есть метод:
public static void Func(object o)
{
   Console.WriteLine(o.ToString());
}
int x = 99;
Func(x);        // Произойдет автоматическая упаковка
```

Возможность автоматического преобразования каждого типа в тип object позволяет создавать универсальные классы, работающие с любыми типами.

2.2.7. Значения по умолчанию для встроенных типов данных

У всех встроенных типов данных в мире .NET есть свои значения по умолчанию. При создании экземпляра класса, его переменным-членам автоматически присваиваются эти значения.

Значением, которым переменная инициализируется по умолчанию (необходимость выполнения этого требования диктуется идеологией безопасности Microsoft .NET) в случае определения посредством типа-значения является 0 (для целого или вещественного типа данных), false (для логического типа данных), '\0' (для строкового типа данных), а в случае определения посредством ссылочного типа – значение пустой ссылки null.

Однако при создании переменных не внутри класса, а внутри метода, значения по умолчанию для них уже применяться не будут. В этой ситуации необходимо ***обязательно*** присвоить им начальные значения (иначе ошибка компилятора!).

Из правила, относящегося к обязательному присвоению значений переменным, создаваемых внутри определения метода, есть одно исключение. **Если переменная действует как исходящий параметр** (с модификатором для параметра метода out) **— присваивать начальное значение этой переменной необязательно.** Причина очевидна — о присвоении значения этому параметру должен позаботиться вызываемый метод. Значения исходящим параметрам присваиваются вызываемым методом обязательно.

2.2.8. Константы

В C# константы определяются при помощи ключевого слова const.

```
public const int myIntConst = 5;
```

Иногда удобно использовать служебный класс для хранения констант. При этом необходимо позаботиться о том, чтобы пользователь не смог создать экземпляр этого класса.

Чтобы запретить создание экземпляров класса:
1 способ) достаточно определить конструктор как private;
2 способ) определив класс как абстрактный (при помощи ключевого слова abstract).

В любом случае при попытке создать экземпляр класса компилятор выдаст сообщение об ошибке. Использование специального класса для хранения всех констант представляется особенно полезным, т.к. C# не позволяет определять глобальные константы.

Замечание. В C# (в отличие от C++) нельзя использовать ключевое слово const как часть объявления метода.

2.2.9. Пользовательские типы

Перечисления, структуры, классы, интерфейсы, массивы и делегаты составляют множество *пользовательских типов*.

Система поддержки общих типов **(Common Type System — CTS)** определяет правила в отношении типов данных.

2.2.9.1. Класс

Класс – тип, поддерживающий всю функциональность объектно-ориентированного программирования, включая наследование и полиморфизм.

Класс (class) – это набор свойств, методов и событий, объединенных в единое целое. В CTS предусмотрены абстрактные члены классов, что обеспечивает возможность применения полиморфизма в производных классах.

Множественное наследование в CTS запрещено!

2.2.9.2. Структура

Структура (structure) – тип, обеспечивающий всю функциональность ООП, кроме наследования.

Структуры CTS могут иметь любое количество конструкторов с параметрами (конструктор без параметров зарезервирован). С помощью конструкторов с параметрами можно (а точнее, нужно) установить значение любого поля объекта структуры в момент создания этого объекта.

Все CTS-совместимые структуры произведены от единого базового класса System.ValueType. Этот базовый класс определяет структуру как тип данных для работы только со значениями, а не со ссылками.

Структуры не могут быть унаследованы от остальных типов данных и они всегда являются «закрытыми», то есть они не могут выступать в качестве базовых для целей наследования.

Рассмотрим пример описания структуры Point, моделирующей точку на плоскости:

```
struct Point
{
        private int x, y;  public Point (int x, int y) {  this.x = x; this.y = y; }
          public void MoveTo (int a,int b) { x=a; y=b;}
```

```
}
```

Координаты точки x, y (в простейшем случае целочисленные), которые используются для определения ее положения, являются атрибутами объекта или полями структуры.

Метод Point является функцией инициализации объекта и называется конструктором. Указатель this является ссылкой на текущий объект.

Замечания:

1. Конструктор может быть только с параметрами!
2. В конструкторе все поля структуры должны быть обязательно проинициализированы!

Метод MoveTo() изменяет текущее местоположение точки на пару целочисленных координат (a,b).

Для использования объекта-структуры необходимо проинициализировать объект-значение в стеке посредством конструктора. Затем можно вызвать метод MoveTo():

```
Point p = new Point(3, 4);
p.MoveTo(10, 20);
```

2.2.9.3. Интерфейс

Интерфейс (interface) – абстрактный тип, реализуемый классами и структурами для обеспечения оговоренной функциональности.

При создании своего собственного интерфейса на .NET-совместимом языке программирования можно произвести этот интерфейс сразу от нескольких базовых интерфейсов.

2.2.9.4. Массив

Массив – пользовательский тип для представления упорядоченного набора значений некоторых (примитивных или пользовательских) типов.

2.2.9.5. Перечисление

Перечисление (enumeration) — это программная конструкция, которая позволяет объединять пары «имя — значение» под указанным именем перечисления.

В CTS все перечисления являются производными от базового класса System.Enum.

Оператор перечисления enum языка C# представляет собой список поименованных констант.

```
enum Color { red, blue, green}
```

Пример использования константы после ее описания имеет следующий вид:

```
Color c = Color.blue;
```

2.2.9.6. Делегат

Делегат (delegate) – пользовательский тип данных для представления ссылок на методы (безопасный для типов эквивалент указателя на функцию в C).

Делегат .NET – это не просто адрес в оперативной памяти, а класс, производный от базового класса MulticastDelegate.

Делегаты очень полезны в тех ситуациях, когда нужно, чтобы одна сущность передала вызов другой сущности.

Делегаты используются в технологии обработки событий .NET.

2.2.10. Операторы typeof и sizeof в языке C#

В языке C# реализована возможность определения типа для того или иного языкового объекта. Для реализации этой возможности используется оператор typeof, который для любого заданного типа возвращает дескриптор этого типа.

```
// Возвращает для данного типа дескриптор типа
Type t = typeof(int);
Console.WriteLine(t.Name);     // тип данного выражения: Int32
```

Оператор sizeof возвращает размер элемента данного типа (в байтах). Данный оператор применим только к типам-значениям и используется только в блоках небезопасного кода (т.к., например, размер структур может изменяться в зависимости от среды реализации). Приведем пример использования оператора sizeof (блоки небезопасного C#-кода выделяются посредством ключевого слова unsafe).

```
unsafe
{
   Console.WriteLine(sizeof(int));
   Console.WriteLine(sizeof(MyEnumType));
   Console.WriteLine(sizeof(MyStructType));
}
```

2.2.11. Иерархия типов

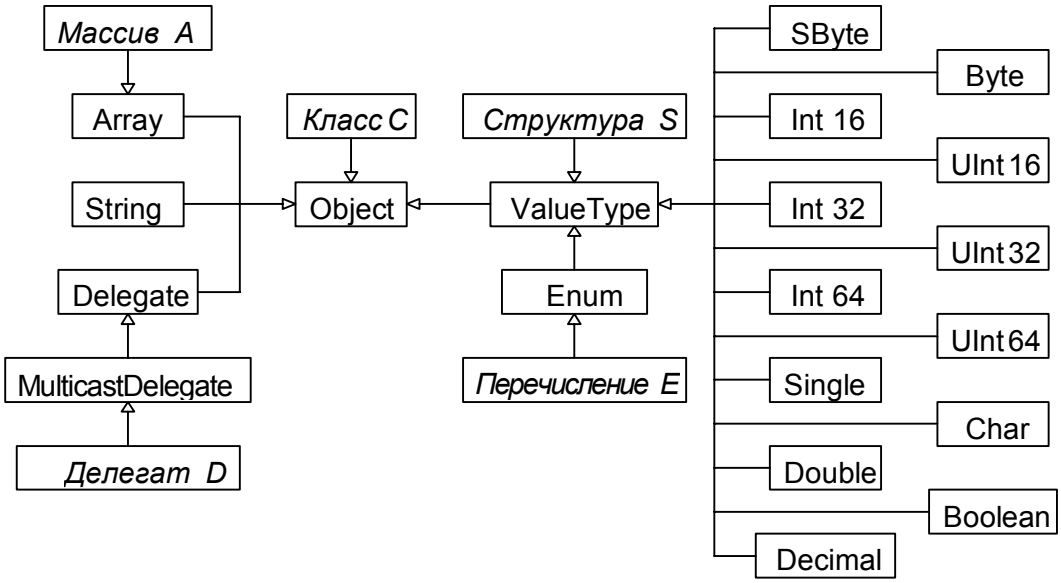


Рис. 5. Иерархия типов .NET Framework

2.2.12. Преобразование типов

Если при вычислении выражения операнды имеют разные типы, то возникает необходимость приведения их к одному типу. Такая необходимость возникает и тогда, когда операнды имеют один тип, но он не согласован с типом операции. Например, при выполнении сложения операнды типа byte должны быть приведены к типу int, поскольку сложение не определено над байтами. При выполнении присваивания x=y тип источника y и тип приемника x должны быть согласованы. Аналогично, при вызове метода также должны быть согласованы типы фактического и формального параметров.

Рассмотрим преобразования при работе с числовыми типами.

Преобразование типов бывает неявным и явным.

Неявное преобразование (*implicit conversion*) выполняется автоматически. При выполнении данного преобразования никогда не происходит потеря точности или переполнение, так как множество значений целевого типа включает множества значений приводимого типа. Для числовых типов неявное преобразование типа A в тип B возможно, если на рис. 6 существует путь из A в B.

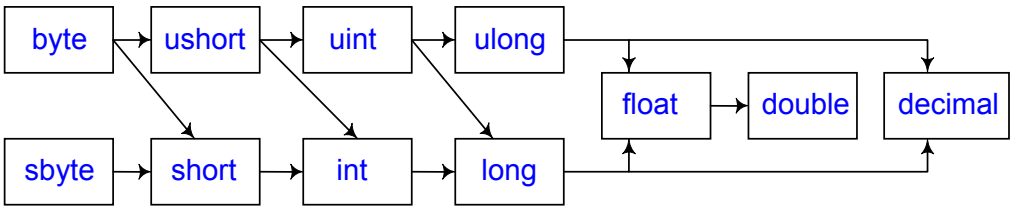


Рис. 6. Схема неявного преобразования числовых типов

Для *явного преобразования* (*explicit conversion*) требуется применять *оператор приведения* в форме (<целевой тип>)<выражение>. При выполнении явного преобразования ответственность за его корректность возлагается на программиста.

```
int k = 100;
byte i;              // тип byte «меньше» типа int
i = (byte) k;        // требуется явное преобразование типов
```

Для более гибкого контроля значений, получаемых при работе с числовыми выражениями, в языке C# предусмотрено использование контролируемого и неконтролируемого контекстов.

Контролируемый контекст объявляется в форме

checked <программный блок>,

либо как оператор

checked(<выражение>).

Если при преобразовании типов выражение в контролируемом контексте получает значение, выходящие за пределы целевого типа, то генерируется либо ошибка компиляции (для константных выражений), либо ошибка времени выполнения (для выражений с переменными).

При использовании *неконтролируемого контекста* выход за пределы целевого типа ведет к автоматическому «урезанию» результата либо путем отбрасывания бит (целые типы), либо путем округления (вещественные типы). *Неконтролируемый контекст* объявляется в форме

unchecked <программный блок>

либо как оператор

unchecked(<выражение>).

Рассмотрим несколько примеров использования контекстов:

```
int i = 1000000;
int k = 1000000;
```

```
int n = i * k;
```

В данном примере n получит значение -727379968, то есть произойдет отбрасывание «лишних» бит числа, получившегося в результате умножения.

Используем неконтролируемый контекст:

```
int i = 1000000;
int k = 1000000;
int n = unchecked(i * k);
```

Значение n осталось прежним (-727379968), таким образом, неконтролируемый контекст применяется по умолчанию.

Теперь проведем вычисления в контролируемом контексте:

```
int i = 1000000;
int k = 1000000;
int n = checked(i * k);
```

При выполнении последнего оператора произойдет генерация исключения System.OverflowException – переполнение.

Важным классом преобразований являются **преобразования в строковый тип и наоборот**. Преобразования в строковый тип всегда определены, поскольку все типы являются потомками базового класса object, а, следовательно, обладают методом ToString() этого класса. Для встроенных типов определена подходящая реализация этого метода. В частности, для всех числовых типов метод ToString() возвращает строку, задающую соответствующее значение типа. Метод ToString() можно вызывать явно, но, если явный вызов не указан, то он будет вызываться неявно, всякий раз, когда требуется преобразование к строковому типу. Преобразования из строкового типа в другие типы должны всегда выполняться явно при помощи методов встроенных или пользовательских классов. Например, класс System.Int32 обладает методом Pasre(), позволяющим преобразовать строку в целое число.

```
System.Console.WriteLine("Input your age");
string s = System.Console.ReadLine();
int Age = System.Int32.Parse(System.Console.ReadLine());
```

Тип char преобразуется в типы sbyte, short, byte явно, а в остальные числовые типы – неявно. Заметим, что преобразование любого числового типа в тип char может быть выполнено, но только в явной форме.

Преобразования пользовательских ссылочных типов выполняются при присваивании и вызове методов. При этом действует стандартное правило ООП: объект типа-потомка преобразуется в объект типа-предка автоматически. Все прочие преобразования должны быть выполнены при помощи оператора приведения. Ответственность за их правильность возлагается на программиста.

В пространстве имен System содержится класс Convert, методы которого поддерживают общий способ выполнения преобразований между типами. Класс Convert содержит набор статических методов вида To<Type>(), где Type – имя встроенного типа CLR (ToBoolean(),ToUInt64() и т. д.). Все методы To<Type>() класса Convert перегружены и каждый из них имеет, как правило, более десятка реализаций с аргументами разного типа. Так что фактически эти методы задают все возможные преобразования между всеми встроенными типами языка C#.

2.2.13. Пространства имен

Пространство имен — это логическая структура для организации имен, используемых в приложении .NET. Основное назначение пространств имен — предупредить возможные конфликты между именами в разных сборках.

2.2.13.1. Важнейшие пространства имен .NET

Таблица 2.5. **Пример пространства имен .NET**

Пространство имен .NET	**Назначение**
System	Содержит множество низкоуровневых классов для работы с простыми типами выполнения математических операций, сборки мусора и т.п.
System Collections	Для работы с контейнерными объектами, такими как ArrayList, Queue, SortedList
System.Data System.Data.Common System.Data.OleDb System.Data.SqlClient	Для обращения к базам данных
System.Diagnostics	Содержаться типы, используемые .NET – совместимыми языками для трассировки и отладки программного кода

System.Drawing System.Drawing.Drawing2D System.Drawing.Printing	Типы для примитивов GDI+ - растровых изображений, шрифтов, значков, поддержки печати
System.Ю	В этом пространстве имен объединены типы, отвечающие за операции ввода-вывода – в файл, буфер и т.п.
System.Net	Содержит типы, относящиеся к передаче данных по сети (запрос-ответ, создание сокетов и т.п.)
System.Reflection System.Reflection.Emit	Классы, предназначенные для обнаружения, создания и вызова во время выполнения пользовательских типов
System.Security	В .NET средства обеспечения безопасности интегрированы как со средой выполнения, так и с библиотекой базовых типов. В этом пространстве имен находятся классы для работы с разрешениями, криптографией и т.п.
System.Threading	Пространство имен для типов, которые используются при работе с потоками (например, Mutex, Thread или Timeout)
System.Web	Классы, которые предназначены для использования в web-приложениях, включая ASP.NET
System.Windows.Forms	Классы для работы с элементами интерфейса Windows – окнами, элементами управления и прочим
System.XML	Множество классов для работы с данными в формате XML

2.2.13.2. Использование пространств имен в коде приложения

Пространство имен — это средство для логической группировки типов.

Слово using используется для обращения к типам в конкретном пространстве имен. После того как определено использование конкретного пространства имен (с использованием ключевого слова using), можно обращаться к типам, содержащимся в этом пространстве. Однако можно обойтись и без using, если использовать полное имя класса.

Рассмотрим два файла X.cs и Y.cs, содержащих исходные тексты программ на языке C#.

```
Файл X.cs                    Файл Y.cs
namespace A                  namespace A
{                            {
    ...                          ...
  namespace B {...}            namespace B {...}
                               namespace C {...}
}                            }
```

В файле X.cs содержатся описания пространств имен A и B, а в файле Y.cs – A, B и C, причем в обоих случаях последующие пространства имен вложены в A. При обращении к вложенному пространству имен нужно указывать его полное имя, например, A.B.

Пространства имен из разных файлов, имеющие один и тот же идентификатор, составляют единую область описания.

Вложенные пространства имен составляют собственную (отдельную) область описания.

Рассмотрим более сложный пример использования пространств имен в языке программирования C#.

Пусть программный проект на языке программирования C# содержит три файла с описаниями структур данных, оформленными в виде отдельных пространств имен:

```
Color.cs              Figures.cs                  Triangle.cs

namespace Util        namespace Util.Figures      namespace  Util.Figures
{                     {                           {
public enum           public class                public class
  Color {...}           Rect {...}                    Triangle {...}
}                     public class                }
                        Circle {...}
                      }
```

В данном случае при использовании полного имени пространства имен (Util.Figures) возможно обойтись без конкретизации классов (Rect), описанных внутри этого пространства имен. Однако, в случае обращения к классам вне данного пространства имен необходимо использовать полное квалификационное имя объекта (Util.Color c):

```
using Util.Figures;
class Test
```

```
{
    Rect r;       // без указания полного имени (т.к. используем Util.Figures)
    Triangle t;
    Util.Color c;     // с указанием полного имени
}
```

2.2.13.3. Использование псевдонимов для имен классов

Еще одна возможность избавиться от конфликтов имен — использовать для имен классов псевдонимы.

```
namespace Ex
{
   using System;
   using Ex1;
   // Создаем псевдоним для класса из другого пространства имен
   using TheHexagon = Ex1.Hexagon;
   class MyApp
   {
      static void Main(string[] args)
      {
         Hexagon h = new Hexagon();
         // Создаем объект при помощи псевдонима
         TheHexagon h2 = new TheHexagon();
      }
   }
}
```

ОГЛАВЛЕНИЕ

Учебное издание

Василенко Жанна Витальевна

ОСНОВЫ ЯЗЫКА C#

Учебно-методическое пособие
для студентов специальностей
1-31 03 03 «Прикладная математика (по направлениям)»,
1-31 03 04 «Информатика»,
1-31 03 05 «Актуарная математика»,
1-31 03 06 «Экономическая кибернетика (по направлениям)»

В авторской редакции

Ответственный за выпуск *Ж. В. Василенко*

Подписано в печать 10.12.2008. Формат 60×84/16. Бумага офсетная.
Гарнитура Таймс. Усл. печ. л. 3,02. Уч.-изд. л. 2,19. Тираж 50 экз. Зак.

Белорусский государственный университет.
ЛИ №02330/0056804 от 02.03.2004.
220030, Минск, проспект Независимости, 4.

Отпечатано с оригинала-макета заказчика
на копировально-множительной технике
факультета прикладной математики и информатики
Белорусского государственного университета.
220030, Минск, проспект Независимости, 4.