ЧАСТЬ II Основы программирования на C#

ГЛАВА 3 Главные конструкции программирования на С#:  
часть I

Каждое исполняемое приложение C# должно содержать класс, определяющий метод Main(), который используется для обозначения точки входа в приложение.

Формально, **объект приложения -** класс, в котором определен метод Main().

Сигнатура метода Main() снабжена ключевым словом static. статические члены имеют область видимости уровня класса (а не уровня объекта) и потому могут вызываться без предварительного создания нового экземпляра класса.

Метод Main() в примере был определен с возвращаемым значением void, т.е. перед выходом из области видимости метода мы не устанавливаем явным образом возвращаемое значение с использованием ключевого слова return.

Внутри метода Main() содержится логика класса Program. Здесь мы работаем с классом Console, который определен в пространстве имен System. В состав его членов входит статический метод WriteLine(), который отправляет текстовую строку и символ возврата каретки в стандартный вывод.

Program-Main-System-Console-WriteLine()

Класс-Метод-Простр. имен-Класс-Метод

Вариации метода Main ()

public static void Main() { }

public static int Main() { }

public static void Main(string[] args) { }

public static int Main(string[] args) { }

Обработка аргументов командной строки

class Program  
{  
 static int Main(string[] args)  
 {  
 *// Обработать любые входные аргументы, используя foreach.* foreach (string arg in args)  
 {  
 Console.WriteLine("Arg: {0}", arg);  
 }  
 Console.ReadLine ();  
 return -1;   
 }  
}

class Program  
{  
 static int Main(string[] args)  
 {  
 *// Получить аргументы с использованием System.Environment.* string[] theArgs = Environment.GetCommandLineArgs();  
 foreach (string arg in theArgs)  
 Console.WriteLine("Arg: {0}", arg);  
 Console.ReadLine();  
 return -1;  
 }  
}

Класс System.Environment

class Program  
{  
 static int Main(string[] args)  
 {  
 ShowEnvironmentDetails();  
  
 Console.ReadLine();  
 return -1;  
 }  
 static void ShowEnvironmentDetails()  
 {  
 *// Вывести информацию о дисковых устройствах  
 // данной машины и другие интересные детали,* foreach (string drive in Environment.GetLogicalDrives())  
 {  
 Console.WriteLine("Drive: {0}", drive); *// Логические устройства* Console.WriteLine("OS: {0}", Environment.OSVersion); *// Версия  
 // операционной системы* Console.WriteLine("Number of processors: {0}",  
 Environment.ProcessorCount); *// Количество процессоров* Console.WriteLine(".NET Version: {0}",  
 Environment.Version); *// Версия платформы .NET* }  
 }  
}

Drive: C:\

OS: Microsoft Windows NT 10.0.19043.0

Number of processors: 32

.NET Version: 6.0.6

Drive: D:\

OS: Microsoft Windows NT 10.0.19043.0

Number of processors: 32

.NET Version: 6.0.6

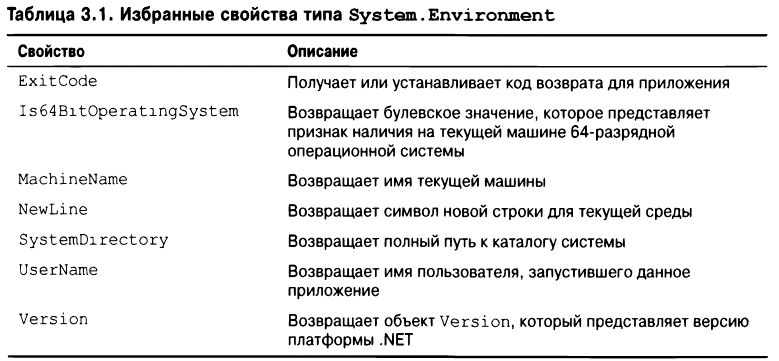
Drive: H:\

OS: Microsoft Windows NT 10.0.19043.0

Number of processors: 32

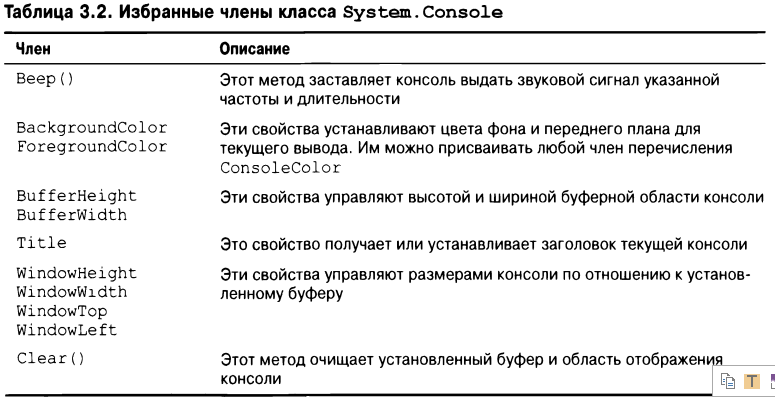
.NET Version: 6.0.6

Также имеются и другие свойства типа Environment:



Класс **System.Console**

**Инкапсуляция -** Упаковка данных и/или функций в единый компонент.



Базовый ввод-вывод с помощью класса **Console**

class Program  
{  
 static void Main(string[] args)  
 {  
 Console.WriteLine("\*\*\*\*\* Basic Console I/O \*\*\*\*\*");  
 GetUserData();  
 Console.ReadLine();  
 Console.WriteLine("Буба");  
 }  
 static void GetUserData()  
 {  
*// Получить информацию об имени и возрасте.* Console.Write("Please enter your name: "); *// Предложить ввести имя* string userName = Console.ReadLine();  
 Console.Write("Please enter your age: "); *// Предложить ввести возраст* string userAge = Console.ReadLine();  
*// Просто ради забавы изменить цвет переднего плана (шрифта).* ConsoleColor prevColor = Console.ForegroundColor;  
 Console.ForegroundColor = ConsoleColor.**Yellow**;  
*// Вывести полученную информацию на консоль.* Console.WriteLine("Hello {0}! You are {1} years old.",  
 userName, userAge);  
*// Восстановить предыдущий цвет переднего плана.* Console.ForegroundColor = prevColor;  
 }  
}

Форматирование числовых данных

class Program  
{  
 static void Main(string[] args)  
 {  
 FormatNumericalData();  
 Console.ReadLine();  
 }  
  
 static void FormatNumericalData()  
 {  
 Console.WriteLine("The value 99999 in various formats:");  
 Console.WriteLine("c format: {0:c}", 99999);  
 Console.WriteLine("d9 format: {0:d9}", 99999);  
 Console.WriteLine("f3 format: {0:f3}", 99999);  
 Console.WriteLine("n format: {0:n}", 99999);  
 *// Обратите внимание, что использование для символа шестнадцатеричного формата  
 // верхнего или нижнего регистра определяет регистр отображаемых символов.* Console.WriteLine ( "Е format: {0:Е}", 99999);  
 Console.WriteLine ("е format: {0:е}", 99999);  
 Console.WriteLine("X format: {0:X}", 99999);  
 Console.WriteLine("x format: {0:x}", 99999);   
 }  
}

**The value 99999 in various formats:**

**c format: 99 999,00 ?**

**d9 format: 000099999**

**f3 format: 99999,000**

**n format: 99 999,00**

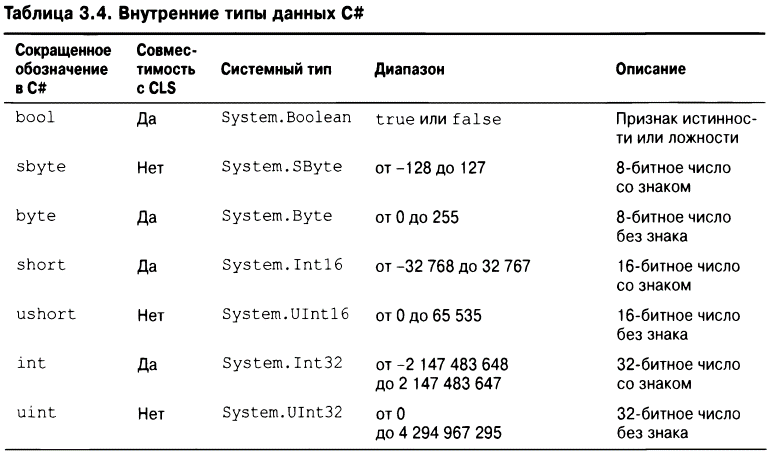
**Е format: Е**

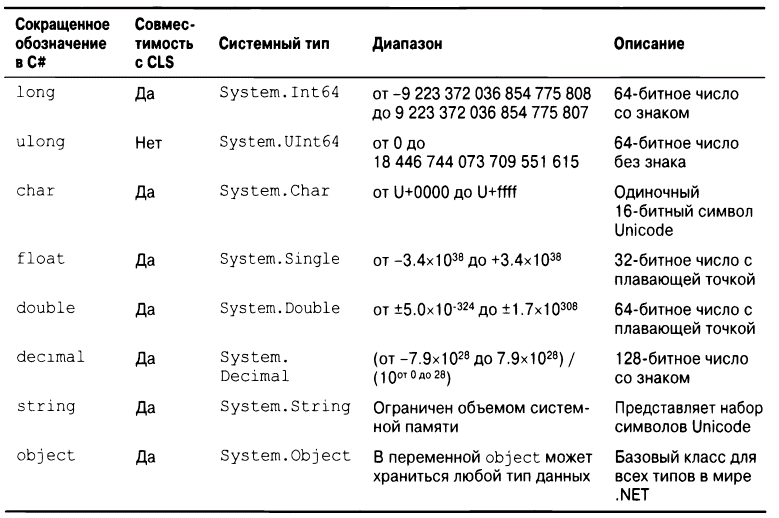
**е format: е**

**X format: 1869F**

**x format: 1869f**

Системные типы данных и соответствующие ключевые слова C#



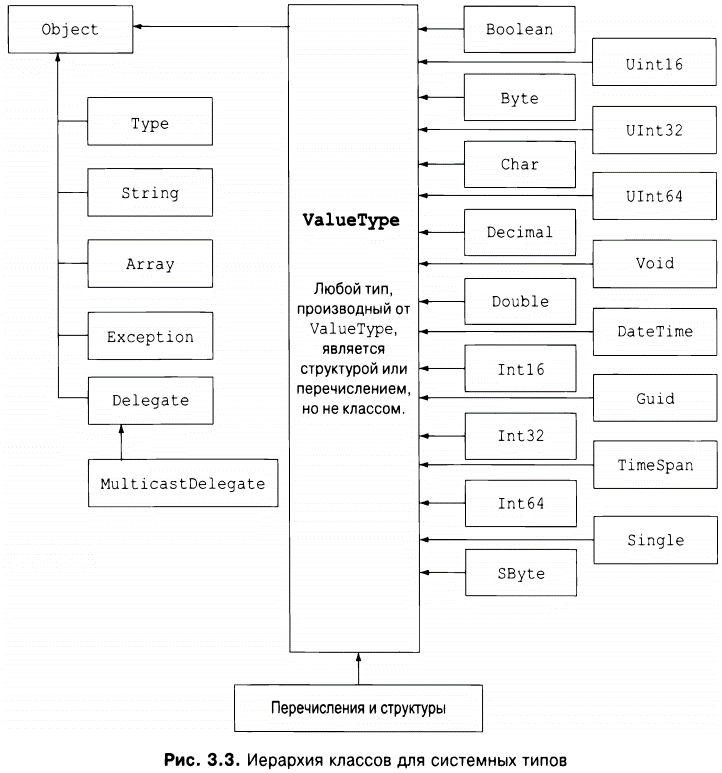


Объявление и инициализация переменных

class Program  
{  
 static void Main(string[] args)  
 {LocalVarDeclarations();  
 Console.ReadLine();  
 }  
 static void LocalVarDeclarations()  
 {  
 Console.WriteLine("=> Data Declarations:");  
 *// Локальные переменные объявляются и инициализируются так:  
 // типДанных имяПеременной - начальноеЗначение;* int myInt = 0;  
 *// Объявлять и присваивать можно также в двух отдельных строках.* string myString;  
 myString = "This is my character data";  
 *// Объявить три переменных типа bool в одной строке,* bool b1 = true, b2 = false, bЗ = b1;  
 *// Использовать тип данных System.Boolean для объявления булевской переменной.* System.Boolean b4 = false;  
 Console.WriteLine("Your data: {0}, {1}, {2}, {3}, {4}, {5}",  
 myInt, myString, b1, b2, bЗ, b4);  
 Console.WriteLine();  
 }  
}

Иерархия классов для типов данных

Типы, находящиеся в верхней части иерархии классов, предоставляют определенное стандартное поведение, которое передается производным типам.



Члены числовых типов данных

Числовые типы .NET поддерживают свойства MaxValue и MinValue, предоставляющие информацию о диапазоне значений, которые способен хранить конкретный тип.

class Program  
{  
 static void Main(string[] args)  
 {DataTypeFunctionality();  
 Console.ReadLine();  
 }  
 static void DataTypeFunctionality ()  
 {  
 Console.WriteLine("=> Data type Functionality:");  
 Console.WriteLine ( "Max of int: {0}", int.**MaxValue**);  
 Console.WriteLine ( "Min of int: {0}", int.**MinValue**);  
 Console.WriteLine("Max of double: {0}", double.**MaxValue**);  
 Console.WriteLine("Min of double: {0}", double.**MinValue**);  
 Console.WriteLine("double.Epsilon: {0}", double.**Epsilon**);  
 Console.WriteLine("double.PositiveInfinity: {0}" ,  
 double.**PositiveInfinity**);  
 Console.WriteLine("double.Negativelnfinity: {0}",  
 double.**NegativeInfinity**);  
 Console.WriteLine();  
 }  
}

=> Data type Functionality:

Max of int: 2147483647

Min of int: -2147483648

Max of double: 1,7976931348623157E+308

Min of double: -1,7976931348623157E+308

double.Epsilon: 5E-324

double.PositiveInfinity: ?

double.Negativelnfinity: -?

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

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

Char

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

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

**=> char type Functionality:**

**char.IsDigit ( 'a') : False**

**char.IsLetter ( 'a ' ) : True**

**char.IsWhiteSpace('Hello There', 5): True**

**char.IsWhiteSpace('Hello There', 6): False**

**char.IsPunctuation('?'): True**

Разбор значений из строковых данных

Перевод String в другой тип с помощью Parse

static void ParseFromStrings()  
{  
 Console.WriteLine("=> Data type parsing:");  
 bool b = bool.Parse("True");  
 Console.WriteLine("Value of b: {0} ", b) ; *// Вывод значения b* double d = double.Parse("123,789") ;  
 Console.WriteLine("Value of d: {0} ", d) ; *// Вывод значения d* int i = int.Parse("8") ;  
 Console.WriteLine ("Value of i: {0}", i) ; *// Вывод значения i* char c = Char.Parse( "w" ) ;  
 Console.WriteLine("Value of c: {0} " , c) ; *// Вывод значения с* Console.WriteLine();  
}

Использование метода TryParse () для разбора значений из строковых данных

Проблема с предыдущим кодом связана с тем, что если строка не может быть аккуратно преобразована в корректный тип данных, то сгенерируется исключение. Например, следующий код потерпит неудачу во время выполнения: **bool b = bool.Parse("Hello");**Решение предусматривает помещение каждого вызова Parse () в блок try-catch, что добавит много кода, или применение метода TryParse (). Метод TryParse () принимает параметр out и возвращает значение bool, которое указывает, успешно ли прошел разбор.

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

=> Data type parsing with TryParse:

Value of b: True

Failed to convert the input (Hello) to a double

Метод TryParse () возвращает true и присваивает разобранное значение переменной, переданной методу. В случае невозможности разбора значения переменной присваивается стандартное значение, а метод TryParse () возвращает false.

Типы System. DateTime и System. TimeSpan

DateTime содержит данные, представляющие специфичное значение даты (месяц, день, год) и времени, которые могут форматироваться разнообразными способами с применением членов этого типа.

TimeSpan позволяет легко определять и трансформировать единицы времени, используя различные ее члены.

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

=> Dates and Times:

The day of 17.10.2015 0:00:00 is Saturday

Daylight savings: False

04:30:00

04:15:00

Biglnteger

Biglnteger может применяться для представления **огромных** числовых значений, которые не ограничены фиксированным верхним или нижним пределом.

using System.Numerics;  
  
class Program  
{  
 static void Main(string[] args)  
 {UseBiglnteger();  
 Console.ReadLine();  
 }  
 static void UseBiglnteger()  
 {  
 Console.WriteLine("=> Use Biglnteger:");  
 BigInteger biggy =  
 BigInteger.Parse("9999999999999999999999999999999999999999999999");  
 Console.WriteLine("Value of biggy is {0}", biggy);  
 *// значение biggy* Console.WriteLine("Is biggy an even value?: {0}", biggy.IsEven);  
 *// biggy - четное?* Console.WriteLine("Is biggy a power of two?: {0}", biggy.IsPowerOfTwo);  
 *// biggy - степень 2?* BigInteger reallyBig = BigInteger.Multiply(biggy,  
 BigInteger.Parse("8888888888888888888888888888888888888888888"));  
 Console.WriteLine("Value of reallyBig is {0}", reallyBig);  
 *// значение reallyBig* }  
}

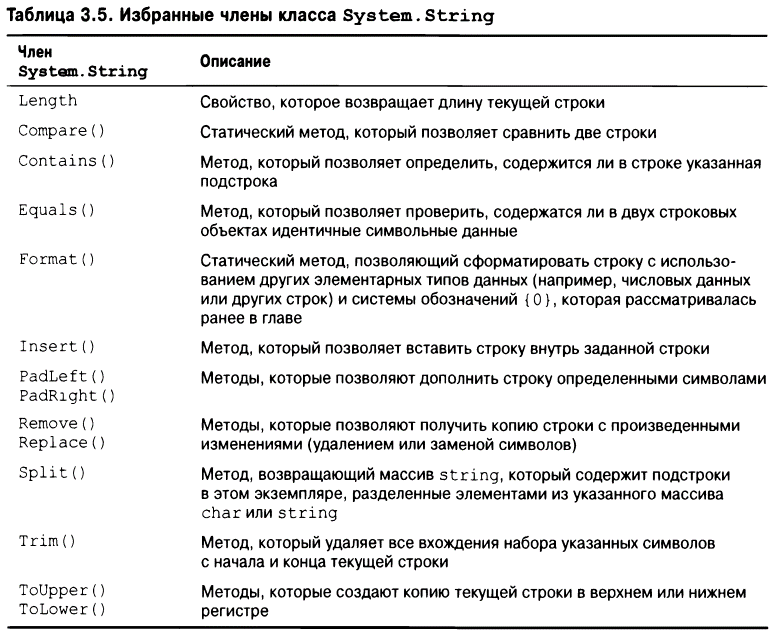
Value of biggy is 9999999999999999999999999999999999999999999999

Is biggy an even value?: False

Is biggy a power of two?: False

Value of reallyBig is 88888888888888888888888888888888888888888879991111111111111111111111111111111111111111112

Работа co строковыми данными



**Пример:**

static void BasicStringFunctionality ()  
{  
 Console.WriteLine("=> Basic String functionality:");  
 string firstName = "Freddy";  
 Console.WriteLine("Value of firstName: {0}", firstName);  
 *// Значение firstName.* Console.WriteLine("firstName has {0} characters.", firstName.Length);  
 *// Длина firstname.* Console.WriteLine("firstName in uppercase: {0}", firstName.ToUpper());  
 *// firstName в верхнем регистре.* Console.WriteLine("firstName in lowercase: {0}", firstName.ToLower ());  
 *// firstName в нижнем регистре.122 Часть II. Основы программирования на C#* Console.WriteLine("firstName contains the letter у?:{0}",firstName.Contains("y"));  
 *// Содержит ли firstName букву у?* Console.WriteLine("firstName after replace: {0}", firstName.Replace("dy", ""));  
 *// firstName после замены.* Console .WriteLine();  
}

**=> Basic String functionality:**

**Value of firstName: Freddy**

**firstName has 6 characters.**

**firstName in uppercase: FREDDY**

**firstName in lowercase: freddy**

**firstName contains the letter у?:True**

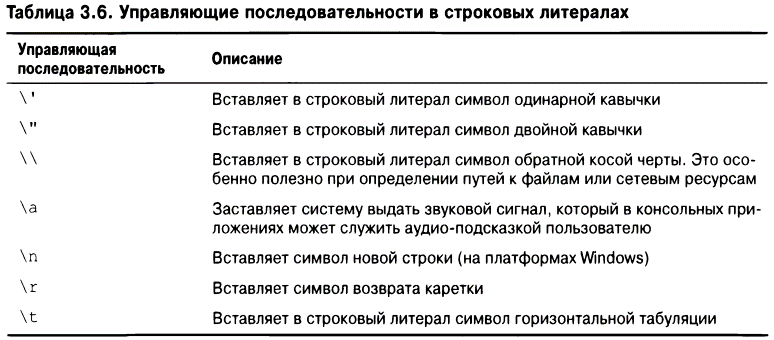
**firstName after replace: Fred**

Конкатенация строк

Переменные string могут соединяться вместе для построения строк большего размера с помощью операции + языка С#. Как вам должно быть известно, такой прием формально называется **конкатенацией строк.**

При обработке символа + компилятор C# выпускает вызов статического метода String. Concat (). В результате конкатенацию строк можно также выполнять, вызывая метод String. Concat () напрямую.

Управляющие последовательности



Определение дословных (сырых) строк

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

Т.е. это то же самое как в питоне r 'строка'

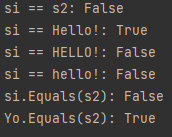
Строки и равенство

Ссылочный тип — это объект, размещаемый в управляемой куче со сборкой мусора.

По умолчанию при выполнении проверки на предмет равенства ссылочных типов (с помощью операций == и ! = языка С#) значение true будет возвращаться в случае, если обе ссылки указывают на один и тот же объект в памяти. Однако, несмотря на то, что тип string в действительности является ссылочным, операции равенства для него были переопределены так, чтобы можно былосравнивать **значения** объектов string, а не ссылки на объекты в памяти.

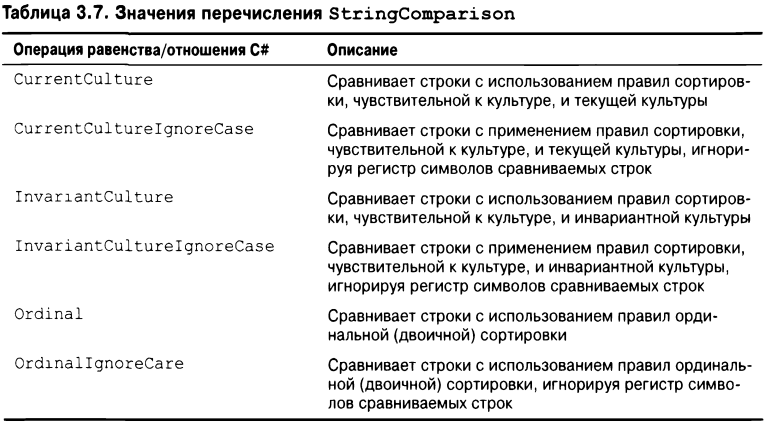
Операции равенства C# выполняют в отношении объектов string посимвольную проверку равенства с учетом регистра и нечувствительную к культуре. Следовательно, строка "Hello!" не равна строке "HELLO!" и также отличается от строки "hello!".

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

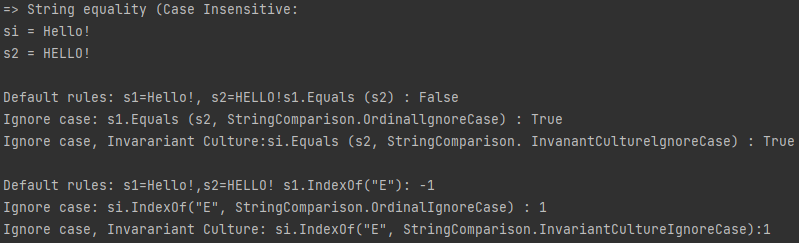
****

Модификация поведения сравнения строк

Применение перегруженных версий перечисленных ранее методов, которые принимают значение перечисления StringComparison, управляющего выполнением сравнения может сделать сравнение строк не зависящим от регистра и многое другое:

****

static void StringEqualitySpecifyingCompareRules()  
{  
Console.WriteLine("=> String equality (Case Insensitive:");  
string s1 = "Hello!";  
string s2 = "HELLO!";  
Console.WriteLine("si = {0}", s1);  
Console.WriteLine("s2 = {0}", s2);  
Console.WriteLine();  
*// Проверить результаты изменения стандартных правил сравнения.*Console .WriteLine ("Default rules: s1={0}, s2={1} s1.Equals (s2) : {2}", s1, s2, s1.Equals(s2));  
Console.WriteLine("Ignore case: s1.Equals (s2, StringComparison.OrdinallgnoreCase) : {0}",  
s1.Equals (s2, StringComparison.**OrdinalIgnoreCase**));  
Console.WriteLine("Ignore case, Invarariant Culture:si.Equals (s2, StringComparison. InvanantCulturelgnoreCase) : {0}",  
s1.Equals(s2, StringComparison.**InvariantCultureIgnoreCase**));  
Console.WriteLine ();  
Console.WriteLine("Default rules: s1={0},s2={1} s1.IndexOf(\"E\"): {2}",  
s1, s2, s1.IndexOf("E"));  
Console.WriteLine("Ignore case: si.IndexOf(\"E\", StringComparison.OrdinalIgnoreCase) : {0}",  
s1.IndexOf("E" , StringComparison.**OrdinalIgnoreCase**) ) ;  
Console.WriteLine("Ignore case, Invarariant Culture: s1.IndexOf(\"E\", StringComparison.InvariantCultureIgnoreCase):{0}",  
s1.IndexOf("E", StringComparison.**InvariantCultureIgnoreCase**));  
Console.WriteLine ();  
}

****

Строки являются неизменяемыми

Один из интересных аспектов класса **System. String** связан с тем, что после присваивания объекту **string** начального значения символьные данные **не могут быть** изменены.

Методы типа **string** на самом деле возвращают новый объект string в модифицированном виде.

Класс string может стать неэффективным и при неправильном употреблении приводить к “разбуханию” кода, особенно при выполнении конкатенации строк или при работе с большимиобъемами текстовых данных. Когда строится приложение, в котором текстовые данные будут часто изменяться (подобное текстовому процессору), то представление обрабатываемых текстовыхданных с применением объектов string будет неудачным решением, т.к. оно практически наверняка (и часто косвенно) приведет к созданию излишних копий строковых данных.

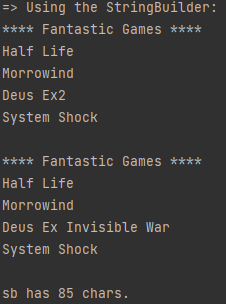
Тип System.Text.StringBuilder

С учетом того, что тип string может оказаться неэффективным при безответственном использовании, библиотеки базовых классов .NET предоставляют пространство имен System. Text.

Уникальность класса StringBuilder в том, что при вызове его членов производится прямое изменение внутренних символьных данных объекта (делая его более эффективным) без получения копии данных в модифицированном формате.

using System.Text;

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

****

Интерполяция строк

static void Stringlnterpolation()  
{  
 *// Некоторые локальные переменные будут включены в крупную строку.* int age = 4;  
 string name = "Soren";  
 *// Использование синтаксиса с фигурными скобками.* string greeting = string.Format("Hello {0} you are {1} years old.", name, age);  
 *// Использование интерполяции строк.* string greeting2 = $"Hello {name} you are {age} years old.";  
}

В переменной greeting2 обратите внимание на то, что конструируемая строка начинается с префикса $.

Фигурные скобки по-прежнему используются для пометки заполнителя под переменную; тем не менее, вместо применения числовой метки имеется возможность указывать непосредственно переменную.

Сужающие и расширяющиепреобразования типов данных

class Program  
{  
 static void Main(string[] args)  
 {  
 Console.WriteLine ("\*\*\*\*\* Fun with type conversions \*\*\*\*\*") ;  
*// Добавить две переменные типа short и вывести результат,* short numb1 = 9, numb2 = 10;  
 Console.WriteLine("{0} + {1} = {2}",  
 numb1, numb2, Add(numb1, numb2));  
 Console.ReadLine ();  
 }  
  
 static int Add(int x, int y)  
 {  
 return x + y;  
 }  
}

****

Легко заметить, что метод Add () ожидает передачи двух параметров int. Тем не менее, в методе Main () ему на самом деле передаются две переменные типа short.

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

Несмотря на то что неявное расширение типов благоприятствовало в предыдущем примере, в других ситуациях оно может стать источником ошибок на этапе компиляции. Например, пусть для переменных numb1 и numb2 установлены значения, которые (при их сложении) превышают максимальное значение типа short. Кроме того, предположим, что возвращаемое значение метода Add () сохраняется в новой локальной переменной short, а не напрямую выводится на консоль.

В данном случае компилятор сообщит об ошибке. Среде CLR не удалось применить **сужающую операцию.** Нетрудно догадаться, что сужающая операция является логической противоположностью расширяющей операции, поскольку предусматривает сохранение большего значения внутри переменной типа данных с меньшим диапазоном допустимых значений.

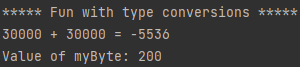
Важно отметить, что все сужающие преобразования приводят к ошибкам на этапе компиляции, даже когда есть основание полагать, что такое преобразование должнопройти успешно. Например, следующий код также вызовет ошибку при компиляции:

*// Снова ошибка на этапе компиляции!*static void NarrowingAttempt()  
{  
 byte myByte = 0;  
 int mylnt = 200;  
 myByte = mylnt;  
 Console.WriteLine("Value of myByte: {0}", myByte);  
}

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

Если нужно проинформировать компилятор о том, что вы готовы мириться с возможной потерей данных из-за сужающей операции, тогда потребуется применить **явное приведение,** используя операцию приведения () языка С#.

class Program  
{  
 static void Main(string[] args)  
 {  
 Console.WriteLine("\*\*\*\*\* Fun with type conversions \*\*\*\*\*");  
 short numb1 = 30000, numb2 = 30000;  
 *// Явно привести int к short (и разрешить потерю данных).* short answer = (short)Add(numb1, numb2);  
 Console.WriteLine("{0} + {1} = {2}",  
 numb1, numb2, answer);  
 NarrowingAttempt();  
 Console.ReadLine();  
 }  
  
 static int Add(int х, int у)  
 {  
 return х + у;  
 }  
 static void NarrowingAttempt()  
 {  
 byte myByte = 0;  
 int myInt = 200;  
 *// Явно привести int к byte (без потери данных) .* myByte = (byte)myInt;  
 Console.WriteLine("Value of myByte: {0}", myByte);  
 }  
}

****

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

Ключевое слово checked

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

По умолчанию, если  
не предпринимаются никакие корректирующие действия, то условия переполнения и  
потери значимости происходят без выдачи сообщений об ошибках.

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

static void ProcessBytes()  
{  
 byte b1 = 100;  
 byte b2 = 250;  
 *//На этот раз сообщить компилятору о необходимости добавления  
 // кода CIL, необходимого для генерации исключениях если возникает  
 // переполнение или потеря значимости.* try  
 {  
 byte sum = checked((byte)Add(b1, b2));  
 Console.WriteLine("sum = {0}", sum);  
 }  
 catch (OverflowException ex) *// если выдается ошибка OverflowException* {  
 Console.WriteLine(ex.Message);  
 }  
}

Поскольку значение sum выходит за пределы допустимого  
диапазона для типа byte, генерируется исключение времени выполнения. Сообщение  
об ошибке выводится посредством свойства Message:

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

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

static void ProcessBytes2()  
{  
 byte b1 = 100;  
 byte b2 = 250;  
 try  
 {  
 checked  
 {  
 byte sum = (byte)Add(b1, b2);  
 Console.WriteLine("sum = {0}", sum);  
 }  
 }  
 catch (OverflowException ex)  
 {  
 Console.WriteLine(ex.Message);  
 }  
}

Настройка проверки переполнения на уровне проекта

В качестве альтернативы компилятор C#  
поддерживает флаг /checked. Когда он указан, все присутствующие в коде арифметические операции будут оцениваться на предмет переполнения, не требуя применения  
ключевого слова checked.

Ключевое слово unchecked

предположим, что проверка переполнения и потери значимости включена  
в масштабах проекта, но есть блок кода, в котором потеря данных приемлема.

Ключевое слово unchecked предназначено  
для отмены генерации исключений, связанных с переполнением, в отдельных случаях.

unchecked используется аналогично checked, т.е. его можно применять  
как к единственному оператору, так и к блоку операторов.

unchecked  
{  
 byte sum = (byte)(b1 + b2);  
 Console.WriteLine("sum = {0}", sum);  
}

Понятие неявно типизированныхлокальных переменных

Язык C# поддерживает возможность неявной типизации локальных переменных с использованием ключевого слова var.

Ключевое слово var может применяться вместо указания конкретного  
типа данных (такого как int, bool или string).

Когда вы поступаете подобным образом,  
компилятор будет автоматически выводить лежащий в основе тип данных на основе начального значения, используемого для инициализации локального элемента данных.

Определение типа переменной:

peremennaya.GetType()

Ограничения неявно типизированных переменных

Неявная типизация применима только к локальным переменным внутри области видимости метода или свойства.

Локальным переменным, которые объявлены с ключевым словом var,  
обязано присваиваться начальное значение в самом объявлении, причем присваивать null в качестве начального значения невозможно.

static void kek()  
{  
 *// Ошибка! Должно быть присвоено значение!* var myData;  
 *// Ошибка! Значение должно присваиваться в самом объявлении!* var mylnt;  
 mylnt = 0;  
 *// Ошибка! Нельзя присваивать null в качестве начального значения!* var myObj = null;  
}

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

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

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

*// Также нормально!*var myInt = 0;  
var anotherInt = myInt;  
string myString = "Wake up!";  
var myData = myString;

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

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

Полезность неявно типизированных локальных переменных

Использование var для объявления локальных переменных  
просто ради интереса особой пользы не принесет. Такой подход может вызвать путаницу у тех, кто будет изучать код, поскольку лишает возможности быстро определить лежащий в основе тип данных и, следовательно, затрудняет понимание общего назначения переменной. Поэтому если вы знаете, что переменная должна относиться к типу int, то сразу и объявляйте ее с типом int!

Фактически можно было бы даже утверждать, что единственным случаем, когда применение ключевого слова var полностью оправдано, является определение данных, возвращаемых из запроса LINQ.

Итерационные конструкции C#

В C# предоставляются четыре итерационные конструкции:  
• цикл for:  
• цикл foreach/in;  
• цикл while:  
• цикл do/while.

Цикл for

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

Цикл foreach

Ключевое слово foreach языка C# позволяет проходить в цикле по всем элементам внутри контейнера без необходимости в проверке верхнего предела.

В отличие от цикла for цикл foreach будет выполнять проход по контейнеру только линейным (п+1) образом (т.е. не получится проходить

*// Проход по элементам массива посредством foreach.*static void ForEachLoopExample()  
{  
 string [] carTypes = {"Ford", "BMW", "Yugo", "Honda" };  
 foreach (string c in carTypes)  
 Console.WriteLine(c) ;  
 int [] myInts = { 10, 20, 30, 40 };  
 foreach (int i in myInts)  
 Console.WriteLine(i);  
}

Циклы while и do/while

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

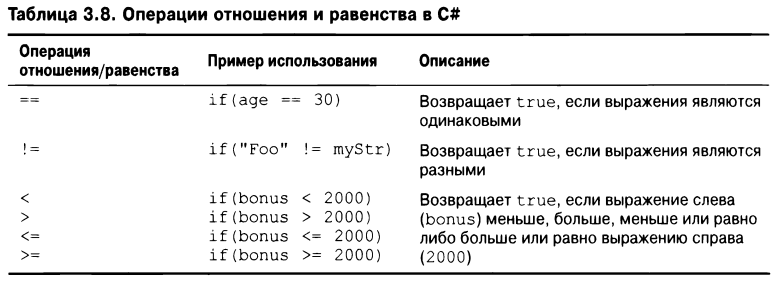
Подобно простому циклу while  
цикл do/while используется, когда какое-то действие должно выполняться неопределенное количество раз. Разница в том, что цикл do/while гарантирует, по крайней  
мере, однократное выполнение своего внутреннего блока кода. С другой стороны, вполне возможно, что цикл while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

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

Конструкции принятия решений и операции отношения/равенства

в C# определены две простые конструкции:  
• оператор if/else;  
• оператор switch.

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



*// Недопустимо, т.к. свойство Length возвращает int, а не bool,*string stringData = "Му textual data";  
if (stringData.Length)  
{  
}

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

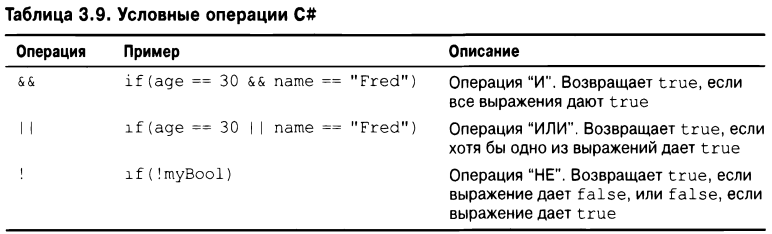
*// Допустимо, т.к. условие возвращает true или false,*if (stringData.Length > 0)  
{  
 Console.WriteLine("string is greater than 0 characters");  
}

Условная операция

Условная операция (?:) является сокращенным способом написания простого оператора if-else. Вот ее синтаксис:  
условие ? первое\_выражение : второе\_выражение;  
Условие представляет собой условную проверку (часть if оператора if-else). Если  
проверка проходит успешно, тогда выполняется код, следующий сразу за знаком вопроса (?). Если результат проверки отличается от true, то выполняется код, находящийся  
после двоеточия (часть else оператора if-else). Приведенный ранее пример кода можно было бы переписать с применением условной операции:

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

С условной операцией связаны некоторые ограничения. Во-первых, типы конструкций первое\_выражение и второе выражение должны быть одинаковыми. Во-вторых, условная операция может использоваться только в операторах присваивания.



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

Оператор switch

Оператор switch позволяет организовать выполнение программы на основе заранее определенного набора вариантов (блок default обрабатывает недопустимый выбор):

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

Модифицированная версия оператора switch, которая оценивает переменную типа string:

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

Использование сопоставления с образцом в операторах switch (нововведение)

В C# 7 операторы switch способны также задействовать  
образец с типами, при котором операторы case могут оценивать тип проверяемой переменной, и выражения case больше не ограничиваются константными значениями.

static void ExecutePatternMatchingSwitch()  
{  
 Console.WriteLine("1 [Integer (5)], 2 [String (\"Hi\")]z 3 [Decimal(2.5)]");  
 Console.Write("Please choose an option: ");  
 string userChoice = Console.ReadLine();  
 object choice;  
*// Стандартный оператор switch, в котором применяется  
// сопоставление с образцом с константами* switch (userChoice)  
 {  
 case "1":  
 choice = 5;  
 break;  
 case "2":  
 choice = "Hi";  
 break;  
 case "3":  
 choice = 2.5;  
 break;  
 default:  
 choice = 5;  
 break;  
 }  
  
 *// Новый оператор switch, в котором применяется  
 // сопоставление с образцом с типами* switch (choice)  
 {  
 case int i:  
 Console.WriteLine("Your choice is an integer.");  
 *// Выбрано целое число* break;  
 case string s:  
 Console.WriteLine("Your choice is a string.");  
 *// Выбрана строка* break;  
 case decimal d:  
 Console.WriteLine("Your choice is a decimal.");  
 *// Выбрано десятичное число* break;  
 default:  
 Console.WriteLine("Your choice is something else");  
 *// Выбрано что-то другое* break;  
 }  
  
 Console.WriteLine();  
}