Программирование на языке C#

В 2000 году компания Microsoft объявила о создании нового языка
программирования - языка C#.
Главным инструментом создания приложений для платформы .Net является ***Microsoft Visual Studio***, которая имеет множество редакций.

Платформа .NET по сути представляла собой новую модель создания приложений, которая включает в себя следующие возможности:

1) использование библиотеки базовых классов, предлагающих целостную объектно-ориентированную модель программирования для всех языков программирования, поддерживающих .NET;
2) полное и абсолютное межъязыковое взаимодействие, позволяющее разрабатывать фрагменты одного и того же проекта на различных языках программирования;
3) общая среда выполнения приложений .NET, независимо от того, на каких языках программирования для данной платформы они были созданы; при этом среда берет на себя контроль за безопасностью выполнения приложений и управление ресурсами;
4) упрощенный процесс развертывания приложения, в результате чего установка приложения может свестись к простому копированию файлов приложения в определенный каталог.

Одним из основных элементов .NET Framework является библиотека классов под общим именем FCL (Framework Class Library), к которой можно обращаться из различных языков программирования, в частности, из C#.

Common Language Runtime - CLR

Кроме FCL в состав платформы .NET входит Common Language Runtime (CLR — единая среда выполнения программ), название которой говорит само за себя - это среда ответственна за поддержку выполнения всех типов приложений, разработанных на различных языках программирования с использованием библиотек .NET.

Среда CLR берет на себя всю низкоуровневую работу, например, автоматическое управление памятью.

Среда CLR обеспечивает интеграцию языков и позволяет объектам, созданным на одном языке, использовать объекты, написанные на другом. Такая интеграция возможна благодаря стандартному набору типов и информации, описывающей тип (метаданным).

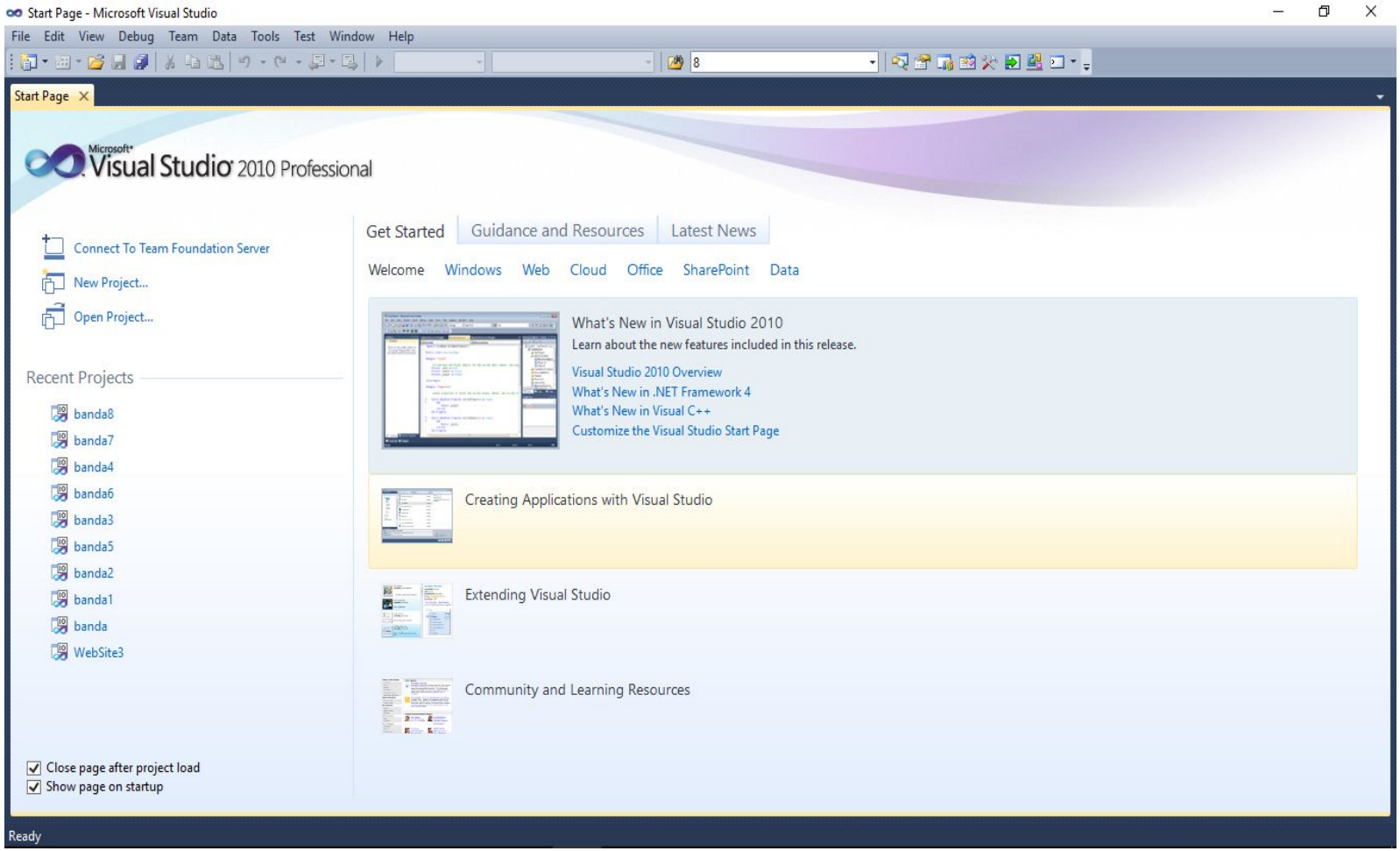
Первый проект в среде Visual Studio

Текст программы:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace pr1
{    class Program
   {       static void Main(string[] args)
      { double x,y,f;
         string s;
         Console.Write("x=");
         s=Console.ReadLine();
         x = Convert.ToDouble(s);
         y = Math.Sin(0.1 * x) + Math.Cos(x);
         f = Math.Tan(1 / x) + 3;
         Console.Write("y={0}    f ={1} ",y, f);
         Console.ReadKey();
      }   }    }
```

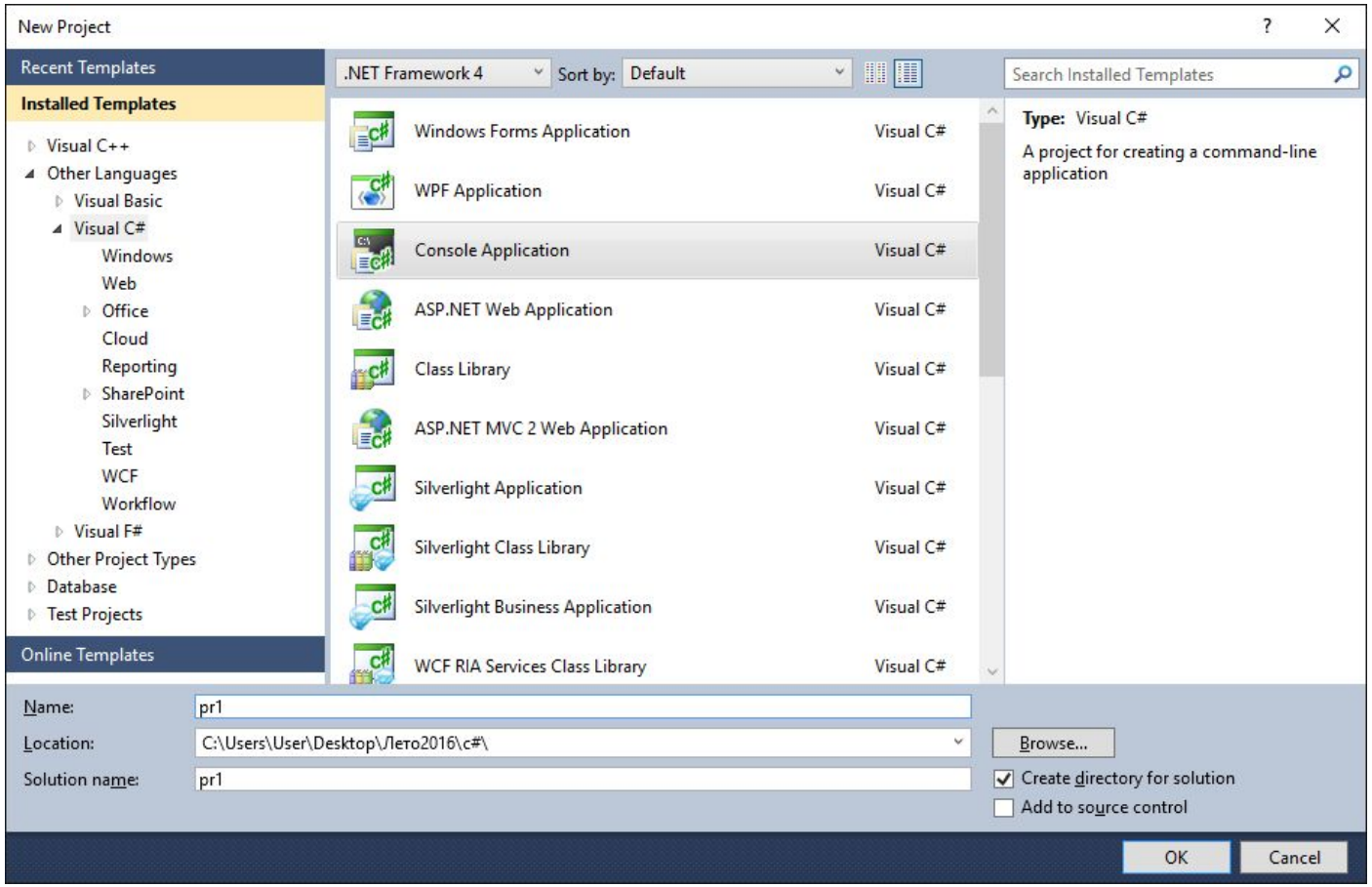
Первая программа

Запускаем Visual Studio



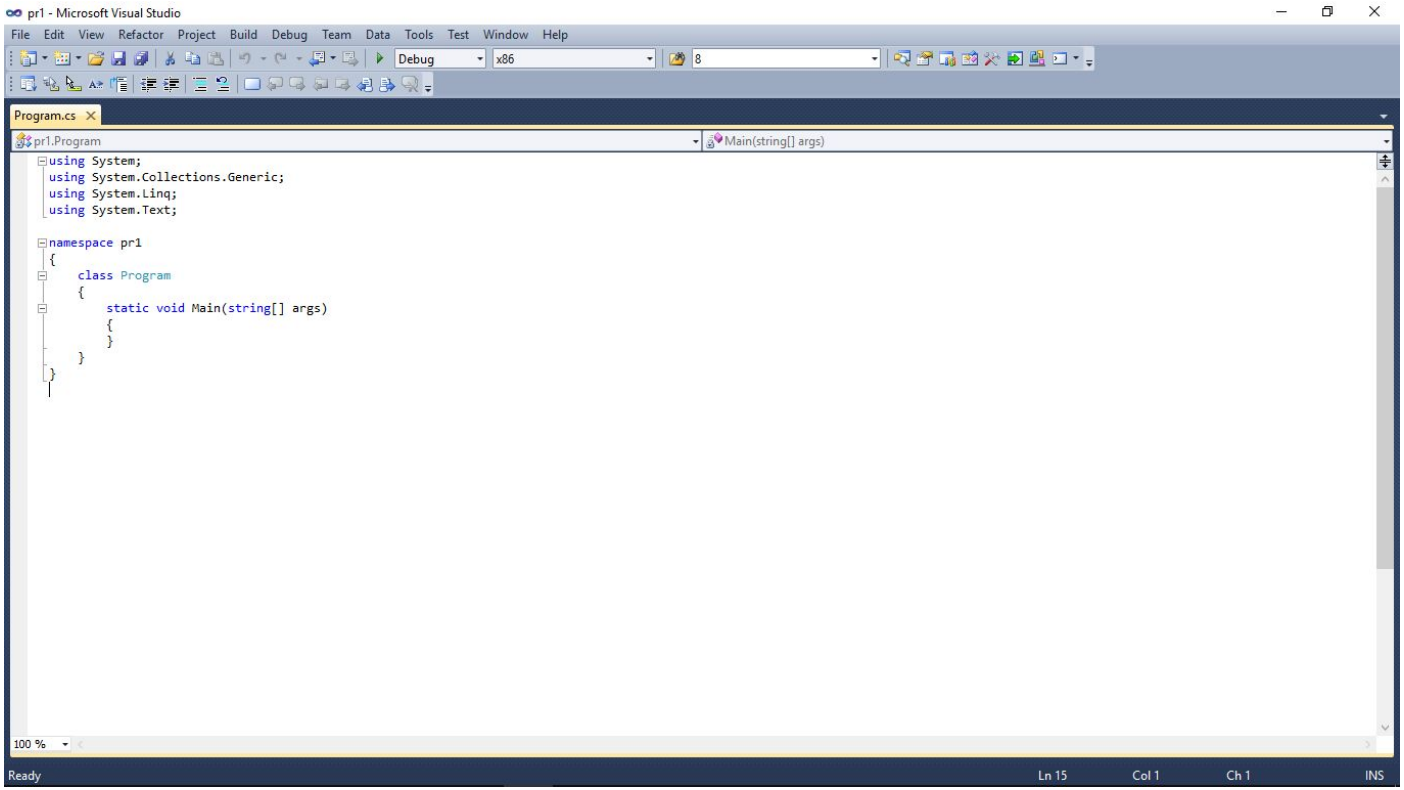
Выбираем New Project

Первая программа



В строке *Name* вводим имя *pr1*, выбираем папку для размещения проекта. Щелкаем по кнопке ОК

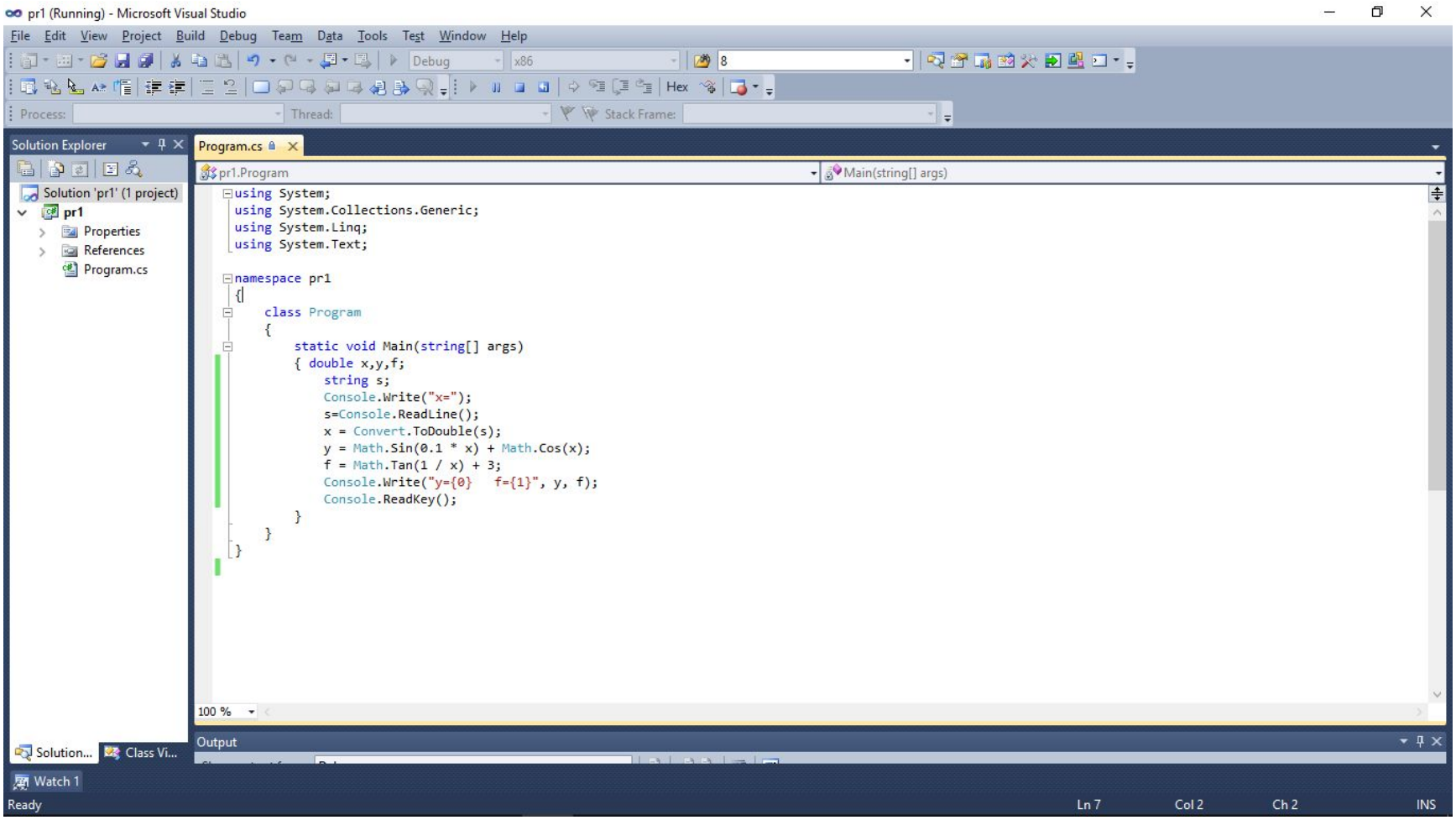
Первая программа



Вводим текст программы

Первая программа

Вводим текст программы



Первая программа

Рассмотрим текст программы:

using System – это директива, которая разрешает использовать имена стандартных классов из пространства имен System непосредственно, без указания имени пространства, в котором они были определены. Так, например, если бы этой директивы не было, то пришлось писать бы ***System.***Console.WriteLine. Писать полное пространство имен каждый раз очень неудобно. При указании директивы using можно писать просто имя, например, Console.WriteLine.

Для консольных программ ключевое слово **namespace** создает свое собственное пространство имен, которое по умолчанию называется именем проекта. В нашем случае пространство имен называется ***pr1***.

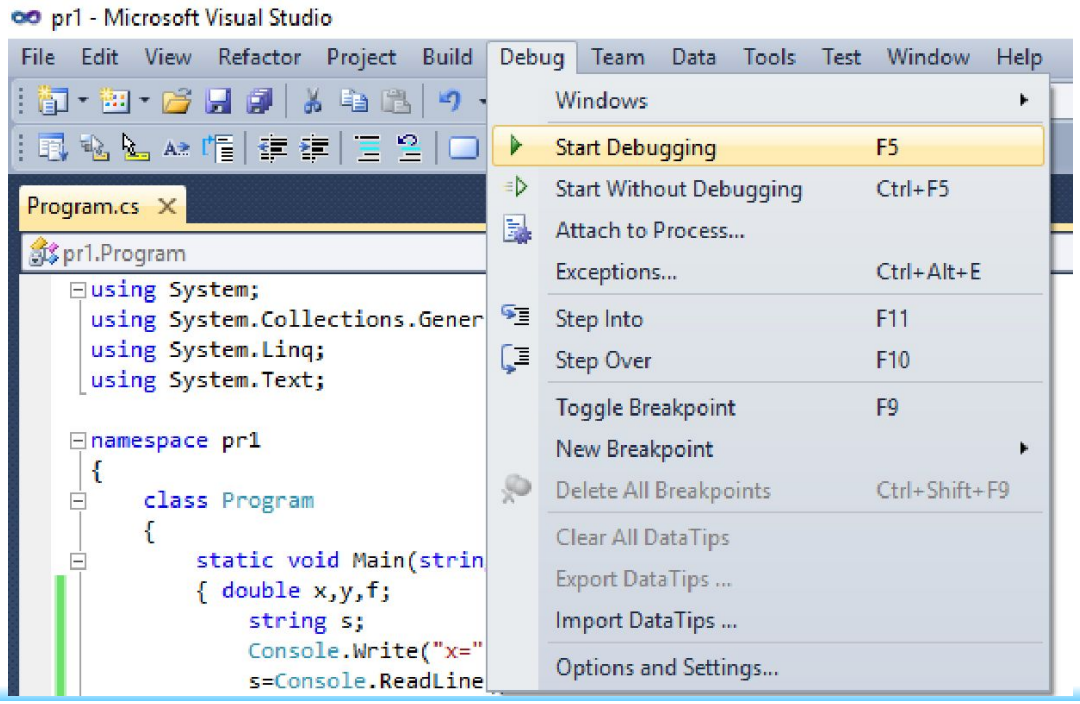
Каждое имя, которое встречается в программе, должно быть уникальным. В больших и сложных приложениях используются библиотеки разных производителей. В этом случае трудно избежать конфликта между используемыми в них именами. Пространства имен предоставляют простой механизм предотвращения конфликтов имен. Они создают разделы в глобальном пространстве имен.

Первая программа

C# – объектно-ориентированный язык, поэтому написанная на нем программа будет представлять собой совокупность взаимодействующих между собой классов.

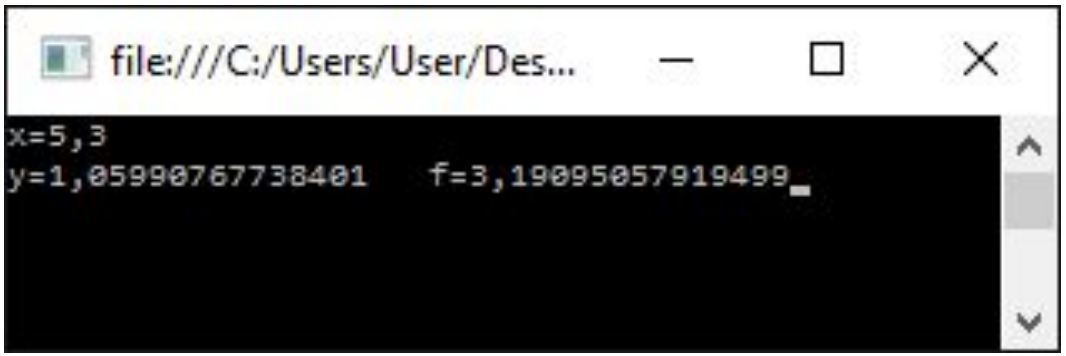
Автоматически создан класс с именем **Program**. Данный класс содержит только один метод – метод **Main()**, который является точкой входа в программу. Это означает, что именно с данного метода начнется выполнение приложения. Каждая консольная программа на языке C# должна иметь метод Main ().

Для запуска программы щелкнем по кнопке  , вместо кнопки можно нажать клавишу *F5* или выполнить команду ⇒



Первая программа

Запускаем программу, вводим исходные данные



Результаты работы приложения

Состав языка

Алфавит – совокупность допустимых в языке символов.

Алфавит языка C# включает:

- прописные и строчные латинские буквы и буквы национальных алфавитов (включая кириллицу);
- арабские цифры от 0 до 9, шестнадцатеричные цифры от A до F;
- специальные знаки:

  " { } , | ; [ ] ( ) + - / % * . \ ' : ? < = > ! & ~ ^ @ _
- пробельные символы: пробел, символ табуляции, символ перехода на новую строку.

Из символов алфавита формируются лексемы (элементы) языка: идентификаторы, ключевые (зарезервированные) слова, знаки операций, константы, разделители (скобки, точка, запятая, пробельные символы).

Состав языка

Идентификатор – это имя программного элемента: константы, переменной, метки, типа, класса, объекта, метода и т.д. Идентификатор может включать латинские буквы и буквы национальных алфавитов, цифры и символ подчеркивания. Прописные и строчные буквы различаются, например, myname, myName и MyName — три различных имени.

Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Пробелы и другие разделители внутри имен не допускаются. Язык C# не налагает никаких ограничений на длину имен, однако для удобства чтения и записи кода не стоит делать их слишком длинными.

Ключевые слова – это зарезервированные идентификаторы, которые имеют специальное значение для компилятора, например, static, int и т.д. Ключевые слова можно использовать только по прямому назначению. Однако если перед ключевым словом поставить символ @, например, @int, @static, то полученное имя можно использовать в качестве идентификатора.

Система типов

В C# типы делятся на три группы:

- базовые типы – предлагаемые языком;
- типы, определяемые пользователем;
- анонимные типы - типы, которые автоматически создаются на основе инициализаторов объектов.

Кроме того, типы C# разбиваются на две другие категории:

- размерные типы (value type)
- ссылочные типы (reference type).

Принципиальное различие между размерными и ссылочными типами состоит в способе хранения их значений в памяти. В первом случае фактическое значение хранится в стеке (или как часть большого объекта ссылочного типа). Адрес переменной ссылочного типа тоже хранится в ***стеке***, но сам объект хранится в ***куче***.

Система типов

Стек – это структура, используемая для хранения элементов по принципу LIFO (Last input – first output или первым пришел - последним ушел). Под стеком понимается область памяти, обслуживаемая процессором, в которой хранятся значения локальных переменных.

Куча – область памяти, используемая для хранения данных, работа с которыми реализуется через указатели и ссылки. Память для размещения таких данных выделяется программистом динамически, а освобождается сборщиком мусора.

Сборщик мусора уничтожает программные элементы в стеке через некоторое время после того, как закончит существование раздел стека, в котором они объявлены. То есть, если в пределах блока (фрагмента кода, помещенного в фигурные скобки {}) объявлена локальная переменная, соответствующий программный элемент будет удален по окончании работы данного блока. Объект в куче подвергается сборке мусора через некоторое время после того, как уничтожена последняя ссылка на него.

Система типов

Стандарт языка включает следующий набор фундаментальных типов.

- Логический тип (bool).
- Символьный тип (char).
- Целые типы. Целые типы могут быть одного из трех размеров - short, int, long, сопровождаемые описателем signed или unsigned, который указывает, как интерпретируется значение, - со знаком или без оного.
- Типы с плавающей точкой. Эти типы также могут быть одного из трех размеров - float, double, long double.
- Тип void, используемый для указания на отсутствие информации.
- Указатели (например, int* - типизированный указатель на переменную типа int).
- Ссылки (например, double& - типизированная ссылка на переменную типа double).
- Массивы (например, char[] - массив элементов типа char).

Язык позволяет конструировать пользовательские типы

- Перечислимые типы (enum) для представления значений из конкретного множества.
- Структуры (struct).
- Классы.

Система типов

Первые три вида типов называются ***интегральными*** или ***счетными***. Значения их перечислимы и упорядочены.

Согласно классификации все типы можно разделить на четыре категории:

- Типы-значения (value), или значимые типы.
- Ссылочные (reference).
- Указатели (pointer).
- Тип void.

Особый статус имеет и тип void, указывающий на отсутствие какого-либо значения.

В языке C# жестко определено, какие типы относятся к ссылочным, а какие - к значимым. К ***значимым*** типам относятся: **логический, арифметический, структуры, перечисление**.

Массивы, строки и классы относятся к ***ссылочным*** типам.

Структуры C# представляют частный случай класса. Определив свой класс как структуру, программист получает возможность отнести класс к значимым типам.

Система типов

Все встроенные типы C# совпадают с системными типами каркаса Net Framework, размещенными в пространстве имен System. Поэтому всюду, где можно использовать имя типа, например, - int, с тем же успехом можно использовать и имя System.Int32.

Логический тип			
Имя типа	Системный тип	Значения	Размер
bool	System.Boolean	true, false	8 бит
Арифметические целочисленные типы			
Имя типа	Системный тип	Диапазон	Размер
sbyte	System.SByte	-128 — 127	Знаковое, 8 Бит
byte	System.Byte	0 — 255	Беззнаковое, 8 Бит
short	System.Short	-32768 —32767	Знаковое, 16 Бит
ushort	System.UShort	0 — 65535	Беззнаковое, 16 Бит
int	System.Int32	(-2*10^9 — 2*10^9)	Знаковое, 32 Бит
uint	System.UInt32	(0 — 4*10^9)	Беззнаковое, 32 Бит
long	System.Int64	(-9*10^18 — 9*10^18)	Знаковое, 64 Бит
ulong	System.UInt64	(0— 18*10^18)	Беззнаковое, 64 Бит

Система типов

Арифметический тип с плавающей точкой			
Имя типа	Системный тип	Диапазон	Точность
float	System.Single	+1.5*10^-45 -/+3.4*10^38	7 цифр
double	System.Double	+5.0*10^-324 -/+1.7*10^308	15-16 цифр

Арифметический тип с фиксированной точкой			
Имя типа	Системный тип	Диапазон	Точность
decimal	System.Decimal	+1.0*10^-28 - +7.9*10^28	28-29 значащих цифр

Символьные типы			
Имя типа	Системный тип	Диапазон	Точность
char	System.Char	U+0000 - U+ffff	16 бит Unicode символ
string	System.String	Строка из символов Unicode	

Объектный тип		
Имя типа	Системный тип	Примечание
object	System.Object	Прародитель всех встроенных и пользовательских типов

Переменные и константы

Переменная представляет собой типизированную область памяти. Программист создает переменную, объявляя ее тип и указывая имя.
При объявлении переменной ее можно инициализировать (присвоить ей начальное значение), а затем в любой момент ей можно присвоить новое значение, которое заменит собой предыдущее.

В языке C# требуется, чтобы переменные были явно проинициализированы до их использования.

Переменные и константы

Константа, в отличие от переменной, не может менять свое значение. Константы бывают трех видов:

- литералы
- типизированные константы
- перечисления.

Число ***32*** является литеральной константой. Его значение всегда равно 32 и его нельзя изменить.

Типизированные константы именуют постоянные значения. Объявление типизированной константы происходит следующим образом:

const <тип> <идентификатор> = <значение>;

Переменные и константы

Перечисление - это особый размерный тип, состоящий из набора именованных констант (называемых списком перечисления). Синтаксис объявления перечисления следующий:

[атрибуты] [модификаторы] enum <имя> [: базовый тип] {список-перечисления констант(через запятую)};

Базовый тип - это тип самого перечисления.

Если не указать базовый тип, то по умолчанию будет использован тип int.

В качестве базового типа можно выбрать любой целый тип, кроме char.

Организация ввода-вывода данных. Форматирование.

Программа при вводе данных и выводе результатов взаимодействует с внешними устройствами. Совокупность стандартных устройств ввода (клавиатура) и вывода (экран) называется консолью. В языке C# нет операторов ввода и вывода. Вместо них для обмена данными с внешними устройствами используются специальные классы. В частности, для работы с консолью используется стандартный класс **Console**, определенный в пространстве имен **System**.

Вывод данных

Вывод данных на экран может выполняться с помощью метода WriteLine, реализованного в классе Console. Существует несколько способов применения данного метода:

- на экран выводится значение идентификатора x

  Console.WriteLine(x);

- на экран выводится строка, образованная последовательным слиянием строки "x=", значения x, строки "y=" и значения y

  Console.WriteLine("x=" + x +"y=" + y);

- на экран выводится строка, формат которой задан первым аргументом метода, при этом вместо параметра {0} выводится значение x, а вместо {1} – значение y

  Console.WriteLine("x={0} y={1}", x, y);

Вывод данных

Пусть нам дан следующий фрагмент программы:

```
int i=3, j=4;
Console.WriteLine("{0} {1}", i, j);
```

При обращении к методу WriteLine через запятую перечисляются три аргумента: "{0} {1}", i, j. Первый аргумент "{0} {1}" определяет формат выходной строки. Следующие аргументы нумеруются с нуля, так переменная i имеет номер 0, j – номер 1. Значение переменной i будет помещено в выходную строку на место параметра {0}, а значение переменной j - на место параметра {1}.

В результате на экран будет выведена строка: 3 4.

Использование управляющих последовательностей

Управляющей последовательностью называют определенный символ, предваряемый обратной косой чертой. Данная совокупность символов интерпретируется как одиночный символ и используется для представления кодов символов, не имеющих графического обозначения (например, символа перевода курсора на новую строку) или символов, имеющих специальное обозначение в символьных и строковых константах.

Вид	**Наименование**
\a	Звуковой сигнал
\b	Возврат на шаг назад
\f	Перевод страницы
\n	Перевод строки
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\\	Обратная косая черта
\'	Апостроф
\"	Кавычки

Управление размером поля вывода

Первым аргументом WriteLine указывается строка вида ***{n, m}*** – где ***n*** определяет номер идентификатора из списка аргументов метода WriteLine, а ***m*** – количество позиций (размер поля вывода), отводимых под значение данного идентификатора. При этом значение идентификатора выравнивается по правому краю. Если выделенных позиций для размещения значения идентификатора окажется недостаточно, то автоматически добавится необходимое количество позиций.

Пример:

```
static void Main() {     double x= Math.E;
Console.WriteLine("E={0,20}", x);
Console.WriteLine("E={0,10}", x); }
```

Управление размещением вещественных данных

Первым аргументом WriteLine указывается строка вида ***{n: ##.###}*** – где ***n*** определяет номер идентификатора из списка аргументов метода WriteLine, а ***##.###*** определяет формат вывода вещественного числа. В данном случае, под целую часть числа отводится две позиции, под дробную – три. Если выделенных позиций для размещения целой части значения идентификатора окажется недостаточно, то автоматически добавится необходимое количество позиций.

Пример:

```
static void Main()
{     double x= Math.E;
Console.WriteLine("E={0:##.###}", x);
Console.WriteLine("E={0:.####}", x);
}
```

Управление форматом числовых данных

Первым аргументом WriteLine указывается строка вида :

{n:<спецификатор>m}

где ***n*** определяет номер идентификатора из списка аргументов метода WriteLine,

<спецификатор> - определяет формат данных,

m – количество позиций для дробной части значения идентификатора.

Спецификаторы

Параметр	*Формат*	*Значение*
C или c	Денежный. По умолчанию ставит денежный знак, определенный текущими региональными настройками. В русской Windows это р.	Задается количество десятичных разрядов.
D или d	Целочисленный (используется только с целыми числами)	Задается минимальное количество цифр. При необходимости результат дополняется начальными нулями
E или e	Экспоненциальное представление чисел	Задается количество символов после запятой. По умолчанию используется значение 6.
F или f	Представление чисел с фиксированной точкой	Задается количество символов после запятой
G или g	Общий формат (или экспоненциальный, или с фиксированной точкой)	Задается количество символов после запятой. По умолчанию выводится целая часть
N или n	Стандартное форматирование с использованием запятых и пробелов в качестве разделителей между разрядами	Задается количество символов после запятой. По умолчанию – 2, если число целое, то ставятся нули
X или x	Шестнадцатеричный формат	
P или p	Процентный	

Пример

```
static void Main()
{
Console.WriteLine("C Format:{0,14:C} \t{0:C2}", 12345.678);
Console.WriteLine("D Format:{0,14:D} \t{0:D6}", 123);
Console.WriteLine("E Format:{0,14:E} \t{0:E8}", 12345.6789);
Console.WriteLine("G Format:{0,14:G} \t{0:G10}",
12345.6789);
Console.WriteLine("N Format:{0,14:N} \t{0:N4}", 12345.6789);
Console.WriteLine("X Format:{0,14:X} ", 1234);
Console.WriteLine("P Format:{0,14:P} ", 0.9);
}
```

Ввод данных

Для ввода данных обычно используется метод ***ReadLine***, реализованный в классе ***Console***.

Данный метод в качестве результата возвращает строку, тип которой ***string***.

```
static void Main()
{   string s = Console.ReadLine();   Console.WriteLine(s); }
```

Для того чтобы получить *числовое значение*, необходимо воспользоваться преобразованием данных.

```
static void Main()
 {        string s = Console.ReadLine();
     int x = int.Parse(s); //преобразование строки в число
Console.WriteLine(x);
}
```

Ввод данных

Для преобразования строкового представления целого числа в тип ***int*** используем метод ***Parse()***, который реализован для всех числовых типов данных. Таким образом, если потребуется преобразовать строковое представление в вещественное, можно воспользоваться методом ***float.Parse()*** или ***double.Parse().***

```
static void Main() {
    double  x =  double.Parse(Console.ReadLine());
Console.WriteLine(x);
}
```

Выражения

Выражением называется совокупность переменных, констант, знаков операций, имен функций, скобок, которая может быть вычислена в соответствии с синтаксисом языка программирования. Результатом вычисления выражения является величина определенного типа. Если эта величина имеет числовой тип, то такое выражение называется *арифметическим*.

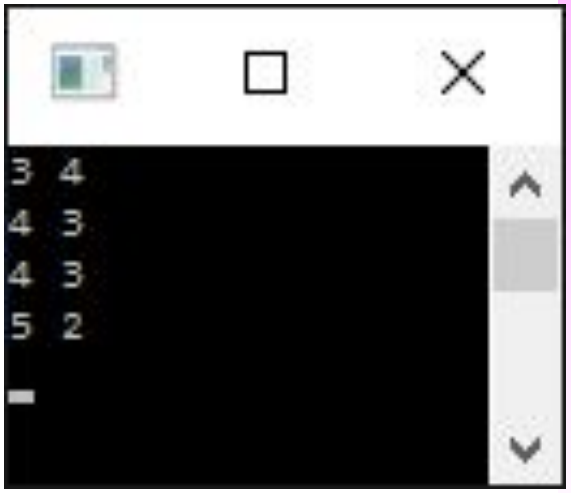
В состав арифметического выражения могут входить: числовые константы; имена переменных; знаки математических операций; математические функции и функции, возвращающие число; открывающиеся и закрывающиеся круглые скобки.

Величины, над которыми выполняются операции, называются *операндами*. В зависимости от количества операндов операции могут быть *унарными* (один операнд) и *бинарными* (два операнда).

Выражения

Инкремент (++) и декремент(--). Эти операции имеют две формы записи - *префиксную*, когда операция записывается перед операндом, и *постфиксную* - операция записывается после операнда. Префиксная операция инкремента (декремента) увеличивает (уменьшает) свой операнд и возвращает измененное значение как результат. Постфиксные версии инкремента и декремента возвращают первоначальное значение операнда, а затем изменяют его. Рассмотрим эти операции на примере.

```
static void Main() {     int i = 3, j = 4;
Console.WriteLine("{0} {1}", i, j);
Console.WriteLine("{0} {1}", ++i, --j);
Console.WriteLine("{0} {1}", i++, j--);
Console.WriteLine("{0} {1}", i, j);
 Console.ReadKey();    }
```



Выражения

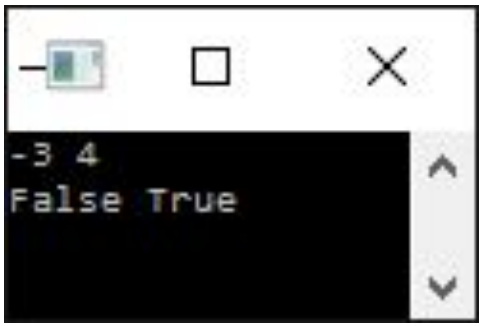
Операция ***new***.

Используется для создания нового объекта. С помощью ее можно создавать как объекты ссылочного типа, так и размерные, например: ***object z=new object(); int i=new int();***

Отрицание.

1. Арифметическое отрицание (-) – меняет знак операнда на противоположный.
2. Логическое отрицание (!) – определяет операцию инверсии для логического типа.

```
static void Main() {     int i = 3, j=-4;
 bool a = true, b=false;
Console.WriteLine("{0} {1}", -i, -j);
Console.WriteLine("{0} {1}", !a, !b);
 Console.ReadKey();
}
```



Выражения

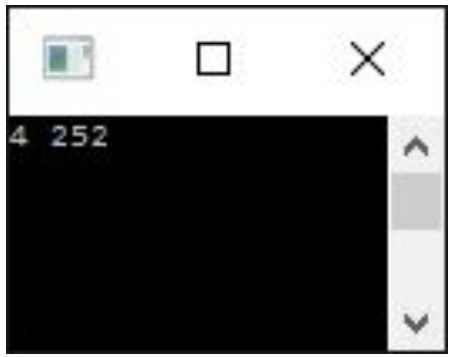
Явное преобразование типа.

Используется для явного преобразования из одного типа в другой.

Формат операции:

(<тип>) <выражение>;

```
static void Main()
{          int  i = -4;
      byte  j = 4;
      int a = (int)j;  //преобразование без потери точности
      byte b = (byte)i; //преобразование с потерей точности
Console.WriteLine ("{0} {1}", a, b);
      Console.ReadKey();
}
```

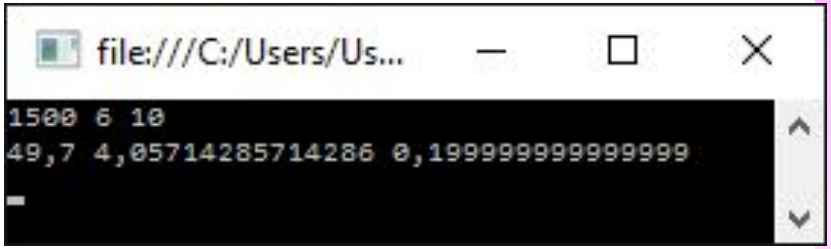


Выражения

Умножение (*), деление (/) и деление с остатком (%).

Операции умножения и деления применимы для целочисленных и вещественных типов данных. Для других типов эти операции применимы, если для них возможно неявное преобразование к целым или вещественным типам. При этом тип результата равен «наибольшему» из типов операндов, но не менее int. Если оба операнда при делении целочисленные, то и результат тоже целочисленный.

```
static void Main()
{
int i = 100, j = 15;
double a = 14.2, b = 3.5;
Console.WriteLine("{0} {1} {2}", i*j, i/j, i%j);
Console.WriteLine("{0} {1} {2}", a * b, a / b, a % b);
 Console.ReadKey();
}
```



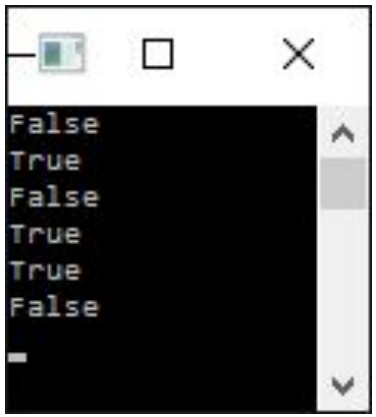
Выражения

Сложение (+) и вычитание (-).

Операции сложения и вычитания применимы для целочисленных и вещественных типов данных. Для других типов эти операции применимы, если для них возможно неявное преобразование к целым или вещественным типам.

Операции отношения (**<, <=, >, >=, ==, !=**). Операции отношения сравнивают значения левого и правого операндов. Результат операции логического типа: *true* – если значения совпадают, *false* – в противном случае.

```
static void Main()
{   int i = 15, j = 15;   Console.WriteLine(i<j); //меньше
Console.WriteLine(i<=j);  //меньше или равно
Console.WriteLine(i>j);  //больше
Console.WriteLine(i>=j); //больше или равно
Console.WriteLine(i==j);  //равно
Console.WriteLine(i!=j);  //не равно
 Console.ReadKey();   }
```

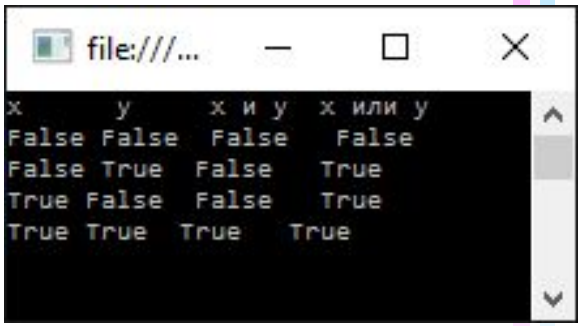


Выражения

Логические операции: ***И (&&), ИЛИ (||).***

Логические операции применяются к операндам логического типа. Результат логической операции ***И*** имеет значение истина тогда и только тогда, когда оба операнда принимают значение истина. Результат логической операции ***ИЛИ*** имеет значение истина тогда и только тогда, когда хотя бы один из операндов принимает значение истина.

```
static void Main()  {
Console.WriteLine("x     y     x и y    x или y");
 Console.WriteLine("{0} {1} {2} {3}", false, false,
false&&false, false||false);
Console.WriteLine("{0} {1} {2} {3}", false, true,
false&&true, false||true);
Console.WriteLine("{0} {1} {2} {3}", true, false,
true&&false, true||false);
Console.WriteLine("{0} {1} {2} {3}", true, true,
 true&&true, true||true);
 Console.ReadKey();  }
```



Выражения

Операции присваивания.

Формат операции простого присваивания (=):

операнд_2 = операнд_1;

В результате выполнения этой операции вычисляется значение операнда_1, и результат записывается в операнд_2. Можно связать воедино сразу несколько операторов присваивания, записывая такие цепочки: a=b=c=100. Выражение такого вида выполняется справа налево: результатом выполнения c=100 является число 100, которое затем присваивается переменной b, результатом чего опять является 100, которое присваивается переменной a. Кроме простой операции присваивания существуют сложные операции присваивания:

***=** **умножение с присваиванием,**

/= **деление с присваиванием,**

%= **остаток от деления с присваиванием,**

+= **сложение с присваиванием,**

-= **вычитание с присваиванием**. В сложных операциях присваивания, например, при сложении с присваиванием, к операнду_2 прибавляется операнд_1, и результат записывается в операнд_2. То есть, выражение ***c* += *a*** является более компактной записью выражения ***c* = *c* + *a*.**

Выражения

Вычисление значения выражения происходит с учетом приоритета операций , которые в нем участвуют.

Если в выражении соседствуют операции одного приоритета, то унарные операции, условная операция и операции присваивания выполняются справа налево, остальные - слева направо.

a = b = c означает ***a=(b=c),***

a+b+c означает ***(a + b) + c***.

Если необходимо изменить порядок выполнения операций, то в выражении необходимо поставить круглые скобки.

Приоритеты операций

Операции языка C# приведены в порядке убывания приоритетов. Операции с разными приоритетами разделены чертой.

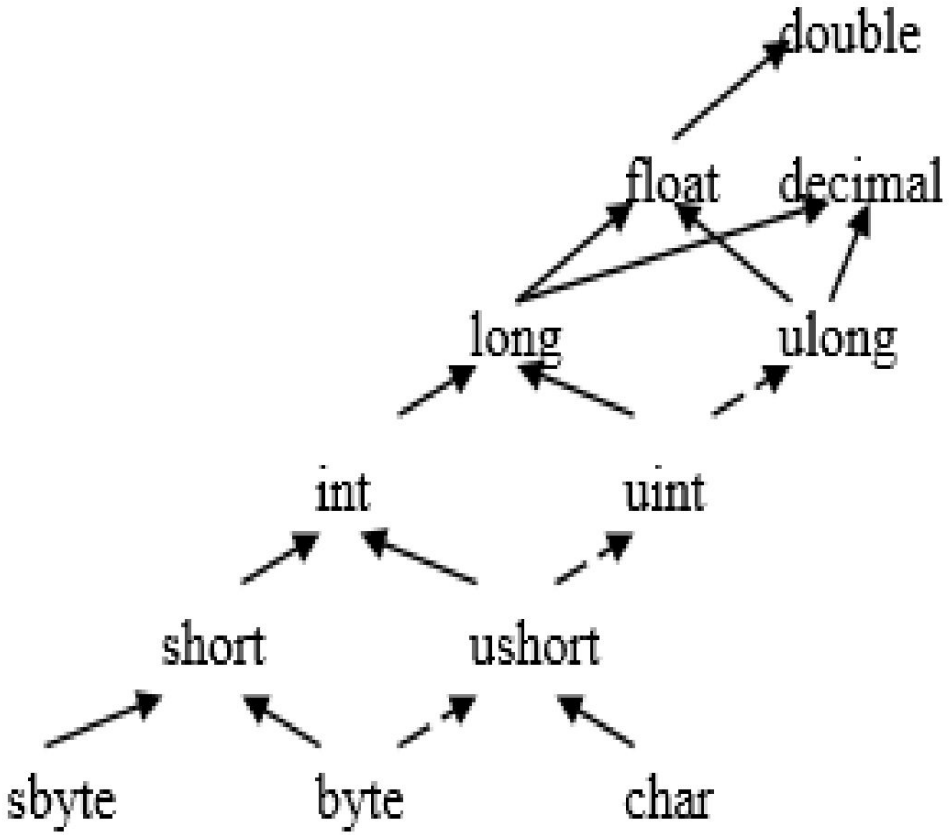
+	Унарный плюс
-	Арифметическое отрицание
!	Логическое отрицание
~	Поразрядное отрицание
++x	Префиксный инкремент
--x	Префиксный декремент
(тип) x	Преобразование типа
*	Умножение
/	Деление
%	Остаток от деления
<<	Сдвиг влево
>>	Сдвиг вправо
<	Меньше
>	Больше
<=	Меньше или равно
>=	Больше или равно
is	Проверка принадлежности типу
as	Приведение типа
==	Равно
!=	Не равно

Продолжение таблицы приоритетов операций

&	Поразрядное И
^	Поразрядное исключающее ИЛИ
|	Поразрядное ИЛИ
&&	Логическое И
||	Логическое ИЛИ
? :	Условная операция
=	Простое присваивание
*=	Умножение с присваиванием
/=	Деление с присваиванием
%=	Остаток от деления с присваиванием
+=	Сложение с присваиванием
-=	Вычитание с присваиванием
<<=	Сдвиг влево с присваиванием
>>=	Сдвиг вправо с присваиванием
&=	Поразрядное И с присваиванием
^=	Поразрядное исключающее ИЛИ с присваиванием
|=	Поразрядное ИЛИ с присваиванием

Преобразование типов в выражениях. Иерархия типов

В выражение могут входить операнды различных типов. Если операнды имеют одинаковый тип, то результат операции будет иметь тот же тип. Если операнды разного типа, то перед вычислениями выполняются преобразования более коротких типов в более длинные для сохранения значимости и точности.



МАТЕМАТИЧЕСКИЕ ФУНКЦИИ ЯЗЫКА C#

Название	***Описание***
Math.Abs(<выражение>)	Модуль
Math.Ceiling(<выражение>)	Округление для большего целого
Math.Cos(<выражение>)	Косинус
Math.E	Число e
Math.Exp(<выражение>)	Экспонента
Math.Floor(<выражение>)	Округление до меньшего целого
Math.Log(<выражение>)	Натуральный логарифм
Math.Log10(<выражение>)	Десятичный логарифм
Math.Max(<выражение1>, <выражение2>)	Максимум из двух значений
Math.Min(<выражение1>, <выражение2>)	Минимум из двух значений
Math.PI	Число π
Math.Pow(<выражение1>, <выражение2>)	Возведение в степень
Math.Round(<выражение>)	Простое округление
Math.Sign(<выражение>)	Знак числа
Math.Sin(<выражение>)	Синус
Math.Sqrt(<выражение>)	Квадратный корень
Math.Tan(<выражение>)	Тангенс

ОПЕРАТОРЫ ЯЗЫКА C#

Все операторы можно разделить на четыре группы:

- операторы следования,
- операторы ветвления,
- операторы цикла ,
- операторы передачи управления.

Операторы следования

Операторы следования выполняются в естественном порядке: начиная с первого до последнего. К операторам следования относятся: *выражение и составной оператор*. Любое выражение, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается вычислением значения выражения или выполнением законченного действия, например, вызовом метода

```
++i;   //оператор инкремента
x+=y;   //оператор сложения с присваиванием
Console.WriteLine(x);  //вызов метода
x=Math.Pow(a,b)+a*b; //вычисление сложного выражения
```

ОПЕРАТОРЫ ЯЗЫКА C#

Составной оператор

Составной оператор или блок представляет собой последовательность операторов, заключенных в фигурные скобки {}.

Блок обладает собственной областью видимости: объявленные внутри блока имена доступны только внутри данного блока или блоков, вложенных в него.

Составные операторы применяются в случае, когда правила языка предусматривают наличие только одного оператора, а логика программы требует нескольких операторов. Если заключить несколько операторов в фигурные скобки, то получится блок, который будет рассматриваться компилятором как единый оператор.

Условный оператор if

используется для разветвления процесса обработки данных на два направления.

Форма сокращенного оператора if:

if (B) S;

где ***B*** – логическое выражение, истинность которого проверяется;

S – оператор: простой или составной.

При выполнении сокращенной формы оператора *if* сначала вычисляется выражение *B*, затем проводится анализ его результата: если *B* истинно, то выполняется оператор *S*; если *B* ложно, то оператор *S* пропускается. Таким образом, с помощью сокращенной формы оператора if можно либо выполнить оператор S, либо пропустить его.

Условный оператор if

Форма полного оператора if:

if (B) S1; else S2;

где ***B*** – логическое выражение, истинность которого проверяется;

S1, S2- оператор: простой или составной.

При выполнении полной формы оператора *if* сначала вычисляется значение выражения *B*, затем анализируется его результат: если *B* истинно, то выполняется оператор *S1*, а оператор *S2* пропускается; если *B* ложно, то выполняется оператор *S2*, а *S1* – пропускается. Таким образом, с помощью полной формы оператора *if* можно выбрать одно из двух альтернативных действий процесса обработки данных.

Примеры.
Сокращенная форма с простым оператором if:
if (a > 0) x=y;

Сокращенная форма с составным оператором
if (++i>0) {x=y; y=2*z;}

Полная форма с простым оператором
if (a > 0 || b<0) x=y; else x=z;

Полная форма с составными операторами
if (i!=j-1) { x= 0; y= 1;} else {x=1; y=0;}

Уровни вложенности

Пример 1.

Уровни вложенности

```
if (A>B)
if (C>D) X=Y;
else X=Z;
else X=R;
```

2 — 1

Пример 2.

Уровни вложенности

```
if (A>B) X=Y;
else if (C>D) X=Z;
     else X=R;
```

2 — 1

Пример 3.

Уровни вложенности

```
if (A < B)
   if (C < D) X =Y;
   else  X = Z;
else
   if (E < F) X= R;
   else  X = Q;
```

2, 2 — 1

Пример 4

Уровни вложенности

```
if (A< B)
   if (C < D)
     if (E < F) X= Q;
     else X = R;
   else X= Z;
else X = Y;
```

3 — 2 — 1

Оператор выбора switch

Оператор выбора switch предназначен для разветвления процесса вычислений по нескольким направлениям.

Формат оператора:

switch (<выражение>) {
case <константное_выражение_1>:[<оператор 1>];
<оператор перехода>;
case <константное_выражение_2>:[<оператор 2>];
<оператор перехода>;
... case <константное_выражение_n>: [<оператор n>];
<оператор перехода>;
[default: <оператор>;] }

Выражение, стоящее за ключевым словом *switch*, должно иметь арифметический, символьный, строковый тип или тип указатель. Все константные выражения должны иметь разные значения, но их тип должен совпадать с типом выражения, стоящего внутри скобок *switc*h или приводиться к нему. Ключевое слово *case* и расположенное после него константное выражение называют также меткой *case*. Выполнение оператора начинается с вычисления выражения, расположенного за ключевым словом *switch*. Полученный результат сравнивается с меткой *case*. Если результат выражения соответствует метке *case*, то выполняется оператор, стоящий после этой метки, за которым обязательно должен следовать оператор перехода: *break, goto* и т.д. В случае отсутствия оператора перехода компилятор выдаст сообщении об ошибке. При использовании оператора *break* происходит выход из *switch* и управление передается оператору, следующему за *switch.* Если же используется оператор *goto*, то управление передается оператору, помеченному меткой, стоящей после *goto*.

Пример.

По заданному виду арифметической операции (сложение, вычитание, умножение и деление) и двум операндам, вывести на экран результат применения данной операции к операндам.

```
static void Main() {  Console.Write("OPER= ");
char oper=char.Parse(Console.ReadLine());  bool ok=true;
Console.Write("A= ");  double a=double.Parse(Console.ReadLine());
Console.Write("B= ");  double b=double.Parse(Console.ReadLine());
double res=0;
switch (oper)  {
case '+' :      res = a + b;    break;
case '-' :      res = a - b;     break;
case '*' :     res = a * b;     break;
case ':' :   if (b != 0) {  res = a / b;  break;    }  else  {   goto default;  }
default:  ok = false;     break;   }
if (ok)   {    Console.WriteLine("{0} {1} {2} = {3}", a, oper, b, res);   }
else    {    Console.WriteLine("error");   }  }
```

Если необходимо, чтобы для разных меток выполнялось одно и то же действие, то метки перечисляются через двоеточие.

```
case ':' : case '/' :  //перечисление меток
 if (b != 0)   {     res = a / b;     break;   }
```

ОПЕРАТОРЫ ЯЗЫКА C#. ***Операторы цикла***

Операторы цикла используются для организации многократно повторяющихся вычислений. К операторам цикла относятся: **цикл с предусловием while, цикл с постусловием do while, цикл с параметром for и цикл перебора foreach.**

Цикл с предусловием while . Оператор цикла *while* организует выполнение одного оператора (простого или составного) неизвестное заранее число раз. Формат цикла *while*:

while (B) S; где *B* – выражение, истинность которого проверяется (условие завершения цикла); *S* – тело цикла (простой или составной оператор). Перед каждым выполнением тела цикла анализируется значение выражения *B*: если оно истинно, то выполняется тело цикла, и управление передается на повторную проверку условия *B*; если значение B ложно – цикл завершается и управление передается на оператор, следующий за оператором *S*. Если результат выражения *B* окажется ложным при первой проверке, то тело цикла не выполнится ни разу.

ОПЕРАТОРЫ ЯЗЫКА C#. ***Операторы цикла***

Цикл с постусловием do while

Оператор цикла do while также организует выполнение одного оператора) неизвестное заранее число раз. Однако в отличие от цикла while условие завершения цикла проверяется после выполнения тела цикла.

Формат цикла do while: ***do S while (B);***

где *B* – выражение, истинность которого проверяется;

S – тело цикла (простой или составной оператор). Сначала выполняется оператор *S*, а затем анализируется значение выражения *B*: если оно истинно, то управление передается оператору *S,* если ложно - цикл завершается, и управление передается на оператор, следующий за условием *B*. Так как условие *B* проверяется после выполнения тела цикла, то в любом случае тело цикла выполнится ***хотя бы один раз***. В операторе *do while*, так же как и в операторе *while*, возможна ситуация зацикливания в случае, если условие *B* всегда будет оставаться истинным.

ОПЕРАТОРЫ ЯЗЫКА C#. ***Операторы цикла***

Пример.
Вывод на экран целых чисел из интервала от 1 до n.

```
static void Main()
{
    Console.Write("N= ");
    int n=int.Parse(Console.ReadLine());
    int i = 1;
    do
        Console.Write(" {0}", i++);
//      Console.Write(" {0}", ++i);
    while (i <= n);
}
```

ОПЕРАТОРЫ ЯЗЫКА C#. ***Оператор foreach***

Оператор ***foreach*** применяется для перебора элементов в специальном образом организованной группе данных. Удобство этого вида цикла заключается в том, что не требуется определять количество элементов в группе и выполнять перебор по индексу.

Синтаксис оператора:

foreach (<тип><имя>in<группа>) <тело цикла>;

где имя определяет локальную по отношению к циклу переменную, которая будет по очереди перебирать все значения из указанной группы; ее тип соответствует базовому типу элементов группы. Ограничением оператора *foreach* является то, что с его помощью можно только просматривать значения элементов. ***Никаких изменений*** ни с самой группой, ни с находящимися в ней данными ***проводить нельзя***.

ОПЕРАТОРЫ ЯЗЫКА C#. ***Вложенные циклы***

Циклы могут быть простые или вложенные (кратные, циклы в цикле).
Вложенными могут быть циклы любых типов:
while,
do while,
for.
Каждый внутренний цикл должен быть полностью вложен во все внешние циклы. «Пересечения» циклов не допускаются.

ОПЕРАТОРЫ ЯЗЫКА C#. ***Операторы безусловного перехода***

В C# есть несколько операторов, изменяющих естественный порядок выполнения команд:
оператор безусловного перехода ***goto***,

- оператор выхода ***break***,
- оператор перехода к следующей итерации цикла ***continue***,
- оператор возврата из метода ***return***,
- оператор генерации исключения ***throw***.

Оператор безусловного перехода *goto*
Формат: ***goto <метка>;***
В теле того же метода должна присутствовать ровно одна конструкция вида: ***<метка>: <оператор>;***
Оператор *goto* передает управление оператору с меткой.

ОПЕРАТОРЫ ЯЗЫКА C#. ***Операторы break и continue***

Оператор ***break*** используется внутри операторов цикла и оператора выбора для обеспечения перехода в точку программы, находящуюся непосредственно за оператором, внутри которого находится ***break***.

Оператор перехода к следующей итерации цикла ***continue*** пропускает все операторы, оставшиеся до конца тела цикла, и передает управление на начало следующей итерации (повторение тела цикла).

```
 static void Main()
{
Console.WriteLine("n=");
int n = int.Parse(Console.ReadLine());
for (int i = 1; i <= n; i++)      {  if (i % 2 == 0)   {   continue;   }
Console.Write(" {0} ", i);      }
}
```

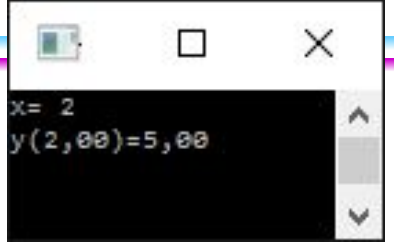
ОПЕРАТОРЫ ЯЗЫКА C#.

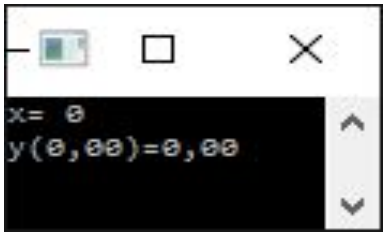
Для произвольных значений аргументов вычислить значение функции, заданной следующим образом:

$$y(x) = \begin{cases} (x^3 + 1)^2, & \text{при } x < 0; \\ 0, & \text{при } 0 \le x < 1; \\ |x^2 - 5x + 1|, & \text{при } x \ge 1. \end{cases}$$

Пример

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace pr3
{    class Program     {        static void Main()
{           Console.Write("x= ");
double x=double.Parse(Console.ReadLine());
double  y;    if (x<0) {  y=Math.Pow(Math.Pow(x,3)+1,2);    }
 else    {     if (x<1) {      y=0;     }
      else      {   y=Math.Abs(x*x-5*x+1);     }    }
        Console.WriteLine ("y({0:f2})={1:f2}",x,y);
        Console.ReadLine();
    } } }
```





Пример

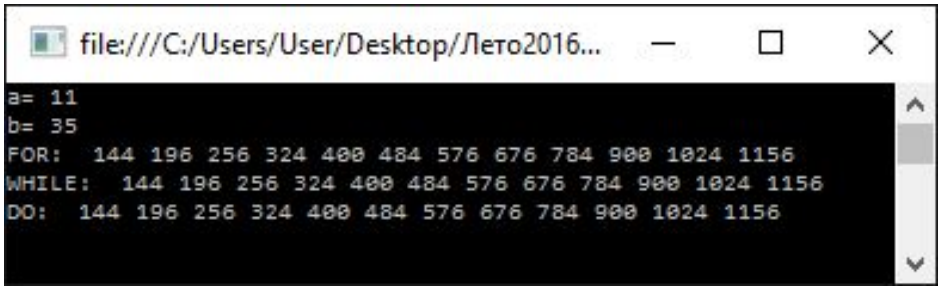
Написать программу, которая выводит на экран квадраты всех четных чисел из диапазона от A до B (A и B целые числа, при этом A≤B).

Указания по решению задачи.

Из диапазона целых чисел от A до B необходимо выбрать только четные числа. Напомним, что четными называются числа, которые делятся на два без остатка. Кроме того, четные числа представляют собой упорядоченную последовательность, в которой каждое число отличается от предыдущего на 2. Решить эту задачу можно с помощью каждого оператора цикла.

Пример

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace pr3
{    class Program
    {   static void Main()
    {     Console.Write("a= ");
  int a = int.Parse(Console.ReadLine());
 Console.Write("b= "); int b = int.Parse(Console.ReadLine());
  int i; Console.Write("FOR: "); a = (a % 2 == 0) ? a : a + 1;
  for (i = a; i <= b; i += 2) { Console.Write(" {0}", i * i); }
Console.Write("\nWHILE: ");
 i = a; while (i <= b) { Console.Write(" {0}", i * i); i += 2; }
Console.Write("\nDO: ");
i = a; do { Console.Write(" {0}", i * i); i += 2; } while (i <= b);
        Console.ReadLine();    }  } }
```



Пример

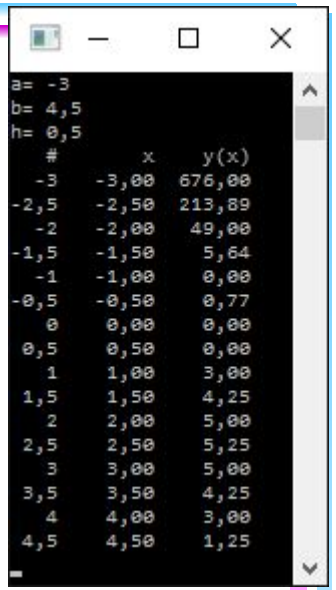
Построить таблицу значений функции:

$$y(x)=\begin{cases}(x^3+1)^2, & \text{при } x<0;\\ 0, & \text{при } 0\le x<1;\\ |x^2-5x+1|, & \text{при } x\ge 1.\end{cases}$$

для $x\in[a,b]$ с шагом h.

Пример

```
namespace pr3
{   class Program
   {       static void Main()
   {       Console.Write("a= ");
      double a=double.Parse(Console.ReadLine());
      Console.Write("b= ");
      double b=double.Parse(Console.ReadLine());
      Console.Write("h= ");
       double h=double.Parse(Console.ReadLine());
      double y;    int i=1;
      Console.WriteLine("{0,4} {1,7} {2,7}", "#", "x", "y(x)");
      for (double x=a; x<=b; x+=h, ++i)    {
         if (x<0)       {     y=Math.Pow(Math.Pow(x,3)+1,2);     }
         else    {     if (x<1)       {      y=0;      }      else
    {      y=Math.Abs(x*x-5*x+1);      }    }
         Console.WriteLine("{1,4} {1,7:f2} {2,7:f2}",i,x,y);     }
      Console.ReadLine();    }  } }
```



МЕТОДЫ

Метод – это функциональный элемент класса, который реализует вычисления или другие действия, выполняемые классом или его экземпляром (объектом).

Метод представляет собой законченный фрагмент кода, к которому можно обратиться по имени. Он описывается один раз, а вызываться может многократно.

Совокупность методов класса определяет, что конкретно может делать класс. Например, стандартный класс Math содержит методы, которые позволяют вычислять значения математических функций.

Синтаксис объявления метода:

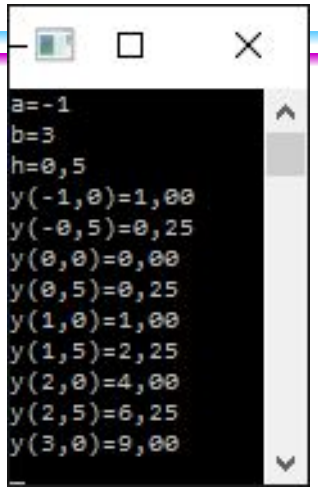
[атрибуты] [спецификторы] тип_результата имя_метода ([список_формальных_параметров]) { тело_метода; return значение; }

МЕТОДЫ

1) атрибуты и спецификторы являются необязательными элементами в описании метода.
2) тип_результата определяет тип значения, возвращаемого методом. Это может быть любой тип, включая типы классов, создаваемые программистом, а также тип void, который говорит о том, что метод ничего не возвращает.
3) имя_метода используется для обращения к нему из других мест программы, является идентификатором.
4) список_формальных_параметров представляет собой последовательность пар, состоящих из типа данных и идентификатора, разделенных запятыми. Формальные параметры - это переменные, которые получают значения, передаваемые методу при вызове. Если метод не имеет параметров, то список_параметров остается пустым.
5) ***return*** – это оператор безусловного перехода, который завершает работу метода и возвращает значение, стоящие после оператора return, в точку его вызова. Тип значения должен соответствовать типу__результата, или приводиться к нему. Если метод не должен возвращать никакого значения, то указывается тип *void*, и в этом случае оператор *return* либо отсутствует, либо указывается без возвращаемого значения.

МЕТОДЫ

```
class Program    {
    static double Func( double x)
    {    return x*x;   }
 static void Main() {  Console.Write("a=");
double a=double.Parse(Console.ReadLine());
Console.Write("b="); double b=double.Parse(Console.ReadLine());
Console.Write("h="); double h=double.Parse(Console.ReadLine());
for (double x = a; x <= b; x += h)       {
double y = Func(x);
Console.WriteLine("y({0:f1})={1:f2}", x, y);  }
 Console.ReadLine();    } }
```



В данном примере метод Func содержит параметр ***x*** типа ***double***. Для того, чтобы метод ***Func*** возвращал в вызывающий его метод *Main* значение выражения *x*x* (типа double), перед именем метода указывается тип возвращаемого значения – *double*, а в теле метода используется оператор передачи управления – *return*. Оператор *return* завершает выполнение метода и передает управление в точку его вызова.

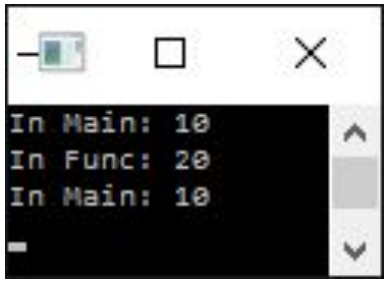
МЕТОДЫ

В C# предусмотрено четыре типа параметров:

- параметры-значения,
- параметры-ссылки,
- выходные параметры и параметры, позволяющие создавать методы с переменным количеством аргументов.

При передаче параметра по значению метод получает копии параметров, и операторы метода работают с этими копиями. Доступа к исходным значениям параметров у метода нет, а, следовательно, нет и возможности их изменить.

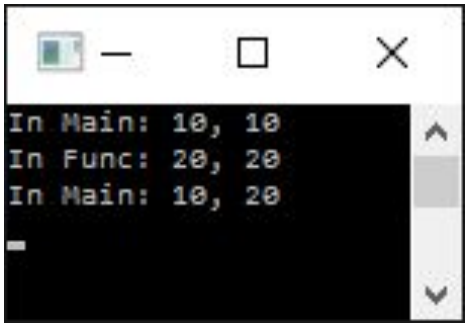
```
namespace pr3
{class Program
    {    static void Func(int x)
        {   x += 10;  Console.WriteLine("In Func: " + x); }
 static void Main()
 {  int a = 10; Console.WriteLine("In Main: {0}", a);
Func(a); Console.WriteLine("In Main: {0}", a);
Console.ReadLine();        }    }      }
```



МЕТОДЫ

При передаче параметров по ссылке метод получает копии адресов параметров, что позволяет осуществлять доступ к ячейкам памяти по этим адресам и изменять исходные значения параметров. Для того чтобы параметр передавался по ссылке, необходимо при описании метода перед формальным параметром и при вызове метода перед соответствующим фактическим параметром поставить спецификатор ***ref***.

```
static void Func(int x, ref int y)
{  x += 10; y += 10;
Console.WriteLine("In Func: {0}, {1}", x, y);
}
 static void Main()     {        int a=10, b=10;
Console.WriteLine("In Main: {0}, {1}", a, b);
Func(a, ref b);
Console.WriteLine("In Main: {0}, {1}", a, b);
}
```

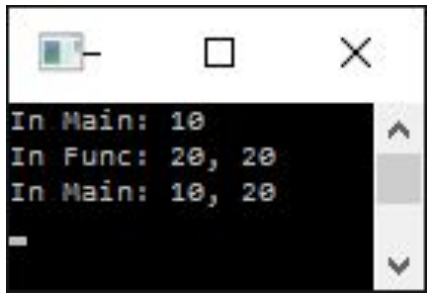


МЕТОДЫ

Передача параметра по ссылке требует, чтобы аргумент был инициализирован до вызова метода .

Однако не всегда имеет смысл инициализировать параметр до вызова метода, например, если метод считывает значение этого параметра с клавиатуры, или из файла. В этом случае параметр следует передавать как выходной, используя спецификатор ***out***.

```
class Program {
static void Func(int x, out int y)
{  x += 10; y = x;
Console.WriteLine("In Func: {0}, {1}", x, y);
}
   static void Main()
{   int a=10, b;
   Console.WriteLine("In Main: {0}", a);
   Func(a, out b);
   Console.WriteLine("In Main: {0}, {1}", a, b);
   Console.ReadLine();  }    }
```



Перегрузка методов

Методы, реализующие один и тот же алгоритм для различного количества параметров или различных типов данных, могут иметь одно и то же имя. Использование таких методов называется ***перегрузкой методов***. Компилятор определяет, какой именно метод требуется вызвать, по типу и количеству фактических параметров.

Max – возвращает значение наибольшей цифры

```
static int Max(int a) { int b = 0;
while (a > 0) {if (a % 10 > b) b = a % 10;  a /= 10;  }        return b;     }
```

Max – возвращает значение наибольшего из двух чисел

```
static int Max(int a, int b)     {   if (a > b) return a;       else return b;     }
```

Max – возвращает значение наибольшего из трех чисел

```
static int Max(int a, int b, int c)      {  if (a > b && a > c) return a;
else if (b > c) return b;    else return c;     }
static void Main()     {
 int a = 1283, b = 45, c = 35740;
Console.WriteLine(Max(a));
 Console.WriteLine(Max(a, b));
 Console.WriteLine(Max(a, b, c));   } }
```

Перегрузка методов

При вызове метода Max выбирается вариант, соответствующий типу и количеству передаваемых в метод аргументов. Если точного соответствия не найдено, выполняются неявные преобразования типов в соответствии с общими правилами. Если преобразование невозможно, выдается сообщение об ошибке. Если выбор перегруженного метода возможен более чем одним способом, то выбирается «лучший» из вариантов (вариант, содержащий меньшее количество и длину преобразований в соответствии с правилами преобразования типов). Если существует несколько вариантов, из которых невозможно выбрать подходящий, выдается сообщение об ошибке.

Перегрузка методов является проявлением *полиморфизма*, одного из основных принципов объектно-ориентированного программирования. Программисту гораздо удобнее помнить одно имя метода и использовать его для работы с различными типами данных, а решение о том, какой вариант метода вызвать, возложить на компилятор.

МАССИВЫ

Массив — набор элементов одного и того же типа, объединенных общим именем.

C#-массивы относятся к *ссылочным* типам данных. При этом имя массива является ссылкой на область кучи (динамической памяти), в которой последовательно размещается набор элементов определенного типа.

Выделение памяти под элементы происходит на этапе объявления или инициализации массива, а за освобождением памяти следит сборщик мусора.

Массивы реализуются в C# как объекты, для которых разработан большой набор методов, реализующих различные алгоритмы обработки элементов массива.

МАССИВЫ

Одномерный массив – это фиксированное количество элементов одного и того же типа, объединенных общим именем, где каждый элемент имеет свой номер. Нумерация элементов массива в C# начинается ***с нуля***, то есть, если массив состоит из 10 элементов, то они будут иметь следующие номера: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Одномерный массив в C# реализуется как объект, поэтому его создание представляет собой двухступенчатый процесс. Сначала объявляется *ссылочная переменная типа массив*, затем выделяется память под требуемое количество элементов базового типа, и ссылочной переменной присваивается адрес нулевого элемента в массиве. Базовый тип определяет тип данных каждого элемента массива. Количество элементов, которые будут храниться в массиве, определяется размером массива.

МАССИВЫ

Для объявления одномерного массива может использоваться одна из следующих форм записи:

1)

базовый_тип [] имя_массива;

Например: ***char [] a;***

Объявлена ссылка на одномерный массив символов (имя ссылки a), которая в дальнейшем может быть использована для: адресации на уже существующий массив; передачи массива в метод в качестве параметра; отсроченного выделения памяти под элементы массива.

2)

базовый_тип [] имя_массива = new базовый_тип [размер];

Например: ***int [] b=new int [10];*** Объявлена ссылка b на одномерный массив целых чисел. Выделена память для 10 элементов целого типа, адрес этой области памяти записан в ссылочную переменную b.

МАССИВЫ

В C# элементам массива присваиваются начальные значения по умолчанию в зависимости от базового типа. Для арифметических типов – ***нули***, для ссылочных типов – ***null***, для символов - ***символ с кодом ноль***.

3)

базовый_тип [] имя_массива={список инициализации};

Например: ***double [] c={0.3, 1.2, -1.2, 31.5};***

Объявлена ссылка с на одномерный массив вещественных чисел. Выделена память под одномерный массив, размерность которого соответствует количеству элементов в списке инициализации - четырем. Адрес этой области памяти записан в ссылочную переменную с. Значение элементов массива соответствует списку инициализации.

МАССИВЫ

Обращение к элементу массива происходит с помощью индекса*:* *указывается имя массива и, в квадратных скобках, номер данного элемента.*

Например, ***a[0], b[8], c[i].***

Так как массив представляет собой набор элементов, объединенных общим именем, то обработка массива обычно производится в *цикле*.

Вывод массива на экран :

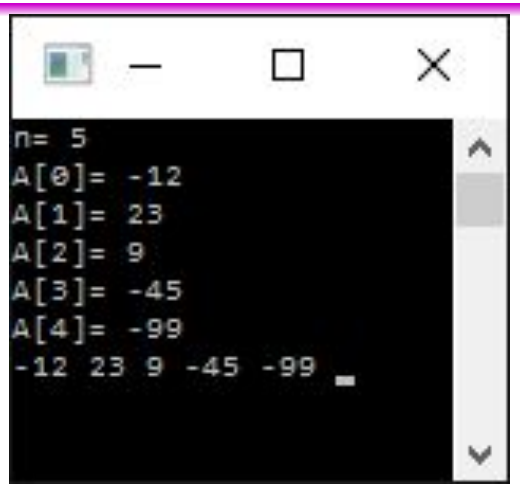
- Использование оператора ***for***

```
static void Main() {
int[] myArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
for (int i = 0; i< 10; i++)
{       Console.WriteLine(myArray[i]);      } }
```

- Использование оператора ***foreach***

```
static void Main() {
 int[] myArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int elem in myArray)
 {   Console.WriteLine(elem);  } }
```

МАССИВЫ



Ввод элементов массива

```
Static void Main() {
 int[] myArray;  //создаем ссылку на массив
Console.Write("n= ");
int n=int.Parse(Console.ReadLine());
myArray=new int [n];  //выделяем память под массив
for (int i=0; i<n; i++)  //вводим элементы массива с клавиатуры
{   Console.Write("A[{0}]= ",i);
myArray[i]=int.Parse(Console.ReadLine());
 }
foreach (int elem in myArray) //выводим массив на экран
 {   Console.Write("{0} ",elem);  } }
```

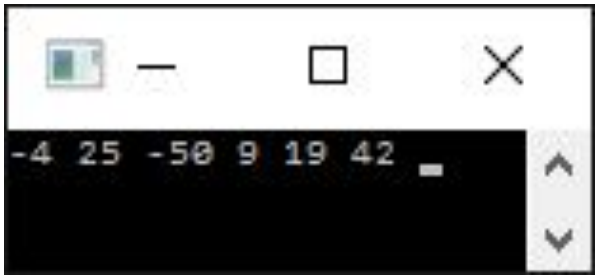
МАССИВЫ

Заполнение массива случайными элементами

Заполнить массив данными можно с помощью генератора случайных чисел

Для этого используется класс ***Random***:

```
Static void Main() {
Random rnd = new Random(); //инициализируем генератор случайных чисел
int[] myArray;
int n = rnd.Next(5, 10); //генерируем случайное число из диапазона [5..10)
myArray = new int[n];
for (int i = 0; i < n; i++)      {
 myArray[i] = rnd.Next(10); // заполняем массив случайными числами
 }
foreach (int elem in myArray)
{        Console.Write("{0} ", elem);      } }
```



МАССИВЫ

Контроль границ массива

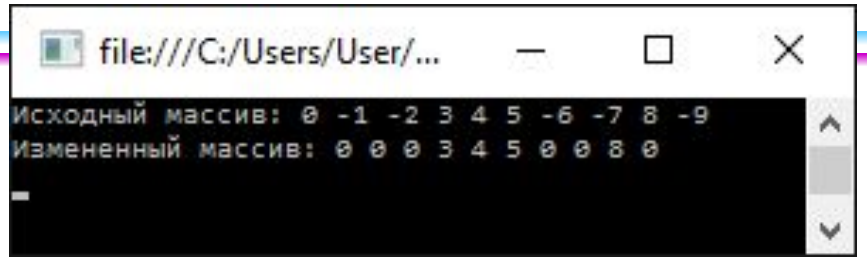
Выход за границы массива в C# расценивается как критическая ошибка, которая ведет к завершению работы программы и генерированию стандартного исключения - ***IndexOutOfRangeException.***

Рассмотрим следующий фрагмент программы:

```
int[] myArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
Console.WriteLine(myArray[10]);
```

В данном случае описан массив из 10 элементов. Так как нумерация элементов массива ведется с нуля, то последний элемент имеет номер 9, и, следовательно, элемента с номером 10 не существует. В этом случае выполнение программы завершится, о чем на консоль будет выдано соответствующее сообщение.

МАССИВЫ



Массив как параметр

Так как имя массива фактически является ссылкой, то он передается в метод по ссылке и, следовательно, все изменения элементов массива, являющегося формальным параметром, отразятся на элементах соответствующего массива, являющегося фактическим параметром. При этом указывать спецификатор *ref* не нужно.

class Program {

static void Print(int[] a) {

foreach (int elem in a) { Console.Write("{0} ", elem); }

Console.WriteLine(); }

static void Change(int[] a, int n) {

for (int i = 0; i< n; i++) { if (a[i] < 0) { a[i] = 0; } } }

static void Main() { int[] myArray = { 0, -1, -2, 3, 4, 5, -6, -7, 8, -9 };

Console.Write("Исходный массив: "); ***Print(myArray);***

Change(myArray, 10); *Console.Write("Измененный массив: ");*

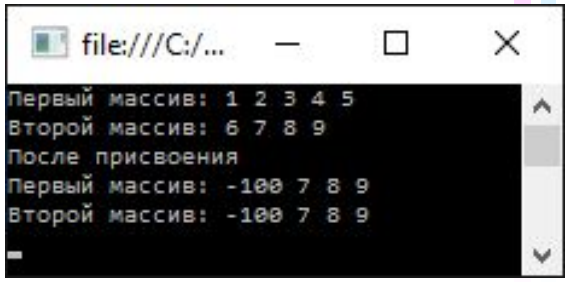
Print(myArray); } }

МАССИВЫ

То, что имя массива является ссылкой, следует учитывать при попытке присвоить один массив другому.

```
class Program {  static void Print(int[] a)    {
foreach (int elem in a)     {     Console.Write("{0} ", elem);    }
Console.WriteLine();  }
 static void Main()
 {   int[] one = { 1, 2, 3, 4, 5};   Console.Write("Первый массив: ");
Print(one);
int[] two = { 6, 7, 8, 9};   Console.Write("Второй массив: ");
Print(two);   one=two;   two[0]=-100;
Console.WriteLine("После присвоения ");
Console.Write("Первый массив: ");   Print(one);
Console.Write("Второй массив: ");   Print(two);  } }
```

Ссылки ***one*** и ***two*** ссылаются на один массив.
А исходный массив, связанный со ссылкой ***one***, оказался потерянным, и будет удален сборщиком мусора.



МАССИВЫ

Массив как объект

В C# массивы реализованы как объекты. Они реализованы на основе базового класса ***Array***, определенного в пространстве имен ***System***. Данный класс содержит различные *свойства* и *методы*.

Например, свойство ***Length*** позволяет определять количество элементов в массиве.

Пример.

```
static void Change(int[] a)
{
for (int i = 0; i<a.Length; i++)
{   if (a[i] > 0)    {    a[i] = 0;   }
} }
```

Информация о длине массива передается в метод *Change* неявным образом вместе с массивом, и нет необходимости вводить дополнительную переменную для хранения размерности массива.

МАССИВЫ

Элемент	Вид	Описание
BinarySearch	статический метод	Осуществляет двоичный поиск в отсортированном массиве.
Clear	статический метод	Присваивает элементам массива значения, определенные по умолчанию, т.е для арифметических типов нули, для ссылочных типов null.
Copy	статический метод	Копирует элементы одного массива в другой массив.
CopyTo	экземплярный метод	Копирует все элементы текущего одномерного массива в другой массив.
IndexOf	статический метод	Осуществляет поиск первого вхождения элемента в одномерный массив. Если элемент найден, то возвращает его индекс, иначе возвращает значение -1.
LastIndexOf	статический метод	Возвращает число размерностей массива. Для одномерного массива Rank возвращает 1, для двумерного – 2 и т.д.

МАССИВЫ

Элемент	Вид	Описание
Length	свойство	Возвращает количество элементов в массиве
Rank	свойство	Возвращает число размерностей массива. Для одномерного массива Rank возвращает 1, для двумерного – 2 и т.д.
Reverse	статический метод	Изменяет порядок следования элементов в массиве на обратный.
Sort	статический метод	Упорядочивает элементы одномерного массива

Вызов статических методов происходит через обращение к имени класса, например, ***Array.Sort(myArray)***. В данном случае обращаемся к статическому методу Sort класса Array и передаем данному методу в качестве параметра объект myArray - экземпляр класса Array. Обращение к свойству или вызов экземплярного метода производится через обращение к экземпляру класса, например, ***myArray.Length*** или ***myArray.GetValue(i)***.

МАССИВЫ

Использование спецификатора params

Чтобы иметь возможность передавать различное количество аргументов, необходимо ввести параметр метода, помеченный спецификатором params. Он размещается в списке параметров последним и является массивом требуемого типа неопределенной длины. В методе может быть только один параметр помеченный спецификатором params.

```
class Program {
static int F(params int []a)  {   int s=0;   foreach (int x in a)   {    s+=x;    }
return s;  }
 static void Main()  {   int a = 1, b = 2, c = 3, d=4;
Console.WriteLine(F());   Console.WriteLine(F(a));
Console.WriteLine(F(a, b));   Console.WriteLine(F(a, b, c));
Console.WriteLine(F(a, b, c, d));  } }
```

Результат работы программы: Недостаточно аргументов 0 1 3 6 10

Как видим, в метод F может быть передано различное количество аргументов, в том числе, и нулевое.

МАССИВЫ. Двумерные массивы

Многомерные массивы имеют более одного измерения. Чаще всего используются двумерные массивы, которые представляют собой таблицы Каждый элемент массива имеет два индекса, первый определяет номер строки, второй – номер столбца, на пересечении которых находится элемент. Нумерация строк и столбцов начинается с нуля.

Объявить двумерный массив можно одним из предложенных способов:

1***)***

базовый_тип [,] имя_массива;

Например: ***int [,] a***;

Объявлена ссылка на двумерный массив целых чисел (имя ссылки a), которая в дальнейшем может быть использована: для адресации на уже существующий массив; передачи массива в метод в качестве параметра; отсроченного выделения памяти под элементы массива.

МАССИВЫ. Двумерные массивы

2)

базовый тип [,] имя_массива = new базовый_тип [размер1, размер2];

Например float [,] a= new float [3, 4];

Объявлена ссылка b на двумерный массив вещественных чисел. Выделена память для 12 элементов вещественного типа, адрес данной области памяти записан в ссылочную переменную b. Элементы массива инициализируются по умолчанию нулями.

3)

базовый_тип [,] имя_массива=={{элементы 1-ой строки}, … , {элементы n-ой строки}};

Например: int [,] a= new int [,]{{0, 1, 2}, {3, 4, 5}};

МАССИВЫ. Двумерные массивы

Обращение к элементу массива происходит с помощью индексов: указывается имя массива и, в квадратных скобках, номер строки и через запятую номер столбца, на пересечении которых находится данный элемент. Например, ***a[0, 0], b[2, 3], c[i, j].*** Так как массив представляет собой набор элементов, объединенных общим именем, то обработка массива обычно производится с помощью вложенных циклов.

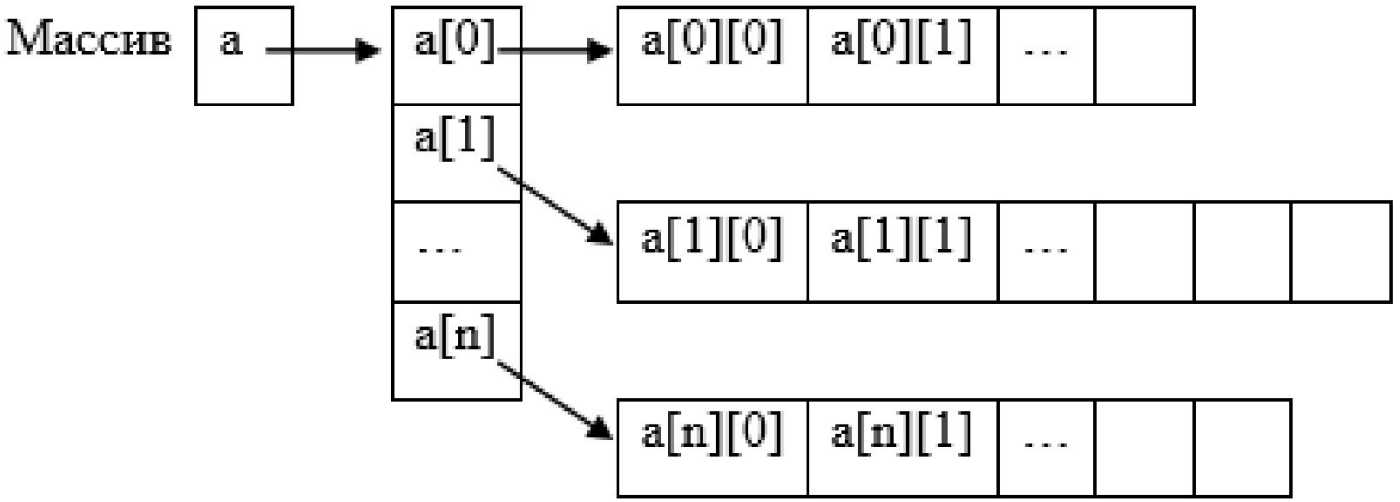
При обращении к свойству ***Length*** для двумерного массива получаем общее количество элементов в массиве.

Чтобы получить количество строк, нужно обратиться к методу ***GetLength*** с параметром ***0***. Чтобы получить количество столбцов – к методу ***GetLength*** с параметром ***1***. В качестве примера рассмотрим программу, в которой сразу будем учитывать два факта: 1) двумерные массивы относятся к ссылочным типам данных; 2) двумерные массивы реализованы как объекты.

```
class Program {
static void Print(int[,] a)  {
for (int i = 0; i<a.GetLength(0); i++)   {
for (int j = 0; j <a.GetLength(1); j++)    {
Console.Write("{0} ", a[i, j]);    }    Console.WriteLine();   }  }
 static void Input(out int[,] a)  {
Console.Write("n= ");   int n=int.Parse(Console.ReadLine());
Console.Write("m= ");   int m=int.Parse(Console.ReadLine());
a=new int[n,m];
for (int i = 0; i<a.GetLength(0); i++)   {
for (int j = 0; j <a.GetLength(1); j++)  {
Console.Write("a[{0},{1}]= ", i, j);
a[i,j]=int.Parse(Console.ReadLine());    }   }  }
 static void Change(int[,] a)  {
for (int i = 0; i<a.GetLength(0); i++)
for (int j = 0; j <a.GetLength(1); j++)
if (a[i, j] % 2 == 0)      {      a[i, j] = 0;          }          }
 static void Main()  {   int [,]a;   Input(out a);
Console.WriteLine("Исходный массив:");   Print(a);   Change(a);
Console.WriteLine("Измененный массив:");   Print(a);  } }
```

МАССИВЫ. Ступенчатые массивы

Структура двумерного ступенчатого массива:



Объявление ступенчатого массива:

тип [][] имя_массива;

Например: ***int[][]a;***

Объявлен одномерный массив ссылок на целочисленные одномерные массивы. При таком описании потребуется не только выделять память под одномерный массив ссылок, но и под каждый из целочисленных одномерных массивов.

МАССИВЫ. Ступенчатые массивы

Пример.

Первый способ выделения памяти:

```
int [][] a= new int [3][];
// Создаем три строки
a[0]=new int [2];
// 0-ая строка ссылается на 2-х элементный одномерный массив
a[1]=new int [3];
// 1-ая строка ссылается на 3-х элементный одномерный массив
a[2]=new int [10];
// 2-ая строка ссылается на 10-х элементный одномерный массив
```

Другой способ выделения памяти:

```
int [][] a= {new int [2], new int [3], new int [10]};
```

Так как каждая строка ступенчатого массива фактически является одномерным массивом, то с каждой строкой можно работать как с экземпляром класса ***Array***. Это является преимуществом ступенчатых массивов перед двумерными массивами.

МАССИВЫ. Ступенчатые массивы

```
class Program {  static void Print(int [][] a)  {
for (int i = 0; i<a.Length; i++)   {
for (int j = 0; j < a[i].Length; j++)     {
 Console.Write("{0} ", a[i][j]);    }    Console.WriteLine();   }  }
static void Input( out int [][] a)  {
Console.Write("n= ");   int n = int.Parse(Console.ReadLine());
a = new int [n][];   for (int i = 0; i<a.Length; i++)   {
Console.Write("введите количество элементов в {0} строке: ", i);
int j = int.Parse(Console.ReadLine());    a[i] = new int[j];
for (j = 0; j < a[i].Length; j++)    {
Console.Write("a[{0}][{1}]= ", i, j);
a[i][j] = int.Parse(Console.ReadLine());    }   }  }
 static void Change (int [][]a)  {   for (int i = 0; i<a.Length; i++)
  {    Array.Sort(a[i]);   }  }
 static void Main()  {   int [][]a;   Input(out a);
Console.WriteLine("Исходный массив:");   Print(a);   Change(a);
Console.WriteLine("Измененный массив:");   Print(a);  } }
```

Примеры

Дан массив из n целых чисел. Написать программу для подсчета суммы этих чисел.

```
class Program {  static int[] Input()   {   Console.Write("n= ");
int n=int.Parse(Console.ReadLine());   int []a=new int[n];
for (int i=0;i<a.Length;++i)    {    Console.Write("a[{0}]= ", i);
a[i]=int.Parse(Console.ReadLine());   }   return a;  }
 static int Sum(int [] a)  {   int sum=0;
foreach (int elem in a )   {    sum+=elem;   }   return sum;  }
 static void Main()  {
int[] a = Input();
Console.WriteLine("Сумма элементов массива={0}",Sum(a));  } }
```

Примеры

Дан массив из n целых чисел. Написать программу, которая определяет наименьший элемент в массиве и его порядковый номер.

```
 class Program {
static int[] Input()   {
Console.Write("n= ");
int n=int.Parse(Console.ReadLine());   int []a=new int [n];
for (int i=0;i<a.Length; i++)    {    Console.Write("a[{0}]= ", i);
a[i]=int.Parse(Console.ReadLine());   }   return a;  }
 static int Min(int[] a, out int index)  {
int min =a[0];     index = 0; for (int i=0; i<a.Length; i++ )
{    if (a[i]<min) {     min=a[i]; index=i; }   }   return min;  }
static void Main()  {
int [] a = Input();   int index;   int min=Min(a, out index);
Console.WriteLine("Наименьший элемент {0} имеет номер {1}",
min, index);  } }
```

Примеры

Дан массив из n целых чисел. Написать программу, которая все наименьшие элементы увеличивает в два раза.

```
class Program {
static int[] Input()  {
Console.Write("n= ");
 int n=int.Parse(Console.ReadLine());  int []a=new int[n];
for (int i=0;i<a.Length; i++)  {  Console.Write("a[{0}]= ", i);
a[i]=int.Parse(Console.ReadLine());  }  return a;  }
 static void Print(int[] a)  {
foreach (int elem in a)  {  Console.Write("{0} ", elem);  }
Console.WriteLine();  }
 static int Min(int[] a)  {
int min=a[0];  for (int i=0; i<a.Length; i++ )  {  if (a[i]<min)  {  min=a[i];
}  }  return min;  }
static void Change(int[] a, int x, int y)  {
 for (int i = 0; i<a.Length; i++)  {  if (a[i] ==x)  {  a[i]*= y;  }  }  }
 static void Main()  {  int[] a = Input();  Console.WriteLine("Исходный
массив:");  Print(a);  int min=Min(a);  const int n=2;  Change(a, min,n);
Console.WriteLine("Измененный массив:");  Print(a);  } }
```

Примеры

Дан массив из n целых чисел. Написать программу, которая подсчитывает количество пар соседних элементов массива, для которых предыдущий элемент равен последующему.

```
class Program {
static int[] Input()    {
Console.Write("n= ");       int n = int.Parse(Console.ReadLine());
int[] a = new int[n];       for (int i = 0; i < a.Length; i++)       {
Console.Write("a[{0}]= ", i);  a[i] = int.Parse(Console.ReadLine());       }
return a;    }
  static int F(int[] a)    {
int k=0;  for (int i = 0; i < a.Length-1; i++)       {
 if (a[i] == a[i+1])  {  ++k;    }       }       return k;    }
   static void Main()    {
int[] a = Input();
Console.WriteLine("k={0}", F(a));    } }
```

Примеры

Дана квадратная матрица, элементами которой являются вещественные числа. Подсчитать сумму элементов главной диагонали.

```
class Program {
static void Print(int[,] a)    {
for (int i = 0; i < a.GetLength(0); i++)      {
for (int j = 0; j < a.GetLength(1); j++)        {   Console.Write("{0} ", a[i, j]);       }
Console.WriteLine();      }    }
static void Input(out int[,] a)    {
Console.Write("n= ");      int n = int.Parse(Console.ReadLine());
a = new int[n, n];      for (int i = 0; i < a.GetLength(0); i++)      {
 for (int j = 0; j < a.GetLength(1); j++)         {
Console.Write("a[{0},{1}]= ", i, j);
a[i, j] =int.Parse(Console.ReadLine());          }      }    }
static int F (int[,] a)    {
int k = 0;      for (int i = 0; i < a.GetLength(0); i++)      {
k += a[i, i];       }       return k;    }
static void Main()    {
int[,] a;      Input(out a);      Console.WriteLine("Исходный массив:");
Print(a); Console.WriteLine("Сум. элем.  на глав. диаг. {0}", F(a)); } }
```

Примеры

Дана прямоугольная матрица n×m, элементами которой являются целые числа. Поменять местами ее строки следующим образом: первую строку с последней, вторую с предпоследней и т.д.

```
class Program {
static void Print(int[][] a)    {
for (int i = 0; i < a.Length; i++)       {   for (int j = 0; j < a[i].Length; j++)
{  Console.Write("{0} ", a[i][j]);    }    Console.WriteLine();       }    }
 static void Input(out int[][] a)    {
Console.Write("n= ");       int n = int.Parse(Console.ReadLine());
Console.Write("m= ");       int m = int.Parse(Console.ReadLine());
a = new int[n][]; for (int i = 0; i < a.Length; i++) { a[i] = new int[m];
for (int j = 0; j < a[i].Length; j++) { Console.Write("a[{0}][{1}]= ", i, j);
a[i][j] = int.Parse(Console.ReadLine());  }         }    }
 static void Change(int[][] a)    {   int[] z;   int n = a.Length;
 for (int i = 0; i < (n / 2); i++) { z = a[i]; a[i] = a[n-i-1]; a[n-i -1] = z;  }      }
 static void Main()    {
 int[][] a;  Input(out a);  Console.WriteLine("Исх. массив:");  Print(a);
Change(a);  Console.WriteLine("Измененный  массив:");  Print(a);    } }
```

Примеры

Дана прямоугольная матрица, элементами которой являются целые числа. Для каждого столбца подсчитать среднее арифметическое его нечетных элементов и записать полученные данные в новый массив. class Program {

```
static void Print(double[] a)    {
foreach (double elem in a) {Console.Write("{0} ", elem);}
Console.WriteLine();    }
 static void Print(int[,] a)    {
for (int i = 0; i < a.GetLength(0); i++)       {
for (int j = 0; j < a.GetLength(1); j++)   {
 Console.Write("{0,5:f2} ", a[i, j]);  }        Console.WriteLine();       }    }
static void Input(out int[,] a)    {
Console.Write("n= ");        int n = int.Parse(Console.ReadLine());
a = new int[n, n];         for (int i = 0; i < a.GetLength(0); i++)         {
for (int j = 0; j < a.GetLength(1); j++) { Console.Write("a[{0},{1}]= ", i, j);
a[i, j] = int.Parse(Console.ReadLine());   }   }    }
 static double[] F(int[,] a)     {
double[] b = new double[a.GetLength(1)];   for (int j = 0; j < a.GetLength(1); j++)
{int k = 0; for (int i = 0; i < a.GetLength(0); i++) {if (a[i, j] % 2 == 1) {
b[j] += a[i, j]; k++;}} if (k != 0) {b[j] /= k; }   }   return b; }
 static void Main()     {  int[,] a;  Input(out a);  Console.WriteLine("Исх.массив:");
Print(a);  double[] b=F(a); Console.WriteLine("Иск. массив:"); Print(b);     } }
```

Символы char

Тип char предназначен для хранения символа в кодировке Unicode.
Кодировка Unicode является двухбайтной, т.е. каждый символ представлен двумя байтами, а не одним, как это сделано кодировке ASCII, используемой в ОС Windows. Из-за этого могут возникать некоторые проблемы, если вы решите, например, работать посимвольно с файлами, созданными в стандартном текстовом редакторе Блокнот.
Символьный тип относится к встроенным типам данных C# и соответствует стандартному классу Char библиотеки .Net из пространства имен System.

Символы char . Основные методы

Метод	Описание
GetNumericValue	Возвращает числовое значение символа, если он является цифрой, и -1 в противном случае.
GetUnicodeCategory	Возвращает категорию Unicode-символа. В Unicode символы разделены на категории, например, цифры (DecimalDigitNumber), римские цифры (LetterNumber), разделители строк (LineSeparator), буквы в нижнем регистре (LowercaseLetter) и т.д.
IsControl	Возвращает true, если символ является управляющим (‘\n’, ‘\t’ или ‘\r’).
IsDigit	Возвращает true, если символ является десятичной цифрой.
IsLetter	Возвращает true, если символ является буквой.
IsLetterOrDigit	Возвращает true, если символ является буквой или десятичной цифрой.
IsLower	Возвращает true, если символ задан в нижнем регистре.
IsNumber	Возвращает true, если символ является числом (десятичным или шестнадцатеричным).
IsPunctuation	Возвращает true, если символ является знаком препинания.
IsSeparator	Возвращает true, если символ является разделителем.
IsUpper	Возвращает true, если символ задан в верхнем регистре.
IsWhiteSpace	Возвращает true, если символ является пробельным (пробел, перевод строки, возврат каретки).
Parse	Преобразует строку в символ, строка при этом должна состоять из одного символа.
ToLower	Преобразует символ в нижний регистр
ToUpper	Преобразует символ в верхний регистр

Символы char

```
class Program {
static void Main()  {
 Console.WriteLine("{0,5} {1,8} {2,15}","код",  "символ",
"назначение");
for (ushort i=0; i<255;i++)
  { char a=(char)i;    Console.Write("\n{0,5} {1,8}", i, a);
if (char.IsLetter(a)) Console.Write("{0,20}","Буква");
if (char.IsUpper(a)) Console.Write("{0,20}","Верхний регистр");
if (char.IsLower(a)) Console.Write("{0,20}","Нижний регистр");
if (char.IsControl(a)) Console.Write("{0,20}","Управл.символ");
if (char.IsNumber(a)) Console.Write("{0,20}","Число");
if (char.IsPunctuation(a)) Console.Write("{0,20}","Знак  препинан");
if (char.IsDigit (a)) Console.Write("{0,20}","Цифра");
if (char.IsSeparator (a)) Console.Write("{0,20}","Разделитель");
 if (char.IsWhiteSpace (a)) Console.Write("{0,20}","Пробел.сим.");
}  } }
```

Символы char

Используя символьный тип можно организовать массив символов и работать с ним на основе базового класса *Array*:

```
class Program {
static void Print(char[] a)  {
foreach (char elem in a)  {
Console.Write(elem);  }  Console.WriteLine();  }
 static void Main()  {
char[] a ={ 'm', 'a', 'X',  'i', 'M', 'u', 'S' , '!', '!', '!' };
Console.WriteLine("Исходный массива:");  Print(a);
for (int x=0;x<a.Length; x++)  {
if (char.IsLower(a[x]))  {  a[x]=char.ToUpper(a[x]);  }  }
Console.WriteLine("Измененный массив a:");  Print(a);
Console.WriteLine();  //преобразование строки в массив символов
char [] b="кол около колокола".ToCharArray();
Console.WriteLine("Исходный массив b:");  Print(b);
Array.Reverse(b);  Console.WriteLine("Измененный массив b:");
Print(b);  } }
```

Строковый тип string

Тип ***string***, предназначенный для работы со строками символов в кодировке Unicode, является встроенным типом C#. Ему соответствует базовый тип класса System.String библиотеки .Net. Тип string относится к ссылочным типам.

Существенной особенностью данного класса является то, что каждый его объект – это неизменяемая (immutable) последовательность символов Unicode. Любое действие со строкой ведет к тому, что создается копия строки, в которой и выполняются все изменения. Исходная же строка не меняется. Такой подход к работе со строками может показаться странным, но он обусловлен необходимостью сделать работу со строками максимально быстрой и безопасной. Например, при наличии нескольких одинаковых строк CLR может хранить их по одному и тому же адресу (данный механизм называется stringinterning), экономя таким образом память.

Строковый тип string

Создать объект типа string можно несколькими способами:

1) string s; // инициализация отложена

2) string s="кол около колокола";

инициализация строковым литералом

3) string s=@"Привет!

4) int x = 12344556; //инициализировали целочисленную переменную

string s=x.ToString(); //преобразовали ее к типу string

5) string s=new string (' ', 20); //конструктор создает строку из 20 пробелов

6) char [] a={'a', 'b', 'c', 'd', 'e'}; //создали массив символов

string v=new string (a); //создание строки из массива символов

7) char [] a={'a', 'b', 'c', 'd', 'e'};

string v=new string (a, 0, 2)

- создание строки из части массива символов, при этом: 0 показывает с какого символа, 2 – сколько символов использовать для инициализации.

Строковый тип string

С объектом типа string можно работать посимвольно, т.е поэлементно:

```
class Program {  static void Main()  {
 string a ="кол около колокола";
Console.WriteLine("Дана строка: {0}", a);
char b='o';   int k=0;
for (int x=0;x<a.Length; x++)   {
  if (a[x]==b)     {     k++;    }   }
Console.WriteLine("Символ {0} содержится в ней {1} раз", b, k );
} }
```

Строковый тип string. Основные методы

Название	*Вид*	*Описание*
Compare	Статический метод	Сравнение двух строк в лексикографическом (алфавитном) порядке. Разные реализации метода позволяют сравнивать строки с учетом или без учета регистра.
CompareTo	Экземплярный метод	Сравнение текущего экземпляра строки с другой строкой.
Concat	Статический метод	Слияние произвольного числа строк.
Copy	Статический метод	Создание копии строки.
Empty	Статическое поле	Открытое статическое поле, представляющее пустую строку.
Format	Статический метод	Форматирование строки в соответствии с заданным форматом.
IndexOf, LastIndexOf	Экземплярные методы	Определение индекса первого или, соответственно, последнего вхождения подстроки в данной строке.
IndexOfAny, LastIndexOfAny	Экземплярные методы	Определение индекса первого или, соответственно, последнего вхождения любого символа из подстроки в данной строке.
Insert	Экземплярный метод	Вставка подстроки в заданную позицию.
Join	Статический метод	Слияние массива строк в единую строку. Между элементами массива вставляются разделители.
Length	Свойство	Возвращает длину строки.
PadLeft, PadRigth	Экземплярные методы	Выравнивают строки по левому или, соответственно, правому краю путем вставки нужного числа пробелов в начале или в конце строки.
Remove	Экземплярный метод	Удаление подстроки из заданной позиции.

Строковый тип string. Основные методы

Replace	Экземплярный метод	Замена всех вхождений заданной подстроки или символа новыми подстрокой или символом.
Split	Экземплярный метод	Разделяет строку на элементы, используя разные разделители. Результаты помещаются в массив строк.
StartWith, EndWith	Экземплярные методы	Возвращают true или false в зависимости от того, начинается или заканчивается строка заданной подстрокой.
Substring	Экземплярный метод	Выделение подстроки, начиная с заданной позиции.
ToCharArray	Экземплярный метод	Преобразует строку в массив символов.
ToLower, ToUpper	Экземплярные методы	Преобразование строки к нижнему или, соответственно, к верхнему регистру.
Trim, TrimStart, TrimEnd	Экземплярные методы	Удаление пробелов в начале и конце строки или только с начала или только с конца соответственно.

Все методы возвращают ссылку на новую строку, созданную в результате преобразования копии исходной строки. Для того чтобы сохранить данное преобразование, нужно установить на него новую ссылку

Строковый тип string. Основные методы

Очень важными методами обработки строк, являются методы разделения строки на элементы - ***Split*** и слияние массива строк в единую строку - ***Join***.

```
class Program {  static void Main()  {
 string poems = "тучки небесные вечные странники";
char[] div = { ' '}; //создаем массив разделителей
// Разбиваем строку на части,
string[] parts = poems.Split(div);
Console.WriteLine("Результат разбиения строки на части: ");
for (int i = 0; i<parts.Length; i++)   {
Console.WriteLine(parts[i]);   }
 // собираем эти части в одну строку, в качестве разделителя используем  символ |
string whole = String.Join(" | ", parts);
Console.WriteLine("Результат сборки: ");
Console.WriteLine(whole);  } }
```

Строковый тип StringBuilder

Строковый тип S*tringBuilder* определен в пространстве имен *System.Text* и предназначен для создания строк, значение которых можно изменять. Объекты данного класса всегда создаются с помощью явного вызова конструктора класса, т.е. через операцию new. Создать объект класса *StringBuilder* возможно одним из следующих способов:

1) //создание пустой строки, размер которой по умолчанию 16 символов

StringBuilder a = new StringBuilder();

2) //инициализация строки и выделение памяти под 4 символа

StringBuilder b = new StringBuilder("abcd");

3) //создание пустой строки и выделение памяти под 100 символов

StringBuilder c = new StringBuilder(100);

4) //инициализация строки и выделение памяти под 100 символов

StringBuilder d = new StringBuilder("abcd", 100);

5) //инициализация подстрокой "bcd", и выделение памяти под 100 символов

StringBuilder d = new StringBuilder("abcdefg", 1, 3,100);

Строковый тип StringBuilder

С объектами класса StringBuilder можно работать посимвольно:

```
using System; //подключили пространство имен для работы склассом
StringBuilder using System.Text;
namespace Example {
class Program  {
static void Main()   {
StringBuilder a = new StringBuilder("кол около колокола");
Console.WriteLine("Дана строка: {0}", a);    char b='o';
int k=0;    for (int x=0;x<a.Length; x++)    {
if (a[x]==b)      {      k++;     }    }
 Console.WriteLine("Символ {0} содержится в ней {1} раз", b, k
);   }  } }
```

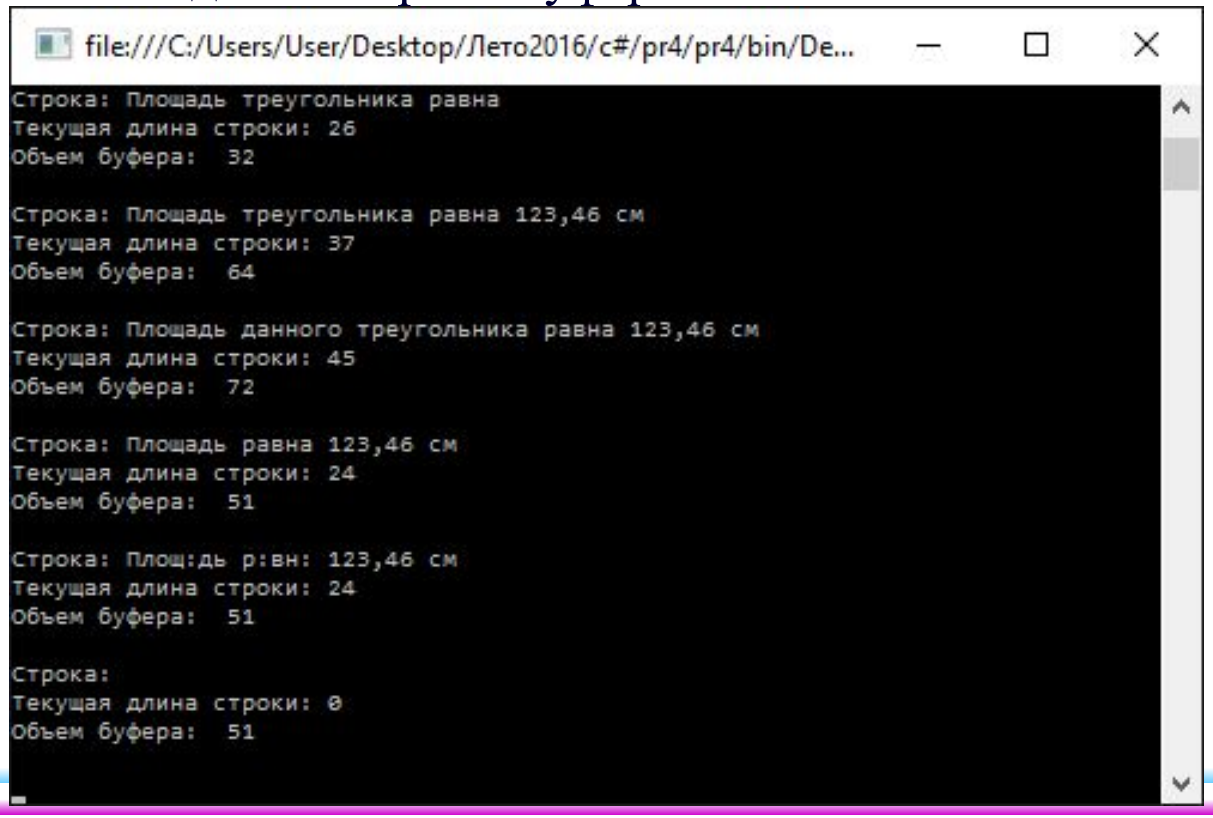
Название	*Член класса*	*Описание*
Append	Экземплярный метод	Производит добавление данных в конец строки. Разные варианты метода позволяют добавлять в строку величины любых встроенных типов, массивы символов, строки и подстроки string.
AppendFormat	Экземплярный метод	Производит добавление форматированной строки в конец данной строки.
Capacity	Свойство	Позволяет получить и установить емкость буфера. Если устанавливаемое значение меньше текущей длины строки или больше максимального, то генерируется исключение ArgumentOutOfRangeException
Insert	Экземплярный метод	Производит вставку подстроки в заданную позицию
Length	Изменяемое свойство	Возвращает длину строки. Присвоение ему значения 0 сбрасывает содержимое и очищает строку
MaxCapacity	Неизменное свойство	Возвращает наибольшее количество символов, которое может быть размещено в строке
Remove	Экземплярный метод	Производит удаление подстроки из заданной позиции
Replace	Экземплярный метод	Производит замену всех вхождений заданной подстроки или символа новой подстрокой или символом
ToString	Экземплярный метод	Преобразование в строку типа string
Chars	Изменяемое свойство	Позволяет обратиться к элементу строки по его номеру. Вместо него можно пользоваться квадратными скобками [].
Equals	Экземплярный метод	Используется для сравнения значения двух строк - возвращает true, только тогда, когда строки имеют одну и ту же длину и состоят из одних и тех же символов
CopyTo	Экземплярный метод	Копирует подмножество символов строки в массив char

Строковый тип StringBuilder

```
class Program {
static void Main()  {
StringBuilder str=new StringBuilder("Площадь");
Console.WriteLine("Максимальный объем буфера: {0} \n ",
str.MaxCapacity);   Print(str);
  str.Append(" треугольника равна");    Print(str);
  str.AppendFormat(" {0:f2} см ", 123.456);    Print(str);
  str.Insert(8, "данного ");   Print(str);
  str.Remove(7, 21);   Print(str);
  str.Replace("a", "…");   Print(str);
str.Length=0; Print(str);  }
 static void Print(StringBuilder a)  {   Console.WriteLine("Строка:
{0} ", a);   Console.WriteLine("Текущая длина строки: {0} ",
a.Length);   Console.WriteLine("Объем буфера:  {0} ",
a.Capacity);   Console.WriteLine();  } }
```

Строковый тип StringBuilder

Все выполняемые действия относились только к одному объекту str. Никаких дополнительных объектов не создавалось. Таким образом, класс StringBuilder применяется тогда, когда необходимо модифицировать исходную строку. Следует обратить внимание на то, что при увеличении текущей длины строки возможно изменение объема буфера, отводимого для хранения значения строки. А именно, если длина строки превышает объем буфера, то он увеличивается в два раза. Обратное не верно, т.е. при уменьшении длины строки буфер остается неизменным.



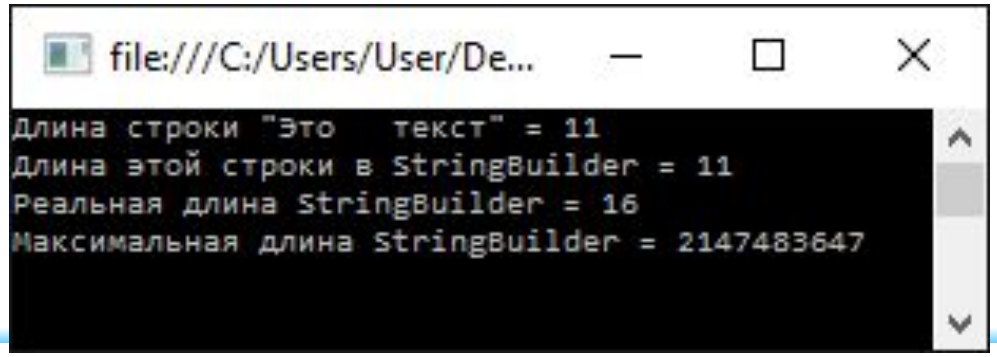
Сравнение классов String и StringBuilder

Основное отличие классов String и StringBuilder заключается в том, что при создании строки типа String выделяется ровно столько памяти, сколько необходимо для хранения инициализирующего значения. Если создается строка как объект класса StringBuilder, то выделение памяти происходит с некоторым запасом. По умолчанию, под каждую строку выделяется объем памяти, равный минимальной степени двойки, необходимой для хранения инициализирующего значения, хотя возможно задать эту величину по своему усмотрению. Например, для инициализирующего значения ”это текст” под строку типа String будет выделена память под 9 символов, а под строку типа StringBuilder – под 16 символов, из которых 9 будут использованы непосредственно для хранения данных, а еще 7 – составят запас, который можно будет использовать в случае необходимости.

Сравнение классов String и StringBuilder

Следующий пример демонстрирует различие между результатами, которые возвращают эти свойства.

```
class Program {  static void Main(string[] args)  {
string s = "Это   текст";
StringBuilder sb = new StringBuilder(s);
 Console.WriteLine("Длина строки \"{0}\" = {1}", s, s.Length);
Console.WriteLine("Длина этой строки в StringBuilder = {0}",
sb.Length);
Console.WriteLine("Реальная длина StringBuilder = {0}",
sb.Capacity);
Console.WriteLine("Максимальная длина StringBuilder = {0}",
sb.MaxCapacity);   Console.Read();  } }
```



Сравнение классов String и StringBuilder

Использование StringBuilder позволяет сократить затраты памяти и времени центрального процессора при операциях, связанных с изменением строк. В то же время, на создание объектов класса StringBuilder также тратится некоторое время и память. Как следствие, в некоторых случаях операции по изменению строк оказывается «дешевле» производить непосредственно с самими строками типа String, а в некоторых – выгоднее использовать StringBuilder. Пусть нам необходимо получить строку, состоящую из нескольких слов «текст» идущих подряд. Сделать это можно двумя способами.

Первый способ (прямое сложение строк):

string str = ""; for (int j = 0; j < count; j++) { str += "текст"; }

Второй способ (использование StringBuilder):

StringBuilder sb = new StringBuilder();

for (int j = 0; j < count; j++) { sb.Append("текст"); }

Строковый тип StringBuilder

Все выполняемые действия относились только к одному объекту str. Никаких дополнительных объектов не создавалось. Таким образом, класс StringBuilder применяется тогда, когда необходимо модифицировать исходную строку. Следует обратить внимание на то, что при увеличении текущей длины строки возможно изменение объема буфера, отводимого для хранения значения строки. А именно, если длина строки превышает объем буфера, то он увеличивается в два раза. Обратное не верно, т.е. при уменьшении длины строки буфер остается неизменным.