Основные понятия языка C#

Основными элементами языка программирования являются *типы данных*, *переменные*, *выражения*, *операторы* и *методы*.

Типы данных

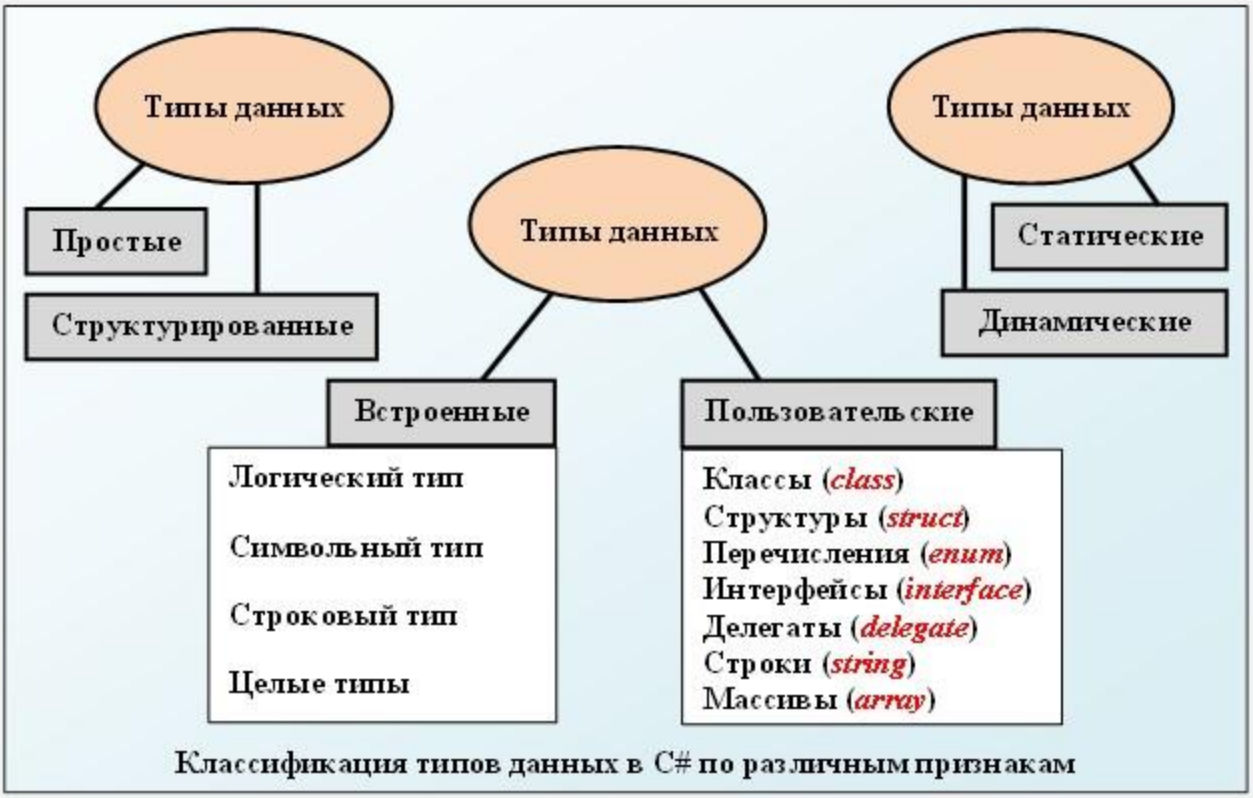
Язык C# является строго типизированным языком. Это означает , что все данные (константы и переменные), с которыми работает программа, имеют явно или неявно заданный тип. ***Под типом данных понимается набор доступных значений, которые может хранить переменная данного типа, и разрешенный набор операций над этими значениями.***

Определение - класс в C# (аналогично определению типа в CTS) *содержит*: 1) данные, задающие свойства объектов класса; 2) методы, определяющие поведение объектов класса; 3) события, которые происходят с объектами. Понятия тип и класс фактически являются синонимами (***класс*** - это хорошо определенный тип данных, ***объект*** - хорошо определенная переменная). Встроенные типы (integer, string) называют по-прежнему типами, а их экземпляры - переменными. C# - язык ООП. В C# сглажено различие между типом и классом. Все типы (встроенные и пользовательские) - одновременно являются классами, связанными отношением наследования. Родительским, базовым классом является класс Object. Все остальные типы (точнее, классы) являются его потомками, наследуя методы этого класса.

Типы языка C# можно классифицировать по разным признакам.

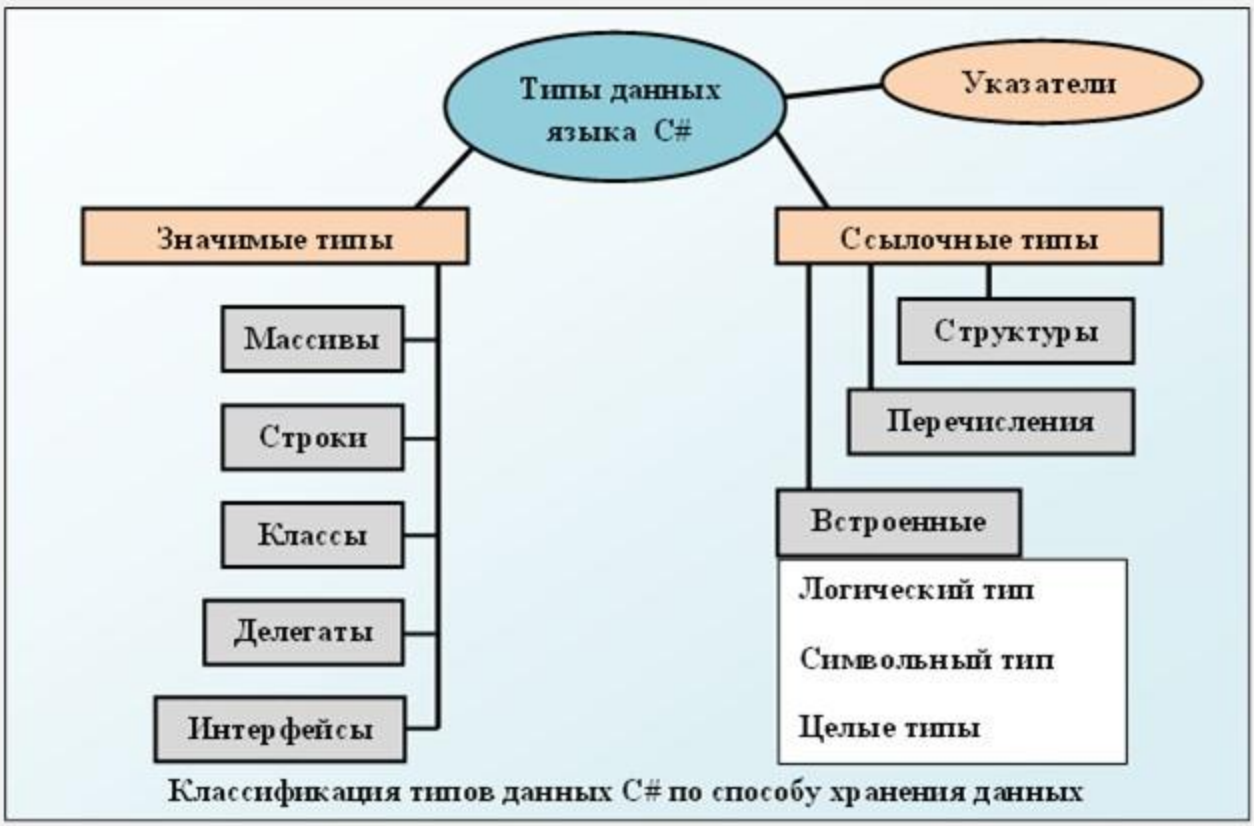
По типу строения элемента типы делятся на ***простые*** (типы не имеют внутренней структуры) и ***структурированные*** (типы состоят из элементов других типов - сложные типы).

По типу своего «создателя» типы делятся на ***встроенные*** (стандартные) и *определяемые* ***программистом*** (пользовательские). Пользовательские составляют саму программу. После того, как тип описан в программе, можно создавать и использовать объекты данного типа, точно так же, как если бы они были встроенными типами. Для ***статического*** типа память выделяется в момент объявления, при этом ее требуемый объем известен. Для данных ***динамического*** типа размер данных в момент объявления может быть неизвестен, память под них выделяется по запросу в процессе выполнения.



Классификация типов данных в C# по различным признакам

Типы C# разделяют по способу хранения на ***значащие типы*** (value types) и ***ссылочные типы*** (reference types).



Классификация типов данных C# по способу хранения данных

Основное различие между ними состоит в том, что **значащие** типы хранят непосредственно свои значения (элементы типов значений представляют собой последовательность битов в памяти – данные значение типа), а **ссылочные** типы хранят ссылки на свои значения (величина ссылочного типа хранит не данные, а ссылку - адрес, по которому расположены данные; сами данные хранятся в динамической памяти). Почти все встроенные типы являются ***значащими*** типами (только типы **Object** и **String** - ссылочные). Все пользовательские типы, кроме структур, являются ***ссылочными***.

Для значимого типа значение переменной хранится непосредственно в стеке. Т.к. значение может быть сложным (состоять из множества скалярных значений), говорят, что значение *разворачивается в стеке*. Поэтому иногда значимый тип называют ***развернутым*** типом. Почти все встроенные типы являются значащими типами (только типы Object и String - ссылочные. Все пользовательские типы, кроме структур, являются ссылочными.

Пример описания:

```
int nInt = new int();
nInt = 7;
float xFloat = 9.5f;
String Sstring = "";
```

Упаковка и распаковка. Для того, чтобы величины ссылочного и значимого типов могли использоваться совместно, необходимо иметь возможность преобразования из одного типа в другой. Преобразование из типа-значения в ссылочный тип называется *упаковкой* (boxing), обратное преобразование - *распаковкой* (unboxing).

```
int    n   = 12345;
object obj = n;       // boxing
int    n2  = (int)obj; // unboxing
```

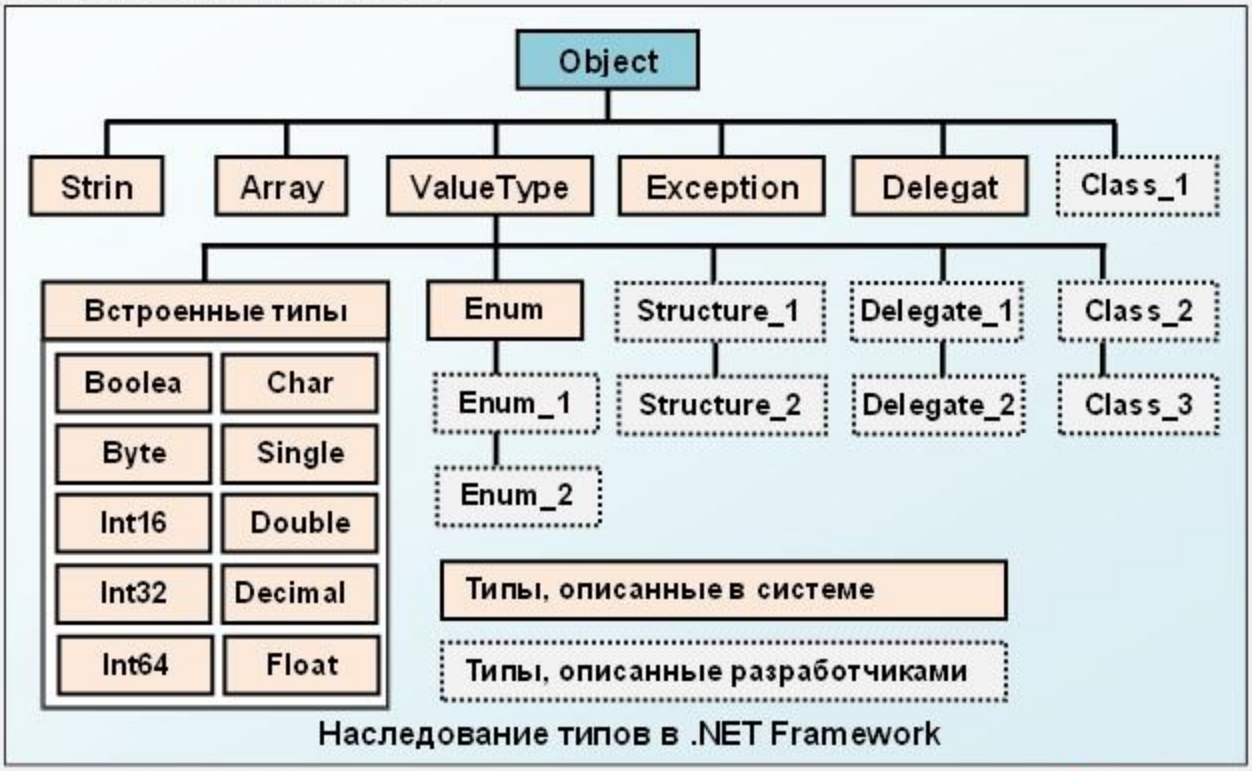
Упаковка - операция приведения к базовому типу (к типу *object*). Сначала в ***heap*** создаётся специальный объект ссылочного типа (*box*) в который копируется содержимое исходного объекта *value*-типа. При этом старый объект *value*-типа не пропадает, им можно пользоваться. Процесс упаковки - это **клонирование с преобразованием** в ссылочный тип. В результате упаковки преобразованный *value*-тип обменял достоинства и недостатки типов - значений, на достоинства и недостатки ссылочных типов. Процесс упаковки - это вынужденная мера, которой по возможности надо избегать.

Распаковка – операция (получение значения из запакованного значения), обратная упаковке не является противоположностью ***упаковке***. Для получения значения, лежащего внутри упаковки, не нужно ни создавать новый объект, ни копировать в него значение. Достаточно получить адрес данных, лежащих внутри упаковки, и использовать этот адрес в операциях с этим значением (*распаковка* - есть операция получения адреса значения внутри *упаковки*, то есть это разновидность операции приведения). Поэтому эффективность ***распаковки*** на порядок выше, чем ***упаковки***.

boxing = new instance + копирование; (instance-экземпляр класса, пример)

unboxing = приведение (вычисление адреса)

В языке C# все типы – *встроенные* и *пользовательские*, связаны *отношением наследования*.



Наследование типов в .NET Framework

Родительским (*базовым*) *классом* всех типов является *класс* ***Object*** (тип object). Все остальные типы являются его *потомками*, наследуя *методы* этого *класса*. У класса *Object* есть 4 наследуемых и 2 защищенных *метода*

Методы класса System.Object

Метод	Описание
Type **GetType**()	Используется с методами отражения для получения информации о типе данного объекта.

Метод	Описание
bool **Equals**()	Сравнивает две ссылки на объекты в период выполнения, чтобы определить, указывают ли они в точности один и тот же объект. Если две переменные ссылаются на один и тот же объект, возвращается *true*. В случае размерных типов этот метод возвращает *true*, если типы переменных идентичны и их значения равны.
int **GetHashCode**()	Возвращает хзш-код объекта (возвращает целочисленное значение, идентифицирующее конкретный экземпляр объекта данного типа). Хэш-функции используются в реализации класса, когда хэш-код объекта нужно поместить в хэш-таблицу для повышения производительности.
string **ToString**()	Используется по умолчанию для получения имени объекта. Его можно переопределить в производных классах, чтобы они возвращали понятное пользователю текстовое представление объекта.
void **Finalize**()	Вызывается в период выполнения для освобождение ресурсов перед сбором мусора. Этот метод можно вызывать, а можно и не делать этого. Поэтому не помещайте в него подлежащий исполнению код. Это правило выливается в нечто под названием *детерминированное завершение* (deterministic finalization
Object **MemberwiseClone**	Представляет *ограниченную копию* (shallow copy) объекта. Под этим я понимаю копию объекта, содержащую ссылки на другие объекты, но не копии этих объектов. Если ваши классы должны поддерживать *полную копию* (deep copy), которая действительно включает копии объектов, на которые она ссылается, то вам нужно реализовать интерфейс *ICloneable* и самому вручную производить клонирование или копирование.

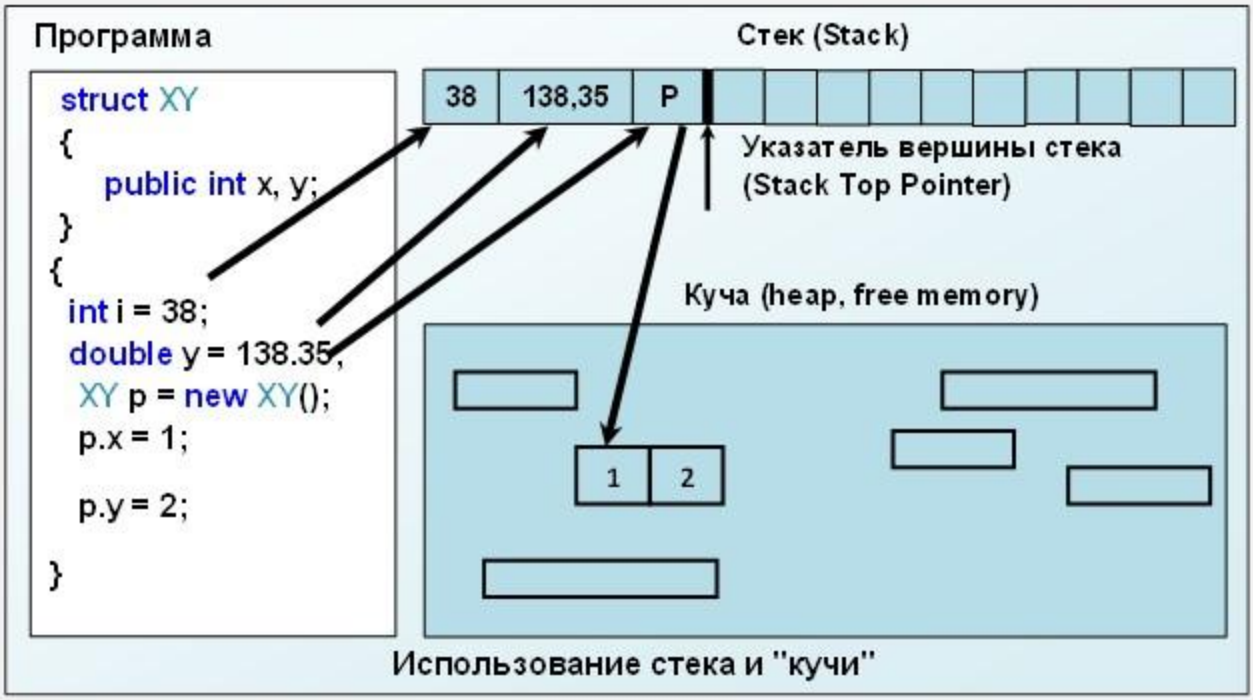
Классы-потомки при создании наследует все свойства и методы класса **Object**. Все *встроенные типы* переопределяют *методы* родителя нужным образом и добавляют собственные *поля*, *свойства и методы*.

Хранение данных в оперативной памяти

При выполнении программы все ее данные хранятся в оперативной памяти ПК. Количество памяти, необходимой для экземпляров данных, и место их хранения, зависит от их типа. При выполнении для хранения данных используются два участка оперативной памяти: ***стек*** (stack) и "***куча***" (heap).

Стек (stack) - линейный участок памяти (массив), который действует как структура данных типа(last-in, first-out - **LIFO**) «Последним пришел – первым ушел». Основной особенностью стека - данные могут добавляться только к вершине стека и удаляться из вершины. Добавление и удаление данных из произвольного места стека невозможно. Операции по добавлению элементов в стек и удаление элементов из стека выполняются очень быстро.

Однако размер стека, как правило, ограничен, и время хранения данных зависит от времени жизни переменной. Для всех локальных переменных методов и передаваемых методам параметров память выделяется в вершине стека. После того, как методы заканчивают работу вся выделенная память в стеке для их переменных автоматически освобождается.



Куча (heap) - это область оперативной памяти, в разных частях которой могут выделяться участки для хранения объектов классов. В отличие от стека, такие участки памяти в "куче" могут выделяться и освобождаться в любом порядке. Хотя программа может хранить элементы данных в "куче", она не может явно удалять их из нее. Вместо этого компонент среды CLR, называемый «Сборщиком мусора» (Garbage Collector, **GC**), автоматически удаляет не используемые участки "кучи", когда он определит, что код программы уже не имеет доступа к ним (не хранит их адреса).

Локальные переменные методов хранятся следующим образом:

Значащие типы хранятся в одном участке памяти, который хранит данные, и размещается в стеке. **Ссылочные типы** требуют для хранения два участка памяти: первый – содержит данные (сами объекты) и всегда размещается в "куче"; второй – размещается в стеке и содержит ссылку, которая указывает на адрес размещения объекта в "куче". **Поля классов** (переменные объекта) хранятся в участке "кучи", выделенном для конкретного объекта. Когда создается объект - выделяется участок "кучи", в котором хранятся все его данные (для методов объекта память не выделяется).

Встроенные типы данных

Все *встроенные типы* языка C# однозначно соответствуют системным типам платформы .Net Framework, описанным в пространстве имен System. Поэтому всюду, где можно использовать имя типа – int, с тем же успехом можно использовать и имя System.Int32.

Основные характеристики встроенных типов языка C#

Имя типа	Системный тип CLR	Значения - диапазон	Размер - точность
Логический тип			
Bool	System.Boolean	true, false	8 бит
Арифметический целочисленный тип (суффиксы: U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu)			
Sbyte	System.SByte	-128 – +127	Знаковое, 8 бит
Byte	System.Byte	0 – 255	Беззнаковое, 8 бит
Short	System. Int16	-32768 – 32767	Знаковое, 16 бит
Ushort	System.UInt16	0 – 65535	Беззнаковое, 16 бит
Int	System.Int32	$\approx(-2*10^9 – 2*10^9)$	Знаковое, 32 бит
Uint	System.UInt32	$\approx(0 – 4*10^9)$	Беззнаковое, 32 бит
Long	System.Int64	$\approx(-9*10^{18} – 9*10^{18})$	Знаковое, 64 бит
Ulong	System.UInt64	$\approx(0 – 18*10^{18})$	Беззнаковое, 64 бит
Арифметический тип с плавающей точкой (суффиксы: F, f, D, d)			
Float	System.Single	$+1.5*10^{-45} – +3.4*10^{38}$	32 бита (точность 7 цифр)
Double	System.Double	$+5.0*10^{-324} – +1.7*10^{308}$	64 бита (точность 15–16 цифр)
Арифметический тип с фиксированной точкой (суффиксы: M, m)			
Decimal	System.Decimal	$+1.0*10^{-28} – +7.9*10^{28}$	28–29 значащих цифр
Символьные типы			
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	Базовый тип всех *встроенных* и пользовательских типов	
Void		Отсутствие какого-либо значения	

Внутреннее представление величины целого типа – целое число в двоичном коде. В знаковых типах старший бит числа интерпретируется как знаковый (0-положительное число, 1-отрицательное). Отрицательные числа представляются в дополнительном коде – все разряды числа, кроме знакового разряда, инвертируются, затем к числу прибавляется единица, и знаковому биту присваивается единица. Беззнаковый тип позволяет представить только положительные числа (старший разряд рассматривается как часть кода числа. ***Внутреннее представление величины вещественного типа*** состоит из двух частей – мантиссы и порядка, каждая часть имеет знак. Длина мантиссы определяет точность числа, а длина порядка – его диапазон. Число $0{,}381*10^4$ хранит цифры мантиссы 381 и порядок 4 (для числа $560{,}3*10^2$ – мантисса 5603, порядок 5). Тип ***decimal*** предназначен для денежных вычислений, в которых критичны ошибки округления. Величины денежного типа нельзя использовать в одном выражении с вещественными величинами без явного преобразования типа. Любой встроенный тип C# соответствует стандартному классу библиотеки .NET, поэтому везде имя встроенного типа можно заменить именем класса библиотеки System (это значит - встроенные типы имеют поля и методы).

Поля встроенных типов:

Поле	Описание	Тип данных
MinValue	Минимальное значение, которое может иметь экземпляр типа	**byte, sbyte, int, uint long, ulong, short, ushort float, double, decimal**
MaxValue	Максимальное значение, которое может иметь экземпляр типа	**byte, sbyte, int, uint long, ulong, short, ushort float, double, decimal**
NegativeInfinity	Представляет минус бесконечность. Значением этой константы является результат деления отрицательного числа на 0. Данная константа возвращается в случае, если результат операции меньше, чем **MinValue**.	**float, double**
PositiveInfinity	Представляет плюс бесконечность. Значением этой константы является результат деления положительного числа на 0. Данная константа возвращается в случае, если результат операции больше, чем **MaxValue**..	**float, double**

Поле	Описание	Тип данных
NaN	Представляет нечисловое значение (NaN). Значением этой константы является результат деления 0 на 0. Данная константа возвращается в том случае, если результат операции является неопределенным.	float, double
Epsilon	Представляет наименьшее ненулевое положительное значение. Значение свойства **Epsilon** отражает наименьшее положительное значение, существенное в числовых операциях или сравнениях при значении экземпляра, равном 0.	float, double
MinusOne	Представление числа -1 в формате decimal - значения	decimal
One	Представление числа 1 в формате decimal - значения	decimal
Zero	Представление числа 0 в формате decimal - значения	decimal

Значения полей арифметических (встроенных) типов C# (.Net)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _Методы_встроенных_типов
{
   class Program
   {
      static void Information_Type(string TypeName, object Value)
      {
  const string TypeFormat  = "  Встроенный тип [C#]{0,10} [.Net
FrameWork]{1,15}";
         const string numberFormat = "{0,48:N0}";//выводить с разделителями
         const string floatFormat  = "{0,48}";
         string MinValueS, MaxValueS, EpsilonS, S = "";
         switch (TypeName)
         {case ("byte"):
               MinValueS = string.Format(numberFormat, byte.MinValue);
               MaxValueS = string.Format(numberFormat, byte.MaxValue); break;
         case ("short"):
            MinValueS = string.Format(numberFormat, short.MinValue);
            MaxValueS = string.Format(numberFormat, short.MaxValue); break;
         case ("int"):
             MinValueS = string.Format(numberFormat, int.MinValue);
```

```
MaxValueS = string.Format(numberFormat, int.MaxValue); break;
            case ("long"):
               MinValueS = string.Format(numberFormat, long.MinValue);
               MaxValueS = string.Format(numberFormat, long.MaxValue);
//MinValueS = long.MinValue.ToString();//MaxValueS = long.MaxValue.ToString();
               break;
            case ("float"):
MinValueS = string.Format(floatFormat, float.MinValue);
               MaxValueS = string.Format(floatFormat, float.MaxValue);
               EpsilonS  = float.Epsilon.ToString();
               S = "\nEpsilon = " + EpsilonS; break;
            case ("double"):
               MinValueS = string.Format(floatFormat, double.MinValue);
               MaxValueS = string.Format(floatFormat, double.MaxValue);
               EpsilonS  = double.Epsilon.ToString();
               S = "\nEpsilon = " + EpsilonS; break;
            case ("decimal"):
               MinValueS = string.Format(numberFormat, decimal.MinValue);
               MaxValueS = string.Format(numberFormat, decimal.MaxValue); break;
            default:
                  MinValueS= "не определен";
                  MaxValueS= "не определен"; break;
           }
           Console.WriteLine(TypeFormat, TypeName, Value.GetType());
           Console.WriteLine("MIN значение = " + MinValueS);
           Console.WriteLine("MAX значение = " + MaxValueS + S);
       }
       static void Main(string[] args)
       {
          byte b = 1;
          short s = 1;
          int i = 1;
          long l = 1;
          float f = 1;
          double d = 1;
          decimal dec = 1;
          Information_Type("byte", b);
          Information_Type("short", s);
          Information_Type("int", i);
```

```
        Information_Type("long", l);
        Information_Type("float", f);
        Information_Type("double", d);
        Information_Type("decimal",dec);
        double D1 = -2.12, D11 = 2.12, D2 = 0, D3 = D1 / D2, D4 = D11 / D2;
        Console.WriteLine("D1 = -2.12, D2 = 0, D3 = D1 / D2 = {0,15}", D3);
        Console.WriteLine("D11 = 2.12, D2 = 0, D4 = D11 / D2 = {0,15}", D4);
        Console.ReadLine();
    }
  }
}
```

Результат выполнения:

```
Встроенный тип [C#]     byte    [.Net FrameWork]    System.Byte
MIN  значение =                          0
MAX значение =                         255
 Встроенный тип [C#]    short    [.Net FrameWork]   System.Int16
MIN  значение =                     -32 768
MAX значение =                      32 767
 Встроенный тип [C#]      int     [.Net FrameWork]   System.Int32
MIN  значение =                  -2 147 483 648
MAX значение =                   2 147 483 647
 Встроенный тип [C#]      long   [.Net FrameWork]   System.Int64
MIN  значение =              -9 223 372 036 854 775 808
MAX значение =               9 223 372 036 854 775 807
 Встроенный тип [C#]      float    [.Net FrameWork]  System.Single
MIN  значение =                  -3,402823E+38
MAX значение =                   3,402823E+38
Epsilon = 1,401298E-45
 Встроенный тип [C#]    double   [.Net FrameWork]  System.Double
MIN  значение =             -1,79769313486232E+308
MAX значение =              1,79769313486232E+308
Epsilon = 4,94065645841247E-324
 Встроенный тип [C#]   decimal   [.Net FrameWork]  System.Decimal
MIN  значение =       -79 228 162 514 264 337 593 543 950 335
MAX значение =        79 228 162 514 264 337 593 543 950 335
D1   = -2.12, D2 = 0, D3 = D1  / D2 =  -бесконечность
D11 =  2.12, D2 = 0, D4 = D11 / D2 =   бесконечность
```

Неявное, явное преобразование типов данных

Над элементами встроенных типов могут осуществляться преобразования из одного типа данных в другой. Такие преобразования могут инициироваться как пользователем (***явные преобразования***), так и системой выполнения программ (***неявные преобразования*** - *автоматические*). *Явные* и *неявные* преобразования типов могут инициироваться пользователем. Неявные преобразования инициируются CTS и производятся автоматически. При этом результат неявного преобразования всегда успешен и не приводит к потере точности. Система CTS среды .NET обеспечивает безопасную типизацию, т.е. гарантирует отсутствие побочных эффектов (переполнение оперативной памяти компьютера, некорректное преобразование типов и т.д.).

Типы T_1 и T_2 считаются **совместимыми**, если: (T_1{источник}$= T_2${приемник})

- оба они есть один и тот же тип;
- оба они вещественные;
- оба они целые и диапазон значений T_2 является подмножеством значений T_1;
- тип T_1 - вещественный, а T_2 - целый;
- тип T_1 - строка, а T_2 - символ;
- тип T_2 является прямым или косвенным потомком класса T_1

Неявное (автоматическое) преобразование типов. При присвоении значения одного типа данных переменной другого типа будет выполнено автоматическое преобразование типов, если

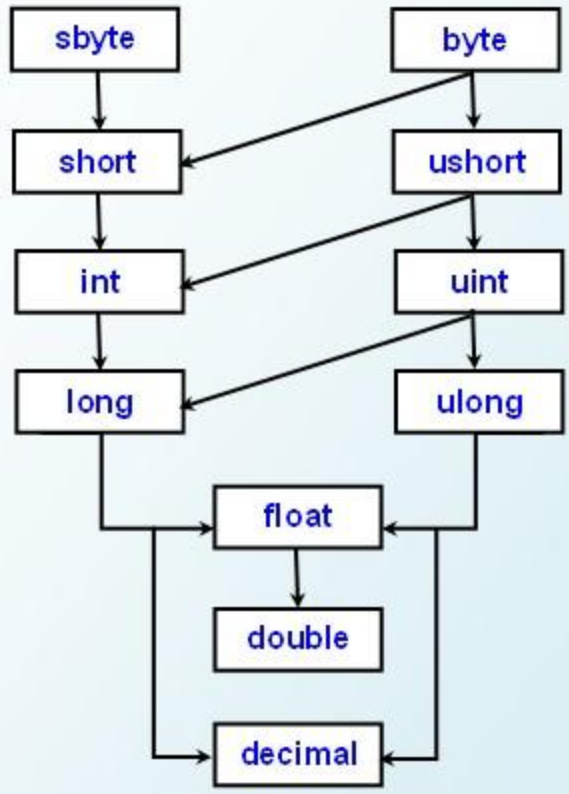
- эти два типа совместимы;
- тип приемника больше (имеет больший диапазон представления чисел) типа источника.

При соблюдении этих условий выполняется преобразование с расширением, или расширяющее преобразование. Например, тип **int** - "большой" тип, чтобы сохранить любое допустимое значение **byte**, а т.к. **int**, так и **byte** - целые типы, здесь может быть применено автоматические преобразование. Для расширяющих преобразований числовые типы, включая целочисленные и с плавающей точкой, совместимы один с другим.

демонстрация автоматического преобразования типов из long в double (преобразование является расширяющим и выполняется автоматически).

```
long L;     double D;
L = 100123285L;
D = L;  //Результат: L = D = 100123285
```

Преобразование типов в выражениях происходит ***неявно*** (автоматически). Если одни из операндов выражения имеет тип, изображенный на более низком уровне, чем другой, то он приводится к типу второго операнда при наличии пути между ними. Если на диаграмме задан путь (стрелками) от **A** к **B**, то это означает существование неявного преобразования из **A** в **B**. Путь, указанный на диаграмме, может быть длинным, но это не означает, что выполняется вся последовательность преобразований данного пути. Наличие пути говорит лишь о существовании неявного преобразования, а само преобразование выполняется только один раз, - из типа источника A в тип назначения B. Если путей несколько, то выбирается наиболее короткий.



Иерархия преобразований

Явное преобразование типов. Для ***явного*** преобразования применяется оператор приведения (операция кастинга).

Синтаксис объявления оператора приведения в C# имеют вид:

<целевой тип> <выражение>

При выполнении явного преобразования ответственность за его корректность полностью возлагается на программиста. Явные преобразования, которые инициируются программистом (или пользователем приложения) требуют явного вызова (при этом они могут завершаться ошибкой или приводить к потере точности).

Демонстрация явного преобразования типов:

```
int i0 = 254;
short i1 = (short)i0;        //Результат: i1 = 254 (без потери точности)
int x = 254568;
short x1 = (short)x;         //Результат: x1 = -7576 (потеря точности)
double d0 = 3.141592536;
float d1 = (float)d0;        //Результат: d1 = 3.141593 (потеря точности)
long d2 = (long)d0;          //Результат: d2 = 3 (потеря точности)
```

Для значимых типов значение **null** не принадлежит множеству возможных значений. Но иногда полезно, чтобы переменная значимого типа имела неопределенное значение. C# позволяет из любого значимого типа данных построить новый тип, отличающийся лишь тем, что множество возможных значений дополнено специальным значением **null**. Такие типы называются типами, допускающими ***неопределенное значение*** (***Nullable Types***).

*Синтаксис объявления типа **T**, допускающего неопределенные значения:*

System.Nullable<T> или эквивалентная форма записи **T?**

Переменные таких типов могут получать значение **null** либо в результате присваивания, либо в процессе вычислений. Если при вычислении выражения один из операндов будет иметь значение **null**, то и результат вычисления выражения будет иметь то же значение **null**. Над переменными этого типа определена специальная операция склеивания: **A ?? B** - результатом вычисления выражения будет операнд **A**, если значение **A** не равно **null**, и **B**, если первый операнд равен **null**.

Преобразование из типа **T** в тип **T?** - безопасное преобразование и потому может выполняться неявно. В другую сторону преобразование является опасным и должно выполняться явно путем - приведения к типу.

```
int x = 3, y = 7;
int? x1 = null, y1, z1;
y1 = x + y;
z1 = x1 ?? y1; //Результат: x1 = , y1 = 10, z1 = 10
```

Переменные и константы

Переменные и типы - тесно связанные понятия. С объектной точки зрения переменная - это экземпляр типа. Переменную можно рассматривать как сущность, обладающую именем, значением и типом. Имя и тип задаются при объявлении переменной и остаются неизменными на все время ее жизни. Значение переменной может меняться в ходе вычислений, эта возможность вариации значений и дало имя понятию переменная (v**ariable**) в математике и программировании. Задание начального значения переменной называется ее *инициализацией*. Важной новинкой C# является требование обязательной инициализации переменной до начала ее использования. Попытка использовать неинициализированную переменную приводит к ошибкам, обнаруживаемым еще на этапе компиляции. Инициализация переменных, как правило, выполняется в момент объявления, хотя и может быть отложена.

Любая программа использует данные в виде переменных и констант.

Переменные **-** это именованные участки памяти, которые могут хранить либо значения некоторого типа (для значащих типов, в стеке), либо ссылки на экземпляры некоторых классов (для ссылочных типов, ссылки на объекты, расположенные в "куче").

В C# переменные описывают:

- *поля классов* (объявляются в классе и создаются для каждого объекта);
- *локальные переменные методов* (создаются при каждом вызове метода).

Объявление переменной. Для создания и использования в программе переменных, их нужно объявить и инициализировать в любом месте метода, но до того, как они будет использоваться.

Синтаксис объявления переменных в C# имеют вид:

[<атрибуты>] [<модификаторы>] <тип> **<имя_переменной>** , где

[<модификаторы>] = { <режим доступа>, static, const }

Атрибуты - средство добавления декларативной (вспомогательной) информации к элементам программного кода. Их назначение: организация взаимодействия между модулями, дополнительная информация об условиях выполнения кода, управление ***сериализацией*** (сохранение информации), отладка и др. Модификаторы прав доступа, обеспечивающие реализацию принципа инкапсуляции, используются при объявлении классов, структур и их составляющих компонентов. При объявлении ***переменных*** указывается их ***тип*** и ***имя***, где ***имя_переменной*** - название **переменной** или название переменной с ***инициализацией***. После типа можно указывать ***список имен***.

Инициализация переменной. Прежде чем использовать переменную, ей необходимо присвоить значение. Инициализацию можно осуществлять либо с помощью присваивания значений или с использованием конструкции **new** и вызовом конструктора по умолчанию. Сделать это можно двумя способами: 1) можно присвоить значение переменной в отдельном операторе после ее объявления; 2) можно сделать это в одном операторе (сначала объявить тип переменной и затем присвоить ей значение).

Синтаксис оператора инициализации переменных имеют вид:

<тип> <имя_переменной> = <Value>

Value - значение, которое присваивается переменной при создании. Значение должно быть совместимо с указанным типом.

```
int x, s;   //переменные без инициализации
int y = 0, u = 77;  //обычный способ инициализаци
float w1 = 0f, w2 = 5.5f, w3 = w1 + w2 + 125.25f;//допустимая инициализация
int z = new int();  //допустимая инициализация в объектном стиле
x = u + y; //теперь x инициализирована
```

Динамическая инициализация. В C# можно инициализировать переменные не только значениями (которые будут получены в результате вычисления выражения) но и динамически, используя любое действительное выражение.

```
double radius = 4, height = 5; // radius - радиус, height - высота
double volume = 3.1416 * radius * radius * height;
//Переменная volume инициализируется динамически во время выполнения
Console.WriteLine("Объем цилиндра равен: " + volume);
```

Неопределенный тип переменной - var. Для переменной можно задать неопределенный тип (**var**) и присвоить значение. В этом случае компилятор **автоматически определит** тип значения и назначит его переменной.

Синтаксис оператора инициализации переменных в C# имеют вид:

<var> <имя_переменной> = < Value >

Value - значение, которое присваивается переменной во время ее создания.

Объявление: **var** name="Петров А.В." аналогично: **string** name="Петров А.В."

При использовании для определения переменной - неопределенного типа - обязательно нужно инициализировать переменную при ее объявлении.

Статический тип dynamic. Использование типа позволяет описывать вызовы методов, операторов, индексаторов и делегатов, а так же обращения к свойствам и полям объета, которые минуют статическую проверку типизации и будут разрешены в процессе выполнения.

```
dynamic d = GetDynamicObject(…);
d.M (7);                  // вызов метода
d.f = d.P;                // получение и установка значений полей и свойств
d[“one”] = d[“two”];  // получение и установка значений индексатора
int i = d + 3;            // вызов оператора
string s = d (5,7);       // описание делегата
```

В отличие от слова ***var***, объект, объявленный как ***dynamic***, может менять тип во время выполнения. При использовании слова ***var*** определение типа объекта откладывается. Но как только он определен компилятором, изменять

его уже нельзя. Что касается объекта ***dynamic***, то можно не просто изменить его тип, но делать это многократно. Это отличается от приведения объекта от одного типа к другому (при приведении объекта создается новый объект, с другим, но совместимым типом).

```
class ExClass
{
    public ExClass() { }
    public ExClass(int v) { }
    public void exMethod1(int i) { }
    public void exMethod2(string str) { }
}
static void Main(string[] args)
{//использование типа dynamic
    ExClass ec = new ExClass();
    //ec.exMethod1(10, 4); Ошибка на этапе компиляции
    //ec.exMethod2("some arg", 4); Ошибка на этапе компиляции
    //ec.exMethod2("arg",4, 23,54); Ошибка на этапе компиляции
    dynamic dyn_ec = new ExClass();
  dyn_ec.exMethod1(10, 4);  Ошибка на этапе выполнения
  dyn_ec.someMethod("some arg", 7, null); Ошибка на этапе выполнения
  dyn_ec.nonexistentMethod(); Ошибка на этапе выполнения
}
```

Если у метода **exMethod1** экземпляра класса **ExClass** в коде имеется только один параметр, то компилятор определяет: вызов метода **ec.exMethod1(10,4)** является недопустимым (т.к. он содержит два аргумента) и приводит к ошибке компилятора. Следующие вызовы: **dyn_ec.exMethod1(10,4)** и др. не проверяется компилятором, поскольку тип dyn_ec является типом **dynamic**. Поэтому ошибка компилятора не возникает. **Ошибка перехватывается** во время выполнения и вызывает **исключение времени выполнения**.

Динамические возможности C# (Dynamic Language Runtime) – это часть исполняющей среды динамического языка. Среда DLR представляет собой среду выполнения, которая добавляет набор служб для динамических языков в среду CLR. Как и CLR, среда DLR является частью .NET Framework и поставляется в установочных пакетах .NET Framework и Visual Studio.

Классы A и B находятся в отношении "**родитель - наследник**", если при объявлении класса **B** класс **A** указан в качестве родительского класса (класс **A** - родитель класса **B,** класс B - наследник класса A).

Классы A и B находятся в отношении "**клиент - поставщик**", если одним из полей класса **B** является объект класса **A** (класс **A** - поставщик класса **B**, класс **B** - клиент класса **A**). Объект класса A "вложен" в класс B (отношение "клиент - поставщик" называют отношением *вложенности* или *встраивания*).

Глобальные переменные уровня модуля В языках программирования переменные могут объявляться на уровне модуля. Такие переменные называются **глобальными**. Их область действия распространяется на весь модуль (можно использовать и в других модулях). Глобальные переменные играют важную роль, обеспечивая способ обмена информацией между различными частями модуля. Обратная сторона глобальных переменных – их *опасность*. Если какая-либо процедура, в которой доступна глобальная переменная, некорректно изменит ее значение, то ошибка может проявиться в другой процедуре, использующей эту переменную. Найти причину ошибки в таких ситуациях бывает чрезвычайно трудно. В языке C# роль модуля играют *классы*, *пространства имен*, *проекты*, *решения*. Поля классов могут рассматриваться как ***глобальные переменные*** класса. Но здесь у них особая роль. ***Данные*** (поля) - основание, на котором существует класс. Каждый экземпляр класса - это отдельный объект. ***Поля*** экземпляра класса (открытые и закрытые) – *глобальная информация*, доступная всем методам класса, обрабатывающим данные. *Статические поля класса* хранят информацию, общую для всех экземпляров класса (такие поля представляют опасность, поскольку каждый экземпляр способен менять их значения).

В других видах модуля - **пространствах имен**, **проектах**, **решениях** - нельзя объявлять **переменные**. В пространствах имен в языке C# можно объявлять только *классы* и их частные случаи: ***структуры***, ***интерфейсы***, ***делегаты***, ***перечисления***. ***Глобальных переменных уровня модуля, в привычном для других языков программирования смысле, в языке C# нет***. Классы не могут обмениваться информацией, используя глобальные переменные. Все взаимодействие между ними обеспечивается способами, стандартными для объектного подхода. Между классами существуют два типа отношений - *клиентские* и *наследования*, а основной способ инициации вычислений - это

вызов метода для объекта или вызов обработчика события. Поля класса и аргументы методов позволяют передавать и получать нужную информацию.

Устранение глобальных переменных - источника (трудно находимых) ошибок повышает надежность создаваемых на языке C# продуктов.

Локальные переменные уровня процедуры (метода) Во всех языках программирования (и в C#) основной контекст, где появляются переменные - это методы (процедуры и функции). Переменные, объявленные на уровне метода, называются **локальными** (переменные локализованы в методе).

В некоторых языках программирования (Паскаль), локальные переменные должны быть объявлены в вершине процедуры. Иногда правило заменяется менее жестким (по сути аналогичным) правилом: при объявлении в любом месте процедуры переменной - ее область видимости распространяется на всю процедуру. В языке C# (как и в C/C++) переменную можно объявлять в любой процедуры. *Область ее видимости распространяется от точки объявления до конца процедуры.* Процедурный блок в C# имеет сложную структуру: в него могут быть вложены блоки, связанные с операторами выбора, цикла и так далее. В каждом таком блоке допустимы вложения блоков. В каждом внутреннем блоке допустимы объявления переменных. **Переменные**, объявленные во внутренних блоках, **локализованы** именно в *этих блоках*, их область видимости и время жизни определяются *этими блоками*. **Локальные переменные** начинают существовать при достижении в блоке *точки объявления* и перестают существовать, когда процесс вычисления завершает выполнение операторов блока. При объявлении локальным переменным отводится память, при завершении вычислений (окончании блока) память освобождается. *Выделение памяти* (начало жизни) *переменной*, объявленной в блоке, происходит не в момент входа в блок, а тогда, когда достигается точка *объявления локальной переменной*.

Глобальные переменные уровня процедуры (метода) Процедура имеет сложную структуру с вложенными блоками (здесь возникает тема глобальных переменных). Переменная, объявленная во внешнем блоке - *глобальная*, по отношению к внутренним блокам. В языках программирования во внутренних блоках разрешается объявлять переменные с именами, совпадающими с именами глобальных переменных. **Конфликта нет**: внутреннее определение локальной переменной сильнее внешнего - область видимости внешней

глобальной переменной сужается и не распространяется на внутренние блоки, где объявлена переменная с подобным именем. В блоке действует локальное объявление блока, при выходе область действия внешнего имени восстанавливается. *В C# во внутренних блоках* ***запрещено использование имен****, совпадающих с именем, использованном во* ***внешнем блоке***.

Глобальные переменные уровня процедуры в C# ***есть***, конфликта имен между глобальными и локальными переменными - нет. Область видимости глобальных переменных процедурного блока распространяется на весь блок, в котором они объявлены, начиная от точки объявления, и не зависит от существования внутренних блоков. Когда говорят, что в C# нет глобальных переменных, то имеется в виду их отсутствие на уровне модуля, а уже потом идет речь об отсутствии конфликтов имен на процедурном уровне.

Области видимости локальных переменных Область видимости переменной (*variable scope*) - часть программы, в которой переменную можно использовать. В общем случае областью видимости локальной переменной является код программы, от строки, в которой она объявляется, до первой фигурной скобки, завершающей блок (метод), в котором она объявлена.

Область видимости переменной определяется правилами:

- **Поле** (*field*) – переменная - член класса, находится в области видимости до тех пор, пока в этой области находится содержащий поле класс;
- **Локальная переменная** (*local variable*) - находится в области видимости до конца блока операторов или метода, в котором она объьявлена;
- **Локальная переменная**, которая объявлена в операторах **for**, **while** (подобных им) видна в пределах тела цикла (оператора).

```
class Test //описание класса
 {
    int  x, y; //координаты точки
    string name; //имя точки
    public Test(int x, int y, string name) //конструктор с параметрами
    {
          this.x = x; this.y = y; this.name = name;
     }
```

```
    public void LocalVar(int x) // x - поле и формальный аргумент метода
    {
      int x=0; //Ошибка! Локальная переменная с именем x - существует
      int y = 77; // y - поле класса, и локальная переменная
      string s = name; //name - имя поля класса Test
      if (s == "Точка1")
      {
              int u = 5; int v = u + y; x += 1;
              Console.WriteLine("y= {0}; u={1}; v={2}; x={3}", y, u, v, x);
       }
       else
       {
              int u = 7; int v = u + y;
              Console.WriteLine("y= {0}; u={1}; v={2}",  y, u, v);
        }
Console.WriteLine("y= {0};u={1};v={2}",y,u,v);//Ошибка! u,v - не существуют
static int Count = 1;//Ошибка! Лок. переменные не могут быть статическими
Console.WriteLine("x= {0};sum={1}", x,sum); //Ошибка! Исп. sum до объяв.
        int i;  long sum = 0;
        for (i = 1; i < x; i++)
        {
     float y = 7.7f; // Ошибка! Коллизия имен: переменная y - уже существует
              sum += i;
         }
         Console.WriteLine("x= {0}; sum ={1}", x, sum);
       } }
```

Если есть пара одноименных переменных: x - в параметре конструктора и x - как переменная класса, то для уточнения переменной класса используется конструкция - this.

```
    class Vector
    {
      float x, y; //Координаты
      public Vector(float x, float y)
      {
         this.x=x;
         this.y=y;
      }     }
```

```
static void Main(string[] args)
{
   int x; // Переменная x известна всему коду в пределах метода Main()
   x = 10;
   if (x == 10)
   {//Создается новая область видимости
      int y = 20;//Эта переменная будет видна только в рамках этого блока.
      // В данном блоке видны обе переменные, x и y.
      Console.WriteLine("Видны x и y: " + x + " " + y);
      x = y * 2;
    } // y = 100; // Ошибка! Переменная y здесь не видна!
    Console.WriteLine("Значение переменной x равно: " + x);
  }
```

Внутри блока переменная может быть объявлена в любом месте, после чего она становится действительной. Если определить переменную в начале метода, она будет доступна всей программе в рамках метода. Если объявить переменную в конце блока, то не нельзя ее использовать до момента ее объявления, т.к. программа не будет иметь к ней доступа. Переменные создаются при их объявлении в какой-либо области видимости (при входе в данную область) и уничтожаются при выходе из области. Это значит, что переменная не будет сохранять свое значение при выходе из своей области видимости (переменная, объявленная в пределах метода, не будет хранить свои значения между вызовами, переменная, объявленная в рамках блока, будет терять свое значение при выходе из блока). Если при объявлении переменной происходит ее инициализация, то переменная будет повторно инициализироваться всегда при входе в блок, где она объявлена.

Создание значений ссылочных переменных При объявлении ссылочной переменной выделяется память для хранения ссылки, память для хранения самого объекта - не выделяется. При выделении памяти для самого объекта необходимо использовать операцию: **new**: **new** <тип>[(параметры)] - операция **new** выделяет и инициализирует память для экземпляра заданного типа (для ссылочных переменных - в куче); после оператора **new** задается имя типа, экземпляр которого создается (в скобках заданы параметры - (вызов конструктора класса) **new TypeName (…)** Если выделение памяти выполняется успешно, то оператор **new** возвращает ссылку на выделенный и инициализированный объект в куче.

```
Box b;          // объявление ссылочной переменной
b = new Box();  // выделение памяти для объекта
```

Два этих оператора могут быть объединены в один:

```
Box b1 = new Box(); // объявление и инициализация
```

Операция ***new*** имеет высший уровень *приоритета* и используется для создания нового объекта заданного типа (вызова конструктора). Результатом выполнения данной операции является ссылка на созданный объект. выполнения данной операции является ссылка на созданный объект.

Константы В C# *константы* задаются в виде *литералов* (набора символов) и *именованных констант*. Набор символов: **y = 7.7;** Значение *константы* "7.7" является одновременно именем и типом. По умолчанию константы с дробной частью имеют тип ***double***. Точный тип константы можно задать с помощью символа, стоящего после литерала (в любом регистре). Такими символами могут быть: *f* – тип float; *d* – тип double; *m* – тип decimal.

Синтаксис объявления ***именнованных констант*** *в C# имеют вид:*

<const> <тип_константы> = < Value >;

Именованные константы обязательно должны быть *инициализированы* и инициализация не может быть отложенной: **const float c = 0.1f;**

Строковые константы Под строковыми константами понимается последовательность символов заключенная в двойные кавычки (""). В C# существуют два вида строковых констант:

- обычные константы (строка символов в кавычках) - "ssss";
- @-*константы*(дословные строковые литералы) - константа с знаком @.

```
s1 = "c:\\c#book\\ch5\\chapter5.doc"; s2 = @"c:\c#book\ch5\chapter5.doc";
```

Операции и выражения ***Операции*** - это действия над данными. Данные, участвующие в операции, часто называют *операндами*. В качестве операнда может выступать константа, переменная или вызов какого-нибудь метода. Для каждой операции используется соответствующий ей знак операции, состоящий из одного или нескольких символов. В результате выполнения операций всегда получается какое-то значение, представляющее собой результат выполнения операции. Переменные и константы могут участвовать (объединяться) в создании выражений с помощью операций. ***Операция*** в языке C# - это термин или символ, получающий на вход один или несколько

операндов (переменных, констант) или выражений (переменных, констант, связанных между собой знаками операций), и возвращающий значение некоторого типа. ***Выражение*** - правило вычисления значения. В выражении участвуют ***операнды***, объединенные *знаками операций* (**b + 2** - выражение, в котором **+** является знаком операции, **b** и **2** - операндами; пробелы внутри знака операции, состоящей из нескольких символов, не допускаются).

Операндами выражения могут быть константы, переменные, выражения, вызовы функций, объединенные знаками операций (скобками). Выражение, завершенное ";" - *оператор*. Символ ";", не относящийся ни к одному оператору - пустой оператор. При вычислении выражения определяется значение и тип. Эти характеристики задаются значениями и типами операндов, входящих в выражение, и правилами вычисления выражения.

Правила вычисления выражения задают:

- приоритет выполнения операций;
- для операций одного приоритета **порядок применения** - слева направо или справа налево;
- преобразование типов операндов и выбор реализации для перегруженных операций;
- тип и значение результата выполнения операции над заданными значениями операндов определенного типа.

Операции, получающие на вход один операнд (операция приращения **++**, операция **new**) называются ***унарными операциями***. Операции, получающие на вход два операнда (арифметические операции **+, –, *, /**) называются ***бинарными операциями***. Операция языка C#, которая получает на вход три операнда (условная операция **?:**) называется ***тернарной***.

Описание операций языка C#

Основные операции	
Выражение	***Описание***
x.y	доступ к элементам типа;
f(x)	вызов метода и делегата;
a[x]	доступ к массиву и индексатору;
new T(...)	создание объекта, класса или делегата;
new T(...) {...}	создание объекта с инициализацией;
New T[...]	создание массива;

Выражение	Описание
–x	отрицательное значение;
!x	логическое отрицание;
~x	поразрядное отрицание;
x++	постфиксное увеличение;
x--	постфиксное уменьшение;
++x	префиксное приращение;
--x	префиксное уменьшение;
(T)x	явное преобразование x в тип T (кастинг);

Мультипликативные операции		Аддитивные операции	
Выражение	*Описание*	*Выражение*	*Описание*
*	умножение;	x + y	сложение, объединение строк
/	деление;	x – y	вычитание
%	остаток;		

Операции сдвига		Операции равенства	
Выражение	*Описание*	*Выражение*	*Описание*
x << y	сдвиг влево;	x == y	равно;
x >> y	сдвиг вправо;	x != y	не равно;

Операции отношения и типа	
Выражение	*Описание*
x < y	меньше;
x > y	больше;
x <= y	меньше или равно;
x >= y	больше или равно;
x is T	Проверяет совместимость **x** с **T** по типу. Если **x** может быть приведен к типу **T**, не вызывая исключение, то возвращается **true**, в противном случае возвращается значение **false**;
x as T	возвращает **x** типа **T** или нулевое значение, если **x** не относится к типу **T**;

Операции назначения	
Выражение	*Описание*
=	присваивание;
x op = y	составные операции присвоения: +=, -=, *=, /=, %=, &=, |=, !=, <<=, >>= ;

Логические, условные операции и Null-операции		
Категория	*Выражение*	*Описание*
Логическое **AND**	x & y	целочисленное поразрядное AND, логическое AND

Категория	*Выражение*	*Описание*
Логическое исключающее **XOR**	**x ^ y**	целочисленное поразрядное исключающее XOR, логическое исключающее XOR
Логическое **OR**	**x | y**	целочисленное поразрядное OR, логическое OR
Условное **AND**	**x && y**	вычисляет y только если x имеет значение true
Условное **OR**	**x || y**	вычисляет y только если x имеет значение false
Объединение нулей	**x ?? y**	равно y, если x = null, в противном случае равно x
Условное ?:	**x ? y:z**	равно y, если x имеет значение true, z если x имеет значение false

Операции выражения выполняются в определенном порядке в соответствии с *приоритетами* (как и в математике). Если выражение содержит несколько операций - порядок выполнения определяется на основе приоритетов, а если операции одинаковые - порядок выполнения задается их ассоциативностью.

Приоритеты и ассоциативность операций языка C#

Приоритет	Категория	Операции	Ассоциативность
0	Первичные (базовые)	(expr) x.y f(x) a[x] new sizeof(t) typeof(t) checked(expr) unchecked(expr)	Слева направо
1	Унарные	+ - ! ~ x++ x-- ++x --x (T)x	Слева направо
2	Мультипликативные (Умножение)	* / %	Слева направо
3	Аддитивные (Сложение)	+ -	Слева направо
4	Сдвиг	<< >>	Слева направо
5	Отношения, проверка типов	< > <= >= is as	Слева направо
6	Эквивалентность	== !=	Слева направо
7	Логическое	&	Слева направо
8	Логическое исключающее ИЛИ (XOR)	^	Слева направо
9	Логическое ИЛИ (OR)	|	Слева направо

Приоритет	Категория	Операции	Ассоциативность
10	Условное И	&&	Слева направо
11	Условное ИЛИ	||	Слева направо
12	Условное выражение	? :	Справа налево
13	Присваивание	= *= /= %= += -= <<= >>= &= ^= |=	Справа налево

Вычисление *выражений* начинается с операций высшего *приоритета*. Сначала вычисляются *выражения* в круглых скобках, определяются значения полей объекта – x.y, вычисляются функции – f(x), переменные с индексами – a[i]. Если есть несколько операций с одинаковыми приоритетами, то они вычисляются в соответствии с их ассоциативностью. Операции с левой ассоциативностью вычисляются слева направо (x * y / z вычисляется как (x*y)/z). Операции с правой ассоциативностью вычисляются справа налево. Операции присваивания и операция (?:) имеют правую ассоциативность. Все другие двоичные операции имеют левую ассоциативность (если в одном выражении есть несколько операций одного приоритета, операции условная и присваивания выполняются *справа налево*, остальные - *слева направо)*.

Для изменения порядка выполнения операций используются *круглые скобки*, уровень их вложенности практически не ограничен. b1 + b2 + b3 означает (b1 + b2) +b3, а b1 = b2 = b3 означает b1 = (b2 = b3).

Тип результата операции зависит от типов операндов, участвующих в операции. Типом арифметической операции является наиболее сложный тип операнда (значение другого операнда преобразуется к более сложному типу: наименее сложный тип **byte**, наиболее сложный **decimal**).

```
int a=5;  double d=2.6;
a *d; // тип результата double      a / 2 // тип результата int
```

Типом результата операций присваивания является тип левого операнда (переменной, которой присваивается значение).

```
int n;          n = a * d;  // тип результата int
```

Типом результата операций отношения является bool:

```
a > 5 // тип результата bool
```

Типом результата логических операций является bool:

```
bool b = true, c = false;      b && c  // тип результата bool
```

Перегрузка операций - существование нескольких реализаций одной и той же операции. Большинство операций C# являются перегруженными, т.е. одна и та же операция может применяться к операндам различных типов, поэтому перед выполнением операции компилятор ведет поиск описания операции, подходящего для используемых типов операндов. Операции, как правило, выполняются над операндами одного типа. Если же операнды разных типов, то предварительно происходит неявное преобразование типа операнда. Оба операнда могут быть одного типа, но преобразование типов будет происходить – если для типов нет соответствующей перегруженной операции (такая ситуация часто возникает, например, операция сложения не определена для младших типов целого типа).

```
byte b1 = 1, b2 = 2, b3;      short sh1;          int in1;
//b3 = b1 + b2;               //ошибка: результат типа int
b3 = (byte)(b1 + b2);
//sh1 = b1 + b2;              //ошибка: результат типа int
sh1 = (short)(b1 + b2); in1 = b1 + b2 + sh1;   // b3= 3   sh1= 3   in1= 6
```

Базовые операции: (x), F(x), new, sizeof, typeof, checked, unchecked

Операция (x). Разновидность оператора «скобки» для управления порядком вычислений, как в математических операциях, так и при вызове методов.

```
class Test
{
    public int i_p=0;
    public double [] Mas = {1.1, 2.2, 3.3, 4.4};
    public double Metod_1 (byte ind, int a1, double a2)
    {    return  (i_p * a1 + a2) / (a1 - Mas[ind] * a2);     }
}
class Program
{
    static void Main(string[] args)
    {
        double dd;
        Test T = new Test();
        T.i_p = 123;
        dd = T.Metod_1(1, 2,  25.5);  //dd = -5,01848
    }
}
```

Операция X.y Оператор «точка» используется для указания члена класса или структуры. Здесь *x* представляет сущность, содержащую в себе член *y*.

```
T.i_p = 123;
```

Операция F(x) Такая разновидность оператора «скобки» применяется для перечисления аргументов методов.

```
dd = T.Metod_1(1, 2,  25.5);
```

Операция A[x] Квадратные скобки используются для индексации массива (также применяются совместно с индексаторами, когда объекты могут рассматриваться как массив).

```
(i_p * a1 + a2) / (a1 - Mas[ind] * a2);
```

Операция new Этот оператор используется для создания экземпляров объектов на основании определения класса (`Test T = new Test();`), массивов (`double[] Mas = new double [4];`).

Операции checked, unchecked применяются при использовании целых выражений. Первая включает, а вторая отключает проверку переполнения при вычислении выражения. Если переполнение в проверяемом выражении, возникает исключение *System.ArithmeticOverflow*. Если переполнение возникает в неконтролируемом выражении, ничего не происходит.

Операция sizeof - возвращает размер значимых типов, заданный в байтах. Она может применяться только в небезопасных блоках. Поэтому проект, в котором она может использоваться, должен быть скомпилирован с включенным свойством /unsafe, разрешающий создание ненадежного кода (в среде Visual Studio флаг устанавливается в диалоговом окне, открываемым командой Проект\Свойства\Построение\Разрешить небезопасный код=Да).

```
unsafe public static void Method_Size()
 {
  Console.WriteLine("Размер типа Boolean = " + sizeof(bool));// Размер =1
  Console.WriteLine("Размер типа double = " + sizeof(double)); // Размер =8
 Console.WriteLine("Размер типа char ="+sizeof(System.Char)); //Размер =2
        int b1 = 1;
        Type t = b1.GetType();
 Console.WriteLine("Тип переменной b1: {0}", t); // Тип b1:   System.Int32
        Console.WriteLine("Размер b1= {0}", sizeof(int)); ); // Размер b1=4
 }
```

Операция typeof, примененная к своему аргументу, возвращает его тип (в роли аргумента может быть имя класса, как встроенного, так и созданного пользователем). Возвращаемый операцией результат имеет тип ***Type***. К экземпляру класса применять операцию typeof - нельзя. Для экземпляра можно вызвать метод **GetType** и получить такой же результат.

```
namespace Опер
{
   class Test
   {
      public int i_p=0;      public double[] Mas;
      public double Metod_1 (byte ind, int a1, double a2)
      {
         Mas = new double [4];          Mas[0] = 1.1;
         return (i_p * a1 + a2) / (a1 - Mas[ind] * a2);
      }
      unsafe public static void Method_Size()
      {
         Console.WriteLine("Размер типа Boolean = " + sizeof(bool));
         Console.WriteLine("Размер типа double = " + sizeof(double));
         Console.WriteLine("Размер типа char = " + sizeof(System.Char));
         int b1 = 1;        Type t = b1.GetType();
         Console.WriteLine("Тип переменной b1: {0}", t);
      Console.WriteLine("Размер переменной b1: {0}", sizeof(System.Int32));
      }
   }
   class Program
   {
      static void  Main(string[] args)
      {
         Type t1 = typeof(Test);
Console.WriteLine("Тип класса Test: {0}", t1);     // Тип Test:Опер.Test
         Type t2 = typeof(Program);
Console.WriteLine("Тип класса Program:{0}",t2);// Тип Test:Опер.Program
      }
   }
}
```

Операция явного преобразования типа используется для явного преобразования из одного типа в другой. Формат операции:

целевой тип><выражение>

Операции присваивания, инкремент, декремент В C# присваивание операция, используемая в выражениях (простое, множественное, сложное).

Простое присваивание:

Переменная = Значение; (значение: выражение, переменная)

```
int i1=256, i2; i2=i1;//i2=256
```

Значение имеет тип, совпадающий с типом переменной, или допускающий *неявное преобразование*.

В противном случае необходимо использовать явное преобразование:

Переменная = (тип переменной слева от=) **Значение**;

```
int i1=256; byte b2; b2=(byte) i1;//b2 = 0
```

Операции присваивания выполняются справа налево:

a = b = c означает **a = (b=c)**.

При присвоении переменных разного типа выполняется преобразование типов. Компилятор пытается выполнить преобразование типа переменной стоящей справа в тип переменной, стоящей слева.

Присваивание переменной стоящей слева (тип T) значения переменной или результата вычисления выражения (типа T1) возможно когда:

- типы **T** и **T1** совпадают;
- тип **T** является родительским для **T1** (в соответствии с наследованием);
- в определении типа **T1** описано явное или неявное преобразование в тип **T**.

Так как классы в C# (встроенные и определенные пользователем) являются потомками класса Object, то переменным класса Object можно присваивать выражения любого типа.

Множественное присваивание

В выражении, называемом множественным *присваиванием*, списку переменных присваивается одно и то же значение.

Переменная №1 = Переменная №2 = … = Переменная №N = Значение;

```
int a, b, c, d;  a = b = c = d = 11;  // a = b = c = d = 11
```

Сложное присваивание - краткая запись для операций присваивания вида:

X = X <operator> (expression); // **x = x * 2;**

Для таких операций используется краткая форма записи:

X <operator>= expression; //**x*=2;**

выражение: **a+= b** компактная запись выражения **a=a+b**

Операции сложного присваивания языка C#

Оператор	Действие
X = Y	X = Y
X += Y	X = X + Y
X -= Y	X = X - Y
X *= Y	X = X * Y
X /= Y	X = X / Y
X %= Y	X = X % Y
X &= Y	X = X & Y
X |= Y	X = X | Y
X ^= Y	X = X ^ Y
X <<= Y	X = X << Y
X >>= Y	X = X >> Y

Операции инкрементации и декрементации Для *присваиваний* вида "**x=x+1**", в которых переменная увеличивается (уменьшается) на 1, используются операции "**++**" (инкрементация) и "--" (декрементация).

Операция инкремент (++) - применяется для увеличения на 1 значения переменной. **A++** значение переменной **A** увеличивается на **1**, и результат сохраняется в переменной **A** (**A++** соответствует выражению: **A=A+1**).

Операция инкремент имеет две формы записи:

- *префиксная:* **++A**: переменная A увеличивается на 1 и затем используется **новое** значение переменной A.
- *постфиксная*: **A++**: переменная A увеличивается на 1, но в текущем выражении используется **старое** значение переменной A.

Постфиксная форма

Операции	*Значение переменной A*	*Значение переменной B*
A=1	1	
B=A++	2	1

Префиксная форма

Операции	*Значение переменной A*	*Значение переменной B*
A=1	1	
B=++A	2	2

Переменная **A** изменилась на **1** и равна **2**, **B** имеет различные результаты.

Операция декремент (--) - применяется для уменьшения на 1 значения переменной. **A--** значение переменной **A** уменьшается на **1**, и результат сохраняется в переменной **A** (**A--** соответствует выражению: **A=A-1**).

Если операция **инкремент** используется как самостоятельный оператор (как предложение на C#), то между обеими формами нет разницы. **A++;** или **++A;** увеличивают значение переменной **A** на 1. Обычно инкремент используют в операторах цикла для увеличения на **1** переменной цикла.

Арифметические операции

byte B1 = 255, B2 = 255; int I1 = 256, I2 = 123; double D1 = 256.1, D2 = 256.1;

Знак операции	Назначение	Пример записи	Результат
+	**сложение**	2+5 2 + 5б,23 I2 + D1 + B1	7 7.23 634,17
-	**вычитание**	4-1 4-1,23 I2 - D1 - B1	3 2.77 -388,17
*	**умножение**	3*5 -3*5,23 I1 * B1 I1 * B1 * D1	15 -15,69 65280 16722777,6
/	**деление (обычное)**	2.4/2 или 2.4/2.0 D1 / I1 B1 / I1 / D1	1,2 1,2 1,0006640625 0,0039036577272
/	**целочисленное деление**	5 B1 / I1 I1 / B1	2 0 1
%	**вычисление остатка при целочисленном делении** (операция вычисления остатка (%) применима только для **целочисленных** операндов)	5 % 2 -5 % 2 B1 % I1 I1 % B1 -I1 % B1	1 -1 255 1 -1

Приоритет операций "**+**" и "**-**" ниже, чем "*****", "**/**" и вычисления остатка. Для изменения порядка вычисления используют круглые скобки: **2*(A+B)**

В C# знак "**/**" - две разные операции. Если оба операнда - вещественные, то выполняется обычное деление, если оба операнда - целые, то выполняется деление нацело и результат будет *целого* типа. Если запрограммировать вычисление математического выражения **1/3sin(2*x)** *как* **1/3*Math.Sin(2*X)** то результат вне зависимости от значения **X** всегда будет равен нулю, так как выражение **1/3** означает деление нацело. Для решения проблемы достаточно один из операндов сделать вещественным **1.0/3*Math.Sin(2*X)**

Вычисление выражений В программировании: выражение - *комбинация* констант, переменных, вызовов методов, объединенных знаками операций и скобками, которая в результате вычисления, возвращает результат. При вычислении выражения определяется его значение и тип. Тип результата выражения зависит от типов переменных, участвующих в выражении. Типом результата выражения является наиболее сложный тип и констант. Наиболее простым типом является ***byte***, а наиболее сложным - ***decimal***.

```
int a = 5; float f;
f = a / 4; // значение f = 1.0, так как результат a/4 int
f = a / 4f;// значение a = 1.25, так как результат float
```

Отрицание арифметическое Операция арифметическое отрицание **(-)** меняет знак операнда на противоположный (для парности имеется операция унарный плюс **+A**). Операция отрицания определена для типов **int**, **long**, **float**, **double**, **decimal**. К величинам других типов ее можно применять, если для них возможно неявное преобразование к этим типам.

```
int val, ii = 3, jj = -4;
val = -ii;   //val = -3          val = -jj;   //val = 4
```

Отрицание логическое Логическое отрицание (!) - определяет операцию инверсия для логического типа (определена для типа bool). Результат операции - ***false***, если операнд равен ***true***, и ***true***, если операнд равен ***false***.

```
bool val1, val2, a_1 = true, b_1 = false;
  val1 =  !a_1;  //val1 = false
  val2 =  !b_1;  //val2 = true
int i_1 = 3, j_1 = -4; bool val1, val2;
  val1 = !(i_1 > j_1);   //!(3   >  -4) = false
  val2 = !(i_1 <= j_1);  //!(3  <= -4) = true
```

Операции сравнения (операции отношения) применяются для сравнения (сопоставления) числовой или символьной информации. Результат выполнения операций - истина (true) или ложь (false). Приоритет выполнения операций отношения ниже, чем у арифметических операций.

Операции сравнения

Название	Знак операции	Пример записи	Результат
Меньше	<	7 < 9 7 < 3 B1 < I1	True False True

Название	Знак операции	Пример записи	Результат
Меньше или равно	<=	7 < = 9 7 <= 3 B1 <= I1	False True False
Проверка на равенство (два символа «=»)	==	7 == 9 7 == 7 B1 == I1	False True False
Проверка на неравенство	!=	7 != 9 7 != 7 B1 != I1	True False True
Больше	>	7 > 9 7 > 3 B1 > I1	False True False
Больше или равно	>=	7 >= 9 7 >= 3 B1 >= I1	False True False
Меньше или равно	<=	7 < = 9 7 <= 3 B1 <= I1	False True False

A+B>=C (Сумма A и B больше C? Если «да», ответ - true, если «нет» - false)

Тип **bool** можно сравнивать только на равенство (неравенство), т.к. **true**, **false** не упорядочиваются (C#: сравнение **true > false** не имеет смысла).

Логические операции используются для составления логических выражений на основе выражений, которые используют операции сравнения (операции отношения). Такие выражения называются булевыми. Значением булевого выражения может быть **true** или **false** (в логических операторах операнды должны иметь тип **bool**, и результат операции всегда будет иметь тип **bool**).

Название	Знак операции	Пример записи	Пояснение
Логическое отрицание (**НЕ**)	!	!X	Если **X** - **true** , то результат - **false** и наоборот
Логическое умножение (**И**)	&	X & Y	Результат - **true** , если истинны оба операнда
Логическое сложение (**ИЛИ**)	|	X | Y	Результат - **true** , если истинен хотя один операнд
Логическое исключающее сложение (**ИЛИ**)	^	X ^ Y	Результат - **true** , если истинен только один из ее операндов
Условное логическое умножение (**И**)	&&	X && Y	Результат - **true** , если истинны оба операнда (второй операнд вычисляется при необходимости)
Условное логическое сложение (**ИЛИ**)	||	X || Y	Результат - **true** , если истинен хотя один операнд (второй операнд вычисляется при необходимости)

Операнды должны быть арифметического типа. Результат логического типа, равен **true** (**false**). Операции сравнения на равенство (неравенство) имеют меньший приоритет, чем другие операции отношения (результат операций отношения вещественных значений в особых случаях: если один из операндов равен **NaN**, результатом для всех операций отношения, кроме **!=**, будет **false**, для операции **!** - результат равен **true**).

Логические операторы **&**, **|**, **^** и **!** выполняют базовые логические операции **И**, **ИЛИ**, **исключающее ИЛИ** и **НЕ** в соответствии с таблицей истинности:

X	Y	X & Y	X | Y	X ^ Y	!X
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

Предупреждение. Правила работы с **логическими выражениями** в языках C# и C++ имеют принципиальные различия. В языке C++ практически для всех типов существует неявное преобразование в логический тип (правило преобразования – ненулевые значения трактуются как истина, нулевое – как ложь). В языке C# неявных преобразований к логическому типу **нет** (даже для целых арифметических типов). Поэтому корректная в языке C++ запись:

int k1 = 7, x = 3, y = 4; **if (k1) ….;** или **if (k1 = 5) ….;**

недопустима в C#. На этапе трансляции возникнет ошибка (вычисляемое условие имеет тип **int**, а неявное преобразование этого типа к типу bool отсутствует). В языке C# более строгие правила действуют и для логических операций. Запись **if(k1 && (x>y))** - корректная в **C++**, приводит к ошибке на **C#**, т.к. операция **&&** определена только для операндов типа **bool**, а в выражении один из операндов имеет тип **int** (**int k1**). В языке C# в данных ситуациях следует использовать: **if(k1>0);** **if ((k1 > 0) && (x > y))** **{ … };**

Бинарные логические операции «**&&** - условное **И**» и «**||** - условное **ИЛИ**» определены над данными типа **bool**. Операции называются условными или сокращенными, т.к. будет ли вычисляться 2-й операнд, зависит от значения уже вычисленного 1-го операнда. В операции «**&&**», если 1-й операнд равен **false**, то 2-й операнд не вычисляется и результат операции равен **false** (в операции «**||**», если 1-й операнд равен значению **true**, то 2-й операнд не вычисляется и результат операции равен **true**).

Ценность условных логических операций не в их эффективности по времени выполнения. Иногда они позволяют вычислить логическое выражение, имеющее смысл, но в котором второй операнд не определен. Пример: задача поиска по образцу в массиве (ищется элемент с заданным значением). Такой элемент в массиве может быть, а может и не быть. Типичное решение задачи с использованием **условного логического** «&& - условное И»:

```
int[] MAS = { 1, 2, 3 };   //Условное And - &&
int Образец = 7;
int i = 0;
while ((i < MAS.Length) && (MAS[i] != Образец))
    i++;
if (i < MAS.Length)
    Console.WriteLine("Образец найден");
else
    Console.WriteLine("Образец не найден");
```

Если значение переменной **Образец** не совпадает ни с одним из значений элементов массива **MAS**, то последняя проверка условия цикла **while** будет выполняться при значении **i**, равном **MAS.Length**. В этом случае 1-й операнд получит значение **false**, и, хотя 2-й операнд при этом не определен, цикл нормально завершит свою работу. Операция «**&** - нормальная **И**» требует вычисления двух операндов, поэтому ее применение в данной программе привело бы к ошибке выполнения (возникновению исключения) в случае, когда образца нет в массиве. Типичное решение задачи поиска по образцу с использованием **логического** «&-логическое И»:

```
int[] MAS_И = { 1, 2, 3 };   //Логическое And - &
int Образец_И = 7;
int i_И = 0;
Образец_И = MAS_И[MAS_И.Length - 1];
while ((i_И < MAS_И.Length) & (MAS_И[i_И] != Образец_И))
   i_И++;
if (i < MAS_И.Length)
   Console.WriteLine("Образец найден");
else
   Console.WriteLine("Образец не найден");
```

Поразрядные (побитовые) операции действуют непосредственно на разряды своих операндов. Они определены только для целочисленных операндов и не могут быть использованы для операндов типа bool, float или double. Поразрядные операции предназначены для тестирования, установки или сдвига битов (разрядов), из которых состоит целочисленное значение. Поразрядные операции часто используются для решения широкого круга задач программирования системного уровня (при опросе информации о состоянии устройства, ее формировании и т.д.).

При сдвиге влево (<<) старшие биты отбрасываются, а младшие биты заполняются нулями. При сдвиге вправо (>>) в типах uint и ulong младшие биты отбрасываются, а старшие биты заполняются нулями.

Поразрядные (побитовые) операции

Название	Знак операции	Пример записи	Результат
Поразрядное И A & B	&	3&5	1
Поразрядное ИЛИ A | B	|	3|5	7
Поразрядное исключающее ИЛИ A ^ B	^	3^5	6
Инверсия ~ A	~	~3	-4
Поразрядный сдвиг влево A << L Двоичное представление числа A сдвигается влево на L позиций - эквивалентно умножению A на 2 в степени L	<<	5<<1	10
Поразрядный сдвиг вправо A >> L Двоичное представление числа A сдвигается вправо на L позиций - эквивалентно делению A на 2 в степени L	>>	5>>1	2

В операциях битовой арифметики действия происходят над двоичным представлением целых чисел (оба операнда переводятся в двоичную систему и над ними производятся логические поразрядные операции).

A	B	A & B И (AND)	A | B ИЛИ (OR)	A ^ B исключающее ИЛИ (OR)	~ A инверсия	A << L сдвиг влево	A >> L сдвиг вправо
0	0	0	0	0	1		
0	1	0	1	1	1		
1	0	0	1	1	0		
1	1	1	1	0	0		

Использование поразрядных операций требует знания правил кодирования положительных и отрицательных целых чисел (*знаковый* и *беззнаковый* тип). Переменные и константы **беззнакового** типа **byte** принимают значения от **0** (соответствующий двоичный код: **00000000** - все нули) до **255** (соответствующий двоичный код: **11111111** - все единицы). Переменные и константы **знакового** типа **sbyte** принимают значения от **-128** (соответствующий двоичный код: **10000000**) до **+127** (соответствующий двоичный код: **01111111**). Это связано с принятым на аппаратном уровне правилом кодирования знаковых целых чисел - для их внутреннего представления используется **дополнительный код**. Если **K** - количество битовых разрядов, отведенное для представления числа **X** (для типа **sbyte K** равно 8), то **дополнительный код** определяется выражением:

$$\text{Дополнительный код}(X) = \begin{cases} X, \text{ если } X >= 0 \\ 2^K - |X|, \text{ если } X < 0 \end{cases}$$

В битовом представлении чисел с использованием дополнительного кода у всех положительных чисел самый ***левый бит*** - равен **0**, а у отрицательных чисел равен **1** (единице). Минимальное число типа **sbyte** равно **-128** (двоичный код: **10000000**). Представление числа **(-1)** - двоичный код: **11111111**. Представление числа 0 - двоичный код: **00000000**, представление числа **(1)** - двоичный код: **00000001**. Зная правила двоичного кодирования отрицательных целых чисел, легко понять, как меняется значение переменной знакового типа при поразрядных операциях (при применении к положительному числу операции поразрядного отрицания **~**, знак числа меняется и на **1** увеличивается его абсолютное значение; при поразрядном отрицании отрицательного числа результат равен уменьшенному на **1** его абсолютному значению).

Представление знаковых и беззнаковых типов

7 6 5 4 3 2 1 0 — byte b = 9 - знаковый тип (основной код)

0 0 0 0 1 0 0 1 — $0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0 = 8+1=9$

7 6 5 4 3 2 1 0 — sbyte b = 9 - беззнаковый тип (основной код) = 9

0 0 0 0 1 0 0 1 — $0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0 = 8+1=9$

7 6 5 4 3 2 1 0 — sbyte b = - 9 - беззнаковый тип (дополнительный код) $=2^8-9=243$

1 1 1 1 0 1 1 1 — $1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+1*2^0=128+64+32+16+4+2+1=247$

Поразрядное отрицание (инверсия) Операция поразрядного отрицания (~), часто называемая побитовой, инвертирует (***инверсия*** - одноместная операция, заменяющая каждый бит операнда на обратный: **0** на **1**, а **1** на **0**; инверсия - это не смена знака операнда) каждый разряд в двоичном представлении операнда типа **byte**, **sbyte**, **int**, **uint**, **long** или **ulong**.

```
   7 6 5 4 3 2 1 0   byte b = 9 (основной код)
   0 0 0 0 1 0 0 1   0*2^7 +0*2^6 +0*2^5 +0*2^4 +1*2^3 +0*2^2 +0*2^1 +1*2^0 = 9
   7 6 5 4 3 2 1 0   ~b = - 10 (дополнительный код: 246 числа - 10)
~b 1 1 1 1 0 1 1 0   1*2^7 +1*2^6 +1*2^5 +1*2^4 +0*2^3 +1*2^2 +1*2^1+0*2^0 = 246
```

```
byte b = 9, val;
val = ~b;// -10
val = (byte)~b;//246
int i;  val = (int)~b;//-10
```

```
byte b = 9, val;
val = ~b;// -10
val = (byte)~b;//246
int i;  val = (int)~b;//-10
```

```
   7 6 5 4 3 2 1 0   sbyte b = -9 (дополнительный код: 247 числа – 9)
   1 1 1 1 0 1 1 1   1*2^7 +1*2^6 +1*2^5 +1*2^4 +0*2^3 +1*2^2 +1*2^1 +1*2^0 = 247
   7 6 5 4 3 2 1 0   ~b = 8 (основной код)
~b 0 0 0 0 1 0 0 0   0*2^7 +0*2^6 +0*2^5 +0*2^4 +1*2^3 +0*2^2 +0*2^1+0*2^0 = 8
```

Если при выполнении операции поразрядного отрицания для беззнакового типа **X** получается отрицательное значение (левый бит имеет значение 1) - то **код** является ***дополнительным*** и для получения десятичного представления полученного значения его необходимо вычесть из величины 2^K (для беззнакового типа **byte** **K=8,** при значении **byte** **X = 9** десятичное значение **~X** будет равно 2^8 – 246(полученный код) = 256 - 246 = **-10**).

Примечание ***byte*** *bb= (**byte**)3;*
(bb=00000011) ~bb = 11111100 *(**bb = 252**$_{10}$ в десятичной системе)*
Для любого знакового типа bb справедливо: **bb + ~bb = 255**$_{10}$
sbyte *bb= 3; (bb=00000011)* ***sbyte*** *BB= - 3; (BB=11111101 в дополнит. коде - 253)*
Для любого безнакового типа bb справедливо:
для положительного значения bb:
~bb = -|bb| + 1 *(~bb = 11111100 в дополнительном коде 252 =* **-4**$_{10}$*)*
для отрицательного значения BB:
~BB = |BB| - 1 *(~BB = 00000010 в десятичной системе . 2 =* **2**$_{10}$*)*

Поразрядное И (&) Операция выполняется над разрядами исходных операндов (операция напоминает логическую операцию **И**, но выполняется с отдельными битами). Результат операции равен **1, е**сли соответствующие биты в обоих операндах равны **1**, иначе он результат равен **0**.

Поразрядное И (&)

	7 6 5 4 3 2 1 0	
		sbyte b = 9 (основной код)
a	0 0 0 0 1 0 0 1	$0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0=9$
	7 6 5 4 3 2 1 0	sbyte b = 10 (основной код)
b	0 0 0 0 1 0 1 0	$0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+1*2^1+0*2^0=10$
	7 6 5 4 3 2 1 0	a & b = 8 (основной код)
a&b	0 0 0 0 1 0 0 0	$0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+0*2^1+0*2^0=8$

	7 6 5 4 3 2 1 0	
		sbyte b = -9 (дополнительный код: 247 числа 9)
a	1 1 1 1 0 1 1 1	$1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+1*2^0=247$
	7 6 5 4 3 2 1 0	sbyte b = 10 (основной код)
b	0 0 0 0 1 0 1 0	$0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+1*2^1+0*2^0=10$
	7 6 5 4 3 2 1 0	a & b = 2 (основной код)
a&b	0 0 0 0 0 0 1 0	$0*2^7+0*2^6+0*2^5+0*2^4+0*2^3+0*2^2+1*2^1+0*2^0=2$

	7 6 5 4 3 2 1 0	
		sbyte b = -9 (дополнительный код: 247 числа 9)
a	1 1 1 1 0 1 1 1	$1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+1*2^0=247$
	7 6 5 4 3 2 1 0	sbyte b =-10 (дополнительный код: 246числа10)
b	1 1 1 1 0 1 1 0	$1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+0*2^0=246$
	7 6 5 4 3 2 1 0	a & b = -10 (дополнительный код: 246 числа 10)
a&b	1 1 1 1 0 1 1 0	$1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+0*2^0=246$

```
sbyte a = 9, b = 10;
sbyte val;
val = a & b;// 8
val = (int)(a&b);//8
```

```
sbyte a = -9, b = 10;
sbyte val;
val = a & b;// 2
val = (int)(a&b);// 2
```

```
sbyte a = -9, b = -10;
sbyte val;
val = a & b;// 2
val = (int)(a&b);//-10
```

Поразрядное ИЛИ (|). При выполнении поразрядной операции над разрядами выполняется операция **ИЛИ**: результат равен **0**, если оба соответствующихразряда равны **0**, иначе результат равен **1**.

	7 6 5 4 3 2 1 0	
		sbyte b = 9 (основной код)
a	0 0 0 0 1 0 0 1	$0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0=9$
	7 6 5 4 3 2 1 0	sbyte b = 10 (основной код)
b	0 0 0 0 1 0 1 0	$0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+1*2^1+0*2^0=10$
	7 6 5 4 3 2 1 0	a | b = 11 (основной код)
a | b	0 0 0 0 1 0 1 1	$0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+1*2^1+1*2^0=11$

	7 6 5 4 3 2 1 0	
a	1 1 1 1 0 1 1 1	sbyte b = -9 (дополнительный код: 247 числа – 9) $1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+1*2^0=247$
b	0 0 0 0 1 0 1 0	sbyte b = 10 (основной код) $0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+1*2^1+0*2^0=10$
a | b	1 1 1 1 1 1 1 1	a | b = -1 (дополнительный код: 255 числа – 1) $1*2^7+1*2^6+1*2^5+1*2^4+1*2^3+1*2^2+1*2^1+1*2^0=255$

	7 6 5 4 3 2 1 0	
a	1 1 1 1 0 1 1 1	sbyte b = -9 (дополнительный код: 247 числа – 9) $1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+1*2^0=247$
b	1 1 1 1 0 1 1 0	sbyte b =-10 (дополнительный код: 246 числа–10) $1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+0*2^0=246$
a | b	1 1 1 1 0 1 1 1	a | b = -10 (дополнительный код: 247 числа –9) $1*2^7+1*2^6+1*2^5+0*2^4+1*2^3+1*2^2+1*2^1+0*2^0=247$

```
sbyte a = 9, b = 10;
sbyte val;
val = a | b;// 11
val =(int)(a | b);//11
```

```
sbyte a = -9, b = 10;
sbyte val;
val = a | b;// -1
val = (int)(a | b);// -1
```

```
sbyte a = -9, b = -10;
sbyte val;
val = a | b;// -9
val = (int)(a | b);// -9
```

Поразрядное исключающее ИЛИ (^). Операция выполняется по правилам: результат равен **1**, если разряды разные, и равен **0**, если они одинаковые.

	7 6 5 4 3 2 1 0	
a	0 0 0 0 1 0 0 1	sbyte b = 9 (основной код) $0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0=9$
b	0 0 0 0 1 0 1 0	sbyte b = 10 (основной код) $0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+1*2^1+0*2^0=10$
a ^ b	0 0 0 0 0 0 1 1	a ^ b = 3 (основной код) $0*2^7+0*2^6+0*2^5+0*2^4+0*2^3+0*2^2+1*2^1+1*2^0=3$

	7 6 5 4 3 2 1 0	
a	1 1 1 1 0 1 1 1	sbyte b = -9 (дополнительный код: 247 числа – 9) $1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+1*2^0=247$
b	0 0 0 0 1 0 1 0	sbyte b = 10 (основной код) $0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+1*2^1+0*2^0=10$
a ^ b	1 1 1 1 1 1 0 1	a ^ b = -1 (дополнительный код: 253 числа – 3) $1*2^7+1*2^6+1*2^5+1*2^4+1*2^3+1*2^2+0*2^1+1*2^0=253$

	7 6 5 4 3 2 1 0	sbyte b = -9 (дополнительный код: 247 числа – 9)
a	1 1 1 1 0 1 1 1	$1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+1*2^0=247$
	7 6 5 4 3 2 1 0	sbyte b =-10 (дополнительный код: 246 числа–10)
b	1 1 1 1 0 1 1 0	$1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+0*2^0=246$
	7 6 5 4 3 2 1 0	a ^ b = 1 (основной код)
a ^ b	0 0 0 0 0 0 0 1	$0*2^7+0*2^6+0*2^5+0*2^4+0*2^3+0*2^2+0*2^1+1*2^0=1$

```
sbyte a = 9, b = 10;
sbyte val;
val = a  ^ b;//  3
val =(int)(a ^b);// 3
```

```
sbyte a = -9, b = 10;
sbyte val;
val = a ^ b;//  -3
val = (int)(a ^ b);//-3
```

```
sbyte a = -9, b = -10;
sbyte val;
val = a ^ b;//  1
val =(int)(a ^ b);// 1
```

Операция исключающее ИЛИ (^) обладает одним интересным свойством, которое позволяет использовать его для кодирования (раскодирования) информации. Если выполнить операцию ^ между значением X и значением Y, а затем снова выполнить операцию ^ между результатом первой операции и тем же значением Y, получим исходное значение X. Это значит, что после выполнения двух операций

R1 = X ^ Y; R2 = R1 ^ Y; значение R2 совпадет со значением X.

Таким образом, в результате выполнения двух последовательных операций ^, использующих одно и то же значение (Y), получается исходное значение (X). Этот принцип можно использовать для создания простой программы шифрования, в которой некоторое целочисленное значение (ключ) – служит для кодирования и декодирования сообщения, состоящего из символов. Для шифрования сообщения операция исключающего ИЛИ применяется первый раз, а для его дешифровки - второй. Простой способ шифрования:

```
   char ch1 = 'М';      char ch2 = 'ы';      char ch3 = '!';
   int key = 88;  // Кодирование (шифрование
Console.WriteLine("Исходное сообщение: " + ch1 + ch2 + ch3);
   ch1 = (char)(ch1 ^ key);)
   ch2 = (char)(ch2 ^ key);
   ch3 = (char)(ch3 ^ key);
Console.WriteLine("Зашифрованное сообщение: " + ch1 + ch2 + ch3);
   ch1 = (char)(ch1 ^ key);// Раскодирование (дешифрование)
   ch2 = (char)(ch2 ^ key);
   ch3 = (char)(ch3 ^ key);
Console.WriteLine("Дешифрованное сообщение: " + ch1 + ch2 + ch3);
```

```
Исходное сообщение: Мы!
Зашифрованное сообщение: фГу
Дешифрованное сообщение: Мы!
```

Операции сдвига Операции сдвига вправо **">>"** и сдвига влево **"<<"** в обычных вычислениях применяются редко. Они полезны, если данные рассматриваются как строка битов. Результатом операции является сдвиг строки битов влево или вправо на **K** разрядов. В применении к обычным целым положительным числам сдвиг вправо равносилен делению нацело на 2^K, а сдвиг влево - умножению на 2^K. Для отрицательных чисел сдвиг влево и деление дают разные результаты, отличающиеся на единицу. В C# операции сдвига определены только для целочисленных типов - **int**, **uint**, **long**, **ulong**. Величина сдвига должна иметь тип **int**. Для знаковых типов используется - **арифметический** сдвиг, для беззнаковых типов - **логический**.

Поразрядный сдвиг влево **(A<<K)** выполняется для разрядов левого операнда на число позиций, равное правому операнду (сдвиг A на K разрядов влево, освободившиеся младшие разряды заполняются - 0). Сдвиг влево на **K** позиций аналогичен умножению числа **A** на 2^K.

	7 6 5 4 3 2 1 0	sbyte a = 9 (основной код)
a =	0 0 0 0 1 0 0 1	$0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0=9$
a<<3	7 6 5 4 3 2 1 0	a << 3 = 72 (основной код)
0 0 0	0 1 0 0 1 0 0 0	$0*2^7+1*2^6+0*2^5+0*2^4+1*2^3+0*2^2+0*2^1+0*2^0=72$

	7 6 5 4 3 2 1 0	sbyte b= -9 (дополнительный код:247 числа–9)
a =	1 1 1 1 0 1 1 1	$1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+1*2^0=247$
a<<3	7 6 5 4 3 2 1 0	a<<3=-72(дополнительный код:184 числа–72)
1 1 1	1 0 1 1 1 0 0 0	$1*2^7+0*2^6+1*2^5+1*2^4+1*2^3+0*2^2+0*2^1+0*2^0=184$

```
sbyte a = 9;
sbyte val;
val = a  <<3; // 72
val =(int)(a <<3); // 72
```

```
sbyte a = -9;
sbyte val;
val = a  <<3; // -72
val =(int)(a <<3); //-72
```

Поразрядный сдвиг вправо **(A>>K)** выполняется для разрядов левого операнда на число позиций, равное правому операнду. По результату работы сдвиг право на **K** позиций аналогичен целочисленному делению исходного числа **A** на 2^K. При сдвиге вправо (типы int и long) старшие биты заполняются 0 (для неотрицательных значений) или 0 (для положительных значений).

7 6 5 4 3 2 1 0		sbyte a = 9 (основной код)
0 0 0 0 1 0 0 1		a = $0*2^7+0*2^6+0*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0=9$
7 6 5 4 3 2 1 0		a>>3 = 1 (основной код)
0 0 0 0 0 0 0 1	0 0 1	$0*2^7+0*2^6+0*2^5+0*2^4+0*2^3+0*2^2+0*2^1+1*2^0=1$

7 6 5 4 3 2 1 0	sbyte a= -9 (дополнительный код:247 числа –9)
1 1 1 1 0 1 1 1	a = $1*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+1*2^0=247$
7 6 5 4 3 2 1 0	a>>3 = -2 (дополнительный код:254 числа–2)
1 1 1 1 1 1 1 0	1 1 1 $1*2^7+1*2^6+1*2^5+1*2^4+1*2^3+1*2^2+1*2^1+0*2^0=-2$

```
sbyte a = 9;
sbyte val;
val = a  << 3; // 72
val =(int)(a <<3); // 72
```

```
sbyte a = -9;
sbyte val;
val = a  << 3; // -72
val =(int)(a <<3); //-72
```

Операции сдвига никогда не приводят к переполнению и потере значимости. Поскольку разряды представления двоичных чисел представляют собой степени числа 2, то операторы сдвига можно использовать в качестве быстрого способа умножения или деления чисел на 2. При сдвиге влево число удваивается, при сдвиге вправо число делится пополам (это будет справедливо до тех пор, пока с одного или другого конца не выдвинутся (и потеряются) значимые биты). Использование операторов сдвига для умножения и деления на 2:

```
int N;
N = 10;
Console.WriteLine("Значение переменной N: " + N);
N = N << 1;// Умножение на 2
Console.WriteLine("Значение переменной N после N = N * 2: " + N);
N = N << 2;// Умножение на 4
Console.WriteLine("Значение переменной N после N = N * 4: " + N);
N = N >> 1;// Деление на 2
Console.WriteLine("Значение переменной N после N = N / 2: " + N);
N = N >> 2;// Деление на 4
Console.WriteLine("Значение переменной N после N = N / 4: " + N);
Console.WriteLine();
N = 10;// Устанавка N в исходное состояние.
Console.WriteLine("Значение переменной N: " + N);
N = N << 30; // Умножение на 2 30 раз - данные потеряны
Console.WriteLine("Значение N после сдвига влево на 30 разрядов: " + N);
```

```
Значение переменной N: 10
Значение переменной N после N = N * 2: 20
Значение переменной N после N = N * 4: 80
Значение переменной N после N = N / 2: 40
Значение переменной N после N = N / 4: 10
Значение переменной N: 10
Значение N после сдвига влево  на 30 разрядов: -2147483648
```

Условная (тернарная) операции ? В C#, как и в C++ определена условная операция **?**. Операция **?** называется *тернарной*, поскольку она работает с тремя операндами. Операция **?** часто используется для замены условных операторов (конструкций ветвления типа **if - then - else**).

Синтаксис объявления условной операции:

(<*Операнд_1*>)?<*Операнд_2*> : <*Операнд_3*>;

где ***Операнд_1*** имеет тип **bool** (или может быть неявно преобразован к типу **bool**). Типы элементов ***Операнд_2*** и ***Операнд_3*** могут отличаться. Если результат вычисления выражения ***Операнда_1*** равен **true**, то результатом условной операции будет значение ***Операнда_2***, в противном случае (если результат вычисления ***Операнда_1*** равен **false**) результатом условной операции будет значение ***Операнда_3***.

Тип результата операции **?** зависит от типов второго и третьего операндов:
- если операнды одного типа, он и становится типом результата операции;
- если существует неявное преобразование типа от операнда_2 к операнду_3 то типом результата операции становится тип операнда_3;
- если существует неявное преобразование типа от операнда_ 3 к операнду_ 2 то типом результата операции становится тип операнда_2;

в противном случае возникает ошибка компиляции.

```
int X=5; int Y=10;
int max = (X > Y) ? X : Y;
Console.WriteLine(" max = "+ max); // max = 10
```

Получение информации о классе Зная тип (класс), получить подробную информацию обо всех методах и полях класса (такая информация - полезна, если класс поставлен сторонними разработчиками). Вся необходимая информация содержится в метаданных, поставляемых вместе с классом. Процесс получения метаданных называется отражением (**reflection**).

Получение информации о всех методах и полях класса

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
namespace Информация_Класс
{
   class Class_Test
   {
      int i_pr = 0;
      public int i_p = 0;
```

```
        public double[] Mas;
        public double Metod_1(byte ind, int a1, double a2)
        {
            Mas = new double[4];
            Mas[0] = 1.1;
            return (i_p * a1 + a2) / (a1 - Mas[ind] * a2);
        }
         unsafe public static void Method_Size()
        {
            Console.WriteLine("Размер типа Boolean = " + sizeof(bool));
            Console.WriteLine("Размер типа double = " + sizeof(double));
            Console.WriteLine("Размер типа char = " + sizeof(System.Char));
            int b1 = 1;            Type  t = b1.GetType();
            Console.WriteLine("Тип переменной b1: {0}", t);
            Console.WriteLine("Размер переменной b1: {0}", sizeof(System.Int32));
        }
        public void Info_Class(string name, object any)
        {
            Type  t = any.GetType();
      Console.WriteLine("n{0}:Тип{1},значение{2}",name,any.GetType(),  any.ToString());
            Console.WriteLine("Методы класса:");
            MethodInfo[]  ClassMethods = t.GetMethods();
            foreach (MethodInfo  curMethod in ClassMethods)
            {
               Console.WriteLine(curMethod);
            }
            Console.WriteLine("Все члены класса:");
            MemberInfo[] ClassMembers = t.GetMembers();
            foreach (MemberInfo curMember in ClassMembers)
            {          Console.WriteLine(curMember.ToString());         }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Class_Test t = new Class_Test();
            t.Info_Class("Целый", 25);
            t.Info_Class("Объект класса t", t);
         }
    }
}
```

Результат выполнения:

```
Целый: Тип- System.Int32 , значение- 25
Методы класса:
Int32 CompareTo(System.Object)
Int32 CompareTo(Int32)
Boolean Equals(System.Object)
Boolean Equals(Int32)
Int32 GetHashCode()
System.String ToString()
System.String ToString(System.String)
System.String ToString(System.IFormatProvider)
System.String ToString(System.String, System.IFormatProvider)
Int32 Parse(System.String)
Int32 Parse(System.String, System.Globalization.NumberStyles)
Int32 Parse(System.String, System.IFormatProvider)
Int32 Parse(System.String, System.Globalization.NumberStyles,
System.IFormatProvider)
Boolean TryParse(System.String, Int32 ByRef)
Boolean TryParse(System.String, System.Globalization.NumberStyles,
System.IFormatProvider, Int32 ByRef)
System.TypeCode GetTypeCode()
System.Type GetType()
Все члены класса:
Int32 CompareTo(System.Object)
Int32 CompareTo(Int32)
Boolean Equals(System.Object)
Boolean Equals(Int32)
Int32 GetHashCode()
System.String ToString()
System.String ToString(System.String)
System.String ToString(System.IFormatProvider)
System.String ToString(System.String, System.IFormatProvider)
Int32 Parse(System.String)
Int32 Parse(System.String, System.Globalization.NumberStyles)
Int32 Parse(System.String, System.IFormatProvider)
Int32 Parse(System.String, System.Globalization.NumberStyles,
System.IFormatProvider)
Boolean TryParse(System.String, Int32 ByRef)
```

Boolean TryParse(System.String, System.Globalization.NumberStyles, System.IFormatProvider, Int32 ByRef)
System.TypeCode GetTypeCode()
System.Type GetType()
Int32 MaxValue
Int32 MinValue

Объект класса t: Тип- Информация_Класс.Class_Test , значение- Информация_Класс.Class_Test
Методы класса:
Double Metod_1(Byte, Int32, Double)
Void Method_Size()
Void Info_Class(System.String, System.Object)
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
System.Type GetType()
Все члены класса:
Double Metod_1(Byte, Int32, Double)
Void Method_Size()
Void Info_Class(System.String, System.Object)
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
System.Type GetType()
Void .ctor()//**Конструктор класса по умолчанию .ctor**, создается компилятором втоматически
Int32 i_p
Double[] Mas

В методе Info_Class создается переменная *t* типа *Type*. Значением этой переменной будет тип аргумента, переданного в метод в качестве значения параметра *any* (*any* имеет базовый тип *object* и потому метод может быть вызван с аргументом, роль которого может играть выражение любого типа). Затем вызываются методы переменной *t* - ***GetMethods()*** и ***GetMembers()***. Эти методы возвращают в качестве значений массивы элементов классов ***MethodInfo*** и ***MemberInfo***. Эти классы содержатся в пространстве имен Reflection и хранят информацию о методах класса (*MethodInfo*) и о полях и методах класса, заданного переменной t (***MemberInfo***).

В методе Main дважды вызывается метод Info_Class:

t.Info_Class("Целый", 25); - аргументом является выражение типа int,

t.Info_Class("Объект класса t", t); - аргументом является объект t, вызывающий метод.

Метод *Info_Class* выдает подробную информацию обо всех элементах этих классов. Для класса Test отображается информация о полях и методах как собственных, так и наследуемых от общего родителя - класса Object (отображается информация только об открытых полях и методах класса, а поскольку поле *i_pr* класса закрыто, то и информации о нем нет). Для класса Int отображается информация о всех полях и методах (в том числе сведения и о статических полях и методах класса).