Лекция 3
Переменные, операции и выражения

1. Переменные
2. Операции и выражения

1. Переменные

Переменная – это именованная область памяти, предназначенная для хранения данных определенного типа. Во время выполнения программы значение переменной можно изменять. Все переменные, используемые в программе, должны быть описаны явным образом. При описании для каждой переменной задаются ее *имя* и *тип*.

Пример описания целой переменной с именем **a** и вещественной переменной **x**:

```
int a; float x;
```

Имя переменной служит для обращения к области памяти, в которой хранится значение переменной. Имя дает программист. Оно должно:

- соответствовать правилам именования идентификаторов C#,
- отражать смысл хранимой величины,
- быть легко распознаваемым.

Пример. Если в программе вычисляется количество каких-либо предметов, лучше назвать соответствующую переменную **quantity** или, на худой конец, **kolich**, но не, скажем, **A**, **tl7_xz** или **prikol**.

СОВЕТ – Желательно, чтобы имя не содержало символов, которые можно перепутать друг с другом, например

l (строчная буква L) и I(прописная буква i)

*Тип переменной выбирается, исходя из **диапазона** и требуемой **точности представления** данных*.

Пример. Нет необходимости заводить переменную вещественного типа для хранения величины, которая может принимать только целые значения, – хотя бы потому, что целочисленные операции выполняются гораздо быстрее.

При объявлении можно присвоить переменной некоторое начальное значение, то есть ***инициализировать*** ее, например:

```
int a, b = 1;  // a типа int не инициализируется
float x = 0.1, y = 0.1f;
```

Здесь описаны:

- переменная **a** типа **int**, начальное значение которой не присваивается;
- переменная **b** типа **int**, ее начальное значение равно **1**;
- переменные **x** и **y** типа **f1oat**, которым присвоены одинаковые начальные значения **0.1**. Разница между ними состоит в том, что:

- для инициализации переменной **x** сначала формируется константа типа **double** (это тип, присваиваемый *по умолчанию* литералам с дробной частью), а затем она преобразуется к типу **float**;

- переменной **y** значение **0.1** присваивается без промежуточного преобразования.

При инициализации можно использовать не только константу, но и выражение – главное, чтобы на момент описания оно было вычисляемым, например:

```
int b = 1, a = 100;
int x = b * a + 25;
```

Инициализировать переменную прямо при объявлении не обязательно, но перед тем, как ее использовать в вычислениях, это сделать все равно придется, иначе компилятор сообщит об ошибке.

Примечание – Рекомендуется всегда инициализировать переменные при описании.

Впрочем, иногда эту работу делает за программиста компилятор, это зависит от местонахождения описания переменной.

Как вы помните, программа на C# состоит из *классов*, внутри которых описывают *методы* и *данные*. Переменные, описанные непосредственно внутри класса, называются *полями класса*.

Им автоматически присваивается так называемое «*значение по умолчанию*» – как правило, это 0 соответствующего типа.

Переменные, описанные внутри метода класса, называются ***локальными*** *переменными*. Их инициализация возлагается на программиста.

ПРИМЕЧАНИЕ – Сейчас рассматриваются только локальные переменные простых встроенных типов данных.

Область действия переменной – область программы, где можно использовать переменную:

- начинается в точке ее описания и
- длится до конца блока, внутри которого она описана.

Блок – это код, заключенный в фигурные скобки.

Основное *назначени*е блока – группировка операторов.

В C# любая переменная описана внутри какого-либо блока:

- класса,
- метода,
- блока внутри метода.

Имя переменной должно быть уникальным в области ее действия. Область действия распространяется на вложенные в метод блоки, из этого следует, что:

во вложенном блоке нельзя объявить переменную с таким же именем, что и в охватывающем его.

```
class X   // начало описания класса X
{
int A;// поле A класса X
int B;// поле B класса X

void Y()  // метод Y класса X
{
   int C://локальная переменная C, область действия - метод Y
   int A;// локальная переменная A (НЕ конфликтует с полем A)
   {// ============ вложенный блок 1 ============
int D;// локальная переменная D, область действия - этот блок
//int A; недопустимо! Ошибка компиляции, конфликт с локальной
//       переменной A
     C = B; // присваивание переменной C поля B класса X (**)
C = this.A;// присваивание переменной C поля A класса X (***)
}// ============ конец блока 1 ==============
```

Разберемся в примере. В нем описан класс X, содержащий три элемента: поле A, поле B и метод Y. Непосредственно внутри метода Y заданы две локальные переменные – C и A.

Внутри метода класса можно описывать переменную с именем, совпадающим с полем класса, потому что существует способ доступа к полю класса с помощью ключевого слова `this` (это иллюстрирует строка, отмеченная символами `***`).

Таким образом, локальная переменная A не «закрывает» поле A класса X, а вот попытка описать во вложенном блоке другую локальную переменную с тем же именем окончится неудачей (эти строки закомментированы).

Если внутри метода нет локальных переменных, совпадающих с полями класса, к этим полям можно обратиться в методе непосредственно (см. строку, помеченную символами `**)`.

Две переменные с именем D не конфликтуют между собой, поскольку блоки, в которых они описаны, не вложены один в другой.

СОВЕТ – Как правило, переменным с большой областью действия даются более длинные имена, а для переменных, вся «жизнь» которых – несколько строк исходного текста, хватит и одной буквы с комментарием при объявлении.

В листинге 1 приведен пример программы, в которой описываются и выводятся на экран локальные переменные.

Листинг 1 – Описание переменных

```
using System;
namespace ConsoleApplication1
{ class Class1
  { static void Main()
    {
       int i = 3;
       double y = 4.12;
       decimal d = 600m;
       string s = "Вася";
       Console.Write("i = "); Console.WriteLine(i);
       Console.Write("y = "); Console.WriteLine(y);
       Console.Write("d = "); Console.WriteLine(d);
       Console.Write("s = "); Console.WriteLine(s);
    }
  }
}
```

Метод **Write** делает то же самое, что и **WriteLine**, но не переводит строку.

ВНИМАНИЕ – Переменные создаются при входе в их область действия (блок) и уничтожаются при выходе. Это означает, что после выхода из блока значение переменной не сохраняется. При повторном входе в этот же блок переменная создается заново.

Именованные константы. Можно запретить изменять значение переменной, задав при ее описании ключевое слово **`const`**, например:

```
const int b = 1;
const float x = 0.1, y = 0.1f; // const распространяется на обе переменные
```

Такие величины называют *именованными константами*, или просто *константами*. Они применяются, чтобы вместо значений констант можно было использовать в программе их имена. Это делает программу более понятной и облегчает внесение в нее изменений.

ПРИМЕЧАНИЕ – Улучшение читабельности происходит только при осмысленном выборе имен констант. В хорошо написанной программе вообще не должно встречаться иных чисел, кроме 0 и 1, все остальные числа должны задаваться именованными константами с именами, отражающими их назначение.

Именованные константы должны обязательно инициализироваться при описании. При инициализации можно использовать не только константу, но и выражение – главное, чтобы оно было вычисляемым на этапе компиляции, например:

```
const int b = 1, a = 100:
const int x = b * a + 25;
```

2. Операции и выражения

Выражение – это правило вычисления значения.

В выражении участвуют *операнды*, объединенные знаками операций. Операндами простейшего выражения могут быть константы, переменные и вызовы функций.

Например, **a + 2** – это выражение, в котором:

+ - знак операции, а **a** и **2** – операнды. Пробелы внутри знака операции, состоящей из нескольких символов, не допускаются. По количеству участвующих в одной операции операндов операции делятся на *унарные*, *бинарные* и *тернарную*.

Таблица 1 – Операции C#

Категория	Знак операции	Название
Первичные	**.**	Доступ к элементу
	x()	Вызов метода или делегата
	x[ ]	Доступ к элементу
	x++	Постфиксный инкремент
	x--	Постфиксный декремент
	new	Выделение памяти
	typeof	Получение типа
	checked	Проверяемый код
	unchecked	Непроверяемый код

Категория	Знак операции	Название
Унарные	+	Унарный плюс
	-	Унарный минус (арифметическое отрицание)
	!	Логическое отрицание
	~	Поразрядное отрицание
	++x	Префиксный инкремент
	--x	Префиксный декремент
	(тип)x	Преобразование типа
Мультипликативные (типа умножения)	*	Умножение
	/	Деление
	%	Остаток от деления
Аддитивные (типа сложения)	+	Сложение
	-	Вычитание
Сдвига	<<	Сдвиг влево
	>>	Сдвиг вправо

Категория	Знак операции	Название
Отношения и проверки типа	`<`	Меньше
	`>`	Больше
	`<=`	Меньше или равно
	`>=`	Больше или равно
	`is`	Проверка принадлежности типу
	`as`	Приведение типа
Проверки на равенство	`==`	Равно
	`! =`	Не равно
Поразрядные логические	`&`	Поразрядная конъюнкция (И)
	`^`	Поразрядное исключающее ИЛИ
	`|`	Поразрядная дизъюнкция (ИЛИ)
Условные логические	`&&`	Логическое И
	`||`	Логическое ИЛИ
Условная	`? :`	Условная операция

Категория	Знак операции	Название
Присваивания	=	Присваивание
	*=	Умножение с присваиванием
	/ =	Деление с присваиванием
	%=	Остаток отделения 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

Примечание – В таблице символ **x** призван показать расположение операнда и не является частью знака операции.

Операции в выражении выполняются в определенном порядке в соответствии с приоритетами, как и в математике.

В табл. 1 операции расположены ***по убыванию приоритетов***, уровни приоритетов разделены в таблице горизонтальными линиями.

Результат вычисления выражения характеризуется значением и типом.

Пример. Пусть **a** и **b** – переменные целого типа и описаны так:

```
int a = 2, b = 5;
```

Тогда

- выражение **a + b** имеет значение **7** и тип **int**,

- выражение **a = b** имеет значение, равное помещенному в переменную **a** (в данном случае – **5**), и тип, совпадающий с типом этой переменной.

Если в одном выражении соседствуют несколько операций одинакового приоритета, *операции присваивания и условная операция* выполняются *справа налево*, остальные – *слева направо*. Для изменения порядка выполнения операций используются круглые скобки, уровень их вложенности практически не ограничен. Например,

a + b + c означает **(a + b) + c,**

a = b = c означает **a = (b = c).**

То есть сначала вычисляется выражение **b = c**, а затем его результат становится правым операндом для операции присваивания переменной **a**.

ПРИМЕЧАНИЕ – Часто перед выполнением операции требуется вычислить значения операндов. Например, в выражении F(i) + G(i++) * H(i) сначала вызываются функции F, G и H, а затем выполняются умножение и сложение. Операнды всегда вычисляются слева направо независимо от приоритетов операций, в которых они участвуют. Кстати, в приведенном примере метод H вызывается с новым значением i (увеличенным на 1).

Тип результата выражения в общем случае формируется по правилам, которые описаны далее.

Преобразования встроенных арифметических типов-значений

При вычислении выражений может возникнуть необходимость в преобразовании типов. Если операнды, входящие в выражение, одного типа и операция для этого типа определена, то результат выражения будет иметь тот же тип.

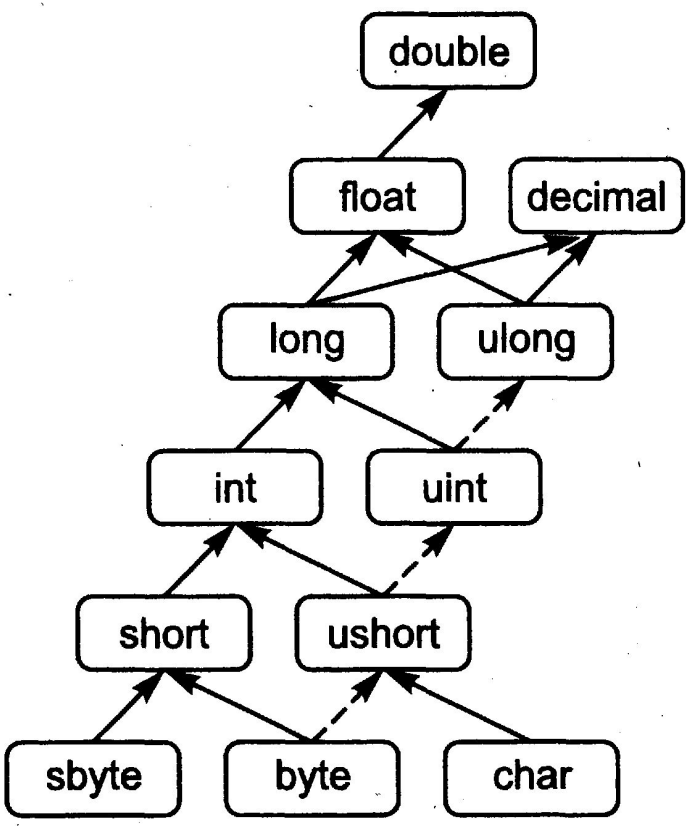
Если операнды разного типа и/или операция для этого типа не определена, перед вычислениями автоматически выполняется преобразование типа по правилам, обеспечивающим приведение более коротких типов к более длинным для сохранения значимости и точности. Автоматическое (*неявное*) преобразование возможно не всегда, а только если при этом не может случиться потеря значимости.

Если неявного преобразования из одного типа в другой не существует, программист может задать явное преобразование типа с помощью операции **(тип)x**. Его результат остается на совести программиста.

Явное преобразование рассмотрим немного позже.

ВНИМАНИЕ – Арифметические операции не определены для более коротких, чем **int**, типов. Это означает, что если в выражении участвуют только величины типов **sbyte**, **byte**, **short** и **ushort**, перед выполнением операции они будут преобразованы в **int**. Таким образом, результат любой арифметической операции имеет тип не менее **int**.

Правила неявного преобразования иллюстрирует рисунок.



Если один из операндов имеет тип, изображенный на более низком уровне, чем другой, то он приводится к типу второго операнда при наличии пути между ними.

Если пути нет, возникает ошибка компиляции.

Если путей несколько, выбирается наиболее короткий, не содержащий пунктирных линий.

Преобразование выполняется не последовательно, а непосредственно из исходного типа в результирующий.

Преобразование более коротких, чем **int**, типов выполняется при присваивании.

Неявного преобразования из **float** и **double** в **decimal** не существует.

Введение в исключения

При вычислении выражений могут возникнуть ошибки, например, переполнение, исчезновение порядка или деление на ноль. В C# есть механизм, который позволяет обрабатывать подобные ошибки и таким образом избегать аварийного завершения программы. Он так и называется: ***механизм обработки исключительных ситуаций (исключений)***.

Если в процессе вычислений возникла ошибка, система сигнализирует об этом с помощью специального действия, называемого ***выбрасыванием*** (генерированием) ***исключения***. Каждому типу ошибки соответствует свое исключение. Поскольку C# – язык объектно-ориентированный, *исключения являются классами, которые имеют общего* ***предка*** *– класс* ***`Exception`****, определенный в пространстве имен* ***`System`***.

Примеры:

- при делении на ноль будет выброшено (сгенерировано) исключение с длинным, но понятным именем **`DivideByZeroException`**,
- при недостатке памяти – исключение **`OutOfMemoryException`**,
- при переполнении – исключение **`OverflowException`**.

ПРИМЕЧАНИЕ – Стандартных исключений очень много, тем не менее, программист может создавать и собственные исключения на основе класса **`Exception`**.

Программист может задать способ обработки исключения в специальном блоке кода, начинающемся с ключевого слова **`catch`** («перехватить»), который будет автоматически выполнен при возникновении соответствующей исключительной ситуации.

Внутри блока можно, например, вывести предупреждающее сообщение или скорректировать значения величин и продолжить выполнение программы. Если этот блок не задан, система выполнит ***действия по умолчанию***, которые обычно заключаются в выводе диагностического сообщения и нормальном завершении программы.

Процессом выбрасывания исключений, возникающих при переполнении, можно управлять.

Для этого служат ключевые слова **`checked`** и **`unchecked`**.

Слово **`checked`** включает проверку переполнения,
слово **`unchecked`** выключает.

При выключенной проверке исключения, связанные с переполнением, не генерируются, а результат операции усекается.

Проверку переполнения можно реализовать для отдельного выражения или для целого блока операторов, например:

```
a = checked (b + c);   // для выражения
unchecked {  // для блока операторов
a = b + c;
}
```

Проверка не распространяется на функции, вызванные в блоке.

Если проверка переполнения включена, говорят, что вычисления выполняются *в проверяемом контексте*,

если выключена – в *непроверяемом*. Проверку переполнения выключают в случаях, когда усечение результата операции необходимо в соответствии с алгоритмом.

Можно задать проверку переполнения во всей программе с помощью ключа компилятора **/checked**, это полезно при отладке программы. Поскольку подобная проверка несколько замедляет работу, в готовой программе этот режим обычно не используется.

3. Основные операции C#

В этом разделе кратко описаны синтаксис и применение всех операций C#, кроме некоторых первичных, которые рассматриваются далее.

Инкремент и декремент. Операции инкремента (**++**) и декремента (**--**), называемые также операциями увеличения и уменьшения на единицу, имеют две формы записи – ***префиксную***, когда знак операции записывается перед операндом, и ***постфиксную***:

- в префиксной форме сначала изменяется операнд, а затем его значение становится результирующим значением выражения,
- в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется.

Листинг 2 иллюстрирует эти операции.

Стандартные операции инкремента существуют для *целых*, *символьных*, *вещественных* и *финансовых* величин, а также для *перечислений*.

Операндом может быть переменная, свойство или индексатор (свойства и индексаторы рассмотрим позже).

Листинг 2 – Операции инкремента и декремента

```
using System;
namespace ConsoleApplication1
{ class Class1
   { static void Main()
      {
         int x = 3, y = 3;
         Console.Write("Значение префиксного выражения:
");
         Console.WriteLine(++x);
         Console.Write("Значение x после приращения: ");
         Console.WriteLine(x);

         Console.Write("Значение постфиксного выражения:
");
         Console.WriteLine(y++);
         Console.Write("Значение y после приращения: ");
         Console.WriteLine(y);
      } //Результат работы программы:
   }   //Значение префиксного выражения: 4
}      //Значение x после приращения: 4
         Значение постфиксного выражения: 3
```

Операция new. Операция **new** служит для создания нового объекта. Формат операции:

```
new тип ([аргументы])
```

С помощью этой операции можно создавать объекты как ссылочных, так и значимых типов, например:

```
object z = new object();
int i = new int();  // то же самое, что int i = 0;
```

Объекты ссылочного типа обычно формируют именно этим способом, а переменные значимого типа чаще создаются так, как описано ранее в разделе «Переменные».

При выполнении операции **new** сначала выделяется необходимый объем памяти (для ссылочных типов в хипе, для значимых – в стеке), а затем вызывается так называемый ***конструктор по умолчанию***, то есть метод, с помощью которого инициализируется объект.

Переменной значимого типа присваивается значение по умолчанию, которое равно нулю соответствующего типа.

Для ссылочных типов стандартный конструктор инициализирует значениями по умолчанию все поля объекта.

Если необходимый для хранения объекта объем памяти выделить не удалось, генерируется исключение **OutOfMemoryException**.

Операции отрицания. Арифметическое отрицание (унарный минус -) меняет знак операнда на противоположный. Стандартная операция отрицания определена для типов **`int`**, **`long`**, **`float`**, **`double`** и **`decimal`**. К величинам других типов ее можно применять, если для них возможно неявное преобразование к этим типам. Тип результата соответствует типу операции.

ПРИМЕЧАНИЕ – Для значений целого и финансового типов результат достигается вычитанием исходного значения из нуля. При этом может возникнуть переполнение. Будет ли при этом выброшено исключение, зависит от контекста.

Логическое отрицание (**`!`**) определено для типа **`bool`**. Результат операции – значение **`false`**, если операнд равен **`true`**, и значение **`true`**, если операнд равен **`false`**.

Поразрядное отрицание (**`~`**), часто называемое побитовым, инвертирует каждый разряд в двоичном представлении операнда типа **`int`**, **`uint`**, **`long`** или **`ulong`**.

Операции отрицания представлены в листинге 3.

Листинг 3 – Операции отрицания

```
using System;
namespace ConsoleApplicationl
{ class Classl
   { static void Main()
     {
        sbyte a = 3, b = -63, c = 126;
        bool d = true;
        Console.WriteLine(-a); // Результат -3
        Console.WriteLine(-c); // Результат -126
        Console.WriteLine(!d); // Результат false
        Console.Writeline(~a); // Результат -4
        Console.WriteLine(~b); // Результат 62
        Console.Wri tellne(~c); // Результат -127
      }
    }
}
```

Явное преобразование типа. Операция используется, как и следует из ее названия, для явного преобразования величины из одного типа в другой.

Это требуется в том случае, когда-не-явного преобразования не существует. При преобразовании из более длинного типа в более короткий возможна потеря информации, если исходное значение выходит за пределы диапазона результирующего типа.

Формат операции:

```
(тип) выражение
```

Здесь **тип** – это имя того типа, в который осуществляется преобразование, а **выражение** чаще всего представляет собой имя переменной, например:

```
long b = 300;
int  a = (int) b;  // данные не теряются
byte d = (byte) a; // данные теряются
```

Преобразование типа часто применяется для ссылочных типов при работе с иерархиями объектов.

Примечание – Эта потеря никак не диагностируется, то есть остается на совести программиста.

Умножение, деление и остаток от деления. ***Операция умножения*** **(*)** возвращает результат перемножения двух операндов.

Стандартная операция умножения определена для типов **`int`**, **`uint`**, **`long`**, **`ulong`**, **`float`**, **`double`** и **`decimal`**. К величинам других типов ее можно применять, если для них возможно неявное преобразование к этим типам. Тип результата операции равен «наибольшему» из типов операндов, но не менее **`int`**.

Если оба операнда целочисленные или типа **`decimal`** и результат операции слишком велик для представления с помощью заданного типа, генерируется исключение **`System.OverflowException`**.

Примечание – В проверяемом контексте. В непроверяемом исключение не выбрасывается, зато отбрасываются избыточные биты.

Все возможные значения для вещественных операндов приведены в табл. 2.

Символами **`x`** и **`y`** обозначены конечные положительные значения, символом **`z`** – результат операции вещественного умножения. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению «бесконечность», если слишком мал, он принимается за **`0.NaN`** (**`not a number`**) означает, что результат не является числом.

Примечание – Об «особых» значениях вещественных величин упоминалось

∞

Таблица 2 – Результаты вещественного умножения

*	+y	-y	+0	-0	+	-	NaN
+x	+z	-z	+0	-0	+	-	NaN
-x	-z	+z	-0	+0	-	+	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+	+	-	NaN	NaN	+	-	NaN
-	-	+	NaN	NaN	-	+	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Операция деления (**/**) вычисляет частное от деления первого операнда на второй.

Стандартная операция деления определена для типов **int**, **uint**, **long**, **ulong**, **float**, **double** и **decimal**. К величинам других типов ее можно применять, если для них существует неявное преобразование к этим типам. Тип результата определяется правилами преобразования, но не меньше **int**.

Если оба операнда целочисленные, результат операции округляется вниз до ближайшего целого числа. Если делитель равен нулю, генерируется исключение **System.DivideByZeroException**.

∞

Если хотя бы один из операндов вещественный, дробная часть результата деления не отбрасывается, а все возможные значения приведены в табл. 3. Символами **x** и **y** обозначены конечные положительные значения, символом **z** – результат операции вещественного деления. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению «бесконечность», если слишком мал, он принимается за 0.

Таблица 3 – Результаты вещественного деления

/	+y	-y	+0	-0	+	-	NaN
+x	+z	-z	+	-	+	-	NaN
-x	-z	+z	-	+	-	+	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+	+	-	+	-	NaN	NaN	NaN
-	-	+	-	+	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Для финансовых величин (тип **decimal**) при делении на 0 и переполнении генерируются соответствующие исключения, при исчезновении порядка результат равен 0.

∞

Операция остатка от деления (**%**) также интерпретируется по-разному для целых, вещественных и финансовых величин. Если оба операнда целочисленные, результат операции вычисляется по формуле **x-(x/y)*y**. Если делитель равен нулю, генерируется исключение **System.DivideByZeroException**. Тип результата операции равен «наибольшему» из типов операндов, но не менее **int**.

Если хотя бы один из операндов вещественный, результат операции вычисляется по формуле **x - n * y**, где **n** – наибольшее целое, меньшее или равное результату деления **x** на **y**. Все возможные комбинации значений операндов приведены в табл. 4. Символами **x** и **y** обозначены конечные положительные значения, символом **z** – результат операции.

Таблица 4 – Результаты вещественного остатка от деления

%	+y	-y	+0	-0	+	-	NaN
+x	+z	-z	NaN	NaN	x	x	NaN
-x	-z	-z	NaN	NaN	-x	-x	NaN
+0	+0	+0	NaN	NaN	+0	-0	NaN
-0	-0	-0	NaN	NaN	-0	-0	NaN
+	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Для финансовых величин (тип **decimal**) при получении остатка от деления на 0 и при переполнении генерируются соответствующие исключения, при исчезновении порядка результат равен 0. Знак результата равен знаку первого операнда.

Листинг 4 – Операции умножения, деления и получения остатка

```
using System;
namespace ConsoleApplication1
{ class Class1
  { static void Main()
    {
     int x = 11; y = 4;
     float z = 4;
     Console.WriteLine(Z * y); // Результат 16
     Console.WriteLine(z * 1e308); // Результат
"бесконечность
     Console.WriteLine( x/y ); // Результат 2
     Console.WriteLine( x/z ); // Результат 2,75
     Console.WriteLine(x % y ); // Результат 3
     Console.WriteLine(1e-324 / 1e-324 ); // Результат NaN
     }
   }
```

Еще раз обращаем ваше внимание на то, что несколько операций одного приоритета выполняются слева направо.

Пример. Рассмотрим выражение `2/x*y`.

Деление и умножение имеют один и тот же приоритет, поэтому сначала 2 делится на x, а затем результат этих вычислений умножается на y. Иными словами, это выражение эквивалентно формуле (`2/x)*y`

Если же мы хотим, чтобы выражение `x * y` было в знаменателе, следует заключить его в круглые скобки или сначала поделить числитель на `x`, а потом на `y`, то есть записать как `2/(x*y)` или `2/x/y`.

Сложение и вычитание. *Операция сложения* (`+`) возвращает сумму двух операндов. Стандартная операция сложения определена для типов `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`.

К величинам других типов ее можно применять, если для них существует неявное преобразование к этим типам. Тип результата операции равен «наибольшему» из типов операндов, но не менее `int`.

Если оба операнда целочисленные или типа `decimal` и результат операции слишком велик для представления с помощью заданного типа, генерируется исключение `System.OverflowException`.

Примечание – В проверяемом контексте. В непроверяемом исключение не выбрасывается, зато отбрасываются избыточные биты.

Все возможные значения для вещественных операндов приведены в табл. 5.

Символами **x** и **y** обозначены конечные положительные значения, символом **z** – результат операции вещественного сложения. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению «бесконечность», если слишком мал, он принимается за 0.

Таблица 5 – Результаты вещественного сложения

+	y	-0	-0	+	-	NaN
x	z	x	x	+	-	NaN
+0	y	+0	+0	+	-	NaN
-0	y	+0	-0	+	-	NaN
-0	-0	-0	NaN	+	NaN	NaN
+	+	+	+	+	NaN	NaN
-	-	-	-	NaN	-	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

Операция вычитания (**-**) возвращает разность двух операндов. Стандартная операция вычитания определена для типов **int**, **uint**, **long**, **ulong**, **float**, **double** и **decimal**.

К величинам других типов ее можно применять, если для них существует неявное преобразование к этим типам. Тип результата операции равен «наибольшему» из типов операндов, но не менее **int**.

Если оба операнда целочисленные или типа **decimal** и результат операции слишком велик для представления с помощью заданного типа, генерируется исключение **System.OverflowException**.

Все возможные значения результата вычитания для вещественных операндов приведены в табл. 6. Символами **x** и **y** обозначены конечные положительные значения, символом **z** – результат операции вещественного вычитания. Если **x** и **y** равны, результат равен положительному нулю. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению «бесконечность» с тем же знаком, что и **x-y**, если слишком мал, он принимается за 0 с тем же знаком, что и **x-y**.

Таблица 6 – Результаты вещественного вычитания

-	y	-0	-0	+	-	NaN
x	z	x	x	-	+	NaN
+0	-y	+0	+0	-	+	NaN
-0	-y	-0	+0	-	+	NaN
+	+	+	+	NaN	+	NaN
-	-	-	-	-	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

Операции сдвига. *Операции сдвига* (`<<` и `>>`) применяются к целочисленным операндам. Они сдвигают двоичное представление первого операнда влево или вправо на количество двоичных разрядов, заданное вторым операндом.

При сдвиге влево (`<<`) освободившиеся разряды обнуляются. При сдвиге вправо (`>>`) освободившиеся биты заполняются нулями, если первый операнд беззнакового типа (то есть выполняется логический сдвиг), и знаковым разрядом – в противном случае (выполняется арифметический сдвиг). Операции сдвига никогда не приводят к переполнению и потере значимости. Стандартные операции сдвига определены для типов **`int`**, **`uint`**, **`long`** и **`ulong`**.

Листинг 5 – Операции сдвига

```
using System;
namespace ConsoleApplication1
{ class Class1
   { static void Main()
     {
       byte a = 3, b = 9;
       sbyte c = 9, d = -9;
       Console.WriteLine(a << 1); //Результат 6
       Console.WriteLine(a << 2); //Результат 12
       Console.WriteLine(b >> 1); // Результат 4
       Console.WriteLine(c >> 1); // Результат 4
       Console.WriteLine(d >> 1); // Результат -5
      }
    }
}
```

Операции отношения и проверки на равенство. Операции отношения (`<`, `<=`, `>`, `>=`, `==`, `!=`) сравнивают первый операнд со вторым.

Операнды должны быть арифметического типа. Результат операции — логического типа, равен `true` или `false`. Правила вычисления результатов приведены в табл. 7.

Таблица 7 – Результаты операций отношения

Операция	Результат
`x == y`	`true, если x равно y, иначе false`
`x != y`	`true, если x не равно y, иначе false`
`x < y`	`true, если x меньше y, иначе false`
`x > y`	`true, если x больше y, иначе false`
`x <= y`	`true, если x меньше или равно y, иначе false`
`x >= y`	`true, если x больше или равно y, иначе false`

ПРИМЕЧАНИЕ – Обратите внимание на то, что операции сравнения на равенство и неравенство имеют меньший приоритет, чем остальные операции сравнения.

Очень интересно формируется результат операций отношения для особых случаев вещественных значений.

Пример. Если один из операндов равен **NaN**, результатом для всех операций, кроме **!=**, будет **false** (для операции **!=** результат равен **true**).

Очевиден факт, что для любых операндов результат операции **x ! = y** всегда равен результату операции **!(x==y)**, однако если один или оба операнда равны **NaN**, для операций **<**, **>**, **<=** и **>=** этот факт не подтверждается. Например, если **x** или **y** равны **NaN**, то **x < y** даст **false**, а **!(x >= y) - true**.

Другие особые случаи рассматриваются следующим образом:

- значения +0 и -0 равны;
- значение - ∞ меньше любого конечного значения и равно другому значению - ∞;
- значение + ∞ больше любого конечного значения и равно другому значению + ∞ .

Поразрядные логические операции. Поразрядные логические операции (**&**, **|**, **^**) применяются к целочисленным операндам и работают с их двоичными представлениями. Операнды сопоставляются побитно.

Стандартные операции определены для типов **int**, **uint**, **long** и **ulong**.

При поразрядной конъюнкции, или поразрядном **И** (обозначается **&**), бит результата равен 1, когда соответствующие биты обоих операндов равны 1.

При поразрядной дизъюнкции, или поразрядном **ИЛИ** (**|**), бит результата равен 1, когда соответствующий бит хотя бы одного из операндов равен 1.

При поразрядном исключающем **ИЛИ** (**^**) бит результата равен 1, когда соответствующий бит только одного из операндов равен 1.

Листинг 6 – Поразрядные логические операции

```
using System;
namespace ConsoleApplication1
{ class Class1
   { static void Main()
      {
        Console.WriteLine( 6 & 5 ); // Результат 4
        Console.WriteLine( 6 | 5 ); // Результат 7
        Console.WriteLine( 6 ^ 5 ); // Результат 3
      }
   }
}
```

Условные логические операции. Условные логические операции **И** (**&&**) и **ИЛИ** (**||**) чаще всего используются с операндами логического типа. Результатом логической операции является **true** или **false**.

Результат операции логическое **И** имеет значение **true**, только если оба операнда имеют значение **true**. Результат операции логическое **ИЛИ** имеет значение **true**, если хотя бы один из операндов имеет значение **true**.

ВНИМАНИЕ – Если значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется. Например, если первый операнд операции И равен false, результатом операции будет false независимо от значения второго операнда, поэтому он не вычисляется.

Листинг 7 – Условные логические операции

```
using System;
namespace ConsoleApplication1
{ class Class1
    { static void Main()
        {Console.WriteLine(true && true); // Результат true
       Console.WriteLine(true && false); // Результат false
       Console.WriteLine(true || true); // Результат true
       Console.WriteLine(true || false); // Результат true
        }
     }
}
```

Условная операция. *Условная операция* (**? :**) – тернарная, то есть имеет *три операнда*. Ее формат:

операнд_1 ? операнд_2 : операнд 3

Первый операнд – выражение, для которого существует неявное преобразование к логическому типу. Если результат вычисления первого операнда равен **true**, то результатом условной операции будет значение второго операнда, иначе – третьего операнда. Вычисляется всегда либо второй операнд, либо третий. Их тип может различаться.

Тип результата операции зависит от типа второго и третьего операндов:

- если операнды одного типа, он и становится типом результата операции (наиболее часто используемый вариант применения операции);
- иначе, если существует неявное преобразование типа от операнда 2 к операнду 3, но не наоборот, то типом результата операции становится тип операнда 3; иначе, если существует неявное преобразование типа от операнда 3 к операнду 2, но не наоборот, то типом результата операции становится тип операнда 2;
- иначе возникает ошибка компиляции.

Условную операцию часто используют вместо условного оператора **if** (он рассматривается далее) для сокращения текста программы.

Пример применения условной операции представлен в листинге 8.

Листинг 8 – Условная операция

```
using System;
namespace ConsoleApplication1
{ class Class1
   { static void Main()
      {
        int a = 11; b = 4;
        int max = b > a ? b : a;
        Console.WriteLine(max);  // Результат 11
           }
       }
}
```

Другой пример применения условной операции: требуется, чтобы некоторая целая величина увеличивалась на 1, если ее значение не превышает n, а иначе принимала значение 1. Это удобно реализовать следующим образом:

```
i = (i < n) ? i + 1: 1;
```

Условная операция правоассоциативна, то есть выполняется справа налево. Например, выражение

`a ? b : c ? d : e` вычисляется как `a ? b : (c ? d : e)`

Операции присваивания. *Операции присваивания* (**=**, **+=**, **-=**, ***=** и т.д.) задают новое значение переменной (свойству, событию или индексатору).

Эти операции могут использоваться в программе как законченные операторы.

Формат операции простого присваивания (**=**):

переменная = выражение

Механизм выполнения операции присваивания такой:

вычисляется выражение и его результат заносится в память по адресу, который определяется именем переменной, находящейся слева от знака операции.

То, что ранее хранилось в этой области памяти, естественно, теряется. Схематично это полезно представить себе так:

Переменная <- Выражение

Напомним, что константа и переменная являются *частными случаями выражения*.

Примеры операторов присваивания:

```
a = b + c / 2;
b = a;    // b = a и a = b – это совершенно разные действия!
a = b;    //  присваивание – это передача данных «налево».
x = 1;
```

Начинающие часто делают ошибку, воспринимая присваивание как аналог равенства в математике.

Чтобы избежать этой ошибки, надо понимать механизм работы оператора присваивания. Рассмотрим пример **(x = x + 0.5).**

Сначала из ячейки памяти, в которой хранится значение переменной **x**, выбирается это значение. Затем к нему прибавляется **0.5**, после чего получившийся результат записывается в ту же самую ячейку, а то, что хранилось там ранее, теряется безвозвратно. Операторы такого вида применяются в программировании очень широко.

Для правого операнда операции присваивания должно существовать неявное преобразование к типу левого операнда. Например, выражение целого типа можно присвоить вещественной переменной, потому что целые числа являются подмножеством вещественных, и информация при таком присваивании не теряется:

```
вещественная_переменная = целое_выражение;
```

Результатом операции присваивания является значение, записанное в левый операнд. Тип результата совпадает с типом левого операнда.

В *сложных операциях присваивания* (**+=**, ***=**, **/=** и т.п.) при вычислении выражения, стоящего в правой части, используется значение из левой части.

Пример. При сложении с присваиванием ко второму операнду прибавляется первый, и результат записывается в первый операнд, то есть выражение **a += b** является более компактной записью выражения **a = a + b**.

Результатом операции сложного присваивания является значение, записанное в левый операнд.

ПРИМЕЧАНИЕ – В документации написано, что тип результата совпадает с типом левого операнда, если он не менее **int**. Это означает, что если, например, переменные **a** и **b** имеют тип **byte**, присваивание

a += b недопустимо и требуется преобразовать тип явным образом: **a += (byte)b**. Однако на практике компилятор ошибку не выдает.

Напомним, что операции присваивания *правоассоциативны*, то есть выполняются справа налево, в отличие от большинства других операций (**a = b = c** означает **a = (b = c)**).