Программирование на C#

2. Операторы

Карбаев Д.С., 2016

Условный оператор

- `if(условие) оператор;`
- Здесь условие представляет собой булево (логическое), выражение, принимающее одно из двух значений: "истина" или "ложь". Если условие истинно, то оператор выполняется. А если условие ложно, то выполнение программы происходит, минуя оператор.
- Пример:
  - `if (10 < 11) Console.WriteLine("10 меньше 11");`
  - `if (10 < 9) Console.WriteLine("не подлежит выводу");`
- Набор операторов отношения, которые можно использовать в условных выражениях

Операция	Значение
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Не равно

Условный оператор

```
static void Main(string[] args) {
    int a, b, c;
    a = 2;  b = 3;
    if (a < b) Console.WriteLine("a меньше b");
    if (a == b) //Не подлежит выводу.
    Console.WriteLine("этого никто не увидит");
    c = a - b; //c содержит -1
    Console.WriteLine("c содержит -1");
    if (c >= 0) Console.WriteLine("значение c
                                   неотрицательно");
    if (c < 0) Console.WriteLine("значение c
                                   отрицательно");
    c = b - a; // теперь c содержит 1
    Console.WriteLine("c содержит 1");
    if (c >= 0) Console.WriteLine("значение c
                                   неотрицательно");
    if (c < 0) Console.WriteLine("значение c
                                   отрицательно ");
}
```

- Пример работы условного оператора if

- Результат выполнения программы:

```
a меньше b
c содержит -1
значение c отрицательно
c содержит 1
значение c неотрицательно
```

Оператор цикла

- **`for (инициализация; условие; итерация) оператор;`**
- В части **«инициализация»** задается начальное значение переменной управления циклом. Часть **«условие»** представляет собой булево выражение, проверяющее значение переменной управления циклом. Если результат проверки истинен, то цикл продолжается. Если же он ложен, то цикл завершается. В части **«итерация»** определяется порядок изменения переменной управления циклом на каждом шаге цикла, когда он повторяется.
- Пример:

```
static void Main(string[] args) {
  int count;
  for (count = 0; count < 5; count = count + 1)
    Console.WriteLine("Это подсчет: " + count);
    Console.WriteLine("Готово!");
}
```

- Результат работы:

```
Это подсчет: 0
Это подсчет: 1
Это подсчет: 2
Это подсчет: 3
Это подсчет: 4
Готово!
```

Стандартный способ записи цикла:

```
for (count = 0; count < 5; count++);
```

Возможности вывода

- При выводе составных строк - части этих строк сцепляются (склеиваются) знаком +, как в следующей строке:

```
Console.WriteLine("Вы заказали " + 2 + " предмета по цене $" + 3 + " каждый.");
```

- При выводе значения с плавающей точкой нельзя определить количество отображаемых десятичных разрядов. Рассмотрим оператор

```
Console.WriteLine("Деление 10 на 3 дает: " + 10.0/3.0);
```

который выводит следующий результат.

```
Деление 10 на 3 дает: 3,33333333333333
```

Возможности вывода

- Для управления форматированием числовых данных служит другая форма метода WriteLine (), позволяющая встраивать информацию форматирования, как показано ниже.

```
WriteLine("форматирующая строка", arg0, arg1, ... , argN);
```

- В этой форме аргументы метода WriteLine() разделяются запятой. А форматирующая строка состоит из двух элементов: обычных печатаемых символов, предназначенных для вывода в исходном виде, а также спецификаторов формата. Последние указываются в следующей общей форме:

```
{argnum, width: fmt}
```

где **argnum** — номер выводимого аргумента, начиная с нуля; например:
спецификатор {0} обозначает аргумент arg0, спецификатор {1} — аргумент arg1 и т.д.
width — минимальная ширина поля;
fmt — формат. Параметры width и fmt являются необязательными.

Возможности вывода

- Начнем с самого простого примера. При выполнение оператора

```
Console.WriteLine("В феврале {0} или {1} дней.", 28, 29);
```

- получается следующий результат.

```
В феврале 28 или 29 дней.
```

- Ниже приведен видоизмененный вариант предыдущего оператора, в котором указывается ширина полей.

```
Console.WriteLine("В феврале {0,10} или {1,5} дней.", 28, 29);
```

- Выполнение этого оператора дает следующий результат.

```
В феврале         28 или    29 дней.
```

Возможности вывода

- Образец требуемого формата задается с помощью символов #, обозначающих разряды чисел. Можно указать десятичную точку и запятые, разделяющие цифры.
- Пример более подходящего вывода результата деления 10 на 3:

```
Console.WriteLine("Деление 10 на 3 дает: {0:#.##}", 10.0/3.0);
```

- Выполнение этого оператора приводит к следующему результату.

```
Деление 10 на 3 дает: 3,33
```

Возможности вывода

- Пример программы вывода таблицы результатов возведения чисел в квадрат и куб

```
static void Main(string[] args)  {
  int i;
  Console.WriteLine("Число\tКвадрат\tКуб");
  for (i = 1; i < 10; i++)
    Console.WriteLine("{0}\t{1}\t{2}", i, i*i, i*i*i);
  }
```

Результат работы:

Число	Квадрат	Куб
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729

Возможности вывода

- Рассмотрим еще один пример. Оператор

```
Console.WriteLine("{0:###,###.##}", 123456.56);
```

дает следующий результат.

```
123 456,56
```

- Для вывода денежных сумм, например, рекомендуется использовать спецификатор формата C.

```
decimal balance;
balance = 12323.09m;
Console.WriteLine("Текущий баланс равен {0:C}", balance);
```

- Результат выполнения этого фрагмента кода выводится в формате денежных сумм (Currency), указываемых в рублях.

```
Текущий баланс равен 12 323,09 р.
```

Литералы

- В C# **литералами** называются постоянные значения, представленные в удобной для восприятия форме.
- В C# литералы могут быть любого простого типа.
  - Символьные: ‘a’, ‘%’
  - Целочисленные: 10, -100
  - С плавающей точкой: 11.123

```
char c=‘5’;
int x = 33;
double b=0.5;
```

- Вопрос: к какому типу следует отнести числовой литерал, например 987 или 0.23?

Литералы

- Правила типизации литералов:
  - у целочисленных литералов должен быть самый мелкий целочисленный тип, которым они могут быть представлены, начиная с типа int. Например: для значения 100 - int, 2147483648 - uint,
    10000000000000 - long или 9223372036854775808 - ulong);
  - литералы с плавающей точкой относятся к типу double.
- Суффиксы позволяют явно указать тип литерала:
  - 10 – int; 12L – long; 100U – uint; 984375UL – ulong.
  - 164.4 (или 164.4D) – double; 10.19F – float; 9.95M – decimal.
- Шестнадцатеричные литералы:

```
int p = 0xFF; // 255 в десятичной системе
int q = 0x1A; // 26 в десятичной системе
```

Управляющие последовательности

- Ряд символов, в числе которых табуляция, перенос строки, одинарные и двойные кавычки, имеют специальное назначение в C#, поэтому их нельзя использовать непосредственно. По этим причинам в C# предусмотрены специальные **управляющие последовательности** символов.

Управляющая последовательность	Описание
\a	Звуковой сигнал (звонок)
\b	Возврат на одну позицию
\f	Перевод страницы (переход на новую страницу)
\n	Новая строка (перевод строки)
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\0	Пустой символ
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта

Область видимости переменных

- В C# локальную переменную разрешается объявлять в любом кодовом блоке. **Кодовый блок** начинается открывающей фигурной скобкой и оканчивается закрывающей фигурной скобкой. Этот блок и определяет **область действия**. Следовательно, всякий раз, когда начинается блок, образуется новая область действия.
- Локальные переменные, объявленные во внешней области действия, будут видимы для кода во внутренней области действия. Но обратное не справедливо: локальные переменные, объявленные во внутренней области действия, не будут видимы вне этой области.

Область видимости переменных

```
static void Main(string[] args)
{
    int x; // Эта переменная доступна для всего кода внутри метода Main().
    x = 10;
    if (x == 10)
    { // начать новую область действия
        int y = 20; // Эта переменная доступна только в данном кодовом блоке.
        // Здесь доступны обе переменные, x и y.
        Console.WriteLine("x и y: " + x + " " + y);
        x = y * 2;
    }
    // y = 100; // Ошибка! Переменна y здесь недоступна.
    // А переменная x здесь по-прежнему доступна.
    Console.WriteLine("x равно " + x);
}
```

Область видимости переменных

- Рассмотрим пример:

```
static void Main(string[] args) {
    int x;
    for (x = 0; x < 3; x++)  {
        int y = -1; // Переменная y инициализируется при каждом входе в блок.
        Console.WriteLine("y равно: " + y); // Здесь всегда выводится -1
        y = 100;
        Console.WriteLine("y теперь равно: " + y);

    }
}
```

Результат выполнения:

```
y равно: -1
y теперь равно: 100
y равно: -1
y теперь равно: 100
y равно: -1
y теперь равно: 100
```

Область видимости переменных

- Несмотря на то, что блоки могут быть вложены, ни у одной из переменных из внутренней области действия не должно быть такое же имя, как и у переменной из внешней области действия.

```
static void Main(string[] args)
 {
     int count; // Объявили переменную count
     for (count = 0; count < 10; count = count + 1)
     {
         Console.WriteLine("Это подсчет: " + count);
         int count; // Недопустимо!!! Нельзя 2й раз объявить.
         for (count = 0; count < 2; count++)
             Console.WriteLine("В этой программе есть ошибка!");
      }
 }
```

Операторы

- Арифметические операторы

Оператор	Действие
+	Сложение
–	Вычитание, унарный минус
*	Умножение
/	Деление
%	Деление по модулю **(остаток от деления)**
––	Декремент
++	Инкремент

Остаток от деления

- Рассмотрим применение оператора деления по модулю:

```
static void Main(string[] args)  {
    int iresult, irem;
    double dresult, drem;
    iresult = 10 / 3; //результат целочисленного деления
    irem = 10 % 3;  //остаток от деления
    dresult = 10.0 / 3.0; //результат вещественного деления
    drem = 10.0 % 3.0; //остаток от деления
    Console.WriteLine("Результат и остаток от деления 10/3: " +
      iresult + " " + irem);
    Console.WriteLine("Результат и остаток от деления 10.0 / 3.0: " +
      dresult + " " + drem);
 }
```

Результат выполнения:

```
Результат и остаток от деления 10/3: 3 1
Результат и остаток от деления 10.0/3.0: 3,33333333333333 1
```

Операторы инкремента (++) и декремента (--)

- Оператор инкремента увеличивает свой операнд на 1, а оператор декремента уменьшает операнд на 1. Следовательно, оператор

`x++;` равнозначен оператору `x = x + 1;`

а оператор

`x--;` равносилен оператору `x = x - 1;`

- Оба оператора инкремента и декремента можно указывать до операнда (в префиксной форме) или же после операнда (в постфиксной форме). Например, оператор

```
x = x + 1;
```

может быть записан в следующем виде:

```
++x; // префиксная форма
```

или же в таком виде:

```
x++; // постфиксная форма
```

Операторы инкремента (++) и декремента (--)

```
static void Main(string[] args) {
   int x, y;
   int i; x = 1; y = 0;
   Console.WriteLine("Ряд чисел, полученных " +
   "с помощью оператора y = y + x++;");
   for (i = 0; i < 10; i++) {
   // постфиксная форма оператора ++
       y = y + x++;
       Console.WriteLine(y + " ");
   }
   x = 1; y = 0;
   Console.WriteLine("Ряд чисел, полученных " +
   "с помощью оператора y = y + ++x;");
   for (i = 0; i < 10; i++) {
   // префиксная форма оператора ++
     y = y + ++x; Console.WriteLine(y + " ");
   }
}
```

Ряд чисел, полученных с помощью оператора y = y + x++	Ряд чисел, полученных с помощью оператора y = y + ++x;
1	2
3	5
6	9
10	14
15	20
21	27
28	35
36	44
45	54
55	65

Операторы отношения и логические операторы

Оператор	Значение
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Оператор	Значение
&	И
|	ИЛИ
^	Исключающее ИЛИ
&&	Укороченное И
||	Укороченное ИЛИ
!	НЕ

Логические операторы

- Операнды логических операторов должны относиться к типу bool, а результат выполнения логической операции также относится к типу bool. Логические операторы **&, |, ^** и **!** поддерживают основные логические операции И, ИЛИ, исключающее ИЛИ и НЕ в соответствии с приведенной ниже таблицей истинности

p	q	p & q	p | q	p ^ q	!p
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

p	q	p & q	p | q	p ^ q	!p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Логические операторы

```
static void Main(string[] args) {
  int i, j;
  bool b1, b2;
  i = 10; j = 11;
  if (i < j) Console.WriteLine("i < j");
  if (i <= j) Console.WriteLine("i <= j");
  if (i != j) Console.WriteLine("i != j");
  if (i == j) Console.WriteLine("Нельзя выполнить");
  if (i >= j) Console.WriteLine("Нельзя выполнить");
  if (i > j) Console.WriteLine("Нельзя выполнить");
  b1 = true;
  b2 = false;
  if (b1 & b2) Console.WriteLine("Нельзя выполнить");
  if (!(b1 & b2)) Console.WriteLine("!(b1& b2) - true");
  if (b1 | b2) Console.WriteLine(“b1 | b2 - true");
  if (b1 ^ b2) Console.WriteLine(“b1 ^ b2 - true");
}
```

- Результат выполнения:

```
i < j
i <= j
i != j
!(b1 & b2) - true
b1 | b2 - true
b1 ^ b2 - true
```

Импликация

- **Импликация** — это двоичная операция, результатом которой является ложное значение только в том случае, если левый ее операнд имеет истинное значение, а правый — ложное. Операция импликации отражает следующий принцип: истина не может подразумевать ложь. Соответствует рассуждениям формата: **если…, то….**

p	q	Результат импликации p и q
true	true	true
true	false	false
false	false	true
false	true	true

- Операция импликации может быть построена на основе комбинации логических операторов ! и |, как в приведенной ниже строке кода.

```
!p | q
```

Импликация

Результат работы:

```
p равно True,
q равно True

Результат импликации
True и True равен True
p равно True, q равно
False
p равно False, q равно
False

Результат импликации
False и True равен
True
p равно False, q равно
False

Результат импликации
False и False равен
True
```

```
static void Main(string[] args) {
  bool p = false, q = false;
  int i, j;
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      if (i == 0) p = true;
      if (i == 1) p = false;
      if (j == 0) q = true;
      if (j == 1) q = false;
        Console.WriteLine("p равно " + p +
                          ", q равно " + q);
      if (!p | q)
        Console.WriteLine("Результат импликации " + p +
                          " и " + q + " равен " + true);
        Console.WriteLine();
      }
    }
 }
```

Условные логические операторы

```
static void Main(string[] args) {
    int n, d;
    n = 10; d = 2;
    if (d != 0 && (n % d) == 0)
        Console.WriteLine(n + " делится нацело на " + d);
    d = 0; // задать нулевое значение переменной d
    // d равно нулю, поэтому второй операнд не вычисляется
    if (d != 0 && (n % d) == 0)
        Console.WriteLine(n + " делится нацело на " + d);
    // Если теперь попытаться сделать то же самое
    // без укороченного логического оператора, то
    //возникнет ошибка из-за деления на ноль.
    if (d != 0 & (n % d) == 0)
        Console.WriteLine(n + " делится нацело на " + d);

}
```

Укороченная (условная) логическая операция И выполняется с помощью оператора **&&**, а укороченная логическая операция ИЛИ — с помощью оператора **||**. Этим укороченным логическим операторам соответствуют обычные логические операторы **&** и **|**.

Условные логические операторы

```
static void Main(string[] args) {
      int i;
      bool someCondition = false;
      i = 0;
      // Значение переменной i инкрементируется,
      // несмотря на то, что оператор if не выполняется.
      if (someCondition & (++i < 100))
          Console.WriteLine("Не выводится");
      Console.WriteLine("Оператор if выполняется: " + i); // выводится 1
      //В данном случае значение переменной i не инкрементируется,
      // поскольку инкремент в укороченном логическом операторе опускается.
      i = 0;
      if (someCondition && (++i < 100))
          Console.WriteLine("Не выводится");
      Console.WriteLine("Оператор if не выполняется: " + i); // выводится 0 !!

 }
```

Оператор присваивания

- Общая форма оператора присваивания:

```
имя_переменной = выражение
```

- Рассмотрим, например, следующий фрагмент кода.

```
int x, y, z;
x = y = z = 100; // присвоить значение 100 переменным x, y и z
```

В приведенном выше фрагменте кода одно и то же значение 100 задается для переменных x, y и z.

- Пример:

```
x = x + 10;
```

можно переписать, используя следующий составной оператор присваивания.

```
x += 10;
```

- Общая форма всех этих операторов имеет следующий вид:

```
имя_переменной op = выражение
```

где **op** — арифметический или логический оператор, применяемый вместе с оператором присваивания.

- Составные операторы присваивания для арифметических и логических операций:

```
+=      -=      *=      /=
%=      &=      |=      ^=
```

Порязрядные операторы

- Поразрядные операторы воздействуют на отдельные двоичные разряды (биты) своих операндов.

Оператор	Значение
&	Поразрядное И
|	Поразрядное ИЛИ
^	Поразрядное исключающее ИЛИ
>>	Сдвиг вправо
<<	Сдвиг влево
~	Дополнение до 1 (унарный оператор НЕ)

- Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ обозначаются следующим образом: &, |, и ~.

p	q	p & q	p | q	p ^ q	~p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Поразрядное И

```
static void Main(string[] args) {
            ushort num;
            ushort i;
            for (i = 1; i <= 10; i++) {
                num = i;
                Console.WriteLine("num: " + num);
                num = (ushort)(num & 0xFFFE); //0xFFFE - 1111111111111110
                Console.WriteLine("num после сброса младшего разряда: "
                + num + "\n");
            }
}
```

Результат выполнения:

```
num: 1
num после сброса младшего разряда: 0
num: 2
num после сброса младшего разряда: 2
num: 3
num после сброса младшего разряда: 2
num: 4
num после сброса младшего разряда: 4
num: 5
num после сброса младшего разряда: 4
num: 6
num после сброса младшего разряда: 6
num: 7
num после сброса младшего разряда: 6
num: 8
num после сброса младшего разряда: 8
num: 9
num после сброса младшего разряда: 8
num: 10
num после сброса младшего разряда: 10
```

Поразрядное И

- Поразрядным оператором И удобно также пользоваться для определения установленного или сброшенного состояния отдельного двоичного разряда. В следующем примере программы определяется, является ли число нечетным.

```
static void Main(string[] args) {
            ushort num;
            num = 16;
            if ((num & 1) == 1) //16 - 10000
                Console.WriteLine("Не выводится.");
            num = 15; //15 - 1111
            if ((num & 1) == 1)
                Console.WriteLine(num + " - нечетное число.");
}
```

- Результат:

```
15 — нечетное число.
```

Поразрядное И

- Возможностью проверять состояние отдельных двоичных разрядов с помощью поразрядного оператора & можно воспользоваться для написания программы, в которой отдельные двоичные разряды проверяемого значения типа byte приводятся в двоичной форме.

```
static void Main(string[] args)  {
  int t; byte val;
  val = 123;
  for (t = 128; t > 0; t = t / 2) {
   if ((val & t) == 1) Console.Write("1");
   if ((val & t) == 0) Console.Write("0");
  }
}
```

```
128 – 10000000
 64 – 01000000
 32 – 00100000
 16 - 00010000
  8 - 00001000
  4 – 00000100
  2 – 00000010
  1 – 00000001
```

- Выполнение этой программы дает следующий результат.

```
01111011
```

Поразрядное ИЛИ

- Пример, где четные числа преобразуются в нечетные

```
static void Main(string[] args) {
  ushort num; ushort i;
  for (i = 1; i <= 10; i++) {
    num = i;  Console.WriteLine("num: " + num);
    num = (ushort)(num | 1);
    Console.WriteLine("num после установки младшего разряда: " + num + "\n");
  }
}
```

Результат выполнения:

```
num: 1
num после установки младшего разряда: 1
num: 2
num после установки младшего разряда: 3
num: 3
num после установки младшего разряда: 3
num: 4
num после установки младшего разряда: 5
num: 5
num после установки младшего разряда: 5
num: 6
num после установки младшего разряда: 7
num: 7
num после установки младшего разряда: 7
num: 8
num после установки младшего разряда: 9
num: 9
num после установки младшего разряда: 9
num: 10
num после установки младшего разряда: 11
```

Поразрядное XOR

```
Принцип XOR:
5 ^ 3 = 6
101 ^ 011 = 110

6 ^ 3 = 5
110 ^ 011 = 101
```

- Пример

```
R1 = X ^ Y; R2 = R1 ^ Y; // R2==X
```

значение переменной R2 оказывается в итоге таким же, как и значение переменной X.

- Пример:

```
static void Main(string[] args) {
  char ch1 = 'H'; char ch2 = 'i';
  char ch3 = '!'; int key = 88;
  Console.WriteLine("Исходное сообщение: "
                    + ch1 +  ch2 + ch3);
  // Зашифровать сообщение
  ch1 = (char)(ch1 ^ key);
  ch2 = (char)(ch2 ^ key);
  ch3 = (char)(ch3 ^ key);
  Console.WriteLine("Зашифрованное сообщение: "
                    + ch1 + ch2 + ch3);
  // Расшифровать сообщение
  ch1 = (char)(ch1 ^ key);
  ch2 = (char)(ch2 ^ key);
  ch3 = (char)(ch3 ^ key);
  Console.WriteLine("Расшифрованное сообщение: "
                    + ch1 + ch2 + ch3);
}
```

- Результат

```
Исходное сообщение: Hi!
Зашифрованное сообщение:  ►1y
Расшифрованное сообщение: Hi!
```

Операторы сдвига

- В C# имеется возможность сдвигать двоичные разряды, составляющие целое значение, влево или вправо на заданную величину. Для этой цели в C# определены два приведенных ниже оператора сдвига двоичных разрядов.

<< Сдвиг влево

>> Сдвиг вправо

- Ниже приведена общая форма для этих операторов:

```
значение << число_битов
значение >> число_битов
```

где число_битов — это число двоичных разрядов, на которое сдвигается указанное значение.

Операторы сдвига

- пример программы, наглядно демонстрирующий действие сдвига влево и вправо.

```
static void Main(string[] args)  {
  int val = 1;
  for (int i = 0; i < 8; i++) {
    for (int t = 128; t > 0; t = t / 2) {
      if ((val & t) != 0) Console.Write("1");
      if ((val & t) == 0) Console.Write("0");
    }
    val = val << 1; // сдвиг влево
  } Console.WriteLine(); val = 128;
  for (int i = 0; i < 8; i++) {
    for (int t = 128; t > 0; t = t / 2) {
      if ((val & t) != 0) Console.Write("1 ");
      if ((val & t) == 0) Console.Write("0 ");
    }  Console.WriteLine();
    val = val >> 1; // сдвиг вправо
   }
}
```

```
Результат выполнения:
00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000

10000000
01000000
00100000
00010000
00001000
00000100
00000010
00000001
```

Операторы сдвига

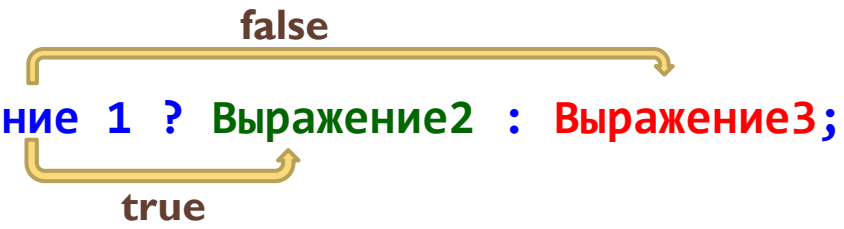
```
static void Main(string[] args) {
  int n = 10;
  Console.WriteLine("Значение переменной n: " + n);
  n = n << 1; // Умножить на 2.
  Console.WriteLine("Значение переменной n после " +
  "операции n = n * 2: " + n);
  n = n << 2; // Умножить на 4.
  Console.WriteLine("Значение переменной n после " +
  "операции n = n * 4: " + n);
  n = n >> 1; // Разделить на 2.
  Console.WriteLine("Значение переменной n после " +
  "операции n = n / 2: " + n);
  n = n >> 2; // Разделить на 4.
  Console.WriteLine("Значение переменной n после " +
  "операции n = n / 4: " + n); Console.WriteLine();
  // Установить переменную n в исходное состояние.
  n = 10;
  Console.WriteLine("Значение переменной n: " + n);
  // Умножить на 2 тридцать раз.
  n = n << 30; // данные теряются
  Console.WriteLine("Значение переменной n после " +
  "сдвига на 30 позиций влево: " + n);
}
```

Результат выполнения программы

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

Оператор “?”

- Оператор ? представляет собой условный оператор и используется вместо определенных видов конструкций if-then-else. Оператор ? еще называют тернарным, поскольку для него требуются три операнда. Общая форма этого оператора:



здесь Выражение1 должно относиться к типу bool, а Выражение2 и Выражение3 — к одному и тому же типу.

- Рассмотрим следующий пример, в котором переменной absval присваивается значение переменной val.

```
absval = val < 0 ? -val : val; // получить абсолютное значение переменной val
```

Оператор “?”

- Пример применения оператора “?”. В данной программе одно число делится на другое, но при этом исключается деление на ноль.

```
static void Main(string[] args) {
  int result;
  for (int i = -5; i < 6; i++) {
    result = i != 0 ? 100 / i : 0;
    if (i != 0)
      Console.WriteLine("100 / " + i + " равно " + result);
  }
}
```

- Выполнение этой программы дает следующий результат:

```
100 / -5 равно -20
100 / -4 равно -25
100 / -3 равно -33
100 / -2 равно -50
100 / -1 равно -100
100 / 1 равно 100
100 / 2 равно 50
100 / 3 равно 33
100 / 4 равно 25
100 / 5 равно 20
```

Оператор “?”

- Значение, которое дает оператор ?, можно использовать в качестве аргумента при вызове метода. А если все выражения в операторе ? относятся к типу bool, то такой оператор может заменить собой условное выражение в цикле или операторе if. В приведенном ниже примере программы выводятся результаты деления числа 100 только на четные, ненулевые значения.

```
static void Main(string[] args) {
    for (int i = -5; i < 6; i++)
       if (i != 0 ? (i % 2 == 0) : false)
           Console.WriteLine("100 / " + i + " равно " + 100 / i);
}
```

Результат:

```
100 / -4 равно -25
100 / -2 равно -50
100 / 2 равно 50
100 / 4 равно 25
```

Бонус: чтение с консоли

- Как уже известно, метод Read() позволяет считать один символ с консоли, который вводит пользователь, а метод ReadLine() позволяет считать строку.
- ***А если пользователь вводит число, как записать его в числовую переменную?***

```
        static void Main(string[] args) {
            Console.Write("Введите целое число: ");
            int x = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Вы ввели x="+x);
            Console.Write("Введите вещественное число: ");
            double y = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Вы ввели y=" + y);
        }
```

Результат:

```
Введите целое число: 5
Вы ввели x=5
Введите вещественное число: 3,4
Вы ввели y=3,4
```

ASCII TABLE

Коды символов

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]