Содержание

Об авторах 7

Карманный справочник по языку C# 7.0 8

Соглашения, используемые в этой книге 8
Использование примеров кода 9
Ждем ваших отзывов! 10
Первая программа на C# 11
Синтаксис 14
Основы типов 17
Числовые типы 26
Булевские типы и операции 33
Строки и символы 35
Массивы 39
Переменные и параметры 43
Выражения и операции 51
Операции для работы со значениями `null` 57
Операторы 59
Пространства имен 68
Классы 72
Наследование 87
Тип `object` 95
Структуры 100
Модификаторы доступа 101
Интерфейсы 103
Перечисления 106
Вложенные типы 109
Обобщения 109
Делегаты 118
События 125
Лямбда-выражения 130

Анонимные методы 135
Операторы `try` и исключения 136
Перечисление и итераторы 144
Типы, допускающие значение `null` 150
Расширяющие методы 154
Анонимные типы 156
Кортежи (C# 7) 157
LINQ 159
Динамическое связывание 184
Перегрузка операций 193
Атрибуты 197
Атрибуты информации о вызывающем компоненте 200
Асинхронные функции 202
Небезопасный код и указатели 212
Директивы препроцессора 216
XML-документация 218
Предметный указатель 222

Модификатор `out`

Аргумент `out` похож на аргумент `ref` за исключением следующих аспектов:

- он не нуждается в присваивании значения перед входом в функцию;
- ему должно быть присвоено значение перед *выходом* из функции.

Модификатор `out` чаще всего применяется для получения из метода нескольких возвращаемых значений.

Переменные `out` и отбрасывание (C# 7)

В версии C# 7 переменные можно объявлять на лету при вызове методов с параметрами `out`:

```
int.TryParse ("123", out int x);
Console.WriteLine (x);
```

Этот код эквивалентен следующему коду:

```
int x;
int.TryParse ("123", out x);
Console.WriteLine (x);
```

Когда вызываются методы с множеством параметров `out`, посредством символа подчеркивания можно "отбрасывать" любые параметры, которые не интересны для кода. Предполагая, что метод `SomeBigMethod()` был определен с пятью параметрами `out`, вот как проигнорировать все параметры кроме третьего:

```
SomeBigMethod (out _, out _, out int x, out _, out _);
Console.WriteLine (x);
```

Модификатор `params`

Для последнего параметра метода может быть указан модификатор `params`, чтобы метод принимал любое количество аргументов определенного типа. Тип параметра должен быть объявлен как массив. Например:

```
static int Sum (params int[] ints)
{
  int sum = 0;
  for (int i = 0; i < ints.Length; i++) sum += ints[i];
  return sum;
}
```

Вызвать метод `Sum()` можно так:

```
Console.WriteLine (Sum (1, 2, 3, 4));  // 10
```

Аргумент `params` может быть также задан как обычный массив. Предыдущий вызов семантически эквивалентен следующему коду:

```
Console.WriteLine (Sum (new int[] { 1, 2, 3, 4 } ));
```

Необязательные параметры

Начиная с версии C# 4.0, в методах, конструкторах и индексаторах можно объявлять *необязательные параметры*. Параметр является необязательным, если в его объявлении указано стандартное значение:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

При вызове метода необязательные параметры могут быть опущены:

```
Foo();  // 23
```

На самом деле необязательному параметру `x` *передается стандартный аргумент* со значением `23` — компилятор встраивает это значение в скомпилированный код на *вызывающей* стороне. Показанный выше вызов `Foo()` семантически эквивалентен такому вызову:

```
Foo (23);
```

потому что компилятор просто подставляет стандартное значение необязательного параметра там, где он используется.

ВНИМАНИЕ!

Добавление необязательного параметра к открытому методу, который вызывается из другой сборки, требует перекомпиляции обеих сборок — точно как в случае, если бы параметр был обязательным.

Стандартное значение необязательного параметра должно быть указано в виде константного выражения или конструктора без параметров для типа значения. Необязательные параметры не могут быть помечены посредством `ref` или `out`.

Обязательные параметры должны находиться *перед* необязательными параметрами в объявлении метода и при его вызове (исключением являются аргументы `params`, которые всегда располагаются в конце). В следующем примере для `x` передается явное значение `1`, а для `y` — стандартное значение `0`:

```
void Foo (int x = 0, int y = 0)
{
  Console.WriteLine (x + ", " + y);
}
void Test()
{
  Foo(1);  // 1, 0
}
```

Чтобы сделать обратное (передать стандартное значение для `x` и указанное явно значение для `y`), потребуется скомбинировать необязательные параметры с *именованными аргументами*.

Именованные аргументы

Вместо распознавания аргумента по позиции его можно идентифицировать по имени. Например:

```
void Foo (int x, int y)
{
  Console.WriteLine (x + ", " + y);
}
void Test()
{
  Foo (x:1, y:2);  // 1, 2
}
```

Именованные аргументы могут встречаться в любом порядке. Следующие вызовы `Foo()` семантически идентичны:

```
Foo (x:1, y:2);
Foo (y:2, x:1);
```

Именованные и позиционные аргументы можно смешивать при условии, что именованные аргументы указаны последними:

```
Foo (1, y:2);
```

Именованные аргументы особенно удобны в сочетании с необязательными параметрами. Например, взгляните на такой метод:

```
void Bar (int a=0, int b=0, int c=0, int d=0) { ... }
```

Его можно вызвать, предоставив только значение для d:

```
Bar (d:3);
```

Это чрезвычайно удобно при работе с API-интерфейсами COM.

Объявление неявно типизированных локальных переменных с помощью var

Часто случается так, что переменная объявляется и инициализируется за один шаг. Если компилятор способен вывести тип из инициализирующего выражения, то на месте объявления типа можно применять ключевое слово var. Например:

```
var x = "строка";
var y = new System.Text.StringBuilder();
var z = (float)Math.PI;
```

Это в точности эквивалентно следующему коду:

```
string x = "строка";
System.Text.StringBuilder y =
  new System.Text.StringBuilder();
float z = (float)Math.PI;
```

Из-за такой прямой эквивалентности неявно типизированные переменные являются статически типизированными. К примеру, приведенный ниже код вызовет ошибку на этапе компиляции:

```
var x = 5;
x = "строка";  // Ошибка на этапе компиляции;
               // x имеет тип int
```

В разделе “Анонимные типы” на стр. 156 мы опишем сценарий, когда использовать ключевое слово var обязательно.

Выражения и операции

Выражение по существу указывает значение. Простейшими разновидностями выражений являются константы (наподобие 123) и переменные (вроде x). Выражения могут видоизменяться и комбинироваться с помощью операций. *Операция* принимает один или более входных *операндов* и дает на выходе новое выражение:

```
12 * 30  // * - операция, а 12 и 30 - операнды
```

Можно строить сложные выражения, поскольку операнд сам по себе может быть выражением, как операнд `(12 * 30)` в следующем примере:

```
1 + (12 * 30)
```

Операции в C# могут быть классифицированы как унарные, бинарные и тернарные в зависимости от количества операндов, с которыми они работают (один, два или три). Бинарные операции всегда применяют *инфиксную* форму записи, при которой операция помещается между двумя операндами.

Операции, которые являются неотъемлемой частью самого языка, называются *первичными*; примером может служить операция вызова метода. Выражение, не имеющее значения, называется *пустым выражением*:

```
Console.WriteLine (1)
```

Поскольку пустое выражение не имеет значения, оно не может использоваться в качестве операнда при построении более сложных выражений:

```
1 + Console.WriteLine (1) //Ошибка на этапе компиляции
```

Выражения присваивания

Выражение присваивания применяет операцию = для присваивания переменной результата вычисления другого выражения. Например:

```
x = x * 5
```

Выражение присваивания — не пустое выражение. На самом деле оно заключает в себе присваиваемое значение и потому может встраиваться в другое выражение. В следующем примере выражение присваивает `2` переменной `x` и `10` переменной `y`:

```
y = 5 * (x = 2)
```

Такой стиль выражения может применяться для инициализации нескольких значений:

```
a = b = c = d = 0
```

Составные операции присваивания являются синтаксическим сокращением, которое комбинирует присваивание с другой операцией.

Например:

```
x *= 2   // эквивалентно x = x * 2
x <<= 1  // эквивалентно x = x << 1
```

(Тонкое исключение из этого правила касается *событий*, которые рассматриваются позже: операции += и −= в них трактуются специальным образом и отображаются на средства доступа `add` и `remove` события.)

Приоритеты и ассоциативность операций

Когда выражение содержит несколько операций, порядок их вычисления определяется приоритетами и ассоциативностью. Операции с более высокими приоритетами выполняются перед операциями, приоритеты которых ниже. Если операции имеют одинаковые приоритеты, то порядок их выполнения определяется ассоциативностью.

Приоритеты операций

Выражение `1 + 2 * 3` вычисляется как `1 + (2 * 3)`, потому что операция `*` имеет более высокий приоритет, чем +.

Левоассоциативные операции

Бинарные операции (кроме операции присваивания, лямбда-операции и операции объединения с `null`) являются *левоассоциативными*; другими словами, они вычисляются слева направо. Например, выражение `8/4/2` вычисляется как `(8/4)/2` по причине левой ассоциативности. Разумеется, порядок вычисления можно изменить, расставив скобки.

Правоассоциативные операции

Операция присваивания, *лямбда-операция*, операция объединения с `null` и условная операция являются *правоассоциативными*; другими словами, они вычисляются справа налево. Правая ассоциативность позволяет компилировать множественное присваивание, такое как `x=y=3`: сначала значение `3` присваивается `y`, а затем результат этого выражения (`3`) присваивается `x`.

Таблица операций

В следующей таблице перечислены операции C# в порядке приоритетов. Операции в одной и той же категории имеют одинаковые приоритеты. Операции, которые могут быть перегружены пользователем, объясняются в разделе “Перегрузка операций” на стр. 193.

Символ операции	Название операции	Пример	Возможность перегрузки
Первичные			
`.`	Доступ к члену	`x.y`	Нет
`->`	Указатель на структуру (небезопасная)	`x->y`	Нет
`()`	Вызов функции	`x()`	Нет
`[]`	Массив/индекс	`a[x]`	Через индексатор
`++`	Постфиксная форма инкремента	`x++`	Да
`--`	Постфиксная форма декремента	`x--`	Да
`new`	Создание экземпляра	`new Foo()`	Нет
`stackalloc`	Небезопасное выделение памяти в стеке	`stackalloc(10)`	Нет
`typeof`	Получение типа по идентификатору	`typeof(int)`	Нет
`nameof`	Получение имени идентификатора	`nameof(x)`	Нет
`checked`	Включение проверки целочисленного переполнения	`checked(x)`	Нет
`unchecked`	Отключение проверки целочисленного переполнения	`unchecked(x)`	Нет
`default`	Стандартное значение	`default(char)`	Нет
Унарные			
`await`	Ожидание	`await myTask`	Нет
`sizeof`	Получение размера структуры	`sizeof(int)`	Нет

Символ операции	Название операции	Пример	Возможность перегрузки
+	Положительное значение	+x	Да
–	Отрицательное значение	-x	Да
!	НЕ	!x	Да
~	Побитовое дополнение	~x	Да
++	Префиксная форма инкремента	++x	Да
--	Префиксная форма декремента	--x	Да
()	Приведение	(int)x	Нет
*	Значение по адресу (небезопасная)	*x	Нет
&	Адрес значения (небезопасная)	&x	Нет
Мультипликативные			
*	Умножение	x * y	Да
/	Деление	x / y	Да
%	Остаток от деления	x % y	Да
Аддитивные			
+	Сложение	x + y	Да
–	Вычитание	x - y	Да
Сдвига			
<<	Сдвиг влево	x << 1	Да
>>	Сдвиг вправо	x >> 1	Да
Отношения			
<	Меньше	x < y	Да
>	Больше	x > y	Да
<=	Меньше или равно	x <= y	Да
>=	Больше или равно	x >= y	Да
is	Принадлежность к типу или его подклассу	x is y	Нет
as	Преобразование типа	x as y	Нет

Символ операции	Название операции	Пример	Возможность перегрузки
Эквивалентности			
==	Равно	x == y	Да
!=	Не равно	x != y	Да
Логическое И			
&	И	x & y	Да
Логическое исключающее ИЛИ			
^	Исключающее ИЛИ	x ^ y	Да
Логическое ИЛИ			
|	ИЛИ	x | y	Да
Условное И			
&&	И	x && y	Через &
Условное ИЛИ			
||	ИЛИ	x || y	Через |
Условная (тернарная)			
?:	Условная	isTrue ? thenThis : elseThis	Нет
Обработка значений `null`			
??	Объединение с `null`	x??y	Нет
?.	`null`-условная	x?.y	Нет
Присваивания и лямбда (самый низкий приоритет)			
=	Присваивание	x = y	Нет
*=	Умножение с присваиванием	x *= 2	Через *
/=	Деление с присваиванием	x /= 2	Через /
+=	Сложение с присваиванием	x += 2	Через +
-=	Вычитание с присваиванием	x -= 2	Через –
<<=	Сдвиг влево с присваиванием	x <<= 2	Через <<

Символ операции	Название операции	Пример	Возможность перегрузки
`>>=`	Сдвиг вправо с присваиванием	`x >>= 2`	Через `>>`
`&=`	Операция И с присваиванием	`x &= 2`	Через `&`
`^=`	Операция исключающего ИЛИ с присваиванием	`x ^= 2`	Через `^`
`|=`	Операция ИЛИ с присваиванием	`x |= 2`	Через `|`
`=>`	Лямбда-операция	`x => x + 1`	Нет

Операции для работы со значениями null

В языке C# определены две операции, предназначенные для упрощения работы со значениями `null`: *операция объединения с* `null` (null coalescing) и *`null`-условная операция* (null-conditional).

Операция объединения с `null`

Операция объединения с `null` обозначается как `??`. Она говорит о следующем: если операнд не равен `null`, тогда возвратить его значение, иначе возвратить стандартное значение. Например:

```
string s1 = null;
string s2 = s1 ?? "пусто"; // s2 получает значение "пусто"
```

Если левостороннее выражение не равно `null`, то правостороннее выражение никогда не вычисляется. Операция объединения с `null` также работает с типами, допускающими `null` (см. раздел "Типы, допускающие значение `null`" на стр. 150).

`null`-условная операция

`null`-условная операция (или *элвис-операция*) обозначается как `?.` и появилась в версии C# 6. Она позволяет вызывать методы и получать доступ к членам подобно стандартной операции точки, но с той разницей, что если находящийся слева операнд равен `null`, то результатом выражения будет `null`, а не генерация исключения `NullReferenceException`:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString();  // Ошибка не возникает;
                            // s равно null
```

Последняя строка кода эквивалентна такой:

```
string s = (sb == null ? null : sb.ToString());
```

Столкнувшись со значением `null`, элвис-операция сокращает вычисление остатка выражения. В следующем примере переменная `s` получает значение `null`, несмотря на наличие стандартной операции точки между `ToString()` и `ToUpper()`:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString().ToUpper // Ошибка
                                   // не возникает
```

Многократное использование элвис-операции необходимо, только если находящийся непосредственно слева операнд может быть равен `null`. Приведенное ниже выражение надежно работает в ситуациях, когда `x` и `x.y` равны `null`:

```
x?.y?.z
```

Оно эквивалентно следующему выражению (за исключением того, что `x.y` оценивается только раз):

```
x == null ? null
          : (x.y == null ? null : x.y.z)
```

Окончательное выражение должно быть способным принимать значение `null`. Показанный далее код не является допустимым, т.к. тип `int` не может принимать `null`:

```
System.Text.StringBuilder sb = null;
int length = sb?.ToString().Length; // Не допускается
```

Исправить положение можно за счет применения типа значения, допускающего `null` (см. раздел "Типы, допускающие значение `null`" на стр. 150):

```
int? length = sb?.ToString().Length;  // Допустимо;
                        // int? может принимать null
```

`null`-условную операцию можно также использовать для вызова метода `void`:

```
someObject?.SomeVoidMethod();
```

Если переменная someObject равна null, то вместо генерации исключения NullReferenceException этот вызов превращается в "отсутствие операции".

null-условная операция может применяться с часто используемыми членами типов, которые описаны в разделе "Классы" на стр. 72, в том числе с *методами*, *полями*, *свойствами* и *индексаторами*. Она также хорошо сочетается с операцией объединения с null:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString() ?? "пусто"; // s получает
                                      // значение "пусто"
```

Последняя строка эквивалентна следующей:

```
string s = (sb == null ? "пусто" : sb.ToString());
```

Операторы

Функции состоят из операторов, которые выполняются последовательно в порядке их появления внутри программы. *Блок операторов* — это последовательность операторов, находящихся между фигурными скобками ({}).

Операторы объявления

Оператор объявления объявляет новую переменную с возможностью ее дополнительной инициализации посредством выражения. Оператор объявления завершается точкой с запятой. Можно объявлять несколько переменных одного и того же типа, указывая их в списке с запятой в качестве разделителя. Например:

```
bool rich = true, famous = false;
```

Объявление константы похоже на объявление переменной за исключением того, что после объявления константа не может быть изменена и объявление обязательно должно сопровождаться инициализацией (см. раздел "Константы" на стр. 83):

```
const double c = 2.99792458E08;
```

Область видимости локальной переменной

Областью видимости локальной переменной или локальной константы является текущий блок. Объявлять еще одну локаль-

ную переменную с тем же самым именем в текущем блоке или в любых вложенных блоках не разрешено.

Операторы выражений

Операторы выражений — это выражения, которые также представляют собой допустимые операторы. На практике такие выражения что-то "делают"; другими словами, выражения:

- выполняют присваивание или модификацию переменной;
- создают экземпляр объекта;
- вызывают метод.

Выражения, которые не делают ничего из перечисленного выше, не являются допустимыми операторами:

```
string s = "foo";
s.Length;  // Недопустимый оператор: он ничего не делает!
```

При вызове конструктора или метода, который возвращает значение, вы не обязаны использовать результат. Тем не менее, если только данный конструктор или метод не изменяет состояние, то такой оператор бесполезен:

```
new StringBuilder(); // Допустим, но бесполезен
x.Equals (y);        // Допустим, но бесполезен
```

Операторы выбора

Операторы выбора предназначены для условного управления потоком выполнения программы.

Оператор `if`

Оператор `if` выполняет некоторый оператор, если результатом вычисления выражения `bool` оказывается `true`. Например:

```
if (5 < 2 * 3)
  Console.WriteLine ("true");  // true
```

Оператором может быть блок кода:

```
if (5 < 2 * 3)
{
  Console.WriteLine ("true");  // true
  Console.WriteLine ("...")
}
```