# C# и .NET | Типы данных

Как и во многих языках программирования, в C# есть своя система типов данных, которая используется для создания переменных.   
  
Тип данных определяет внутреннее представление данных, множество значений, которые может принимать объект, а также допустимые действия, которые можно применять над объектом.

В языке C# есть следующие базовые типы данных:

bool: хранит значение true или false (логические литералы). Представлен системным типом System.Boolean

bool alive = true;  
  
bool isDead = false;

byte: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом   
  
System.Byte

byte bit1 = 1;  
  
byte bit2 = 102;

sbyte: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом   
  
System.SByte

sbyte bit1 = -101;  
  
sbyte bit2 = 102;

short: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом   
  
System.Int16

short n1 = 1;  
  
short n2 = 102;

ushort: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом   
  
System.UInt16

ushort n1 = 1;  
  
ushort n2 = 102;

int: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом   
  
System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:

int a = 10;  
  
int b = 0b101; // бинарная форма b =5  
  
int c = 0xFF; // шестнадцатеричная форма c = 255

uint: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом   
  
System.UInt32

uint a = 10;  
  
uint b = 0b101;  
  
uint c = 0xFF;

long: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом   
  
System.Int64

long a = -10;  
  
long b = 0b101;  
  
long c = 0xFF;

ulong: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом   
  
System.UInt64

ulong a = 10;  
  
ulong b = 0b101;  
  
ulong c = 0xFF;

float: хранит число с плавающей точкой от -3.4\*1038 до 3.4\*1038 и   
  
занимает 4 байта. Представлен системным типом System.Single

double: хранит число с плавающей точкой от ±5.0\*10-324 до ±1.7\*10308 и   
  
занимает 8 байта. Представлен системным типом System.Double

decimal: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от ±1.0\*10-28 до ±7.9228\*1028,   
  
может хранить 28 знаков после запятой и занимает 16 байт. Представлен системным типом System.Decimal

char: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом   
  
System.Char. Этому типу соответствуют символьные литералы:

char a = 'A';  
  
char b = '\x5A';  
  
char c = '\u0420';

string: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.

string hello = "Hello";  
  
string word = "world";

object: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом   
  
System.Object, который является базовым для всех других типов и классов .NET.

object a = 22;  
  
object b = 3.14;  
  
object c = "hello code";

Например, определим несколько переменных разных типов и выведем их значения на консоль:

string name = "Tom";  
  
int age = 33;  
  
bool isEmployed = false;  
  
double weight = 78.65;  
  
  
  
Console.WriteLine($"Имя: {name}");  
  
Console.WriteLine($"Возраст: {age}");  
  
Console.WriteLine($"Вес: {weight}");  
  
Console.WriteLine($"Работает: {isEmployed}");

Для вывода данных на консоль здесь применяется интерполяция: перед строкой ставится знак $ и после этого мы можем вводить в строку в фигурных скобках значения переменных.   
  
Консольный вывод программы:

Имя: Tom  
  
Возраст: 33  
  
Вес: 78,65  
  
Работает: False

## Использование суффиксов

При присвоении значений надо иметь в виду следующую тонкость: все вещественные литералы (дробные числа) рассматриваются как значения типа   
  
double. И чтобы указать, что дробное число представляет тип float или тип decimal,   
  
необходимо к литералу добавлять суффикс: F/f - для float и M/m - для decimal.

float a = 3.14F;  
  
float b = 30.6f;  
  
  
  
decimal c = 1005.8M;  
  
decimal d = 334.8m;

Подобным образом все целочисленные литералы рассматриваются как значения типа int. Чтобы явным образом указать, что целочисленный литерал представляет значение   
  
типа uint, надо использовать суффикс U/u, для типа long - суффикс L/l, а для типа   
  
ulong - суффикс UL/ul:

uint a = 10U;  
  
long b = 20L;  
  
ulong c = 30UL;

## Использование системных типов

Выше при перечислении всех базовых типов данных для каждого упоминался системный тип. Потому что название встроенного типа по сути представляет собой сокращенное обозначение системного типа. Например, следующие переменные будут эквивалентны по типу:

int a = 4;  
  
System.Int32 b = 4;

## Неявная типизация

Ранее мы явным образом указывали тип переменных, например, int x;. И компилятор при запуске уже знал, что x хранит целочисленное значение.

Однако мы можем использовать и модель неявной типизации:

var hello = "Hell to World";  
  
var c = 20;

Для неявной типизации вместо названия типа данных используется ключевое слово var. Затем уже при компиляции компилятор сам выводит   
  
тип данных исходя из присвоенного значения. Так как по умолчанию все целочисленные значения рассматриваются как значения типа   
  
int, то поэтому в итоге переменная c будет иметь тип int. Аналогично переменной hello присваивается строка, поэтому эта переменная будет иметь тип string

Эти переменные подобны обычным, однако они имеют некоторые ограничения.

Во-первых, мы не можем сначала объявить неявно типизируемую переменную, а затем инициализировать:

// этот код работает  
  
int a;  
  
a = 20;  
  
  
  
// этот код не работает  
  
var c;  
  
c= 20;

Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной null:

// этот код не работает  
  
var c=null;

Так как значение null, то компилятор не сможет вывести тип данных.

НазадСодержаниеВперед

Contacts: metanit22@mail.ru

Copyright © Евгений Попов, metanit.com, 2024. Все права защищены.

# C# и .NET | Преобразования базовых типов данных

При рассмотрении типов данных указывалось, какие значения может иметь тот или иной тип и сколько байт памяти он может занимать. В прошлой теме   
  
были расмотрены арифметические операции. Теперь применим операцию сложения к данным разных типов:

byte a = 4;  
  
int b = a + 70;

Результатом операции вполне справедливо является число 74, как и ожидается.

Но теперь попробуем применить сложение к двум объектам типа byte:

byte a = 4;  
  
byte b = a + 70; // ошибка

Здесь поменялся только тип переменной, которая получает результат сложения - с int на byte. Однако при попытке скомпилировать программу мы получим ошибку на этапе компиляции.   
  
И если мы работаем в Visual Studio, среда подчеркнет вторую строку красной волнистой линией, указывая, что в ней ошибка.

При операциях мы должны учитывать диапазон значений, которые может хранить тот или иной тип. Но в данном случае число 74, которое мы ожидаем получить, вполне укладывается   
  
в диапазон значений типа byte, тем не менее мы получаем ошибку.

Дело в том, что операция сложения (да и вычитания) возвращает значение типа int, если в операции участвуют целочисленные типы данных с разрядностью меньше   
  
или равно int (то есть типы byte, short, int). Поэтому результатом операции a + 70 будет объект, который имеет длину в памяти 4 байта.   
  
Затем этот объект мы пытаемся присвоить переменной b, которая имеет тип byte и в памяти занимает 1 байт.

И чтобы выйти из этой ситуации, необходимо применить операцию преобразования типов. Операция преобразования типов предполагает указание в скобках того типа, к которому надо преобразовать значение:

(тип\_данных\_в\_который\_надо\_преобразовать)значение\_для\_преобразования;

Так, изменим предыдущий пример, применив операцию преобразования типов:

byte a = 4;  
  
byte b = (byte)(a + 70);

## Сужающие и расширяющие преобразования

Преобразования могут быть сужающие (narrowing) и расширяющие (widening). Расширяющие преобразования расширяют размер объекта в памяти. Например:

byte a = 4; // 0000100  
  
ushort b = a; // 000000000000100

В данном случае переменной типа ushort присваивается значение типа byte. Тип byte занимает 1 байт (8 бит),   
  
и значение переменной a в двоичном виде можно представить как:

00000100

Значение типа ushort занимает 2 байта (16 бит). И при присвоении переменной b значение переменной a расширяется до 2 байт

0000000000000100

То есть значение, которое занимает 8 бит, расширяется до 16 бит.

Сужающие преобразования, наоборот, сужают значение до типа меньшей разядности. Во втором листинге статьи мы как раз имели дело с сужающими преобразованиями:

ushort a = 4;  
  
byte b = (byte) a;

Здесь переменной b, которая занимает 8 бит, присваивается значение ushort, которое занимает 16 бит. То есть из 0000000000000100   
  
получаем 00000100. Таким образом, значение сужается с 16 бит (2 байт) до 8 бит (1 байт).

## Явные и неявные преобразования

В случае с расширяющими преобразованиями компилятор за нас выполнял все преобразования данных, то есть преобразования были неявными   
  
(implicit conversion). Такие преобразования не вызывают каких-то затруднений. Тем не менее стоит сказать пару слов об общей механике подобных преобразований.

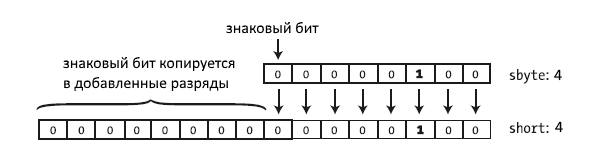
Если производится преобразование от безнакового типа меньшей разрядности к безнаковому типу большой разрядности,   
  
то добавляются дополнительные биты, которые имеют значени 0. Это называется дополнение нулями или zero extension.

byte a = 4; // 0000100  
  
ushort b = a; // 000000000000100

Если производится преобразование к знаковому типу, то битовое представление дополняется нулями, если число положительное,   
  
и единицами, если число отрицательное. Последний разряд числа содержит знаковый бит - 0 для положительных и 1 для   
  
отрицательных чисел. При расширении в добавленные разряды компируется знаковый бит.

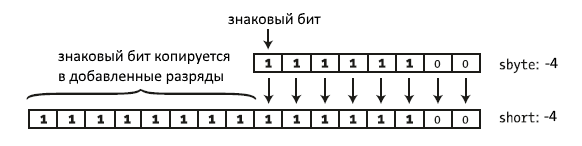
Рассмотрим преобразование положительного числа:

sbyte a = 4; // 0000100  
  
short b = a; // 000000000000100



Преобразование отрицательного числа:

sbyte a = -4; // 1111100  
  
short b = a; // 111111111111100



При явных преобразованиях (explicit conversion) мы сами должны применить операцию преобразования (операция ()).   
  
Суть операции преобразования типов состоит в том, что перед значением указывается в скобках тип, к которому надо привести данное значение:

int a = 4;  
  
int b = 6;  
  
byte c = (byte)(a+b);

Расширяющие преобразования от типа с меньшей разрядностью к типу с большей разрядностью компилятор проводит неявно. Это могут быть следующие цепочки преобразований:

byte -> short -> int -> long -> decimal

int -> double

short -> float -> double

char -> int

Все безопасные автоматические преобразования можно описать следующей таблицей:

Тип

В какие типы безопасно преобразуется

byte

short, ushort, int, uint, long, ulong, float, double, decimal

sbyte

short, int, long, float, double, decimal

short

int, long, float, double, decimal

ushort

int, uint, long, ulong, float, double, decimal

int

long, float, double, decimal

uint

long, ulong, float, double, decimal

long

float, double, decimal

ulong

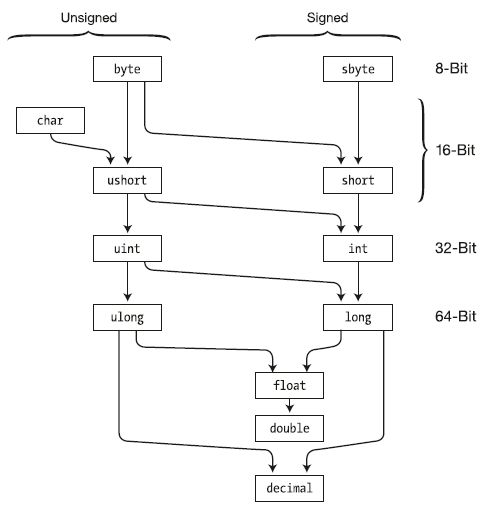
float, double, decimal

float

double

char

ushort, int, uint, long, ulong, float, double, decimal



В остальных случаях следует использовать явные преобразования типов.

Также следует отметить, что несмотря на то, что и double, и decimal могут хранить дробные данные, а decimal имеет большую разрядность, чем double, но все равно   
  
значение double нужно явно приводить к типу decimal:

double a = 4.0;  
  
decimal b = (decimal)a;

## Потеря точности данных

Рассмотрим другую ситуацию, что будет, например, в следующем случае:

int a = 33;  
  
int b = 600;  
  
byte c = (byte)(a+b);  
  
Console.WriteLine(c); // 121

Результатом будет число 121, так число 633 не попадает в допустимый диапазон для типа byte, и старшие биты будут усекаться. В итоге получится число 121.   
  
Поэтому при преобразованиях надо это учитывать. И мы в данном случае можем либо взять такие числа a и b, которые в сумме дадут число не больше 255,   
  
либо мы можем выбрать вместо byte другой тип данных, например, int.

НазадСодержаниеВперед

Contacts: metanit22@mail.ru

Copyright © Евгений Попов, metanit.com, 2024. Все права защищены.

# C# и .NET | Арифметические операции

В C# используется большинство операций, которые применяются и в других языках программирования. Операции представляют определенные действия над операндами -   
  
участниками операции. В качестве операнда может выступать переменной или какое-либо значение (например, число).   
  
Операции бывают унарными (выполняются над одним операндом), бинарными - над двумя операндами и тернарными - выполняются над тремя операндами. Рассмотрим все виды операций.

Бинарные арифметические операции:

+

Операция сложения двух чисел:

int x = 10;  
  
int z = x + 12; // 22

-

Операция вычитания двух чисел:

int x = 10;  
  
int z = x - 6; // 4

\*

Операция умножения двух чисел:

int x = 10;  
  
int z = x \* 5; // 50

/

операция деления двух чисел:

int x = 10;  
  
int z = x / 5; // 2  
  
  
  
double a = 10;  
  
double b = 3;  
  
double c = a / b; // 3.33333333

При делении стоит учитывать, что если оба операнда представляют целые числа, то результат также будет округляться до целого числа:

double z = 10 / 4; //результат равен 2

Несмотря на то, что результат операции в итоге помещается в переменную типа double, которая позволяет сохранить дробную часть, но в самой операции участвуют два литерала,   
  
которые по умолчанию рассматриваются как объекты int, то есть целые числа, и результат то же будет целочисленный.

Для выхода из этой ситуации необходимо определять литералы или переменные, участвующие в операции, именно как типы double или float:

double z = 10.0 / 4.0; //результат равен 2.5

%

Операция получение остатка от целочисленного деления двух чисел:

double x = 10.0;  
  
double z = x % 4.0; //результат равен 2

Также есть ряд унарных операций, в которых принимает участие один операнд:

++

Операция инкремента

Инкремент бывает префиксным: ++x - сначала значение переменной x увеличивается на 1,   
  
а потом ее значение возвращается в качестве результата операции.

И также существует постфиксный инкремент: x++ - сначала значение переменной x возвращается в качестве результата операции,   
  
а затем к нему прибавляется 1.

int x1 = 5;  
  
int z1 = ++x1; // z1=6; x1=6  
  
Console.WriteLine($"{x1} - {z1}");  
  
  
  
int x2 = 5;  
  
int z2 = x2++; // z2=5; x2=6  
  
Console.WriteLine($"{x2} - {z2}");

--

Операция декремента или уменьшения значения на единицу. Также существует префиксная форма декремента (--x) и   
  
постфиксная (x--).

int x1 = 5;  
  
int z1 = --x1; // z1=4; x1=4  
  
Console.WriteLine($"{x1} - {z1}");  
  
  
  
int x2 = 5;  
  
int z2 = x2--; // z2=5; x2=4  
  
Console.WriteLine($"{x2} - {z2}");

При выполнении сразу нескольких арифметических операций следует учитывать порядок их выполнения. Приоритет операций от наивысшего к низшему:

Инкремент, декремент

Умножение, деление, получение остатка

Сложение, вычитание

Для изменения порядка следования операций применяются скобки.

Рассмотрим набор операций:

int a = 3;  
  
int b = 5;  
  
int c = 40;  
  
int d = c---b\*a; // a=3 b=5 c=39 d=25  
  
Console.WriteLine($"a={a} b={b} c={c} d={d}");

Здесь мы имеем дело с тремя операциями: декремент, вычитание и умножение. Сначала выполняется декремент переменной c, затем умножение b\*a, и в конце вычитание. То есть фактически набор операций выглядел так:

int d = (c--)-(b\*a);

Но с помощью скобок мы могли бы изменить порядок операций, например, следующим образом:

int a = 3;  
  
int b = 5;  
  
int c = 40;  
  
int d = (c-(--b))\*a; // a=3 b=4 c=40 d=108  
  
Console.WriteLine($"a={a} b={b} c={c} d={d}");

## Ассоциативность операторов

Как выше было отмечено, операции умножения и деления имеют один и тот же приоритет, но какой тогда результат будет в выражении:

int x = 10 / 5 \* 2;

Стоит нам трактовать это выражение как (10 / 5) \* 2 или как 10 / (5 \* 2)? Ведь в зависимости от трактовки   
  
мы получим разные результаты.

Когда операции имеют один и тот же приоритет, порядок вычисления определяется ассоциативностью операторов. В зависимости от ассоциативности   
  
есть два типа операторов:

Левоассоциативные операторы, которые выполняются слева направо

Правоассоциативные операторы, которые выполняются справа налево

Все арифметические операторы являются левоассоциативными, то есть выполняются слева направо. Поэтому выражение 10 / 5 \* 2   
  
необходимо трактовать как (10 / 5) \* 2, то есть результатом будет 4.

НазадСодержаниеВперед

Contacts: metanit22@mail.ru

Copyright © Евгений Попов, metanit.com, 2024. Все права защищены.

# C# и .NET | Массивы

Массив представляет набор однотипных данных. Объявление массива похоже на объявление переменной за тем исключением, что после указания типа ставятся квадратные скобки:

тип\_переменной[] название\_массива;

Например, определим массив целых чисел:

int[] numbers;

После определения переменной массива мы можем присвоить ей определенное значение:

int[] nums = new int[4];

Здесь вначале мы объявили массив nums, который будет хранить данные типа int. Далее используя операцию new,   
  
мы выделили память для 4 элементов массива: new int[4]. Число 4 еще называется длиной массива.   
  
При таком определении все элементы получают значение по умолчанию, которое предусмотренно для их типа. Для типа int значение по умолчанию - 0.

Также мы сразу можем указать значения для этих элементов:

int[] nums2 = new int[4] { 1, 2, 3, 5 };  
  
  
  
int[] nums3 = new int[] { 1, 2, 3, 5 };  
  
  
  
int[] nums4 = new[] { 1, 2, 3, 5 };  
  
  
  
int[] nums5 = { 1, 2, 3, 5 };

Все перечисленные выше способы будут равноценны.

Подобным образом можно определять массивы и других типов, например, массив значений типа string:

string[] people = { "Tom", "Sam", "Bob" };

Начиная с версии C# 12 для определения массивов можно использовать выражения коллекций, которые предполагают   
  
заключение элементов массива в квадратные скобки:

int[] nums1 = [ 1, 2, 3, 5 ];  
  
int[] nums2 = []; // пустой массив

## Индексы и получение элементов массива

Для обращения к элементам массива используются индексы. Индекс представляет номер элемента в массиве, при этом нумерация начинается с   
  
нуля, поэтому индекс первого элемента будет равен 0, индекс четвертого элемента - 3.

Используя индексы, мы можем получить элементы массива:

int[] numbers = { 1, 2, 3, 5 };  
  
  
  
// получение элемента массива  
  
Console.WriteLine(numbers[3]); // 5  
  
  
  
// получение элемента массива в переменную  
  
var n = numbers[1]; // 2  
  
Console.WriteLine(n); // 2

Также мы можем изменить элемент массива по индексу:

int[] numbers = { 1, 2, 3, 5 };  
  
  
  
// изменим второй элемент массива  
  
numbers[1] = 505;  
  
  
  
Console.WriteLine(numbers[1]); // 505

И так как у нас массив определен только для 4 элементов, то мы не можем обратиться, например, к шестому элементу.   
  
Если мы так попытаемся сделать, то мы получим ошибку во время выполнения:

int[] numbers = { 1, 2, 3, 5 };  
  
  
  
Console.WriteLine(numbers[6]); // ! Ошибка - в массиве только 4 элемента

## Свойство Length и длина массива

каждый массив имеет свойство Length, которое хранит длину массива. Например, получим длину выше созданного массива numbers:

int[] numbers = { 1, 2, 3, 5 };  
  
  
  
Console.WriteLine(numbers.Length); // 4

Для получения длины массива после названия массива через точку указывается свойство Length: numbers.Length.

## Получение элементов с конца массива

Благодаря наличию свойства Length, мы можем вычислить индекс последнего элемента массива - это длина массива - 1. Например, если длина массива - 4 (то есть   
  
массив имеет 4 элемента), то индекс последнего элемента будет равен 3. И, используя свойство Length, мы можем легко получить элементы с конца массива:

int[] numbers = { 1, 2, 3, 5};  
  
  
  
Console.WriteLine(numbers[numbers.Length - 1]); // 5 - первый с конца или последний элемент  
  
Console.WriteLine(numbers[numbers.Length - 2]); // 3 - второй с конца или предпоследний элемент  
  
Console.WriteLine(numbers[numbers.Length - 3]); // 2 - третий элемент с конца

Однако при подобном подходе выражения типа numbers.Length - 1, смысл которых состоит в том, чтобы получить какой-то определенный элемент с конца массива,   
  
утяжеляют код. И, начиная, с версии C# 8.0 в язык был добавлен специальный оператор ^,   
  
с помощью которого можно задать индекс относительно конца коллекции.

Перепишем предыдущий пример, применяя оператор ^:

int[] numbers = { 1, 2, 3, 5};  
  
  
  
Console.WriteLine(numbers[^1]); // 5 - первый с конца или последний элемент  
  
Console.WriteLine(numbers[^2]); // 3 - второй с конца или предпоследний элемент  
  
Console.WriteLine(numbers[^3]); // 2 - третий элемент с конца

## Перебор массивов

Для перебора массивов мы можем использовать различные типы циклов. Например, цикл foreach:

int[] numbers = { 1, 2, 3, 4, 5 };  
  
foreach (int i in numbers)  
  
{  
  
 Console.WriteLine(i);  
  
}

Здесь в качестве контейнера выступает массив данных типа int. Поэтому мы объявляем переменную с типом int

Подобные действия мы можем сделать и с помощью цикл for:

int[] numbers = { 1, 2, 3, 4, 5 };  
  
for (int i = 0; i < numbers.Length; i++)  
  
{  
  
 Console.WriteLine(numbers[i]);  
  
}

В то же время цикл for более гибкий по сравнению с foreach. Если foreach последовательно извлекает элементы контейнера и только для чтения,   
  
то в цикле for мы можем перескакивать на несколько элементов вперед в зависимости от приращения счетчика, а также можем изменять элементы:

int[] numbers = { 1, 2, 3, 4, 5 };  
  
for (int i = 0; i < numbers.Length; i++)  
  
{  
  
 numbers[i] = numbers[i] \* 2;  
  
 Console.WriteLine(numbers[i]);  
  
}

Также можно использовать и другие виды циклов, например, while:

int[] numbers = { 1, 2, 3, 4, 5 };  
  
int i = 0;  
  
while(i < numbers.Length)  
  
{  
  
 Console.WriteLine(numbers[i]);  
  
 i++;  
  
}

## Многомерные массивы

Массивы характеризуются таким понятием как ранг или количество измерений. Выше мы рассматривали   
  
массивы, которые имеют одно измерение (то есть их ранг равен 1) - такие массивы можно представлять в виде ряда (строки или столбца) элемента.   
  
Но массивы также бывают многомерными. У таких массивов количество измерений (то есть ранг) больше 1.

Массивы которые имеют два измерения (ранг равен 2) называют двухмерными. Например, создадим одномерный и двухмерный массивы, которые имеют одинаковые элементы:

int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };  
  
  
  
int[,] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };

Визуально оба массива можно представить следующим образом:

0

1

2

3

4

5

0

1

2

3

4

5

Поскольку массив nums2 двухмерный, он представляет собой простую таблицу. Все возможные способы определения двухмерных массивов:

int[,] nums1;  
  
int[,] nums2 = new int[2, 3];  
  
int[,] nums3 = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };  
  
int[,] nums4 = new int[,] { { 0, 1, 2 }, { 3, 4, 5 } };  
  
int[,] nums5 = new [,]{ { 0, 1, 2 }, { 3, 4, 5 } };  
  
int[,] nums6 = { { 0, 1, 2 }, { 3, 4, 5 } };

Массивы могут иметь и большее количество измерений. Объявление трехмерного массива могло бы выглядеть так:

int[,,] nums3 = new int[2, 3, 4];

Соответственно могут быть и четырехмерные массивы и массивы с большим количеством измерений. Но на практике   
  
обычно используются одномерные и двухмерные массивы.

Определенную сложность может представлять перебор многомерного массива. Прежде всего надо учитывать, что длина такого массива - это совокупное количество элементов.

int[,] numbers = { { 1, 2, 3 }, { 4, 5, 6 }};  
  
foreach (int i in numbers)  
  
 Console.Write($"{i} ");

В данном случае длина массива numbers равна 6. И цикл foreach выводит все элементы массива в строку:

1 2 3 4 5 6

Но что если мы хотим отдельно пробежаться по каждой строке в таблице? В этом случае надо получить количество элементов в размерности.   
  
В частности, у каждого массива есть метод GetUpperBound(номер\_размерности), который возвращает индекс последнего   
  
элемента в определенной размерности. И если мы говорим непосредственно о двухмерном массиве, то первая размерность (с индексом 0)   
  
по сути это и есть таблица. И с помощью выражения

numbers.GetUpperBound(0) + 1

можно получить количество строк таблицы, представленной двухмерным массивом. А через

numbers.Length / количество\_строк

можно получить количество элементов в каждой строке:

int[,] numbers = { { 1, 2, 3 }, { 4, 5, 6 }};  
  
  
  
int rows = numbers.GetUpperBound(0) + 1; // количество строк  
  
int columns = numbers.Length / rows; // количество столбцов  
  
// или так  
  
// int columns = numbers.GetUpperBound(1) + 1;  
  
  
  
for (int i = 0; i < rows; i++)  
  
{  
  
 for (int j = 0; j < columns; j++)  
  
 {  
  
 Console.Write($"{numbers[i, j]} \t");  
  
 }  
  
 Console.WriteLine();  
  
}

1 2 3  
  
4 5 6

## Массив массивов

От многомерных массивов надо отличать массив массивов или так называемый "зубчатый массив":

int[][] nums = new int[3][];  
  
nums[0] = new int[2] { 1, 2 }; // выделяем память для первого подмассива  
  
nums[1] = new int[3] { 1, 2, 3 }; // выделяем память для второго подмассива  
  
nums[2] = new int[5] { 1, 2, 3, 4, 5 }; // выделяем память для третьего подмассива

Здесь две группы квадратных скобок указывают, что это массив массивов, то есть такой массив, который в свою очередь содержит в себе   
  
другие массивы. Причем длина массива указывается только в первых квадратных скобках, все последующие квадратные скобки должны быть пусты:   
  
new int[3][]. В данном случае у нас массив nums содержит три массива. Причем размерность каждого из этих массивов может не совпадать.

Альтернативное определение массива массивов:

int[][] numbers = {   
  
 new int[] { 1, 2 },   
  
 new int[] { 1, 2, 3 },   
  
 new int[] { 1, 2, 3, 4, 5 }   
  
};

1

2

1

2

3

1

2

3

4

5

Используя вложенные циклы, можно перебирать зубчатые массивы. Например:

int[][] numbers = new int[3][];  
  
numbers[0] = new int[] { 1, 2 };  
  
numbers[1] = new int[] { 1, 2, 3 };  
  
numbers[2] = new int[] { 1, 2, 3, 4, 5 };  
  
foreach(int[] row in numbers)  
  
{  
  
 foreach(int number in row)  
  
 {  
  
 Console.Write($"{number} \t");  
  
 }  
  
 Console.WriteLine();  
  
}  
  
  
  
// перебор с помощью цикла for  
  
for (int i = 0; i<numbers.Length;i++)  
  
{  
  
 for (int j =0; j<numbers[i].Length; j++)  
  
 {  
  
 Console.Write($"{numbers[i][j]} \t");  
  
 }  
  
 Console.WriteLine();  
  
}

## Основные понятия массивов

Суммируем основные понятия массивов:

Ранг (rank): количество измерений массива

Длина измерения (dimension length): длина отдельного измерения массива

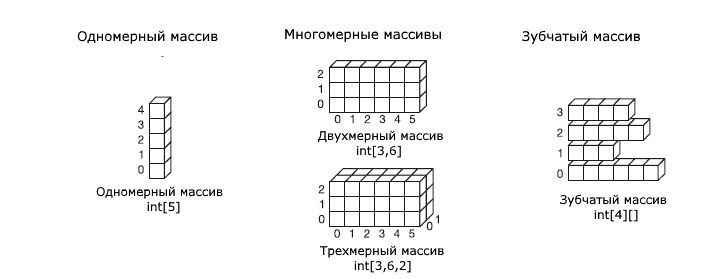
Длина массива (array length): количество всех элементов массива

Например, возьмем массив

int[,] numbers = new int[3, 4];

Массив numbers двухмерный, то есть он имеет два измерения, поэтому его ранг равен 2. Длина первого измерения - 3, длина второго измерения - 4. Длина массива (то есть общее количество элементов) - 12.

Примеры массивов:



НазадСодержаниеВперед

Contacts: metanit22@mail.ru

Copyright © Евгений Попов, metanit.com, 2024. Все права защищены.

# C# и .NET | Конструкция if..else и тернарная операция

Условные конструкции - один из базовых компонентов многих языков программирования, которые направляют работу программы по одному из путей   
  
в зависимости от определенных условий. Одной из таких конструкций в языке программирования C# является конструкция if..else

Конструкция if/else проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код.

Ее простейшая форма состоит из блока if:

if(условие)  
  
{  
  
 выполняемые инструкции  
  
}

После ключевого слова if ставится условие. Условие должно представлять значение типа bool.   
  
Это может быть непосредственно значение типа bool или результат условного выражения или другого выражения, которое возвращает значение типа bool.   
  
И если это условие истинно (равно true), то срабатывает код, который помещен далее после условия внутри фигурных скобок.

Например:

int num1 = 8;  
  
int num2 = 6;  
  
if(num1 > num2)  
  
{  
  
 Console.WriteLine($"Число {num1} больше числа {num2}");  
  
}

В данном случае у нас первое число больше второго, поэтому выражение num1 > num2 истинно и возвращает   
  
true, следовательно, управление переходит к строке Console.WriteLine("Число {num1} больше числа {num2}");

Если блок if содержит одну инструкцию, то мы можем его сократить, убрав фигурные скобки:

int num1 = 8;  
  
int num2 = 6;  
  
if (num1 > num2)  
  
 Console.WriteLine($"Число {num1} больше числа {num2}");  
  
  
  
// или так  
  
if (num1 > num2) Console.WriteLine($"Число {num1} больше числа {num2}");

Также мы можем соединить сразу несколько условий, используя логические операторы:

int num1 = 8;  
  
int num2 = 6;  
  
if(num1 > num2 && num1==8)  
  
{  
  
 Console.WriteLine($"Число {num1} больше числа {num2}");  
  
}

В данном случае блок if будет выполняться, если num1 > num2 равно true и num1==8 равно true.

Но что, если мы захотим, чтобы при несоблюдении условия также выполнялись какие-либо действия? В этом случае мы можем добавить блок else:

int num1 = 8;  
  
int num2 = 6;  
  
if(num1 > num2)  
  
{  
  
 Console.WriteLine($"Число {num1} больше числа {num2}");  
  
}  
  
else  
  
{  
  
 Console.WriteLine($"Число {num1} меньше числа {num2}");  
  
}

Блок else выполняется, если условие после if ложно, то есть равно false. Если блок else содержит толко одну   
  
инструкцию, то оять же мы можем его сократить, убрав фигурные скобки:

int num1 = 8;  
  
int num2 = 6;  
  
if(num1 > num2)  
  
 Console.WriteLine($"Число {num1} больше числа {num2}");  
  
else  
  
 Console.WriteLine($"Число {num1} меньше числа {num2}");

Но в примере выше при сравнении чисел мы можем насчитать три состояния: первое число больше второго, первое число меньше второго и числа равны.   
  
Используя конструкцию else if, мы можем обрабатывать дополнительные условия:

int num1 = 8;  
  
int num2 = 6;  
  
if(num1 > num2)  
  
{  
  
 Console.WriteLine($"Число {num1} больше числа {num2}");  
  
}  
  
else if (num1 < num2)  
  
{  
  
 Console.WriteLine($"Число {num1} меньше числа {num2}");  
  
}  
  
else  
  
{  
  
 Console.WriteLine("Число num1 равно числу num2");  
  
}

При необходимости можно добавить несколько выражений else if:

string name = "Alex";  
  
  
  
if (name == "Tom")  
  
 Console.WriteLine("Вас зовут Tomas");  
  
else if (name == "Bob")  
  
 Console.WriteLine("Вас зовут Robert");  
  
else if (name == "Mike")  
  
 Console.WriteLine("Вас зовут Michael");  
  
else  
  
 Console.WriteLine("Неизвестное имя");

## Тернарная операция

Тернарную операция также позволяет проверить некоторое условие и в зависимости от его истинности выполнить некоторые действия. Она имеет следующий синтаксис:

[первый операнд - условие] ? [второй операнд] : [третий операнд]

Здесь сразу три операнда. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то   
  
возвращается второй операнд; если условие равно false, то третий. Например:

int x=3;  
  
int y=2;  
  
  
  
int z = x < y ? (x+y) : (x-y);  
  
Console.WriteLine(z); // 1

Здесь первый операнд (то есть условие) представляет выражение x < y. Если оно равно true, то возвращается   
  
второй операнд - (x+y), то есть результат операции сложения. Если условие равно false, то возвращается   
  
третий операнд - (x-y).

Результат тернарной операции (то есть второй или третий операнд в зависимости от условия) присваивается переменной z.

НазадСодержаниеВперед

Contacts: metanit22@mail.ru

Copyright © Евгений Попов, metanit.com, 2024. Все права защищены.

# C# и .NET | Циклы

Циклы являются управляющими конструкциями, позволяя в зависимости от определенных условий выполнять некоторое действие множество раз. В   
  
C# имеются следующие виды циклов:

for

foreach

while

do...while

## Цикл for

Цикл for имеет следующее формальное определение:

for ([действия\_до\_выполнения\_цикла]; [условие]; [действия\_после\_выполнения])  
  
{  
  
 // действия  
  
}

Объявление цикла for состоит из трех частей. Первая часть объявления цикла - некоторые действия, которые выполняются один раз   
  
до выполнения цикла. Обычно здесь определяются переменные, которые будут использоваться в цикле.

Вторая часть - условие, при котором будет выполняться цикл. Пока условие равно true, будет выполняться цикл.

И третья часть - некоторые действия, которые выполняются после завершения блока цикла. Эти действия выполняются каждый раз при завершении блока цикла.

После объявления цикла в фигурных скобках помещаются сами действия цикла.

Рассмотрим стандартный цикл for:

for (int i = 1; i < 4; i++)  
  
{  
  
 Console.WriteLine(i);  
  
}

Здесь первая часть объявления цикла - int i = 1 - создает и инициализирует переменную i.

Вторая часть - условие i < 4. То есть пока переменная i меньше 4, будет выполняться цикл.

И третья часть - действия, выполняемые после завершения действий из блока цикла - увеличение переменной i на единицу.

Весь процесс цикла можно представить следующим образом:

Определяется переменная int i = 1

Проверяется условие i < 4. Оно истинно (так как 1 меньше 4), поэтому выполняется блок цикла, а именно инструкция   
  
Console.WriteLine(i), которая выводит на консоль значение переменной i

Блок цикла закончил выполнение, поэтому выполняется треться часть объявления цикла - i++. После этого переменная i будет равна 2.

Снова проверяется условие i < 4. Оно истинно (так как 2 меньше 4), поэтому опять выполняется блок цикла -   
  
Console.WriteLine(i)

Блок цикла закончил выполнение, поэтому снова выполняется выражение i++. После этого переменная i будет равна 3.

Снова проверяется условие i < 4. Оно истинно (так как 3 меньше 4), поэтому опять выполняется блок цикла -   
  
Console.WriteLine(i)

Блок цикла закончил выполнение, поэтому снова выполняется выражение i++. После этого переменная i будет равна 4.

Снова проверяется условие i < 4. Теперь оно возвражает false, так как значение переменной i НЕ меньше 4,   
  
поэтому цикл завершает выполнение. Далее уже выполняется остальная часть программы, которая идет после цикла

В итоге блок цикла сработает 3 раза, пока значение i не станет равным 4. И каждый раз это значение будет увеличиваться на 1. Однократное выполнение блока цикла   
  
называется итерацией. Таким образом, здесь цикл выполнит три итерации. Результат работы программы:

1  
  
2  
  
3

Если блок цикла for содержит одну инструкцию, то мы можем его сократить, убрав фигурные свобки:

for (int i = 1; i < 4; i++)  
  
 Console.WriteLine(i);  
  
  
  
// или так  
  
for (int i = 1; i < 4; i++) Console.WriteLine(i);

При этом необязательно именно в первой части цикла объявлять переменную, а в третий части изменять ее значение - это могут быть любые действия. Например:

var i = 1;  
  
  
  
for (Console.WriteLine("Начало выполнения цикла"); i < 4; Console.WriteLine($"i = {i}"))  
  
{  
  
 i++;  
  
}

Здесь опять же цикл срабатывает, пока переменная i меньше 4, только приращение переменной i происходит в блоке цикла. Консольный вывод данной программы:

Начало выполнения цикла  
  
i = 2  
  
i = 3  
  
i = 4

Нам необязательно указывать все условия при объявлении цикла. Например, мы можем написать так:

int i = 1;  
  
for (; ;)  
  
{  
  
 Console.WriteLine($"i = {i}");  
  
 i++;  
  
}

Формально определение цикла осталось тем же, только теперь блоки в определении у нас пустые: for (; ;). У нас нет   
  
инициализированной переменной, нет условия, поэтому цикл будет работать вечно - бесконечный цикл.

Мы также можем опустить ряд блоков:

int i = 1;  
  
for (; i<4;)  
  
{  
  
 Console.WriteLine($"i = {i}");  
  
 i++;  
  
}

Этот пример по сути эквивалентен первому примеру: у нас также есть переменная-счетчик, только определена она вне цикла. У нас есть условие выполнения цикла.   
  
И есть приращение переменной уже в самом блоке for.

Также стоит отметить, что можно определять несколько переменных в объявлении цикла:

for (int i = 1, j = 1; i < 10; i++, j++)  
  
 Console.WriteLine($"{i \* j}");

Здесь в первой части объявления цикла определяются две переменных: i и j. Цикл выполняется, пока i не будет равна 10. После каждой итерации   
  
переменые i и j увеличиваются на единицу. Консольный вывод программы:

1  
  
4  
  
9  
  
16  
  
25  
  
36  
  
49  
  
64  
  
81

## Цикл do..while

В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while. И пока это условие истинно,   
  
цикл повторяется.

do  
  
{  
  
 действия цикла  
  
}  
  
while (условие)

Например:

int i = 6;  
  
do  
  
{  
  
 Console.WriteLine(i);  
  
 i--;  
  
}  
  
while (i > 0);

Здесь код цикла сработает 6 раз, пока i не станет равным нулю. Но важно отметить, что цикл do гарантирует хотя бы единократное выполнение действий,   
  
даже если условие в инструкции while не будет истинно. То есть мы можем написать:

int i = -1;  
  
do  
  
{  
  
 Console.WriteLine(i);  
  
 i--;  
  
}  
  
while (i > 0);

Хотя у нас переменная i меньше 0, цикл все равно один раз выполнится.

## Цикл while

В отличие от цикла do цикл while сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:

while (условие)  
  
{  
  
 действия цикла  
  
}

Например:

int i = 6;  
  
while (i > 0)  
  
{  
  
 Console.WriteLine(i);  
  
 i--;  
  
}

## Цикл foreach

Цикл foreach предназначен для перебора набора или коллекции элементов. Его общее определение:

foreach(тип\_данных переменная in коллекция)  
  
{  
  
 // действия цикла  
  
}

После оператора foreach в скобках сначала идет определение переменной. Затем ключевое слово in и далее коллекция,   
  
элементы которой надо перебрать.

При выполнении цикл последовательно перебирает элементы коллекции и помещает их в переменную, и таким образом в блоке цикла мы можем выполнить с ними некоторые действия.

Например, возьмем строку. Строка по сути - это коллекция символов. И .NET позволяет перебрать все элементы строки - ее символы с помощью цикла foreach.

foreach(char c in "Tom")  
  
{  
  
 Console.WriteLine(c);  
  
}

Здесь цикл foreach пробегается по всем символам строки "Tom" и каждый символ помещает в символьную переменную c. В блоке цикла   
  
значение переменной c выводится на консоль. Поскольку в строке "Tom" три символа, то цикл выполнится три раза. Консольный вывод программы:

T  
  
o  
  
m

Стоит отметить, что переменная, которая определяется в объявлении цикла, должна по типу соответствовать типу элементов перебираемой коллекции. Так, элементы строки - значения типа   
  
char - символы. Поэтому переменная c имеет тип char. Однако в реальности не всегда бывает очевидно, какой тип представляют   
  
элементы коллекции. В этом случае мы можем определить переменную с помощью оператора var:

foreach(var c in "Tom")  
  
{  
  
 Console.WriteLine(c);  
  
}

В дальнейшем мы подробнее рассмотрим, что представляют собой коллекции в .NET и какие именно коллекции можно перебирать с помощью цикла foreach.

## Операторы continue и break

Иногда возникает ситуация, когда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором   
  
break.

Например:

for (int i = 0; i < 9; i++)  
  
{  
  
 if (i == 5)  
  
 break;  
  
 Console.WriteLine(i);  
  
}

Хотя в условии цикла сказано, что цикл будет выполняться, пока счетчик i не достигнет значения 9, в реальности цикл сработает 5 раз.   
  
Так как при достижении счетчиком i значения 5, сработает оператор break, и цикл завершится.

0  
  
1  
  
2  
  
3  
  
4

Теперь поставим себе другую задачу. А что если мы хотим, чтобы при проверке цикл не завершался, а просто пропускал текущую итерацию.   
  
Для этого мы можем воспользоваться оператором continue:

for (int i = 0; i < 9; i++)  
  
{  
  
 if (i == 5)  
  
 continue;  
  
 Console.WriteLine(i);  
  
}

В этом случае цикл, когда дойдет до числа 5, которое не удовлетворяет условию проверки, просто пропустит это число и перейдет к следующей итерации:

0  
  
1  
  
2  
  
3  
  
4  
  
6  
  
7  
  
8

Стоит отметить, что операторы break и continue можно применять в любом типе циклов.

## Вложенные циклы

Одни циклы могут быть вложенными в другие. Например:

for (int i = 1; i < 10; i++)  
  
{  
  
 for (int j = 1; j < 10; j++)  
  
 {  
  
 Console.Write($"{i \* j} \t");  
  
 }  
  
 Console.WriteLine();  
  
}

В данном случае цикл for (int i = 1; i < 10; i++) выполняется 9 раз, то есть имеет 9 итераций. Но в рамках каждой итерации   
  
выполняется девять раз вложенный цикл for (int j = 1; j < 10; j++). В итоге данная программа выведет таблицу умножения.

НазадСодержаниеВперед

Contacts: metanit22@mail.ru

Copyright © Евгений Попов, metanit.com, 2024. Все права защищены.

# C# и .NET | Задачи с массивами

Познакомившись с циклами, переменными, условными конструкциями и массивами, рассмотрим несколько задач для работы с массивами.

## Количество положительных чисел

Найдем количество положительных чисел в массиве:

int[] numbers = { -4, -3, -2, -1, 0, 1, 2, 3, 4 };  
  
int result = 0;  
  
foreach(int number in numbers)  
  
{  
  
 if(number > 0)  
  
 {  
  
 result++;  
  
 }  
  
}  
  
Console.WriteLine($"Число элементов больше нуля: {result}");

Здесь создаем вспомогательную переменную result, которая будет содержать количество положительных чисел. В цикле прохожим по массиву и, если его элемент больше нуля,   
  
добавляем к переменной result единицу.

## Инверсия массива

Вторая задача - инверсия массива, то есть переворот его в обратном порядке:

int[] numbers = { -4, -3, -2, -1,0, 1, 2, 3, 4 };  
  
   
  
int n = numbers.Length; // длина массива  
  
int k = n / 2; // середина массива  
  
int temp; // вспомогательный элемент для обмена значениями  
  
for(int i=0; i < k; i++)  
  
{  
  
 temp = numbers[i];  
  
 numbers[i] = numbers[n - i - 1];  
  
 numbers[n - i - 1] = temp;  
  
}  
  
foreach(int i in numbers)  
  
{  
  
 Console.Write($"{i} \t");  
  
}

Поскольку нам надо изменять элементы массива, то для этого используется цикл for. Алгоритм решения задачи подразумевает перебор элементов до середины массива,   
  
которая в программе представлена переменной k, и обмен значений элемента, который имеет индекс i, и элемента с индексом n-i-1.

## Программа сортировки массива

Теперь возьмем задачу посложнее - простейшую сортировку массива:

int[] nums = { 54, 7, -41, 2, 4, 2, 89, 33, -5, 12 };  
  
  
  
// сортировка  
  
int temp;  
  
for (int i = 0; i < nums.Length - 1; i++)  
  
{  
  
 for (int j = i + 1; j < nums.Length; j++)  
  
 {  
  
 if (nums[i] > nums[j])  
  
 {  
  
 temp = nums[i];  
  
 nums[i] = nums[j];  
  
 nums[j] = temp;  
  
 }  
  
 }  
  
}  
  
  
  
// вывод  
  
Console.WriteLine("Вывод отсортированного массива");  
  
for (int i = 0; i < nums.Length; i++)  
  
{  
  
 Console.WriteLine(nums[i]);  
  
}

Для сортировки массива выполняем проходы по массиву и сравниваем элементы. Поскольку нам надо последовательно сравнивать каждый элемент массива с каждым (за исключением сравния с самим собой),   
  
то здесь применятся вложенный цикл.

Во внешнем цикле мы берем элемент, который будем сравнивать:

for (int i = 0; i < nums.Length - 1; i++)

Далее запускаем вложенный цикл, который начинается, со следующего элемента, и из которого извлекаем элементы, с которыми будем сравнивать тот элемент, которые берется из массива во внешнем цикле:

for (int j = i + 1; j < nums.Length; j++)

Если элемент с меньшим индексом больше элемента с большим индексом, то меняем элементы местами.

if (nums[i] > nums[j])  
  
{  
  
 temp = nums[i];  
  
 nums[i] = nums[j];  
  
 nums[j] = temp;  
  
}

В конце выводим все элементы.

НазадСодержаниеВперед

Contacts: metanit22@mail.ru

Copyright © Евгений Попов, metanit.com, 2024. Все права защищены.

# C# и .NET | Методы

Если переменные хранят некоторые значения, то методы содержат собой набор инструкций, которые выполняют определенные действия.   
  
По сути метод - это именованный блок кода, который выполняет некоторые действия.

Общее определение методов выглядит следующим образом:

[модификаторы] тип\_возвращаемого\_значения название\_метода ([параметры])  
  
{  
  
 // тело метода  
  
}

Модификаторы и параметры необязательны.

Ранее мы уже использовали как минимум один метод - Console.WriteLine(), который выводит информацию на консоль.   
  
Теперь рассмотрим, как мы можем создавать свои методы.

## Определение метода

Определим один метод:

void SayHello()  
  
{  
  
 Console.WriteLine("Hello");  
  
}

Здесь определен метод SayHello, который выводит некоторое сообщение. К названиям методов предъявляются в принципе те же требования, что и к названиям переменных. Однако, как правило,   
  
названия методов начинаются с большой буквы.

Перед названием метода идет возвращаемый тип данных. Здесь это тип void, который указыает,   
  
что фактически ничего не возвращает, он просто производит некоторые действия.

После названия метода в скобках идет перечисление параметров. Но в данном случае скобки пустые, что означает, что метод не принимает никаких параметров.

После списка параметров в круглых скобках идет блок кода, который представляет набор выполняемых методом инструкций. В данном случае блок метода SayHello   
  
содержит только одну инструкцию, которая выводит строку на консоль:

Console.WriteLine("Hello");

Но если мы запустим данный проект, то мы не увидим никакой строки, которую должен выводить метод SayHello. Потому что после определения   
  
метод еще надо вызвать, чтобы он выполнил свою работу.

## Вызов методов

Чтобы использовать метод SayHello, нам надо его вызвать. Для вызова метода указывается его имя, после которого в скобках идут значения для его параметров (если метод принимает параметры).

название\_метода (значения\_для\_параметров\_метода);

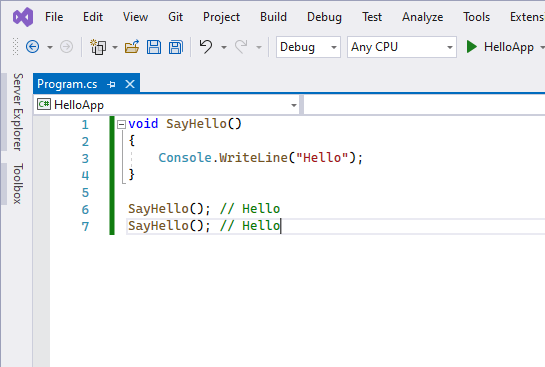
Например, вызов метода SayHello будет выглядеть следующим образом:

SayHello();

Поскольку метод не принимает никаких параметров, то после названия метода идут пустые скобки.

Объединим определение и вызов метода:

void SayHello()  
  
{  
  
 Console.WriteLine("Hello");  
  
}  
  
  
  
SayHello(); // Hello  
  
SayHello(); // Hello



Консольный вывод программы:

Hello  
  
Hello

Преимуществом методов является то, что их можно повторно и многократно вызывать в различных частях программы. Например, в примере   
  
выше два раза вызывается метод SayHello.

При этом в данном случае нет разницы, сначала определяется метод, а потом вызывается или наоборот. Например, мы могли бы написать и так:

SayHello(); // Hello  
  
SayHello(); // Hello  
  
  
  
void SayHello()  
  
{  
  
 Console.WriteLine("Hello");  
  
}

Определим и вызовем еще несколько методов:

void SayHelloRu()  
  
{  
  
 Console.WriteLine("Привет");  
  
}  
  
void SayHelloEn()  
  
{  
  
 Console.WriteLine("Hello");  
  
}  
  
void SayHelloFr()  
  
{  
  
 Console.WriteLine("Salut");  
  
}  
  
  
  
  
  
string language = "en";  
  
  
  
switch (language)  
  
{  
  
 case "en":   
  
 SayHelloEn();  
  
 break;  
  
 case "ru":  
  
 SayHelloRu();  
  
 break;  
  
 case "fr":  
  
 SayHelloFr();  
  
 break;  
  
}

Здесь определены три метода SayHelloRu(), SayHelloEn() и SayHelloFr(), которые также имеют   
  
тип void, не принимают никаких параметров и также выводит некоторую строку на консоль. Условно говоря, они выводят приветствие на   
  
определенном языке.

В конструкции switch проверяется значение переменной language, которая условно хранит код языка,   
  
и в зависимости от ее значения вызывается определенный метод. Так, в данном случае на консоль будет выведено

Hello

## Сокращенная запись методов

Если метод в качестве тела определяет только одну инструкцию, то мы можем сократить определение метода. Например, допустим у нас есть метод:

void SayHello()  
  
{  
  
 Console.WriteLine("Hello");  
  
}

Мы можем его сократить следующим образом:

void SayHello() => Console.WriteLine("Hello");

То есть после списка параметров ставится оператор =>, после которого идет выполняемая инструкция.

НазадСодержаниеВперед

Contacts: metanit22@mail.ru

Copyright © Евгений Попов, metanit.com, 2024. Все права защищены.

# C# и .NET | Параметры методов

В прошлой теме был определен метод SayHello, который выводит на консоль некоторое сообщение:

void SayHello()  
  
{  
  
 Console.WriteLine("Hello");  
  
}  
  
  
  
SayHello(); // Hello

Но минусом подобного метода является то, что он выводит одно и то же сообщение. И было бы неплохо, если бы мы могли бы динамически определять,   
  
какое сообщение будет выводить метод на экран, то есть передать из вне в метод это сообщение. Для этого в языке C# мы можем использовать параметры.

Параметры позволяют передать в метод некоторые входные данные. Параметры определяются через заятую в скобках после названия метода в виде:

тип\_метода имя\_метода (тип\_параметра1 параметр1, тип\_параметра2 параметр2, ...)  
  
{  
  
 // действия метода  
  
}

Определение параметра состоит из двух частей: сначала идет тип параметра и затем его имя.

Например, определим метод PrintMessage, который получает извне выводимое сообщение:

void PrintMessage(string message)  
  
{  
  
 Console.WriteLine(message);  
  
}  
  
  
  
PrintMessage("Hello work"); // Hello work  
  
PrintMessage("Hello METANIT.COM"); // Hello METANIT.COM  
  
PrintMessage("Hello C#"); // Hello C#

Здесь метод PrintMessage() принимает один параметр, который называется message и имеет тип string.

Чтобы выполнить метод, который имеет параметры, при вызове после имени метода в скобках ему передаются значения для его параметров, например:

PrintMessage("Hello work");

Здесь параметру message передается строка "Hello work". Значения, которые передаются параметрам, еще называются аргументами.   
  
То есть передаваемая строка "Hello work" в данном случае является аргументом.

Иногда можно встретить такие определения как формальные параметры и фактические параметры.   
  
Формальные параметры - это собственно параметры метода (в данном случае message), а фактические параметры - значения, которые   
  
передаются формальным параметрам. То есть фактические параметры - это и есть аргументы метода.

Определим еще один метод, который складывает два числа:

void Sum(int x, int y)  
  
{  
  
 int result = x + y;  
  
 Console.WriteLine($"{x} + {y} = {result}");  
  
}  
  
  
  
Sum(10, 15); // 10 + 15 = 25

Метод Sum имеет два параметра: x и y. Оба параметра представляют тип int. Поэтому при вызове данного метода нам обязательно надо   
  
передать на место этих параметров два числа. Внутри метода вычисляется сумма переданных чисел и выводится на консоль.

При вызове метода Sum значения передаются параметрам по позиции. Например, в вызове Sum(10, 15) число 10   
  
передается параметру x, а число 15 - параметру y.

Также параметры могут использоваться в сокращеной версии метода:

void Sum(int x, int y) => Console.WriteLine($"{x} + {y} = { x + y }");  
  
  
  
Sum(10, 15); // 10 + 15 = 25

Передаваемые параметру значения могут представлять значения переменных или результат работы сложных выражений, которые возвращают   
  
некоторое значение:

void Sum(int x, int y) => Console.WriteLine($"{x} + {y} = { x + y }");  
  
  
  
int a = 10, b = 15, c = 6;  
  
Sum(a, b); // 10 + 15 = 25  
  
Sum(3, c); // 3 + 6 = 9  
  
Sum(14, 4 + c); // 14 + 10 = 24

Если параметрами метода передаются значения переменных, то таким переменным должно быть присвоено значение. Например, следующая программа не скомпилируется:

void Sum(int x, int y)  
  
{  
  
 Console.WriteLine($"{x} + {y} = { x + y }");  
  
}  
  
  
  
int a;  
  
int b = 15;  
  
Sum(a, b); // ! Ошибка

## Соответствие параметов и аргументов по типу данных

При передаче значений параметрам важно учитывать тип параметров: между аргументами и параметрами должно быть соответствие по типу. Например:

void PrintPerson(string name, int age)  
  
{  
  
 Console.WriteLine($"Name: {name} Age: {age}");  
  
}  
  
  
  
PrintPerson("Tom", 24); // Name: Tom Age: 24

В данном случае первый параметр метода PrintPerson() представляет тип string, поэтому при вызове метода   
  
мы должны передать этому параметру значение типа string, то есть строку. Второй параметр представляет тип int,   
  
поэтому должны передать ему целое число, которое соответствует типу int.

PrintPerson("Tom", 24);

Также мы можем передать параметрам значения тех типов, которые автоматически могут быть преобразованы в тип параметров. Например:

void PrintPerson(string name, int age)  
  
{  
  
 Console.WriteLine($"Name: {name} Age: {age}");  
  
}  
  
  
  
byte b = 37;  
  
PrintPerson("Tom", b); // Name: Tom Age: 37

Здесь параметру типа int передается значение типа byte, но комилятор может автоматически преобразовать значение типа byte к тиу int. Поэтому здесь ошибки не возникнет.   
  
Какие преобразования типов могут быть выполнены автоматически, рассматривалось в одной из предыдущих тем: Преобразования базовых типов данных

Данные других типов мы передать параметров не можем. Например, следующий вызов метода PrintPerson будет ошибочным:

PrintPerson(45, "Bob"); // Ошибка! несоответствие значений типам параметров

## Необязательные параметры

По умолчанию при вызове метода необходимо предоставить значения для всех его параметров. Но C# также позволяет использовать необязательные параметры.   
  
Для таких параметров нам необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

void PrintPerson(string name, int age = 1, string company = "Undefined")  
  
{  
  
 Console.WriteLine($"Name: {name} Age: {age} Company: {company}");  
  
}

Здесь параметры age и company являются необязательными, так как им присвоены значения. Поэтому при вызове метода мы можем не передавать для них данные:

void PrintPerson(string name, int age = 1, string company = "Undefined")  
  
{  
  
 Console.WriteLine($"Name: {name} Age: {age} Company: {company}");  
  
}  
  
  
  
PrintPerson("Tom", 37, "Microsoft"); // Name: Tom Age: 37 Company: Microsoft  
  
PrintPerson("Tom", 37); // Name: Tom Age: 37 Company: Undefined  
  
PrintPerson("Tom"); // Name: Tom Age: 1 Company: Undefined

Консольный вывод программы:

Name: Tom Age: 37 Company: Microsoft  
  
Name: Tom Age: 37 Company: Undefined  
  
Name: Tom Age: 1 Company: Undefined

## Именованные параметры

В предыдущих примерах при вызове методов значения для параметров передавались в порядке объявления этих параметров в методе. То есть аргументы передавались параметрам по позиции.   
  
Но мы можем нарушить подобный порядок, используя именованные параметры:

void PrintPerson(string name, int age = 1, string company = "Undefined")  
  
{  
  
 Console.WriteLine($"Name: {name} Age: {age} Company: {company}");  
  
}  
  
  
  
PrintPerson("Tom", company:"Microsoft", age: 37); // Name: Tom Age: 37 Company: Microsoft  
  
PrintPerson(age:41, name: "Bob"); // Name: Bob Age: 41 Company: Undefined  
  
PrintPerson(company:"Google", name:"Sam"); // Name: Sam Age: 1 Company: Google

Для передачи значений параметрам о имени при вызове метода указывается имя параметра и через двоеточие его значение: name:"Tom"

Консольный вывод программы:

Name: Tom Age: 37 Company: Microsoft  
  
Name: Bob Age: 41 Company: Undefined  
  
Name: Sam Age: 1 Company: Google

НазадСодержаниеВперед

Contacts: metanit22@mail.ru

Copyright © Евгений Попов, metanit.com, 2024. Все права защищены.

# C# и .NET | Массив параметров и ключевое слово params

Во всех предыдущих примерах мы использовали постоянное число параметров. Но, используя ключевое слово   
  
params, мы можем передавать неопределенное количество параметров:

void Sum(params int[] numbers)  
  
{  
  
 int result = 0;  
  
 foreach (var n in numbers)  
  
 {  
  
 result += n;  
  
 }  
  
 Console.WriteLine(result);  
  
}  
  
  
  
int[] nums = { 1, 2, 3, 4, 5};  
  
Sum(nums);  
  
Sum(1, 2, 3, 4);  
  
Sum(1, 2, 3);  
  
Sum();

Сам параметр с ключевым словом params при определении метода должен представлять одномерный массив того типа, данные которого мы собираемся использовать.   
  
При вызове метода на место параметра с модификатором params мы можем передать как отдельные значения, так и массив   
  
значений, либо вообще не передавать параметры. Количество передаваемых значений в метод неопределено, однако все эти значения должны соответствовать типу параметра с params.

Если же нам надо передать какие- то другие параметры, то они должны указываться до параметра с ключевм словом params:

void Sum(int initialValue, params int[] numbers)  
  
{  
  
 int result = initialValue;  
  
 foreach (var n in numbers)  
  
 {  
  
 result += n;  
  
 }  
  
 Console.WriteLine(result);  
  
}  
  
  
  
int[] nums = { 1, 2, 3, 4, 5};  
  
Sum(10, nums); // число 10 - передается параметру initialValue  
  
Sum(1, 2, 3, 4);  
  
Sum(1, 2, 3);  
  
Sum(20);

Здесь метод Sum имеет обязательный параметр initialValue, поэтому при вызове метода для него нужно обязательно передать значение. Поэтому первое значение при вызове   
  
метода будет передаваться этому параметру.

Однако после параметра с модификатором params мы НЕ можем указывать другие параметры. То есть   
  
следующее определение метода недопустимо:

//Так НЕ работает  
  
void Sum(params int[] numbers, int initialValue)  
  
{}

## Массив в качестве параметра

Также этот способ передачи параметров надо отличать от передачи массива в качестве параметра:

void Sum(int[] numbers, int initialValue)  
  
{  
  
 int result = initialValue;  
  
 foreach (var n in numbers)  
  
 {  
  
 result += n;  
  
 }  
  
 Console.WriteLine(result);  
  
}  
  
  
  
int[] nums = { 1, 2, 3, 4, 5};  
  
Sum(nums, 10);  
  
  
  
// Sum(1, 2, 3, 4); // так нельзя - нам надо передать массив

Так как метод Sum принимает в качестве параметра массив без ключевого слова params, то при его вызове нам обязательно надо передать в качестве первого параметра   
  
массив. Кроме того, в отличие от метода с параметром params после параметра-массива могут располагаться другие параметры.

НазадСодержаниеВперед

Contacts: metanit22@mail.ru

Copyright © Евгений Попов, metanit.com, 2024. Все права защищены.