**C# для чайников**

**Вступление**

Платформа .NET позволяет создавать распределенные приложения.

Преимуществами распределенных приложений являются:

1. возможность сконцентрировать в подходящих местах программный код, выполняющий специализированные задачи.   
   Т.е. становится не нужно на каждой машине поддерживать программное обеспечение, выполняющее все функции
2. разные платформы могут использовать один и тот же код за счет того, что это специализированное программное обеспечение имеет общеизвестный интерфейс.   
   Например сотовые телефоны с их специализированным программно-аппаратным обеспечением могут использовать код web-серверов реализованных по-другому.
3. Становится возможным предоставление платных услуг, например, вычислительных мощностей или информации.   
   Здесь фирма предоставляющие услуги владеет одной частью системы, а пользователи - другой.

С точки зрения разработчика, платформа .NET это совокупность Common Language Runtime (CLR), .NET Framework Classes и высокоуровневых средств, таких как WinForms и Active Server Pages (ASP .NET)

Common Language Runtime управляет выполнением кода, написанного для платформы .NET, Framework Classes позволяют работать с операционной системой, WinForms нужны для создания пользовательского интерфейса, а ASP для формирования web-страниц.

Код для платформы может быть скомпилирован из исходных текстов, написанных на разных языках. Один из таких языков - C#.

**Шаг 2. Синтаксис C#**

Принято начинать описание языка с примера Hello World. Можете посмотреть [этот](http://www.development.efficientit.com/csharp/code/console/helloworld.html)

Прежде всего, нужно сказать, что C# учитывает регистр идентификаторов. Компилятор выдаст ошибку, если, к примеру, написть слово Console с маленькой буквы.

Во-вторых, каждый оператор заканчивается точкой с запятой. Поэтому циклы нужно писать как минимум в две строки, а лучше всегда использовать составные операторы. Так

while (i < 10) SomeFunction();

OtherFunction();

менее читабельно, чем

while (i < 10)

{

SomeFunction();

}

В C# не бывает операторов вне функций, а функций вне классов. Поэтому в примере описан класс World и функция Main. Выполнение программы начинается с метода Main одного из классов программы. Стоит обратить внимание на то, что Main пишется с заглавной (большой) буквы. Имя класса может отличаться от имени того файла, в котором класс находится.

Директивы using позволяют сократить имена идентификаторов из используемых библиотек. Так, например, объект Console на самом деле называется System.Console, но так как используется директива using, можно писать просто Console.

Комментарии в C# бывают нескольких видов:

* до конца строки - //
* многострочные - /\* \*/
* для документации - ///

**Строки**

В тексте программы не всегда можно написать все символы, поэтому те символы, которые нельзя написать непосредственно, заменяют на цепочки из других символов.

В C# есть два вида такой замены.

1. Использование замен при помощи символа

|  |  |
| --- | --- |
| **Символ, который надо заменить** | **Цепочка** |
| ' | ' |
| " | \" |
|  | \\ |
| Звонок | [\a](file:///\\a) |
| Забой | [\b](file:///\\b) |
| Перевод страницы | [\f](file:///\\f) |
| Новая строка | [\n](file:///\\n) |
| Возврат каретки | [\r](file:///\\r) |
| Табуляция | [\t](file:///\\t) |
| Вертикальная табуляция | [\v](file:///\\v) |
| Символ Unicode, заданный своим номером, например u200 | [\u](file:///\\u) |
| Символ Unicode, заданный своим номером в 16-ричной системе счисления, например xc8 | [\x](file:///\\x) |
| Символ с кодом ноль | \0 |

2. Строки с удвоенными кавычками (как в ZX Spectrum)   
Здесь нужно поставить перед строкой символ @ а затем удваивать кавычки внутри строки. Если такая строка занимает несколько строк программы, то будут сохранены символы перевода на новую строку и пробелы.

Пример: если нужно записать строку

C:My Documents

то можно писать так:

string path = "C:My Documents";

а можно - так:

string path = @"C:MyDocuments";

**Переменные**

переменные в C# обязательно имеют тип и могут содержать только объекты этого типа (и производных).

Есть три вида переменных

* значения
* ссылки
* указатели (это для специальных участков программы)

Переменные, содержащие значения непосредственно содержат значения, по одному значению на переменную. Для того, чтобы самому определить тип переменных-значиний можно использовать перечисления или структуры

Ссылки непосредственно содержат данные, позволяющие определить местоположение значения. Таким образом могут существовать несколько переменных ссылочного типа, ссылающиеся на одно значение. Если изменить значение через одну переменную такого типа, то изменится значение для всех таких переменных.

**Переменные-значения**

Ключевые слова языка C# отображаются (отображаются - являются сокращениями для имен) на типы пакета System, поэтому во второй колонке приведены названия этих типов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **тип C#** | **тип из .Net Framework** | **Знаковый бит** | **Сколько байтов** | **Граничне значения** |
| sbyte | System.Sbyte | есть | 1 | от -128 до 127 |
| short | System.Int16 | есть | 2 | от -32768 до 32767 |
| int | System.Int32 | есть | 4 | от -2147483648 до 2147483647 |
| long | System.Int64 | есть | 8 | от -9223372036854775808 до 9223372036854775807 |
| byte | System.Byte | нет | 1 | от 0 до 255 |
| ushort | System.Uint16 | нет | 2 | от 0 до 65535 |
| uint | System.UInt32 | нет | 4 | от 0 до 4294967295 |
| ulong | System.Uint64 | нет | 8 | от 0 до 18446744073709551615 |
| float | System.Single | есть | 4 | Примерно от ±1.5 x 10-45 до ±3.4 x 1038 , 7 значащих цифр |
| double | System.Double | есть | 8 | Примерно от ±5.0 x 10-324 до ±1.7 x 10308 , 15 или 16 значащих цифр |
| decimal | System.Decimal | есть | 12 | Примерно от ±1.0 x 10-28 до ±7.9 x 1028 , 28 или 29 значащих цифр |
| char | System.Char | не применимо | 2 | любой символ Unicode (16-ти битный) |
| bool | System.Boolean | не применимо | 1 / 2 | true и false |

Есть еще (immutable) класс string , объекты которого обязаны всегда создаваться заново, поэтому после выполнения кода

string s1 = "hello";

string s2 = s1;

s1 = "goodbye";

строка s1 будет равна "goodbye", а s2 - "hello"

**Переменные - ссылки**

Переменные этого вида на самом деле хранят адрес начала памяти, в которой находятся данные, на которые эти переменные ссылаются.

object x = new object();

x.myValue = 10;

object y = x;

y.myValue = 20; // после этого оператора x.myValue и y.myValue равняюся 20

Новые типы могут быть определены при помощи конструкций 'class', 'interface' и 'delegate'.

**Boxing**

C# позволяет преобразовывать переменные-значения в переменные-ссылки и обратно.

int i = 123;

object box = i;

if (box is int)

{

Console.Write("Box contains an int"); // this line is printed

i = (int)box;

}

**Перечисления**

Перечисление - это тип, объекты которого могут иметь значения только из заданного набора. При определении типа можно указать значения, которые будут образовывать такой набор. Например это могут быть дни недели, месяцы года, виды валют, поддерживаемые системой, или что-то еще (что?).

public enum DAYS

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday,

}

По умолчанию базовым типом для перечисления является тип integer. Первому значению сопоставляется число ноль, следующему 1 и т.д. (поэтому Sunday == 6). После последнего значения можно ставить запятую, а можно и не ставить

Можно изменить базовый тип для перечисления

enum byteEnum

: byte

{

Zero,

One

}

Кроме того, можно задать значения первому элементу перечисления (последующие будут пронумерованы автоматически) или даже каждому элементу

public enum MONTHS

{

Apr = 4,

Aug = 8,

Dec = 12,

Feb = 2,

Jan = 1,

Jul = 7,

Jun = 6,

Mar = 3,

May = 5,

Nov = 11,

Oct = 1,

Sep = 9,

}

Можно распечатать символьное название константы (используя ToString())

using System;

public class EnumTest

{

public enum DAYS

: byte

{

Monday = 1,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

public static void Main()

{

object[] dayArray = Enum.GetValues(typeof(EnumTest.DAYS));

foreach (DAYS day in dayArray)

{

Console.WriteLine("{0} == {1}", day, (byte)day);

}

}

}

Здесь функция WriteLine заменяет {n} на n-ый аргумент функции после строки

**Операторы**

В таблице приведен список операторов и помечены те из них, которые можно перегрузить

|  |  |  |  |
| --- | --- | --- | --- |
| **Вид оператора** | **Название** | **Пример** | **Перегружаем?** |
| Первичные | Группировка | (a+b) | Нет |
| Доступ | A.B | Нет |
| Доступ к элементу структуры | A->B | Нет |
| Вызов метода | f(x) | Нет |
| Постинкремент | c++ | Да |
| Постдекремент | c-- | Да |
| Вызов конструктора | c = new Coord(); | Нет |
| Создание массива на стеке | int\* c = stackalloc int[10] | Нет |
| Размер структуры | sizeof (int) | Нет |
| Включение режима проверки переполнения | checked {byte c = (byte) d;} | Нет |
| Выключение режима проверки переполнения | unchecked {byte c = (byte) d;} | Нет |
| Унарные (с одним операндом) | Положительное значение | +10 | Нет |
| Отрицательное значение | -10 | Нет |
| Отрицание | !(c==d) | Да |
| Побитовое отрицание | ~(int x) | Да |
| Прединкремент | ++c | Да |
| Преддекремент | --c | Да |
| Преобразование типа | (myType)c | Нет |
| Присваивание адреса | int\* c = d; | Нет |
| Взятие адреса значения | int\* c = &d; | Нет |
| Выяснение типа | Совместимость типа | a is String | Нет |
| Получение класса, описывающего тип | typeof (int) | Нет |
| Арифметические | Умножение | c \* d | Да |
| Деление | c / d | Да |
| Остаток | c % d | Да |
| Сложение | c + d | Да |
| Вычитание | c - d | Да |
| Побитовый сдвиг вправо | c >> 3 | Да |
| Побитовый сдвиг влево | c << 3 | Да |
| Логические и операции сравнения | Меньше чем | c < d | Да |
| Больше чем | c > d | Да |
| Меньше или равно | c < =d | Да |
| Больше или равно | c > = d | Да |
| Равно | c == d | Да |
| Не равно | c != d | Да |
| Побитовое И | c & d | Да |
| Побитовое ИЛИ | c | d | Да |
| Логическое И | c && d | Нет |
| Логическое ИЛИ | c || d | Нет |
| Условный оператор | (d < 10) ? 5 : 15 | Нет |

**Перегрузка операторов**

Для того, чтобы перегрузить оператор, нужно использовать ключевое слово 'operator'.

public static bool operator == (Value a, Value b)

{

return a.Int == b.Int;

}

Если операторы образуют пары, то нужно перегрузить оба оператора пары

== и !=

< и

> <

= и

>

=

**Циклы**

В С# есть четыре вида циклов

* while
* do-while
* for
* foreach

Переменные объявленные в цикле (в том числе в заголовке цикла for и foreach) не видны снаружи цикла (так же как в стандартном C++)

**while**

Пока истинно условие выполняется оператор - тело цикла.

int a =

3;

while (a

< 3)

{

System.Console.WriteLine(a);

a++;

}

**do-while**

Здесь сначала один раз выполняется тело цикла, затем проверяется условие. Если условие истинно, то выполнение тела цикла повторяется.

int a =

4;

do {

System.Console.WriteLine(a);

a++;

}

while

(a

< 3)

**for**

Этот цикл чаще всего используют когда число повторений известно заранее.

for (int a =

0;

a < 5; a ++)

{

System.Console.WriteLine(a);

}

Есть частная разновидность - бесконечный цикл - for (;;){...}   
В C# рекомендуют бесконечный цикл писать как while(true){...}

**foreach**

Этот цикл полезен для перебора всех элементов в массиве, классах типа ArrayList и Hashtable (и вообще всех классах, реализующих IEnumerable)

int[] a = new int[]{1,2,3};

foreach (int b in a)

{

System.Console.WriteLine(b);

}

При этом переменной b присваивается по очереди каждый элемент массива. Если переменная b имеет содержит непосредственно значение (а не ссылку на объект), то изменять элементы массива через нее не получится. Если же переменная b - объект, то через нее можно менять текущий элемент массива.

**Операторы перехода**

К операторам перехода относятся

* break
* continue
* goto
* return
* throw

**break**

Этот оператор может стоять внутри циклов for, do-while и while, а также внутри оператора switch. Выполнение оператора break приводит к выходу из цикла или оператора switch.

**continue**

Этот оператор может стоять внутри циклов for, do-while и while. Выполнение оператора continue приводит к переходу на проверку условия цикла

**goto**

Оператор goto используется для перехода на метку и внутри оператора switch. Переходы на метку использовать не рекомендуется.

int a =

0;

start: System.Console.WriteLine(a);

a++;

if

(a

< 5)

{

goto start;

}

**Оператор выбора (условный оператор)**

Оператор выбора используется для выполнения или не выполнения некоторого оператора в зависимости от условия. Если условие истинно, то вложенный оператор выполняется.   
Кроме того, может присутствовать альтернативная ветвь, которая выполняется в том случае, если условие ложно.

if (a == 5)

System.Console.WriteLine("A is 5");

else

System.Console.WriteLine("A is not 5");

Еще бывает операция выбора

(a == 5) ? true : false ;

**Оператор ветвления**

Здесь выполняется одна из альтернативных ветвей в зависимости от значения управляющего выражения

switch (nValue)

{

case 1: case 2: case 3: case 4: case 5:

Console.WriteLine("Rabbit walk through the street" );

break; // this line is necessary

}

switch (someString)

{

default:

break;

case null:

break;

case "leavetakings" : // two cases for code

case "greetings" :

goto default;

case "goodbye" :

goto case "leavetakings";

case "error":

// int i = 5;

// if you uncomment previous line you should get

// "control cannot fall through from one case to another"

error

case "next case":

throw new ApplicationException();

}

Отличительной возможностью C# является возможность ветвления по строкам и запрет на переход между блоками без явного на это указания (при компиляции выдается ошибка)

Есть возможность сопоставить одному блоку несколько значений (в примере - "leavetakings" и "greetings"). Есть возможность передать управление на другой блок (goto case x; goto default;).

В случае если делается ветвление по переменной типа string, значение которой равно null, а соответствующий case отсутствует, будет выполняться ветвь default (если такая есть). null - вполне полноправное значение переменной типа string и NullPointerException оператором switch выбрасываться не будет.

**Массивы**

Массивы имеют тип System.Array и не копируются при присваивании (т.е. это ссылки а не значения). Массивы можно использовать в циклах foreach. Свойство Length содержит общее число элементов массива (по всем измерениям).

**Одномерные массивы (векторы в C++)**

Тип массива включает в себя информацию о базовом типом элементов, которые может содержать массив, и о размерности (количестве индексов) массива.

Одномерные массивы определяются так:

int[] arr = new int[100];

Здесь arr - имя массива, 100 - количество элементов в массиве, первый элемент - arr[0], последний arr[99].

Заполнить массив можно тремя способами:

* int i = new int[2];
* i[0] = 1;
* i[1] = 2;
* int[] i = new int[] {1,2};
* int[] i = {1,2};

Используя класс System.Array можно сделать массив с произвольным значением индекс первого элемента.

**Прямоугольные массивы (Такой конструкции нет в Java)**

Двумерный массив 2 x 3:

int[,] squareArray1 = new int[2,3];

int[,] squareArray2 = {{1,

2, 3}, {4, 5,

6}}; for(i =

0; i < squareArray2.GetLength(0);

i++) { for (j =

0;

j < squareArray2.GetLength(1); j++)

{

Console.WriteLine(squareArray[i,j]);

}

}

Чтобы получить граничные индексы элементов в некотором измерении, нкжно воспользоваться функциями GetUpperBound(int i) и GetLowerBound(int i), где i - номер измерения (считая с нуля). GetLength(int i) вернет количество элементов в некотором измерении.

**Массивы из массивов (Jagged Arrays)**

Они подходят для текстовых редакторов.

// memory allocation and object zeroing

int[][] jag1 = new int[2][];

jag1[0] = new int [4];

jag1[1] = new int [6];

for (int i = 0; i < jag1.GetLength(0); i++)

{

for (int j =

0; j <

jag1[h].GetLength(0); j++) { Console.WriteLine(jag1[i][j]);

} } // creating

an initialized array int[][] jag2 = new int[][]

{

new int[] {1,

2, 3,

4}, new int[] {5,

6, 7, 8, 9,

10} }; for (int i =

0; i < jag2.Length; i++)

{ for

(int j =

0;

j < jag2[h].Length; j++)

{

Console.WriteLine(jag2[i][j]);

}

}

**Indexer'ы**

Индексаторы (как бы их по-русски поудачнее назвать?) могут пригодиться если нужно:

* сделать разреженный массив;
* хранить массив не в памяти, а, к примеру, на диске;
* вычислять значения массива на лету и хранить только измененные значения

Синтаксис:

public double this[int i, int j]

{

set

{

MyCodeToStoreValue(i, j, value);

}

get

{

double res = MyMethodToRetrieveValue(i, j);

return res;

}

}

Здесь value - ключевое слово, которое обозначает значение

**Исключения**

Исключения могут быть выброшены двумя способами: либо оператором throw (в этом случае оно выбрасывается всегда). Либо при возникновении особой ситуации (например деления на ноль) - это случается не всегда (и по-идее должно не случаться)

Исключения ловятся блоками try-catch:

try

{

throw new Exception();

}

catch (Exception ex)

{

}

finally

{

}

Блок finally выполняется независимо от того - было исключение или нет. (Это альтернатива деструкторам)

Можно написать несколько блоков catch подряд, в этом случае выбирается первый подходящий по типу. Поэтому следует писать обработчики специальных исключений сначала, а базовых исключений в конце. Свои исключения стоит наследовать от ApplicationException