Programación C#

Funciones

Índice

Tipos de programación

Programación monolítica

Programación realizada empleando únicamente el bloque ***main.***

En programas más complejos este bloque main puede volverse difícil de gestionar.

Programación modular

Permite dividir un problema en varios subproblemas que,unidos , ayudan a resolver el problema principal.

Cada uno de esos subproblemas se resuelve mediante un ***módulo*** específico.

Estos ***módulos*** también se suelen llamar ***funciones***,***procedimientos*** o ***subrutinas***.

Ventajas

Permite repartir una tarea compleja en varios pasos simples.

Permite reutilizar el código: si queremos hacer una cosa varias veces, utilizamos varias veces el módulo que la hace.

Los ***módulos*** que se generan son más cortos y fáciles de mantener.

Concepto de función

Cada uno de los módulos en que se puede dividir un programa consta de:

-Un nombre para identificarla/invicarla

-Parámetros

-Instrucciones que ejecuta

-Resultado que produce

Definición de funciones

Se define el tipo de **retorno**,hay 2 tipos:

-No tiene retorno(**static void ()**

-Si tienen retorno(**int**,**string**,**float**,etc)

Indicamos el nombre de la función,con unos paréntesis(el nombre suele ser un verbo).

Entre llaves,colocamos el código de la función.

Podemos definir tantas como queramos dentro de una clase, en el orden que queramos.

```
static void Saludar()
{
    Console.WriteLine("Hola, buenas");
    Console.WriteLine("Bienvenido/a")
}
```

Invocación o llamada a funciones

```
static void Saludar()
{
    Console.WriteLine("Hola, buenas");
    Console.WriteLine("Bienvenido/a");
}

static void Main()
{
    Saludar();
}
```

Parámetros

Datos que necesita la función para completar su tarea.

Se indican dentro de los paréntesis, separados por comas, indicando de qué tipo es cada uno y su nombre (como si fueran variables).

```
/* Función que saluda al nombre que se le indica como parámetro */
static void Saludar(string nombre)
{
    Console.WriteLine("Hola {0}", nombre);
}

/* Función que muestra la suma de sus dos parámetros enteros */
static void MostrarSuma(int n1, int n2)
{
    Console.WriteLine("{0} + {1} = {2}", n1, n2, n1+n2);
}

/* Invocación de funciones */
static void Main()
{
    Saludar("Nacho");   // Hola Nacho
    MostrarSuma(7, 3);  // 7 + 3 = 10
}
```

Paso de parámetros al main

Los programas que compilamos generan un ejecutable que se puede lanzar desde el terminal.

Algunos comandos útiles de los sistemas operativos admiten una serie de parámetros adicionales.

Ejemplos: sudo apt-get install, dir *.txt, etc.

Estos parámetros que se pasan desde fuera al programa se pueden recoger con un array de strings como parámetro del Main.

```
// miPrograma.exe Nacho 20

static void Main(string[] args)
{
    Console.WriteLine(args[0]); // Nacho
    Console.WriteLine(args[1]); // 20
}
```

Parámetros con valores por defecto

Podemos asignar un valor por defecto a los parámetros de una función.

De este modo, si no le pasamos ese parámetro, tomará su valor por defecto.

Es **IMPORTANTE** que los parámetros con valores por defecto se sitúen **AL FINAL** de la lista de parámetros.

A la hora de llamar a la función, podemos omitir parámetros siempre desde el final (nunca en posiciones intermedias).

```
static string Linea(int veces, char simbolo = '*')
{
    return new string(letra, veces);
}

static void Saludar(string nombre, string apellido =
"García", int dia = 1)
{
    Console.WriteLine("Hola {0} {1}, hoy es día {2}",
nombre, apellido, dia);
}

static void Main()
{
    string texto1 = Linea(5, '-');  // -----
    string texto2 = Linea(8);       // ********

    Saludar("Nacho");       // Hola Nacho García, hoy es día
1
    Saludar("Nacho", 3);    // ERROR! El segundo
parámetro es el apellido
}
```

Parámetros con nombre

Desde hace poco también se permite llamar a una función cambiando el orden de los parámetros, siempre que en la llamada indiquemos el nombre de cada parámetro para que se puedan emparejar.

```
static void DibujarRectangulo(int nBase, int nAltura)
{ ... }

static void Main()
{
    DibujarRectangulo(nAltura: 3, nBase: 8);
}
```

Tipo de retorno

La palabra ***void*** indica que la función no tiene ningún resultado(Habitual cuando las funciones se limitan a sacar datos por pantalla).

En caso de que la función produzca algún resultado remplazamos ***void*** por el tipo de resultado producido.

en este caso la función devuelve el resultado producido usando la instrucción ***return***.

En estos casos es habitual asignar el resultado devuelto a una variable del mismo tipo, o usarlo en una expresión.

```
/* Esta función devuelve el cuadrado del número que recibe como parámetro */
static int Cuadrado(int n)
{
    return n * n;
}

/* Esta función devuelve la suma de sus dos parámetros enteros */
static int Sumar(int n1, int n2)
{
    return n1 + n2;
}

/* Llamada a funciones */
static void Main()
```

```
{
    int cuad = Cuadrado(4);
    Console.WriteLine(cuad);         // 16

    int resultado = Sumar(7, 3);
    Console.WriteLine(resultado);    // 10
    Console.WriteLine(Sumar(4, 2));  // 6
}
```

Variables locales y globales

Las variables declaradas dentro de un bloque sólo existen en ese bloque (variables **locales**)

-Una variable declarada dentro de un **if** sólo existe en ese **if**.

-Una variable declarada en una **función** sólo existe en esa **función**.

Las variables declaradas fuera de cualquier bloque (dentro de la **clase**) son **globales**.

Compartir información entre módulos

Podríamos pensar que una forma de hacerlo es usando variables globales.

Desaconsejado en general porque puede producir efectos colaterales entre módulos (que uno modifique una variable global inesperadamente).

Lo mejor es pasar información a través de los **parámetros**.

Las variables temporal e i son locales de la función Potencia, no existen fuera de la misma.

```
static int Potencia(int numBase, int numExponente)
{
    int temporal = 1;
    for (int i = 1; i <= numExponente; i++)
        temporal *= numBase;
    return temporal;
}

static void Main()
{
    Console.WriteLine(Potencia(4, 3));  // 64
}
```

Conflictos en nombres de variables

Si una variable global se llama igual que otra local, prevalece la local.

```
class Prueba
{
   static int x = 20;

   static void Main()
   {
      int x = 3;
      Console.WriteLine(x);       // 3
   }
}
```

Si una variable local a una función se llama igual que otra local a otra función, son variables independientes. Cada una existe en su función.

```
static void CambiaN()
{
   int n = 5;
   n++;
}

static void Main()
{
   int n = 1;
   Console.WriteLine(n);   // 1
   CambiaN();
   Console.WriteLine(n);   // 1
}
```

Paso de parámetros

Por valor

Los tipos simples (enteros, reales, cadenas, caracteres...) SIEMPRE se pasan por valor a una función.

Esto significa que se pasa una COPIA del valor, con lo que no se puede modificar la variable original.

```
static void CambiarValor(int n)
{
    n++;
}

static void Main()
{
    int numero = 0;
    CambiarValor(numero);
    Console.WriteLine(numero);  // 0
}
```

Por referencia

Los datos complejos (arrays, structs, objetos...) SIEMPRE se pasan por referencia.

Podemos modificar el contenido de estos datos siempre que no modifiquemos su referencia (es decir, siempre que no los reasignemos enteros).

```
static void CambiarArray(int[] numeros)
{
    for (int i = 0; i < numeros.Length; i++)
    {
        numeros[i]++;
    }
}

static void CambiarArray2(int[] numeros)
{
    numeros = new int[5] {0, 10, 20, 30, 40};
}

static void Main()
{
    int[] datos = {1, 2, 3, 4, 5};
    CambiarArray(datos);            // datos = {2, 3, 4, 5, 6}
    CambiarArray2(datos);           // datos = {2, 3, 4, 5, 6}
}
```

Por referencia de tipos simples

Usamos la palabra **ref** en el parámetro (tanto en la definición como en la llamada).

Es necesario que la variable que pasamos como parámetro tenga un valor inicial.

```
static void CambiarValor(ref int n)
{
   n++;
}

static void Main()
{
   int numero = 0;
   CambiarValor(ref numero);
   Console.WriteLine(numero);  // 1
}
```

Parámetros de salida

samos la palabra **out** en el parámetro (tanto en la definición como en la llamada).

No es necesario que la variable tenga un valor inicial.

```
static void Mayor(int n1, int n2, out int resultado)
{
    if (n1 > n2)
        resultado = n1;
    else
        resultado = n2;
}

static void Main()
{
    int mayor;
    Mayor(3, 7, out mayor);
    Console.WriteLine(mayor);   // 7
}
```

Algunas funciones útiles

Tryparse

Alternativa a usar las funciones Convert.ToXXXX y tener que capturar las posibles excepciones en la conversión.

Parámetros: dato a convertir (típicamente texto) y parámetro de tipo out donde almacenar la conversión.

Devuelve un booleano indicando si se ha podido convertir o no el dato.

Más compacto y fácil de usar.

```
int numero;

Console.WriteLine("Escribe un número del 1 al 10");
if (Int32.TryParse(Console.ReadLine(), out numero) &&
    numero >= 0 && numero <= 10)
{
    Console.WriteLine("Número correcto");
}
else
{
    Console.WriteLine("Número incorrecto");
}
```

Números aleatorios

Creamos una variable de tipo **Random.**

Debemos crearla **UNA** sola vez para todos los números que queramos generar.

Usamos la función **Next** de dicha variable, indicando el rango de valores deseado.

Si no indicamos nada, se genera un entero aleatorio positivo.

```
Random r = new Random();
int valor1 = r.Next(100, 200);  // Entre 100 y 199 inclusive
int valor2 = r.Next(0, 11);     // Entre 0 y 10 inclusive
int valor3 = r.Next();          // Entero aleatorio positivo
```

Uso avanzado de la consola

Console.Clear()

Limpia la consola.

Console.SetCursorPosition(int x, int y)

coloca el cursor en las coordenadas indicadas como parámetro, siendo y = 0 la fila superior de la consola.

Console.ForegroundColor

permite obtener o cambiar el color de texto.

Console.BackgroundColor

permite obtener o cambiar el color de fondo.

Console.ResetColor()

restablece los colores por defecto (fondo y texto).

```
/* Dibujamos fila de 10 asteriscos rojos en línea 3 */
Console.SetCursorPosition(0, 2);
Console.ForegroundColor = ConsoleColor.Red;
for(int i = 1; i <= 10; i++)
    Console.Write("*");
```

Thread.Sleep

Pausa el programa la cantidad de milisegundos que indiquemos.

Debemos incluir el espacio **System.Threading**.

```
using System.Threading;
...
Console.WriteLine("Hola");
Thread.Sleep(2000);
```

```
Console.WriteLine("Este mensaje saldrá 2 segundos después");
```

Recursividad

Un módulo recursivo resuelve un problema llamándose a sí mismo con una versión cada vez más simple del problema a resolver.

En el código del módulo encontraremos siempre dos elementos:

Caso(s) base: caso más simple de todos, al que poco a poco van tendiendo las llamadas recursivas.

Caso(s) recursivo(s): llamadas al propio módulo descomponiendo el problema en otro más simple.

Tipos

Simple

en la definición de la función sólo hay una llamada a sí misma cada vez.

Ejemplo: factorial.

Múltiple

en la definición de la función hay más de una llamada a sí misma cada vez.

Ejemplo: serie de Fibonacci.

Indirecta

una función A llama a otras funciones, que al final acaban llamando de nuevo a A.

Usos

Muchos problemas se pueden expresar de forma más simple y corta de forma recursiva.

En algunos tipos de juegos (por ejemplo, búsquedas en laberintos o mapas, o juegos de tablero por turnos), es muy utilizada para buscar posibles jugadas contra el adversario.

Ejemplos

Factorial

El factorial de un número n se representa por n!, y se calcula multiplicando todos los números desde 1 hasta n.

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Representación del concepto de factorial.

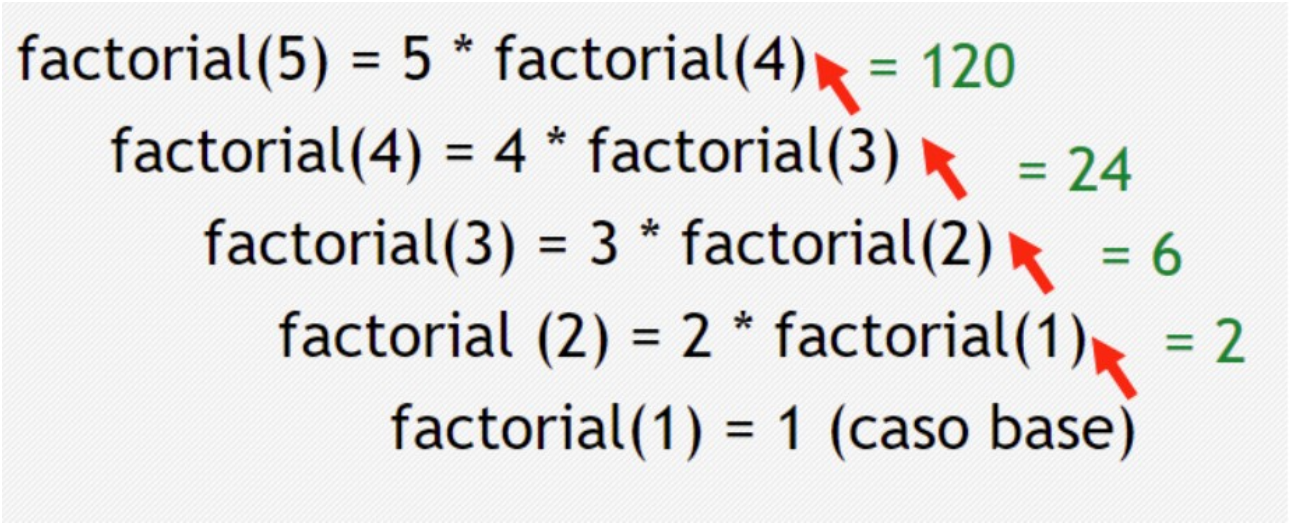
Podemos representar el factorial de n como n por el factorial del número anterior (n-1), y repetir este proceso hasta llegar al caso más simple (factorial de 1 = 1).

```
5! = 5 * 4! = 5 * 4 * 3! = 5 * 4 * 3 * 2! = 5 * 4 * 3 * 2 * 1!
= 5 * 4 * 3 * 2 * 1
```

```
static int Factorial(int n)
{
    // Caso base
    if (n == 1 || n == 0)
        return 1;
    else
        return n * Factorial(n - 1);
```

```
}
```

Pila de llamadas recursivas



Serie de Fibonacci

Se comienza con los números 0 y 1.

El resto de elementos se consigue sumando los dos anteriores.

```
0 1 1 2 3 5 8 13 21 ...
```

Podemos expresarlo así:

```
Fibonacci(0) = 0
```

Fibonacci(1) = 1

Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)