# Introducción a C# desde Java

Al igual que Java, C# es un lenguaje orientado a objetos y por tanto todo se distribuye en clases. Sin embargo, hay un par de conceptos estructurales que están por encima de las clases que conviene conocer.

## Namespaces (Espacios de nombres)

Es una forma de agrupar los diferentes elementos: clases, interfaces, módulos, estructuras, etc., de forma lógica y jerarquizada. Una especie de librería. Es equivalente a los *packages* de Java.

Por ejemplo, en una aplicación de gestión, que es un *namespace*, puedo tener dentro a su vez un *namespace* donde se agrupan las clases de contabilidad y otro en el que se agrupan las de control de stocks.

Cuando queremos acceder a un *namespace* ajeno, se debe usar el nombre de dicha *namespace*, de forma similar a como accedemos a las propiedades o métodos de los objetos. O usar la sentencia *using* en la parte superior del código.

Un espacio de nombres puede estar repartido entre varios ejecutables o *dll's* o incluso en un mismo programa se pueden establecer diversos espacios de nombres.

## Ensamblados

Son bloques ejecutables. Es decir, son los clásicos *.exe* y *.dll* que componen una aplicación, incluyendo, además, recursos como imágenes, textos, etc., con la ventaja de no tener que cargarlos desde un soporte externo. Puedes verlo de forma equivalente a los *.jar* de *Java*.

Una *DLL* (*Dinamic Link Library*) es un módulo que tiene en su interior funciones ejecutables, clases, etc. y que pueden ser llamadas desde distintas aplicaciones. Por ejemplo, las *DirectX* son un conjunto de *DLL* que proporcionan clases y funciones para manejo de gráficos, audio, etc. Son como librerías, pero ya compiladas y preparadas para ejecutar.

Puedes ver las funciones de las *DLL's* instaladas en tu *Windows* mediante programas como: http://www.nirsoft.net/utils/dll_export_viewer.html

## Estructura básica de un programa en C#

Al crear un nuevo proyecto de consola (.Net Framework) podemos ver la estructura mínima de un programa C#:

```
using System;

usingSystem.Collections.Generic;

using System.Linq;

using System.Text

namespace ConsoleApplication1
 {
     class Program
     {
         static void Main(string[] args)
         {
             Console.WriteLine("Bienvenido a la programación en C#");
             Console.ReadLine();
         }
     }
 }
```

Los using iniciales son similares a los import de Java.

El namespace es equivalente al package pero con la salvedad de que tiene llaves de inicio y fin de forma que en un archivo podría tener varios namespaces.

La función principal de C# es la estática Main (fíjate en que empieza por mayúscula).

Las líneas dentro del Main realizan tareas ya conocidas: WriteLine escribe una línea en la consola terminándola en un retorno de carro y ReadLine es similar al nextLine() del Scanner de Java.

Por supuesto lo anterior lo genera automáticamente el VS (Visual Studio) y no todas las líneas son necesarias en todos los programas. Podríamos reducirlo aún más:

```
 using System;
 class Program
 {
     static void Main()
     {
         Console.WriteLine("Bienvenido a la programación en C#");
         Console.ReadLine();
     }
}
```

De hecho, si crearas un proyecto consola .Net Core aparecería así, pues las otras librerías son de Windows.

Hemos dejado sólo el using System ya que contiene la clase Console. Si quisiéramos quitar este using podríamos hacerlo metiendo el System en la línea de Console de la siguiente forma:

```
class Program
{
    static void Main()
    {
        System.Console.WriteLine("Bienvenido a la programación en C#");
        System.Console.ReadLine();
    }
}
```

También se ha modificado el Main ya que dispone de varias sobrecargas con y sin parámetros.

Teniendo estas cuestiones básicas en cuenta empezaremos a ver los distintos elementos del lenguaje diferenciándolos con el Java.

## Comentarios

Existen al igual que en java los comentarios de línea mediante doble barra // y los de varias líneas mediante barra-asterisco /* */

En el caso de los comentarios de documentación en el caso de C# se realizan con triple barra /// y se usan tags XML. Con programas como el SandCastle de Microsoft se puede generar documentación CHM o HTML a partir de estos comentarios. Probablemente veamos algo más adelante.

## Lista de palabras reservadas

No pueden ser usadas como identificadores

```
abstract, as, base, bool, break, byte, case, catch, char, checked,
class, const,continue, decimal, default, delegate, do, double, else,
enum, event, explicit, extern, false, finally, fixed, float, for,
foreach, goto, if, implicit, in, int,interface, internal, lock, is,
long, namespace, new, null, object, operator, out, override, params,
private, protected, public, readonly, ref, return, sbyte, sealed,
short, sizeof, stackalloc, static, string, struct, switch, this,
throw, true, try, typeof, uint, ulong, unchecked, unsafe, ushort,
using, virtual, void, while
```

**Se podría usar** como identificador una palabra reservada si se le **antepone el carácter @**. Esto permite usar luego si es un nombre de una propiedad, acceder a través del objeto y dicho nombre sin @. Por ejemplo, miObjeto.class si se definió la propiedad @class. Ojo, es una práctica que puede ser confusa y por ello no es recomendable.

## Literales

Los literales se manejan prácticamente igual que en Java. Los recordamos:

Enteros: simplemente se escribe la cifra. Se admite decimal y hexadecimal.

Si se desea escribir el número en hexadecimal, se antepone el prefijo 0x. Por ejemplo, 0xFF.

Solo desde la versión 7 de C# se admite también en binario con el prefijo 0b. Por ejemplo, 0b1001 sería 9 en decimal.

Para negativos, se antepone el signo menos (–).

Reales: admiten tanto notación habitual con punto como científica con E o e. Ejemplo: 15.21, 3.02e10.

Lógicos: true y false.

Carácter: se encierran entre comillas simples: 'a', 'P', ...

También se pueden representar mediante su código UNICODE. Para ello se usa la "secuencia de escape" \u. Es decir, si se pone entre comillas simples \u, el C# entiende que lo que va a continuación es el código hexadecimal del carácter que se desea escribir. Por ejemplo '\u002A' hace referencia al asterisco (de hecho, es lo mismo que escribir '*').

Se pueden ver los distintos valores UNICODE de los caracteres en el mapa de caracteres (charmap).

Además del uso del código UNICODE, algunos caracteres especiales y de control tienen su propia secuencia de escape al igual que en Java:

| **Carácter** | **Código de escape Unicode** | **Código de escape especial** |
|---|---|---|
| Comilla simple | \u0027 | \' |
| Comilla doble | \u0022 | \' |
| Carácter nulo | \u0000 | \0 |
| Alarma | \u0007 | \a |
| Retroceso | \u0008 | \b |
| Salto de página | \u000C | \f |
| Nueva línea | \u000A | \n |
| Retorno de carro | \u000D | \r |
| Tabulación horizontal | \u0009 | \t |
| Tabulación vertical | \u000B | \v |
| Barra invertida | \u005C | \\ |

Cadena: va entre comillas dobles ("esto es una cadena").

Si se desea que no se interpreten los caracteres de escape, se debe anteponer la arroba a la cadena. En general la arroba hace que la cadena sea todo lo que se encuentra entre la primera y la última comilla doble.

Por ejemplo:

```
Console.WriteLine(@"Un \ttabulador\nlínea distinta");
```

Escribirá en pantalla:

```
Un \ttabulador\nlínea distinta
```

Nulo: palabra reservada null. Se usará principalmente para inicializar objetos.

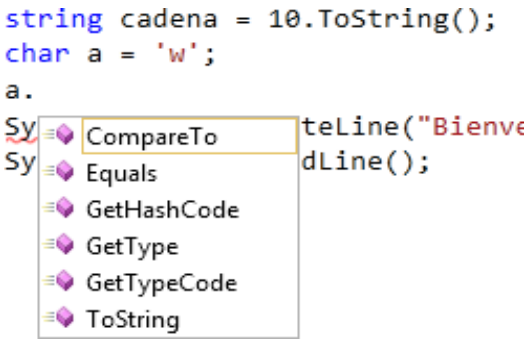
## Tipos de datos y variables

En C# todos los tipos de datos son objetos (herencia común de object). Todos. Es decir, así como en Java existían tipos primitivos (int, char, double, …) estos no existen en C# de forma que cualquier valor o variable es un objeto y se podrá tratar como tal con sus propiedades y sus métodos. Por tanto, el siguiente código es correcto:

```
string cadena = 10.ToString();
```

El propio número 10 es un objeto y por eso admite el operador punto.

Si declaras un char, al usar el operador punto ocurre lo siguiente:




Se despliegan las posibilidades que tengo para el char. Son pocas porque hereda directamente de la clase object. Por tanto, en C# se puede guardar cualquier tipo en un object ya que todo hereda de este.

Sin embargo, los tipos que considerábamos primitivos en Java, en C#, siendo objetos, realmente no se comportan como tales en cierto aspecto. Cuando asignas un entero a otro, tenemos dos variables apuntando (referenciando) al mismo dato si no que se hace copias.

Esto conlleva a que se diferencia en C# dos tipos de variables: **variables valor** y **variables referencia**. Las variables valor se pueden manejar en cuanto a igualdad como los primitivos de Java, directamente contienen el valor, no una referencia al mismo.

Por el contrario, los tipos referencia sí contienen una referencia al dato. En cuanto a tipos base, sólo hay dos por referencia: object y string y este último tiene sobrecargado el operador igual (sí, en C# se pueden sobrecargar operadores) de forma que funciona como en una variable por valor. También tiene sobrecargado el operador == para hacer comparaciones más cómodas.

Declaración de variables: Idéntica a Java.

Ejemplos:

```
byte a;              char car = 'A';
int b, num1;         long num2 = 152, num3 = 998792;
```

En C#, los tipos base realmente son alias (forma distinta de escribirlo. Se puede crear un alias con la directiva using) de tipos del CTS común a todos los lenguajes .Net. La tabla de equivalencias es la siguiente:

| Tipo CTS | Descripción | Bits | Rango de valores | Alias |
|---|---|---|---|---|
| SByte | *Bytes* con signo | 8 | [-128, 127] | **sbyte** |
| Byte | *Bytes* sin signo | 8 | [0, 255] | **byte** |
| Int16 | Enteros cortos con signo | 16 | [-32.768, 32.767] | **short** |
| UInt16 | Enteros cortos sin signo | 16 | [0, 65.535] | **ushort** |
| Int32 | Enteros normales | 32 | [-2.147.483.648, 2.147.483.647] | **int** |
| UInt32 | Enteros normales sin signo | 32 | [0, 4.294.967.295] | **uint** |
| Int64 | Enteros largos | 64 | [-9.223.372.036.854.775.808, 9.223.372.036.854.775.807] | **long** |
| UInt64 | Enteros largos sin signo | 64 | [0-18.446.744.073.709.551.615] | **ulong** |
| Single | Reales con 7 dígitos de precisión | 32 | $[1,5\times10^{-45} - 3,4\times10^{38}]$ | **float** |
| Double | Reales de 15-16 dígitos de precisión | 64 | $[5,0\times10^{-324} - 1,7\times10^{308}]$ | **double** |
| Decimal | Reales de 28-29 dígitos de precisión | 128 | $[1,0\times10^{-28} - 7,9\times10^{28}]$ | **decimal** |
| Boolean | Valores lógicos | 32 | **true**, **false** | **bool** |
| Char | Caracteres Unicode | 16 | ['\u0000', '\uFFFF'] | **char** |
| String | Cadenas de caracteres | Variable | El permitido por la memoria | **string** |
| Object | Cualquier objeto | Variable | Cualquier objeto | **object** |

En C#, se escribirá habitualmente el alias, pero el programador tendrá en cuenta el tipo CTS que está por debajo.

Pero a pesar de ser objetos, los tipos valor como int o float no admiten ser asignados a null. Esto en ocasiones es un inconveniente (imagina una BBDD que no tiene un valor determinado para un entero, debería ser null). Para solventar esto último se puede usar los tipos "nullable" que es exactamente la misma definición que ya conocemos pero acabada en ?. Es decir int?, float?, bool?, etc. De esta forma se puede hacer:

```
int c = null; //Error!!!

int? a = null;

float? b = null;
```

## Constantes

Se definen de la siguiente forma

```
const <tipo_constante> <nombre_constante> = <valor>;
```

Por ejemplo

```
const double pi=3.14159265;
```

Cómo en Java, hay que tener en cuenta el tipo base que se le asigna, para lo cual conviene usar en ciertos casos los sufijos:

```
const float pi=3.14F;
```

Sufijos enteros:

| Sufijo | Tipo del literal entero |
|---|---|
| ninguno | **int** |
| **L** ó **l (no recomendado, confusión con 1)** | **long** |
| **U** ó **u** | **int** |
| **UL, Ul, uL, ul, LU, Lu, lU** ó **lu** | **ulong** |

Sufijos reales:

| Sufijo | Tipo del literal real |
|---|---|
| **F** ó **f** | **float** |
| ninguno, **D** ó **d** | **double** |
| **M** ó **m** | **decimal** |

## Operadores

Son los mismos que en Java. Los resumimos:

### Aritméticos:

```
+    suma
-    resta
*    multiplicación
/    división
%    resto
=    asignación
```

### Lógicos:

```
&&: And (Se puede usar & que evalúa ambos resultados)
||: Or (Se puede usar | que evalúa ambos resultados)
!: not
^:xor
==  "igual que"
!= "distinto de"
<,>,<=,  >=   menor,  mayor,  menor  o  igual,  mayor  o  igual
respectivamente
```

### Bit a bit:

```
& and
| or
~ not (AltGr+4)
^ xor
```

```
<< desplazamiento a la izquierda
>> desplazamietno a la derecha
```

Asignación:

```
=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= , ++, --
```

De concatenación de cadenas y caracteres:

```
+, +=
```

## Preincremento y postincremento

Los operadores ++ y – pueden usarse antes o después de la variable con resultado distinto si se hace en combinación con otra operación. Esto en ocasiones puede ser útil para ciertas tareas. Ejecuta y analiza este ejemplo:

```
int a = 10;

int b = 10;

int c, d;


c = a++;

d = ++b;

Console.WriteLine("a: {0},\tb:{1}", a, b);

Console.WriteLine("c: {0},\td:{1}", c, d);
```

En el caso de c=a++, primero se asigna a c y luego se incrementa a (postincremento). En el segundo, d=++b, primero se incrementa b y luego se asigna el resultado a c(preincremento).

Veamos algún ejemplo práctico para mostrar por pantalla una indexación en 1 o en 0 según interese.:

```
int i = 0;
do {
        Console.WriteLine(i++); ;
} while (i < 5);
Console.WriteLine();
i = 0;
do {
        Console.WriteLine(++i); ;
} while (i < 5);
```

## Conversión de tipos

El casting que aprendimos a usar en Java funciona también en C#:

```
int a;

long b = 13;

a = (int)b;
```

Pero tenemos también métodos de conversión y comandos relacionados que vemos a continuación:

Una forma típica es usar la clase Convert de la siguiente forma:

```
Destino = Convert.ToXXXX(Origen);
```

donde XXXX representa el tipo destino. Convert contiene una serie de métodos estáticos para realizar conversiones. Por ejemplo:

```
long b;

b = Convert.ToUInt32("121");
```

Hay que conocer el tipo CTS para saber qué queremos convertir.

También se puede usar el método Parse que existe de forma estática en algunas de las clases:

```
long b;

b = long.Parse("121");
```

Tiene el mismo resultado que el caso anterior.

La diferencia entre los dos casos es que el Convert primero comprueba que el string no sea null, si lo es devuelve 0. El Parse lanza una excepción ArgumentNullException si el string es nulo.

Para convertir cualquier tipo a cadena se usa el método ToString() que viene heredado ya de object. Es más, en nuestras clases se puede sobreescribir el método para que funcione según nuestros intereses.

Existe también, aunque lo usaremos menos, para hacer conversiones, el operador as:

```
expresión_o _variable as tipo_de_dato
```

Esto devuelve la variable en el nuevo tipo. De todas formas, esto solo vale para tipos por referencia (punteros), ya que no hace exactamente una conversión de tipo si no que se conserva el tipo original, pero fuerza al programa a usarlo como el nuevo tipo. Esto hace que sea más eficiente, pero hay que ser más cuidadoso al usarlo. Es otra manera de realizar un casting. Se aplica en polimorfismo.

## Números aleatorios

Usaremos la clase Random para instanciar un objeto que nos permita generar números aleatorios:

```
Random generador = new Random();
```

El objeto generador dispone entre otros de los siguientes métodos de generación de números aleatorios:

Next(): Devuelve un número int aleatorio positivo.

Next(int max): Devuelve un número aleatorio positivo menor que el valor máximo especificado.

Next(int min, int max): Devuelve un número aleatorio que se encuentra en el intervalo especificado. Incluye min pero no max.

NextDouble(): Devuelve un número de punto flotante de doble precisión que es mayor o igual que 0.0 y menor que 1.0.

Por ejemplo:

```
Random generador = new Random();

double d;

int i;

d = generador.NextDouble(); //Nº real entre 0 y 1 (excluido)

i = generador.Next(1, 7);//Nº entero entre 1 y 7 (excluido, es decir, de 1 a 6)
```

Nunca debe meterse el new Random() en un bucle para generar varios números aleatorios pues usa como semilla el mismo instante temporal (por ser muy rápido) y siempre sacaría el mismo número.

## Declaración mediante var

En C# se pueden hacer declaraciones implícitas no diciendo qué tipo es, pero dejando que sea el compilador quién decida usando el tipo var.

```
var i = 10; // tipado implícito

int i = 10; // tipado explícito
```

No la usaremos mucho pues es necesaria principalmente para gestionar lo que se denominan tipos anónimos que probablemente no veremos por falta de tiempo.

## E/S en Proyectos de Consola

Usaremos para leer datos Console.ReadLine() que es una función que nos devuelve un dato que introduce el usuario de forma similar al nextLine del Scanner de Java. Siempre devuelve un string por lo que se deben hacer las conversiones pertinentes.

Existe también Console.ReadKey() que espera a la pulsación de cualquier tecla y la devuelve como tipo ConsoleKeyInfo en el que no profundizaremos. Esto puede ser útil para establecer interacción con el usuario sin tener eco por pantalla.

Para escribir datos en pantalla usaremos básicamente Console.WriteLine() o Console.Write(). Este último no hace un retorno de carro.

Dentro de una cadena en un Write o WriteLine se pueden usar cualquiera de las secuencias de escape vistas, como por ejemplo el retorno de carro (\n).

Cuando queremos mostrar varias variables dentro de un texto podemos hacer lo siguiente:

• Concatenamos usando los operadores ya vistos y teniendo en cuenta las conversiones necesarias.

• Usamos modificadores con llaves: {0} {1}, … insertadas dentro de la constante cadena. Esos modificadores corresponden con el mismo orden de colocación de las variables al final del WriteLine (similar al funcionamiento del printf).

Ejemplo:

```
int a = 10, b = 3, totalsuma, totalresta;

string moneda;

totalsuma = a + b;

totalresta = a - b;

moneda = "Euros";

Console.WriteLine("El valor {0} sumado con el {1} resulta un total de {2} {3}. Pero dicho valor {0} si se le resta {1} queda un total de {4} {3}\n", a, b, totalsuma, moneda, totalresta);

Console.WriteLine("Para mostrar {llaves}, si no hay ninguna variable para mostrar, no hay problema.");

Console.WriteLine("Si hay alguna, se usan dobles llaves\n Llaves:{{}} Moneda: {0}", moneda);
```

Mediante estas, también se puede dar formato de números decimales y de espacio ocupado por una variable:

{modificador, nºdigitos:.000} Tantos 0 como cifras decimales

```
double pi = 3.14159265;

Console.WriteLine("{0,10:.000}", pi); // También es válido "{0,10:F3}"
```

Existe la función estática String.Format que devuelve la cadena con formato de la misma forma que trabaja WriteLine. Más información sobre formatos:

http://msdn.microsoft.com/es-es/library/txafckwd(v=vs.80).aspx

https://docs.microsoft.com/es-es/dotnet/api/system.string.format?redirectedfrom=MSDN&view=net- 5.0#overloads

Incluso un programador puede definir nuevos formatos mediante los interfaces IFormatProvider e ICustomFormatter como se ve en el segundo enlace de los dos anteriores.

Finalmente hay que comentar que la clase Console dispone de otros métodos que mejoran el manejo y aspecto de la consola. Por ejemplo, SetCursorPosition permite colocar el cursor en cierta coordenada o ForegroundColor y BackgroundColor permite cambiar los colores del texto y fondo de la consola. Esto permite hacer aplicaciones más amigables con el usuario o incluso videojuegos sencillos en consola.

https://docs.microsoft.com/es-es/dotnet/api/system.console?view=net-5.0#common-operations

## Control de flujo

En C# las sentencias de control de flujo son prácticamente idénticas a Java, por lo que las veremos por encima parándonos sólo en aquellos puntos diferenciadores.

Las sentencias selectivas **if**, **if-else** son idénticas a Java.

Las estructuras repetitivas **while**, **do-while** y **for** también son idénticas a las equivalentes de Java. Las cláusulas **break**, **return** y **continue** funcionan igual que en Java.

Existe un bucle for extendido (**foreach**) que veremos más adelante.

## Sentencia switch

Es similar al switch de Java con la salvedad que además de break se pueden poner otras sentencias de ruptura del caso. Además, dispone de más posibilidades. Estructura:

```
switch (<expresión>)
{
        case <valor1>:   <bloque1>
                         <siguienteAcción>
        case <valor2>:   <bloque2>
                         <siguienteAcción>
        ...
        default:         <bloqueDefault>
                         <siguienteAcción>
 }
```

El manejo es igual que en Java, sólo cambia el elemento marcado como <siguienteAcción> colocado tras cada bloque de instrucciones indica qué es lo que ha de hacerse tras ejecutar las instrucciones del bloque que lo preceden.

Puede ser uno de estos tres tipos de instrucciones y es obligatorio ponerlas salvo que en el case no haya nada lo que lo haría pasar al siguiente:

```
goto case <valor_i>;

goto default;

break;
```

Si es un **goto case** indica que se ha de seguir ejecutando el bloque de instrucciones asociado en el switch a la rama del <valor_i> indicado.

Si es un **goto default** indica que se ha de seguir ejecutando el bloque de instrucciones de la rama default.

Si es un **break** indica que se ha de seguir ejecutando la instrucción siguiente al switch, al igual que sucedía en Java.

Además, como ocurría en la versión 7 de Java el switch de C# admite cadenas.

Ejemplo:

```
string nom;
Console.WriteLine("Por favor introduce tu nombre");
nom = Console.ReadLine();
switch (nom)
{
        case "adios":
        case "Adios":
        case "ADIOS":
                Console.WriteLine("Hasta la próxima");
                break;
        case "Curro":
                Console.WriteLine("Acceso denegado, cámbiate el nombre");
                goto case "ADIOS";
        default:
                Console.WriteLine("Hola {0}, bienvenido a este programa", nom);
                break;
}
```

Lo visto aquí es un ejemplo de la potencia de esta sentencia. Pero permite mucho más como aplicación directa de polimorfismo, usar sentencias case condicionales o incluso establecer rangos (desde C# 7). Puedes ver más ejemplos en:

https://www.dotnetperls.com/switch

https://stackoverflow.com/questions/20147879/switch-case-can-i-use-a-range-instead-of-a-one-number

## Métodos

La creación de métodos o funciones en C# es prácticamente igual a la vista el año pasado en Java. La principal diferencia se encuentra en el paso de parámetros y es donde nos vamos a centrar.

Veíamos en Java que los parámetros es una forma de darle datos a la función para trabajar con ellos y, si es necesario, que la función devuelva algún dato mediante la sentencia return.

En C#, además de esta posibilidad de uso de parámetros llamada, paso de **parámetros por valor** o **parámetros de entrada**, existe otra posibilidad y es la de pasarle como parámetro una variable que sí sea modificada. A esto se le llama **parámetros de salida o parámetros por referencia**.

Para pasar un parámetro por referencia simplemente se especifica con la abreviación ref tanto en la definición de la función como en la llamada.

```
static void suma (int a, int b, ref int c)
{
        c=a+b;
}

static void Main()
{
        int res=0;
        suma(12, 34, ref res);
        Console.WriteLine("Resultado: {0}", res);
}
```

En el caso anterior es necesario inicializar res en el programa principal si no se provoca un error de compilación indicando que la variable no está inicializada. Esto se puede evitar usando una variante del parámetro por referencia denominado out que obliga a que el o los parámetros por referencia sean inicializados dentro de la función por lo que no es obligatorio inicializarlos antes.

```
static void suma (int a, int b, out int c)
{
        c=a+b;
}

static void Main()
{
        int res;
```

```
        suma(12, 34, out res);
        Console.WriteLine(("Resultado: {0}", res);
}
```

Ahora no da error de compilación porque el out implica que res se inicializará dentro de la función.

### Parámetros con nombre y opcionales

En C# está permitido pasar los parámetros en cualquier orden si se especifica el nombre del parámetro.

También si se inicializa un parámetro con un valor, dicho parámetro se convierte en opcional. Veamos un ejemplo de ambas cosas. Sea la función:

```
        static void suma10(int a, int b = 10)
        {
                Console.WriteLine(a + b);
        }
```

Puede ser llamada desde el Main de estas tres formas:

```
        suma10(3, 4);
        suma10(b:4,a:3);
        suma10(3);
```

## Directivas del preprocesador (o del compilador)

En ocasiones conviene darle ciertas indicaciones al compilador para que varíe la forma en que realiza la compilación. Para ello se usan las directivas del preprocesador. Lo que hacen es "preprocesar" el código antes de ser compilado.

El formato de dar esas indicaciones es:

#<nombre de la directiva> <valor de la directiva>

Veamos algunos ejemplos que se ampliarán durante el curso de ser necesarios. La principal utilidad es indicar que partes del código serán compiladas y cuáles no. Para ello se usan las siguientes directivas.

### Directiva define

El formato es:

#define <identificador>

Esta directiva indica si cierto identificador está definido. Por ejemplo

#define PRUEBA

No es obligatorio, pero es costumbre escribir los identificadores del preprocesador con mayúsculas. Es obligatorio que las directivas **define** se encuentren al **principio del código**.

### Directivas de compilación condicional

Son las directivas que nos permiten decidir qué se compila y qué no se compila en nuestro código (normalmente esto va asociado a que tenemos código de depuración que en algunas ocasiones interesa compilar y en otras no). La estructura es

```
#if <condición1>
        <código1>
#elif <condición2>
        <código2>
...
#else
        <códigoElse>
#endif
```

**elif** es **equivalente** a **else if**

Escribe y ejecuta el siguiente ejemplo:

```
#define PRUEBA
using System;
class A
{
        public static void Main()
        {
                #if PRUEBA
                        Console.Write ("Esto es una prueba");
                        #if TRAZA
                        Console.Write(" con traza");
                        #elif !TRAZA
                                Console.Write(" sin traza"); #endif
                        #endif
        }
}
```

Prueba ahora a definir el identificador TRAZA y volver a ejecutarlo.

Como veremos, el carácter ! Indica NOT. Otras posibilidades:

**!** para "not"

**&&** para "and"

**||** para "or"

**==** operadores relacionales de igualdad

**!=** desigualdad

**()** paréntesis

**true** y **false**

Por ejemplo:

```
#if TRAZA // Se cumple si TRAZA está definido.
#if TRAZA==true // Ídem al ejemplo anterior aunque con una sintaxis menos cómoda
#if !TRAZA // Sólo se cumple si TRAZA no está definido.
#if TRAZA==false //Ídem al ejemplo anterior pero con una sintaxis menos cómoda
```

```
#if TRAZA == PRUEBA        // Solo se cumple si tanto TRAZA como PRUEBA están definidos o si ninguno lo está.
#if TRAZA != PRUEBA        // Solo se cumple si TRAZA esta definido y PRUEBA no o viceversa
#if TRAZA && PRUEBA // Solo se cumple si están definidos TRAZA y PRUEBA. #if TRAZA || PRUEBA // Solo se cumple si están definidos TRAZA o PRUEBA. #if false // Nunca se cumple (por lo que es absurdo ponerlo)
```

Existen más directivas que se verán si fueran necesarias.