

# Introducción a C# .NET

## Módulo 1

# Sintaxis y semántica del lenguaje C#

# Variables y constantes

## ¿Qué es una variable?

Una **variable** representa una posición en memoria en la que se puede almacenar un valor de un determinado tipo. A esa posición se le da cierto nombre. Los nombres de las variables tienen ciertas reglas, por ejemplo, no puedo tener un nombre de variable formada por dos palabras y un espacio en el medio.

Para poder utilizar la variable sólo hay que definirla indicando cuál será su nombre y cuál será el tipo de datos que podrá almacenar.

Se hace siguiendo la siguiente sintaxis:

```
<tipoVariable> <nombreVariable>;
```

Una variable puede ser definida dentro de una definición de clase, en cuyo caso se la denominaría **campo**. También puede definirse como una **variable local** a un método, que es una variable definida dentro del código del método a la que sólo puede accederse desde dentro de dicho código.

Otra opción es definirla como **parámetro** de un método. Estas son variables que almacenan los valores de llamada al método y que, como las variables locales, sólo pueden ser accedidas desde código ubicado dentro del método.

El siguiente ejemplo muestra cómo definir variables de todos estos casos:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;
```

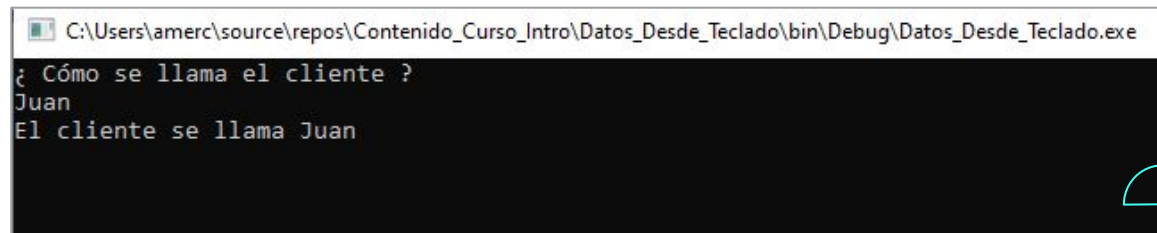
...

```
namespace Datos_Desde_Teclado
{
    class CapturaDatos
    {
        static void Main(string[] args)
        {
            string Nombre; // Declaración de la variable Nombre, de tipo string
            Console.WriteLine("¿ Cómo se llama el cliente ?");

            //Captura del valor ingresado y asignación de ese valor a variable Nombre,
            //a través de Console.ReadLine();
            Nombre = Console.ReadLine();

            //Mostrar en la posición 0 la variable Nombre
            Console.WriteLine("El cliente se llama {0}", Nombre);
            Console.ReadKey();
        }
    }
}
```

En este sencillo programa, el compilador reservará memoria para la variable **Nombre**. En su ejecución, primero preguntará por el nombre del cliente y después que se escriba y al presionar *Enter*, mostrará cómo se llama:



```
C:\Users\amerc\source\repos\Contenido_Curso_Intro\Datos_Desde_Teclado\bin\Debug\Datos_Desde_Teclado.exe
¿ Cómo se llama el cliente ?
Juan
El cliente se llama Juan
```

También podemos inicializar el valor de una variable en el momento de declararla, sin que esto suponga un obstáculo para poder modificar después su valor:

```
int num = 10;
```

## ¿Qué es una constante?

Las **constantes** hacen que el compilador reserve **un espacio de memoria para almacenar un dato**, pero en este caso **ese dato es siempre el mismo** y no se puede modificar durante la ejecución del programa.

**Ejemplo:** uno claro, sería almacenar el valor de Pi en una constante para no tener que poner el número en todas las partes donde lo podamos necesitar. Otros ejemplos de constantes son, la cantidad de días de la semana, la cantidad de meses del año, etc.

Las constantes se declaran de un modo similar a las variables, basta con añadir la palabra **const** en la declaración.

Veamos un ejemplo en la próxima pantalla.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

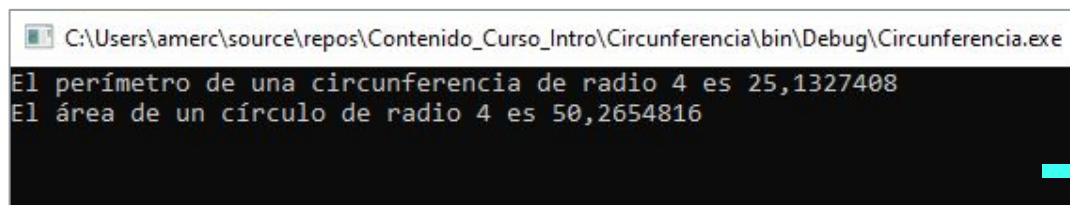
namespace Circunferencia
{
    class Perimetro
    {
        static void Main(string[] args)
        {
            const double PI = 3.1415926; // Declaración de constante PI, de tipo double
            double Radio = 4; //Declaración e inicialización de constante Radio, de tipo double

            Console.WriteLine("El perímetro de una circunferencia de radio {0} es {1}", Radio, 2 * PI * Radio);
            Console.WriteLine("El área de un círculo de radio {0} es {1}", Radio, PI * Math.Pow(Radio, 2));

            Console.ReadKey();
        }
    }
}
```



La salida en la consola de este programa sería la siguiente:



```
C:\Users\amerc\source\repos\Contenido_Curso_Intro\Circunferencia\bin\Debug\Circunferencia.exe
El perímetro de una circunferencia de radio 4 es 25,1327408
El área de un círculo de radio 4 es 50,2654816
```

# Tipos de dato

## Introducción

El sistema de **tipos de datos** suele ser la parte más importante de cualquier lenguaje de programación. Para que una aplicación sea eficiente con el menor consumo posible de recursos, es esencial el uso correcto de los distintos tipos de datos. Muchos de los lenguajes orientados a objetos proporcionan los tipos agrupándolos de dos formas: los **tipos primitivos del lenguaje**, como números o cadenas, y el resto de **tipos creados a partir de clases que instanciados serán objetos**.

Esto genera muchas dificultades, ya que los tipos primitivos no son y no pueden tratarse como *objetos*, es decir, no se pueden derivar y no tienen nada que ver unos con otros. Sin embargo, en C# (más propiamente en **.NET Framework**) se cuenta con un **sistema de tipos unificado, el CTS (*Common Type System*)**, que proporciona todos los tipos de datos como clases derivadas de la **clase de base `System.Object`** (incluso los literales pueden tratarse como objetos).

Es importante aclarar que: *“hacer que todos los datos que va a manejar un programa sean objetos, puede provocar que baje el rendimiento de la aplicación (performance y uso de recursos)”*.

Para solventar este problema, .NET Framework divide los tipos en dos grandes grupos: los **tipos valor** y los **tipos referencia**.

Cuando se declara una variable que es de un **tipo valor** se está reservando un espacio de memoria en la pila (*stack*) para que almacene los datos reales que contiene esta variable.

Por ejemplo, en la declaración:

```
int num = 10;
```

Se está reservando un espacio de 32 bits en la pila (una variable de tipo `int` es un objeto de la clase `System.Int32`), en los que se almacena el 10, que es el contenido de la variable.

Esto hace que la variable “num” se pueda tratar directamente como si fuera de un tipo primitivo en lugar de un objeto, mejorando notablemente el rendimiento. Como consecuencia, una variable de **tipo valor** nunca puede contener el valor `null` (referencia nula).

Durante la ejecución de un programa, la memoria se distribuye en tres bloques: la **pila** (*stack*), el **montón** (*heap*) y la **memoria global**.

La **pila** es una estructura en la que los elementos se van apilando de modo que el último elemento en entrar en la pila es el primero en salir (estructura LIFO, o sea, *Last In First Out*).

El **montón** es un bloque de memoria contiguo en el cual la memoria no se reserva en un orden determinado como en la pila, sino que se va reservando aleatoriamente según se va necesitando. Cuando el programa requiere un bloque del montón, este se sustrae y se retorna un puntero (variable que contiene una dirección de memoria), al principio del mismo.

La **memoria global** es el resto de memoria de la máquina que no está asignada ni a la pila ni al montón, y es donde se colocan el método `main` y las funciones que éste invocará.

En el caso de una variable que sea de un **tipo referencia**, lo que se reserva es un espacio de memoria en el montón para almacenar el valor, pero lo que se devuelve internamente es una referencia al objeto, es decir, un puntero a la dirección de memoria que se ha reservado. En este caso, una variable de un tipo referencia si puede contener una referencia nula (`null`).



¿Por qué esta “separación” de tipos y posiciones de memoria?

Porque una variable de un tipo valor funcionará como un tipo primitivo siempre que sea necesario, pero podrá funcionar también como un tipo referencia, es decir como un objeto, cuando se necesite que sea un objeto.

Hacer esto en otros lenguajes, como Java, es imposible, dado que los tipos primitivos en Java no son objetos.

Más adelante ampliaremos estos conceptos.



Los **tipos de dato básicos** son ciertos tipos de dato tan comúnmente utilizados en la escritura de aplicaciones que en C# se ha incluido una sintaxis especial para tratarlos.

Por ejemplo, para representar números enteros de 32 bits con signo se utiliza el tipo de dato **System.Int32** definido en la BCL, aunque a la hora de crear un objeto **a**, de este tipo, que represente el valor 2, se usa la siguiente sintaxis:

```
System.Int32 a = 2;
```

Para este tipo, que es de uso muy frecuente, también se ha predefinido en C# el alias **int**, por lo que la definición de variable anterior queda así de compacta:

```
int a = 2;
```



**System.Int32** no es el único tipo de dato básico incluido en C#. En el espacio de nombres **System** se han incluido otros tipos de dato como los siguientes:

Tipo	Descripción	Bits	Rango de valores	Alias
<b>SByte</b>	Bytes con signo	8	[-128, 127]	Sbyte
<b>Byte</b>	Bytes sin signo	8	[0, 255]	byte
<b>Int16</b>	Enteros cortos con signo	16	[-32.768, 32.767]	short
<b>UInt16</b>	Enteros cortos sin signo	16	[0, 65.535]	ushort
<b>Int32</b>	Enteros normales	32	[-2.147.483.648, 2.147.483.647]	Int
<b>UInt32</b>	Enteros normales sin signo	32	[0, 4.294.967.295]	uint
<b>Int64</b>	Enteros largos	64	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]	Long

Tipo	Descripción	Bits	Rango de valores	Alias
<b>UInt64</b>	Enteros largos sin signo	64	[0-18.446.744.073.709.551.615]	Ulong
<b>Single</b>	Reales con 7 dígitos de precisión	32	[1,5×10-45 - 3,4×1038]	Float
<b>Double</b>	Reales de 15-16 dígitos de precisión	64	[5,0×10-324 - 1,7×10308]	double
<b>Decimal</b>	Reales de 28-29 dígitos de precisión	128	[1,0×10-28 - 7,9×1028]	decimal
<b>Boolean</b>	Valores lógicos	32	<b>true, false</b>	Bool
<b>Char</b>	Caracteres Unicode	16	['\u0000', '\uFFFF']	Char
<b>String</b>	Cadenas de caracteres	Variable	El permitido por la memoria	String
<b>Object</b>	Cualquier objeto	Variable	Cualquier objeto	Object

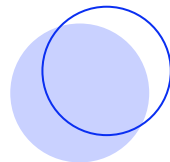


## Tipos de dato básicos

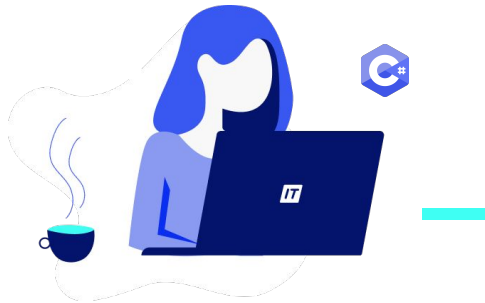
Pese a su sintaxis especial, en C# los tipos básicos **son tipos del mismo nivel que cualquier otro tipo** del lenguaje. Es decir, **todos heredan del tipo base que es `System.Object`**, y pueden tratarse como objetos de dicha clase por cualquier método que espere un **`System.Object`**.

Esto es muy útil para el diseño de rutinas genéricas que admitan parámetros de cualquier tipo, y es una ventaja importante de C# frente a lenguajes similares, como Java, donde los tipos básicos no son considerados objetos.

El valor que por omisión se le da a los campos de tipos básicos, consiste en poner a cero todo el área de memoria que ocupen. Esto se traduce en que los campos de tipos básicos numéricos se inicializan por omisión con el valor 0, los de tipo **`bool`** lo hacen con **`false`**, los de tipo **`char`** con **`'\u0000'`** (carácter nulo), y los de tipo **`string`** y **`object`** con **`null`**.



Para los casos de los tipos de variables **numéricas** se puede especificar (durante la asignación de un valor), el *tipo* de dato numérico de ese *valor* mediante el uso de **sufijos**.



Los sufijos para los valores literales de los distintos tipos de datos numéricos son los siguientes:

- **L** (mayúscula o minúscula): long ó ulong, por este orden;
- **U** (mayúscula o minúscula): int ó uint, por este orden;
- **UL** ó **LU** (independientemente de que esté en mayúsculas o minúsculas): ulong;
- **F** (mayúscula o minúscula): single;
- **D** (mayúscula o minúscula): double;
- **M** (mayúscula o minúscula): decimal;

# Palabras reservadas

**Las palabras reservadas no pueden ser usadas como nombre de variable.** En el siguiente slide se muestran algunas de las palabras reservadas más relevantes del lenguaje C#.

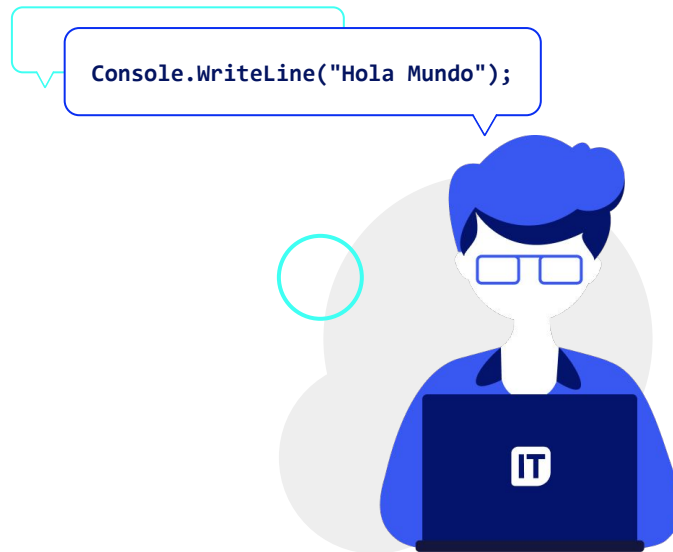


abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
do	double	else	enum	event
explicit	extern	false	finally	fixed
float	for	foreach	goto	if
implicit	in	value	int	interface
internal	long	new	null	out
override	params	private	protected	public
readonly	ref	return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

# Construcciones del lenguaje

## Introducción

Cuando se desarrolla un programa, luego de declarar las variables, constantes, clases, etc., se debe especificar qué hacer con/entre estos elementos. Para ello iremos escribiendo **líneas de código** (para C# cada línea de código termina con un “;” – punto y coma).



## Sentencias

**Una sentencia es una orden** que se le da al programa para realizar una tarea específica, por ejemplo, mostrar un mensaje en la pantalla, declarar una variable (para reservar espacio en memoria), inicializarla, llamar a un método, etc.

Para C# cada sentencia termina con un “;” (punto y coma). Normalmente, las sentencias se ponen unas debajo de otras, aunque sentencias cortas pueden colocarse en una misma línea. He aquí algunos ejemplos de sentencias:

- `int x = 10;`
- `using System.IO;`
- `Console.WriteLine("Hola Mundo");`

En C#, los caracteres espacio en blanco se pueden emplear libremente. Es muy importante para la legibilidad de un programa la colocación de unas líneas debajo de otras empleando tabuladores. El editor del IDE nos ayudará plenamente en esta tarea sin apenas percibirlo, pues irá “indentando automáticamente” las líneas de código.



## Bloques de código

Un bloque de código es un grupo de sentencias que se comportan como una unidad. Se usan para indicar el comienzo y fin de un conjunto de líneas (sentencias) de código. Por lo general, se los usa para delimitar *namespaces*, clases, estructuras, enumerados y métodos.

Un bloque de código está limitado por las llaves de apertura “{” y cierre “}”. Veamos un ejemplo:

```
static void Main() {  
    Linea1;  
    Linea2;  
}
```

Cada una de esas líneas implica una **expresión** que el compilador de C# analizará, verificando que su sintaxis sea la correcta. Para ello disponemos de los **operadores**.



**¡Sigamos  
trabajando!**