

# Enciclopedia de Microsoft®

# Visual C#™

## 2ª EDICIÓN

- Entornos de desarrollo:
  - Visual Studio 2005
  - o
  - Visual C# 2005 Express
  - y
  - Visual Web Developer 2005 Express
- 
- Resumen del lenguaje C#
- 
- Programación orientada a objetos
- 
- Aplicaciones con interfaz gráfica
- 
- Barras de herramientas
- 
- Cajas de diálogo
- 
- Tablas y árboles
- 
- Dibujar y pintar



- Interfaz para múltiples documentos
- 
- Construcción de controles
- 
- Programación con hilos
- 
- Acceso a una base de datos
- 
- Interacción con Office
- 
- Páginas Web
- 
- Formularios Web
- 
- Servicios Web
- 
- Seguridad de aplicaciones ASP.NET
- 
- AJAX
- 
- Móviles

**Fco. Javier Ceballos**

**Alfaomega**  **Ra-Ma®**



Incluye CD-ROM con  
Microsoft .NET Framework  
SDK, EDI y las aplicaciones  
contenidas en el libro

cación, y si no, puede requerir guardar la aplicación en cualquier instante ejecutando la orden *Guardar todo* del menú *Archivo*.

Si desplegamos el menú *Archivo*, nos encontraremos, además de con la orden *Guardar todo*, con dos órdenes más: *Guardar nombre-fichero* y *Guardar nombre-fichero como...* La orden *Guardar nombre-fichero* guarda en el disco el fichero actualmente seleccionado y la orden *Guardar nombre-fichero como...* realiza la misma operación, y además nos permite cambiar el nombre, lo cual es útil cuando el fichero ya existe.

No es conveniente que utilice los nombres que Visual C# asigna por defecto, porque pueden ser fácilmente sobrescritos al guardar aplicaciones posteriores.

## Verificar la aplicación

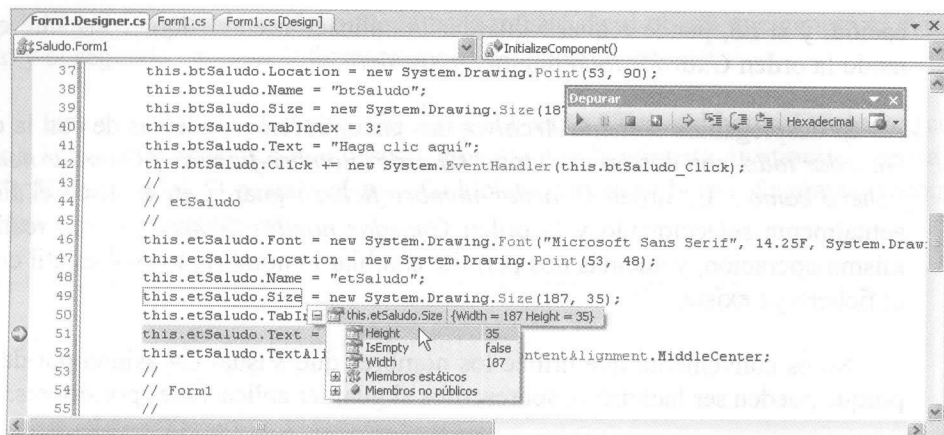
Para ver cómo se ejecuta la aplicación y los resultados que produce, hay que seleccionar la orden *Iniciar sin depurar* del menú *Depurar* o pulsar *Ctrl+F5*.

Si durante la ejecución encuentra problemas o la solución no es satisfactoria y no es capaz de solucionarlos por sus propios medios, puede utilizar, fundamentalmente, las órdenes *Paso a paso por instrucciones* (*F11*), *Paso a paso por procedimientos* (*F10*), *Insertar o quitar un punto de interrupción* (*F9*), todas ellas del menú *Depurar*, para hacer un seguimiento paso a paso de la aplicación, y las órdenes del menú *Depurar > Ventanas*, para observar los valores que van tomando las variables y expresiones de la aplicación.

La orden *Paso a paso por instrucciones* permite ejecutar cada método de la aplicación paso a paso. Esta modalidad se activa y se continúa pulsando *F11*. Si no quiere que los métodos invocados a su vez por el método en ejecución se ejecuten línea a línea, sino de una sola vez, utilice la tecla *F10* (*Paso a paso por procedimientos*). Para detener la depuración pulse las teclas *Mayús+F5*.

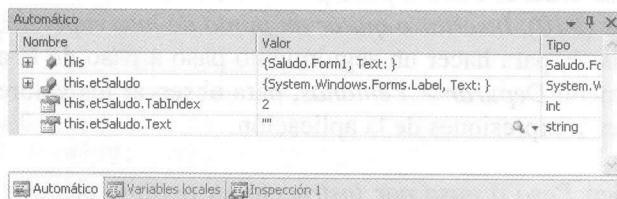
La orden *Insertar o quitar un punto de interrupción* (*F9*) permite colocar una pausa en cualquier línea. Esto permite ejecutar la aplicación hasta la pausa en un solo paso (*F5*), y ver en la ventana *Automático* los valores que tienen las variables en ese instante. Para poner o quitar una pausa, se coloca el cursor donde se desea que tenga lugar dicha pausa y se pulsa *F9*, o bien se hace clic con el ratón sobre la barra situada a la izquierda del código.

Alternativamente al menú de depuración, puede utilizar la barra de herramientas de depuración. La figura siguiente muestra esta barra dentro de la ventana de código en un proceso de depuración. La línea de código sombreada es la siguiente sentencia a ejecutar.



También puede utilizar el ratón para arrastrar el puntero de ejecución (observe la flecha en el margen izquierdo de la ventana anterior) a otro lugar dentro del mismo método con la intención de alterar el flujo normal de ejecución.

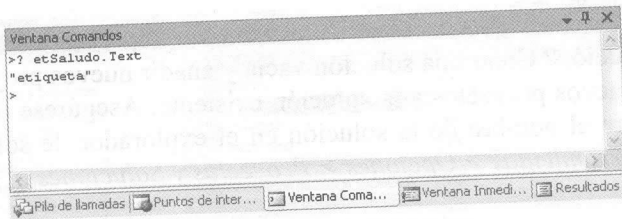
Durante el proceso de depuración, puede ver en la ventana *Automático* los valores de las variables y expresiones que desee. Además, en la ventana *Inspección* puede escribir la expresión que desea ver.



También, puede seleccionar en la ventana de código la expresión cuyo valor quiere inspeccionar y ejecutar *Inspección rápida...* Una forma más rápida de hacer esto último es situando el puntero del ratón sobre la expresión; le aparecerá una etiqueta con el valor, como se puede observar en la ventana de código anterior.

Así mismo, según se observa en la figura siguiente, puede ejecutar en la *ventana de comandos* cualquier sentencia de una forma inmediata. Para mostrar u ocultar esta ventana ejecute la orden *Otras ventanas > Ventana de comandos* del menú *Ver*. El resultado del ejemplo mostrado es el contenido de la propiedad *Text* de la etiqueta *etSaludo* (observe el uso del símbolo ?).

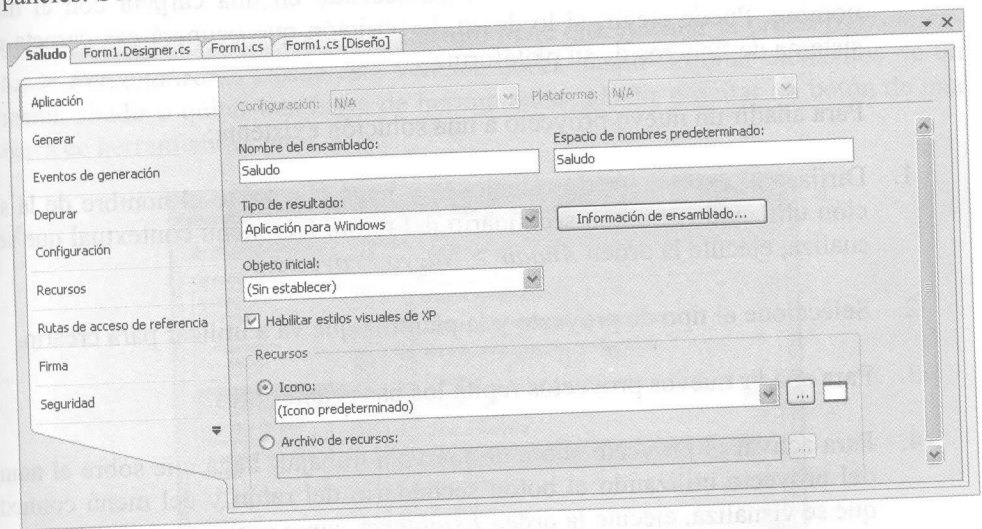




Una vez iniciada la ejecución de la aplicación, si se pulsa la tecla *F5*, la ejecución continúa desde la última sentencia ejecutada en un método hasta finalizar ese método o hasta otro punto de parada.

## Propiedades del proyecto

Para establecer las propiedades del proyecto actual hay que ejecutar la orden *Proyecto > Propiedades de nombre-proyecto...* Se le mostrará una ventana con varios paneles. Seleccione el deseado y modifique las propiedades que considere.



## Crear soluciones de varios proyectos

Una solución agrupa uno o más proyectos. Por omisión, cuando se crea un nuevo proyecto, en la misma carpeta física se crea la solución (fichero con extensión *.sln*) a la que pertenece, con el mismo nombre que el proyecto. Esta solución permite que los ficheros que forman parte del proyecto se almacenen bajo una estructura de directorios que facilite su posterior localización así como las tareas de compartir la solución con otros desarrolladores de un supuesto equipo.



¿Qué tenemos que hacer si necesitamos agrupar varios proyectos bajo una misma solución? Crear una solución vacía y añadir nuevos proyectos a la solución o añadir nuevos proyectos a la solución existente. Asegúrese de que se va a mostrar siempre el nombre de la solución en el explorador de soluciones. Para ello, ejecute *Herramientas > Opciones > Proyectos y Soluciones > Mostrar siempre la solución*.

Para crear una nueva solución vacía:

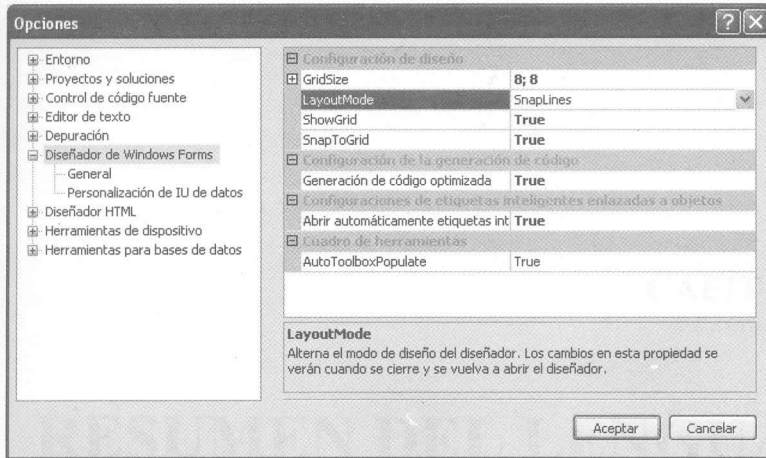
1. Ejecute la orden *Archivos > Nuevo > Proyecto*.
2. Como tipo de proyecto, seleccione *Otros Tipos de Proyectos*.
3. Y como plantilla, seleccione *Solución en Blanco*.
4. Finalmente, introduzca el nombre que desea dar a la solución. Se creará un fichero *.sln* con el nombre dado, almacenado en una carpeta con el mismo nombre. Puede elegir, si lo desea, la posición que ocupará esa carpeta en el sistema de ficheros de su plataforma.

Para añadir un nuevo proyecto a una solución existente:

1. Dirijase al explorador de soluciones y haga clic sobre el nombre de la solución utilizando el botón secundario del ratón. Del menú contextual que se visualiza, ejecute la orden *Añadir > Nuevo Proyecto...*
2. Seleccione el tipo de proyecto y la plantilla que va a utilizar para crearlo.
3. Para añadir nuevos proyectos repita los pasos anteriores.
4. Para activar el proyecto sobre el que va a trabajar, haga clic sobre el nombre del proyecto utilizando el botón secundario del ratón y del menú contextual que se visualiza, ejecute la orden *Establecer como proyecto de inicio*.

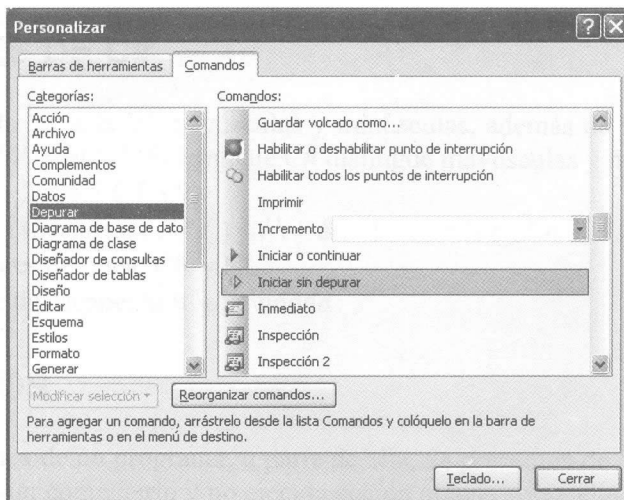
## Opciones del EDI

Para mostrar la ventana que observa en la figura siguiente tiene que ejecutar la orden *Herramientas > Opciones...* Desde esta ventana podrá establecer opciones para el entorno de desarrollo, para los proyectos y soluciones, para el diseñador, para el depurador, etc. Por ejemplo, para que el depurador sólo navegue a través del código escrito por el usuario, no sobre el código añadido por los asistentes, tiene que estar activada la opción "habilitar sólo mi código"; *Herramientas > Opciones > Depuración > General > Habilitar sólo mi código*.



## Personalizar el EDI

Para personalizar el entorno de desarrollo tiene que ejecutar la orden *Herramientas > Personalizar...* Desde esta ventana podrá añadir o quitar elementos de un menú, añadir o quitar una barra de herramientas, añadir o quitar un botón de una barra de herramientas, etc.



## CAPÍTULO 3

©F.J.Ceballos/RA-MA

# RESUMEN DEL LENGUAJE

---

En este capítulo veremos, a modo de resumen, los elementos que aporta C# para escribir un programa, puesto que ya fueron expuestos ampliamente en mi otro libro *Microsoft C# – Curso de programación*. Lo que se va a exponer en este capítulo lo irá utilizando en menor o mayor medida en los capítulos sucesivos. Por lo tanto, límitese ahora simplemente a realizar un estudio con el fin de informarse de los elementos con los que contamos.

## CARACTERES DE C#

- Letras de la ‘a’ a la ‘z’ mayúsculas y minúsculas, además de las letras acentuadas, la ‘ñ’, y el ‘\_’. El lenguaje C# distingue mayúsculas y minúsculas.
- Dígitos: 0 1 2 3 4 5 6 7 8 9.
- Caracteres especiales: . : ' " ( ) < / \ + & ^ \* - = > % @ ! # \$
- Terminadores de línea: CR y NL.
- Secuencias de escape: \n \t \r \a \udddd

## COMENTARIOS

Cuando una línea de un programa, o parte de ella, va precedida de // C# interpreta esa línea como un comentario y no ejecuta acción alguna sobre ella. También, una o más líneas delimitadas por los caracteres /\* y \*/ se considerarán un comentario. Por ejemplo:

```
/* Cálculo de  
 * la velocidad media  
 */
```

```
suma = 0; // Se inicia la variable suma con el valor 0
```



TIPOS

Los tipos en C# se clasifican en: tipos *valor* y tipos *referencia*. Una variable de un tipo *valor* almacena directamente un valor (datos en general), mientras que una variable de un tipo *referencia* lo que permite almacenar es una referencia a un objeto (posición de memoria donde está el objeto). Por ejemplo:

```
int suma = 0;           // suma almacena un entero.
string cadena = null;  /* cadena permitirá almacenar una
                        referencia a un objeto string. */
```

La tabla siguiente resume los tipos intrínsecos en C#.

Tipo C#	Estructura C#	Bytes	Rango de valores
bool	Boolean	?	true y false
byte	Byte	1	0 a 255
char	Char	2	0 a 65535 (U+0000 a U+ffff)
DateTime	DateTime	8	1/Enero/1 a 31/Diciembre/9999 00:00:00 AM a 11:59:59 PM
decimal	Decimal	16	+/-79228162514264337593543950335 ó +/-7.9228162514264337593543950335E+28
double	Double	8	+/-1.79769313486231570E+308
int	Int32	4	-2147483648 a +2147483647
long	Int64	8	-9223372036854775808 a +9223372036854775807
object	Object	4	Cualquier tipo puede ser almacenado en una variable de tipo <b>Object</b>
sbyte	SByte	1	-128 a 127
short	Int16	2	-32768 a 32767
float	Single	4	+/-3.4028235E+38
string	String	?	0 a 2 billones de caracteres UNICODE
uint	UInt32	4	0 a 4294967295
ulong	UInt64	8	0 a 18446744073709551615
ushort	UInt16	2	0 a 65535
Estructuras			Tipos valor definidos por el usuario

(? = depende de la plataforma de desarrollo)

Todos los tipos primitivos expuestos tienen una estructura (variable de tipo **struct**) de datos asociada; por ejemplo, el tipo **double** es un alias de **System.Double** (estructura **Double** del espacio de nombres **System**), **char** es un alias de **System.Char**, **bool** es un alias de **System.Boolean**, **int** es un alias de **System.Int32**, etc. Por lo tanto, un dato de un tipo primitivo es un objeto.

## Tipos enumerados

Los tipos enumerados son tipos definidos por el usuario. Para crearlos se utiliza la palabra **enum**. Por ejemplo:

```
class Test
{
    enum día
    {
        lunes,
        martes,
        miércoles,
        jueves,
        viernes,
        sábado,
        domingo
    };

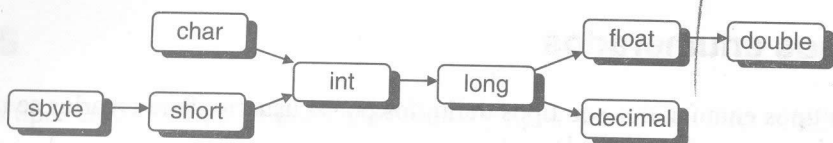
    static void Main(string[] args)
    {
        día díaSem = día.jueves;
        if (díaSem == día.domingo)
            System.Console.Write("fiesta");
        else
            System.Console.WriteLine(díaSem); // escribe 3
    }
}
```

Los valores con los que se forma un tipo enumerado se corresponden con las constantes enteras 0, 1, 2, etc. Según esto, en el ejemplo anterior *lunes* se corresponde con 0, *martes* con 1, *miércoles* con 2, etc. Análogamente al tipo *días*, puede definir cualquier otro tipo enumerado de datos. Esto, en algunas ocasiones, ayudará a escribir un código más legible.

## CONVERSIÓN ENTRE TIPOS PRIMITIVOS

Cuando C# tiene que evaluar una expresión en la que intervienen operandos de diferentes tipos, primero convierte, sólo para realizar las operaciones solicitadas, los valores de los operandos al tipo del operando cuya precisión sea más alta. Cuando se trate de una asignación, convierte el valor de la derecha al tipo de la variable de la izquierda siempre que no haya pérdida de información; en otro caso, C# exige que la conversión se realice explícitamente.

La figura siguiente resume los tipos con signo, colocados de izquierda a derecha de menos a más precisos; las flechas indican las conversiones implícitas permitidas:



C# permite una conversión explícita (conversión forzada) del tipo de una expresión mediante una construcción denominada *cast*, que tiene la forma:

(tipo) expresión

Cualquier valor de un tipo entero o real puede ser convertido explícitamente a o desde cualquier tipo numérico. No se pueden realizar conversiones entre los tipos enteros o reales y el tipo **bool**.

## LITERALES

Un literal en C# puede ser: un entero, un real, un valor booleano, un carácter, una cadena de caracteres, una fecha y hora, y **null**.

- El lenguaje C# permite especificar un literal entero en base 10 y 16. Por ejemplo:

```

256      número decimal 256
0x100    número decimal 256 expresado en hexadecimal
  
```

- Un literal real está formado por una parte entera, seguido por un punto decimal, y una parte fraccionaria. También se permite la notación científica, en cuyo caso se añade al valor una *e* o *E*, seguida por un exponente positivo o negativo. Por ejemplo:

```

-17.24
27e-3
  
```

Una constante real tiene siempre tipo **double**, a no ser que se añada a la misma una *f* o *F*, en cuyo caso será de tipo **float**, o *m* o *M*, en cuyo caso es de tipo **decimal**.

- Los literales de un solo carácter son de tipo **char**. Este tipo de literales está formado por un único carácter encerrado entre *comillas simples*. Una secuencia de escape es considerada como un único carácter. Ejemplo:

```

' '      espacio en blanco
'x'      letra minúscula x
'\n'     retorno de carro más avance de línea
'\u0007' pitido
  
```



- Un literal de cadena de caracteres es una secuencia de caracteres encerrados entre comillas dobles. Las cadenas de caracteres en C# son objetos de tipo **string** (estructura **System.String**). Esto es, cada vez que en un programa se utilice un literal de caracteres, C# crea de forma automática un objeto **String** con el valor del literal.
- Los literales fecha y hora son de tipo **string** y pueden ser convertidos en un objeto **DateTime** utilizando el método **Parse** de éste. Este tipo de literales está formado por una fecha (mes, día, año separados por / o -) y una hora (horas, minutos, segundos separados por dos puntos, más AM o PM si el formato es de 12 horas). A continuación se muestra un ejemplo:

```
string sFechaHora = "15/08/2010 06:25:07 PM";
```

```
DateTime fechaHora = DateTime.Parse(sFechaHora);
Console.WriteLine("Fecha y hora: {0}", fechaHora);
System.Console.WriteLine(fechaHora.ToString());
System.Console.WriteLine(" - {0}", fecha
```

```
// Otra forma de construir un objeto DateTime:
```

```
DateTime fechaHora = new System.DateTime(2010,08,15,18,25,7);
```

## IDENTIFICADORES

Los identificadores son nombres dados a tipos, literales, variables, clases, interfaces, métodos, espacios de nombres y sentencias de un programa. La sintaxis para formar un identificador es la siguiente:

```
{letra|_} [{letra|dígito|_}]...
```

Pueden tener cualquier número de caracteres (realmente pueden tener hasta 1023 caracteres).

## PALABRAS CLAVE

Las palabras clave son identificadores predefinidos que tienen un significado especial para el compilador C#. Por lo tanto, un identificador definido por el usuario no puede tener el mismo nombre que una palabra clave. El lenguaje C# tiene las siguientes palabras clave:

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	is	lock	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
volatile	while			

Las palabras clave deben escribirse siempre en minúsculas, tal y como están.

## DECLARACIÓN DE CONSTANTES SIMBÓLICAS

Declarar una constante simbólica significa decirle al compilador C# el nombre de la constante y su valor. Esto se hace utilizando el calificador **const**.

```
const tipo identificador = cte[, identificador = cte.]...
```

Un ejemplo puede ser el siguiente:

```
const double cte3 = 3.1415926;
```

## DECLARACIÓN DE VARIABLES

Una variable representa un espacio de memoria para almacenar un valor de un determinado tipo. La sintaxis para declarar una variable es la siguiente:

```
tipo identificador[, identificador]...
```

En el ejemplo siguiente se declaran tres variables de tipo **short**, una variable de tipo **int** y dos variables de tipo **string**:

```
class CElementosCSharp
{
    short día, mes, año = 2010;

    void Test()
    {
        int contador = 0;
        string Nombre = "", Apellidos = "";
        día = 20;
        Apellidos = "Ceballos";
    }
}
```

```
// ...
}
// ...
}
```

La declaración de una variable puede realizarse a *nivel de la clase* (variables miembro o atributos de la clase), a *nivel del método* (dentro de la definición de un método) o a nivel de un bloque de código. Dependiendo de dónde se declare, su uso estará limitado a la clase, al método o al bloque de código que la defina. Este espacio del programa al que queda limitado una variable se denomina *ámbito* de esa variable.

Una variable miembro de una clase puede ser declarada en cualquier parte dentro de la clase siempre que sea fuera de todo método y estará disponible para todo el código de esa clase.

Una variable declarada dentro de un método es una variable local al método. Los parámetros de un método son también variables locales al método. Y una variable declarada dentro de un bloque correspondiente a una sentencia compuesta también es una variable local a ese bloque.

En general, una variable local existe y tiene valor desde su punto de declaración hasta el final del bloque donde está definida. Cada vez que se ejecuta el bloque que la contiene, la variable local es nuevamente definida, y cuando finaliza la ejecución del mismo, la variable local deja de existir. Un elemento con carácter local es accesible solamente dentro del bloque al que pertenece.

Las variables miembro de una clase son iniciadas por omisión por el compilador C# para cada objeto que se declare de la misma: las variables numéricas con 0, los caracteres con '\0' y las referencias a las cadenas de caracteres y el resto de las referencias a otros objetos con **null**. También pueden ser iniciadas explícitamente, como ocurre con *año*. En cambio, las variables locales no son iniciadas por el compilador C#. Por lo tanto, es nuestra obligación iniciarlas, de lo contrario el compilador visualizará un mensaje de aviso en todas las sentencias que accedan a esas variables antes de ser iniciadas.

## OPERADORES

Los operadores son símbolos que indican cómo son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, relacionales, lógicos, unitarios, a nivel de bits, de asignación y operador condicional.



## Operadores aritméticos

Operador	Operación
+	<i>Suma.</i> Los operandos pueden ser enteros o reales.
-	<i>Resta.</i> Los operandos pueden ser enteros o reales.
*	<i>Multipliación.</i> Los operandos pueden ser enteros o reales.
/	<i>División.</i> Los operandos pueden ser enteros o reales. Si ambos operandos son enteros el resultado es entero. En el resto de los casos el resultado es real.
%	Los operandos pueden ser enteros o reales. Si ambos operandos son enteros el resto será entero; en otro caso, el resto será real.

Cuando en una operación aritmética los operandos son de diferentes tipos, ambos son convertidos al tipo del operando de precisión más alta.

## Operadores de relación

Operador	Operación
<	¿Primer operando <i>menor que</i> el segundo?
>	¿Primer operando <i>mayor que</i> el segundo?
<=	¿Primer operando <i>menor o igual que</i> el segundo?
>=	¿Primer operando <i>mayor o igual que</i> el segundo?
!=	¿Primer operando <i>distinto que</i> el segundo?
==	¿Primer operando <i>igual que</i> el segundo?

## Operadores lógicos

El resultado de una operación lógica (AND, OR, XOR y NOT) es un valor booleano verdadero o falso (**true** o **false**).

Operador	Operación
&& o &	<i>AND.</i> Da como resultado <b>true</b> si al evaluar cada uno de los operandos el resultado es <b>true</b> . Si uno de ellos es <b>false</b> , el resultado es <b>false</b> . Si se utiliza && (no &) y el primer operando es <b>false</b> , el segundo operando no es evaluado.
o	<i>OR.</i> El resultado es <b>false</b> si al evaluar cada uno de los operandos el resultado es <b>false</b> . Si uno de ellos es <b>true</b> , el resultado es <b>true</b> . Si se

utiliza `||` (no `|`) y el primer operando es **true**, el segundo operando no es evaluado (el carácter `|` es el Unicode 124).

**!** *NOT*. El resultado de aplicar este operador es **false** si al evaluar su operando el resultado es **true**, y **true** en caso contrario.

**^** *XOR*. Da como resultado **true** si al evaluar cada uno de los operandos el resultado de uno es **true** y el del otro **false**; en otro caso el resultado es **false**.

## Operadores unitarios

Operador	Operación
<code>~</code>	Complemento a 1 (cambiar ceros por unos y unos por ceros). El carácter <code>~</code> es el Unicode 126. El operando debe de ser de un tipo <b>int</b> , <b>uint</b> , <b>long</b> o <b>ulong</b> .
<code>-</code>	Cambia de signo al operando (esto es, se calcula el complemento a 2 que es el complemento a 1 más 1). El operando puede ser de un tipo numérico.

## Operadores a nivel de bits

Los operandos para los operadores `&`, `|` y `^` tienen que ser de un tipo entero o **bool**, y para los de desplazamiento el primer operando debe ser de tipo **int**, **uint**, **long** o **ulong** y el segundo de tipo **int**.

Operador	Operación
<code>&amp;</code>	Operación AND a nivel de bits.
<code> </code>	Operación OR a nivel de bits (carácter Unicode 124).
<code>^</code>	Operación XOR a nivel de bits.
<code>&lt;&lt;</code>	Desplazamiento a la izquierda rellenando con ceros por la derecha.
<code>&gt;&gt;</code>	Desplazamiento a la derecha rellenando con el bit de signo por la izquierda.

## Operadores de asignación

Los operandos tienen que ser del mismo tipo o bien el operando de la derecha tiene que poder ser convertido implícita o explícitamente al tipo del operando de la izquierda.

Operador	Operación
++	Incremento.
--	Decremento.
=	Asignación simple.
*=	Multiplicación más asignación.
/=	División más asignación.
%=	Módulo más asignación.
+=	Suma más asignación.
-=	Resta más asignación.
<<=	Desplazamiento a la izquierda más asignación.
>>=	Desplazamiento a la derecha más asignación.
&=	Operación AND sobre bits más asignación.
=	Operación OR sobre bits más asignación.
^=	Operación XOR sobre bits más asignación.

## Operador condicional

El operador condicional (?), llamado también operador ternario, se utiliza en expresiones condicionales, que tienen la forma siguiente:

*operando1 ? operando2 : operando3*

La expresión *operando1* debe ser una expresión booleana. Si el resultado de la evaluación de *operando1* es **true**, el resultado de la expresión condicional es *operando2*. Si el resultado de la evaluación de *operando1* es **false**, el resultado de la expresión condicional es *operando3*.

```
double a = 10.2, b = 20.5, mayor = 0;
mayor = (a > b) ? a : b;
```

## PRIORIDAD Y ORDEN DE EVALUACIÓN

La tabla que se presenta a continuación resume las reglas de prioridad y asociatividad de todos los operadores. Las líneas se han colocado de mayor a menor prioridad. Los operadores escritos sobre una misma línea tienen la misma prioridad.

Una expresión entre paréntesis, siempre se evalúa primero. Los paréntesis tienen mayor prioridad y son evaluados de más internos a más externos.



Operador	Asociatividad
() [] . new typeof	izquierda a derecha
- ~ ! ++ -- (tipo)expresión	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >= is as	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
	izquierda a derecha
&&	izquierda a derecha
	izquierda a derecha
?:	derecha a izquierda
= *= /= %= += -= <<= >>= >>>= &=  = ^=	derecha a izquierda

En C#, todos los operadores *binarios* excepto los de asignación y el operador ternario son evaluados de izquierda a derecha. En el siguiente ejemplo, primero se asigna *z* a *y* y a continuación *y* a *x*.

```
int x = 0, y = 0, z = 15;
x = y = z;      // resultado x = y = z = 15
```

En C#, los operadores *unitarios* son evaluados de derecha a izquierda. En el siguiente ejemplo, primero se aplica *~* a *y* y a continuación *-* al resultado.

```
int x = 0; int y = 1;
x = -~y;      // resultado x = 2, y = 1
```

## SENTENCIAS

Una sentencia C# puede formarse a partir de: una palabra clave (**for**, **while**, **if ... else**, etc.), expresiones, declaraciones o llamadas a métodos. Cuando se escriba una sentencia hay que tener en cuenta las siguientes consideraciones:

- Toda sentencia simple termina con un punto y coma (;).
- Dos o más sentencias pueden aparecer sobre una misma línea, separadas una de otra por un punto y coma.
- Una sentencia nula consta solamente de un punto y coma.

## PENSAR EN OBJETOS

Según lo estudiado hasta ahora, usted podría pensar en un programa como si fuera una lista de instrucciones que le indican a la máquina qué hacer. En cambio, desde la perspectiva de la POO un programa es un conjunto de objetos que dialogan entre sí para realizar las distintas tareas programadas. Para aclararlo, consideremos el ejemplo de la entidad bancaria mencionado anteriormente y pensemos en una concreta: XYZ. Podemos ver a esta entidad como a un objeto que tiene que comunicarse con otros muchos objetos (Bolsa, otras entidades bancarias, empresas, etc.) para lograr sus fines: ganar dinero. A su vez, la entidad XYZ tendrá un montón de sucursales distribuidas por toda la geografía. Cada sucursal es otro objeto, de diferentes características que la entidad bancaria, que se comunicará con otras sucursales para satisfacer las peticiones de sus clientes. Pero y, ¿qué es un cliente? Pues otro objeto con sus propias características que se comunicará con otros objetos (sucursales, otros clientes, empresas, etc.) para realizar operaciones desde sus cuentas (transferencias, cargos, ingresos, etc.). Pero y las cuentas, ¿no son también objetos? Evidentemente. Vemos entonces que escribir un programa de gestión para el banco XYZ supondría crear objetos banco, sucursal, cliente, cuenta, etc., que deben comunicarse entre sí para poder responder a las operaciones solicitadas en cada momento.

## Clases y objetos

Del ejemplo expuesto anteriormente, podemos deducir que la POO se basa en la observación de que, en el mundo real, los objetos se construyen a partir de otros objetos. La combinación de estos objetos es un aspecto de dicha programación, pero también incluye mecanismos y características que hacen que la creación y el uso de objetos sea sencillo y flexible. Un mecanismo importantísimo es la *clase*, y el encapsulamiento y la herencia son dos propiedades o características poderosas.

¿Qué es una *clase* de objetos? Pongamos un ejemplo: piense en un molde para hacer flanes; el molde es la clase y los flanes, los objetos. Esto es, si disponemos de un molde de un litro para hacer flanes de vainilla (ingredientes: leche, vainilla, azúcar, etc.), el molde agrupa las propiedades comunes a todos los flanes de vainilla, pero no todos los flanes tienen por qué tener la misma cantidad de cada ingrediente. Esto es, una *clase* equivale a la generalización de un tipo específico de objetos, pero cada objeto que construyamos de esa clase tendrá sus propios datos.

Un *objeto* de una determinada clase se crea en el momento en que se invoca al operador **new** para dicha *clase*. Por ejemplo, la siguiente línea crea un objeto de la clase o tipo *CCuenta* y asigna a la variable *cuenta01* una referencia al mismo.

```
CCuenta cuenta01 = new CCuenta(); // nueva cuenta
```

Algunos autores emplean el término instancia (traducción directa de *instance*) en el sentido de que una instancia es la representación concreta y específica de una clase; por ejemplo, *cuenta01* es una instancia de la clase *CCuenta*. Desde este punto de vista, los términos instancia y objeto son lo mismo. En este libro se prefiere, no obstante, utilizar el término *objeto*, o bien *ejemplar*.

Cuando se escribe un programa utilizando un lenguaje orientado a objetos, no se definen objetos verdaderos, se definen clases de objetos, donde una clase se ve como una plantilla para múltiples objetos con características similares. Afortunadamente, no tendrá que escribir todas las clases que necesite en su programa, porque .NET proporciona una biblioteca de clases estándar para realizar las operaciones más habituales que podamos requerir.

## Mensajes y métodos

Un programa orientado a objetos se compone solamente de objetos. Cada uno de ellos es una entidad que tiene unas propiedades particulares, los *atributos*, y unas formas de operar sobre ellos, los *métodos*. Por ejemplo, una ventana de una aplicación Windows es un objeto. El color de fondo, la anchura, la altura, etc. son atributos. Las rutinas, lógicamente transparentes al usuario, que permiten maximizar la ventana, minimizarla, etc. son los métodos.

Cuando se ejecuta un programa orientado a objetos, los objetos están recibiendo, interpretando y respondiendo a *mensajes* de otros objetos. En la POO, un *mensaje* está asociado con un *método*, de tal forma que cuando un objeto recibe un mensaje la respuesta a ese mensaje es ejecutar el método asociado. Por ejemplo, cuando un usuario quiere maximizar una ventana de una aplicación Windows, lo que hace simplemente es pulsar el botón de la misma que realiza esa acción. Eso provoca que Windows envíe un mensaje a la ventana para indicar que tiene que maximizarse. Como respuesta a este mensaje se ejecutará el método programado para ese fin.

Un *método* se escribe en una *clase* de objetos y determina cómo tiene que actuar el objeto cuando recibe el *mensaje* vinculado con ese método. A su vez, un *método* puede también enviar *mensajes* a otros objetos solicitando una acción o información. En adición, los atributos definidos en la clase permitirán almacenar información para dicho objeto.

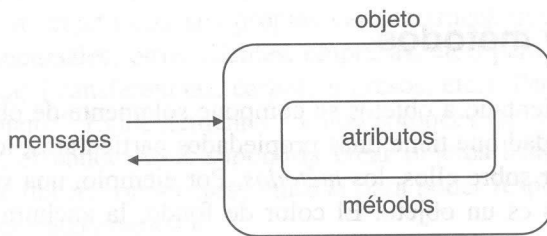
Según lo expuesto, podemos decir que la ejecución de un programa orientado a objetos realiza fundamentalmente tres tareas:

1. Crea los objetos necesarios.

2. Los mensajes enviados a unos y a otros objetos dan lugar a que se procese internamente la información.
3. Finalmente, cuando los objetos no son necesarios, son borrados.

## DISEÑO DE UNA CLASE DE OBJETOS

Cuando escribimos un programa orientado a objetos, lo que hacemos es diseñar un conjunto de clases, desde las cuales se crearán los objetos necesarios cuando el programa se ejecute. Cada una de estas clases incluye dos partes fácilmente diferenciables: los *atributos* y los *métodos*. Los atributos definen el estado de cada uno de los objetos de esa clase y los métodos, su comportamiento.



Normalmente, los atributos, la estructura más interna del *objeto*, se ocultan a los usuarios del objeto, manteniendo como única conexión con el exterior los *mensajes*. Esto quiere decir que los atributos de un objeto solamente podrán ser manipulados por los *métodos* del propio objeto. Este conjunto de métodos recibe el nombre de *interfaz*: medio de comunicación con un objeto.

Escribamos una clase de objetos. Siguiendo el ejemplo comentado al principio de este capítulo, podemos crear una clase de objetos *CCuenta* que represente una cuenta bancaria. Abra *Visual Studio* o *Microsoft Visual C#* y cree un nuevo proyecto, bajo la plantilla *Aplicación de consola*, denominado *Clase*. Cambie el nombre *Program.cs* por *Test.cs* (obsérvese que la clase *Program* pasa a llamarse *Test*). Añada al proyecto un nuevo elemento de tipo *Clase* y guárdelo en el fichero *CCuenta.cs*. Finalmente, complete la clase siguiendo paso a paso lo indicado a continuación:

```
class CCuenta
{
    // Cuerpo de la clase: atributos y métodos
}
```

Observamos que para declarar una clase hay que utilizar la palabra reservada **class** seguida del nombre de la clase y del cuerpo de la misma. El cuerpo de la clase incluirá entre { y } sus miembros: atributos y métodos.

## Atributos

Los atributos son las características individuales que diferencian un objeto de otro. El color de una ventana Windows la diferencia de otras; el D.N.I. de una persona la identifica frente a otras; el número de una cuenta la distingue entre otras; etc. Pensando en la clase de objetos *CCuenta*, elegimos los atributos de interés que van a definir esta clase de objetos:

- ◇ *nombre*: nombre del cliente del banco al que pertenece la cuenta.
- ◇ *cuenta*: número de la cuenta.
- ◇ *saldo*: saldo actual de la cuenta.
- ◇ *tipoDeInterés*: tipo de interés en tanto por cien.

Todos los atributos son definidos en la clase por variables:

```
class CCuenta
{
    private string nombre;
    private string cuenta;
    private double saldo;
    private double tipoDeInterés;

    // ...
}
```

Observe que se han definido cuatro atributos: dos de ellos, *nombre* y *cuenta*, pueden contener una cadena de caracteres (una cadena de caracteres es un objeto de la clase **string** perteneciente a la biblioteca .NET). Los otros dos atributos, *saldo* y *tipoDeInterés*, son de tipo **double**.

Anteriormente dijimos que, generalmente, los atributos de un objeto de una clase se ocultan a los usuarios del mismo. ¿Qué quiere decir esto? Que un usuario que utilice la clase *CCuenta* en su programa no podrá escribir su código basado directamente en estos atributos, sino que tendrá que acceder a ellos a través de los métodos que implemente la clase, como veremos a continuación; de esta forma, un usuario de la clase *CCuenta* no podrá asignar cualquier valor a los atributos de la misma. Esta protección es la que se consigue justamente con el modificador **private** (generalmente se utilizan los modificadores **private** o **public**). Un miembro declarado privado (**private**) es accesible solamente por los métodos de su propia clase. Esto significa que no se puede acceder a él por los métodos de cualquier otra clase, incluidas las subclases.



```

    }
    saldo = saldo + cantidad;
}

public void reintegro(double cantidad)
{
    if (saldo - cantidad < 0)
    {
        System.Console.WriteLine("Error: no dispone de saldo");
        return;
    }
    saldo = saldo - cantidad;
}

public double TipoDeInterés
{
    get
    {
        return tipoDeInterés;
    }
    set
    {
        if (value < 0)
        {
            System.Console.WriteLine("Error: tipo no válido");
            return;
        }
        tipoDeInterés = value;
    }
}
}

```

## Trabajando con objetos

Para poder crear objetos de esta clase y trabajar con ellos, tendremos que escribir un programa. Según lo estudiado en los capítulos anteriores, en un programa tiene que haber una clase con un método **Main**, puesto que éste es el punto de entrada y de salida del programa. Este requerimiento se puede satisfacer de tres formas. Vamos a comentarlas sobre el ejemplo que estamos desarrollando:

1. Añadir a la clase *CCuenta* un método **Main** declarado **static** que incluya el código del programa (crear objetos *CCuenta* y realizar operaciones con ellos).
2. Añadir, en el mismo fichero fuente en el que está almacenada la clase *CCuenta*, otra clase, por ejemplo una clase estándar *Test*, que incluya el método **Main** declarado **static**.

3. Tenemos un fichero fuente, *CCuenta.cs*, con la clase *CCuenta*. Añadir en la misma carpeta (directorio) otro fichero fuente, *Test.cs*, con una clase (en nuestro caso ya la tenemos y se llama *Test*) que incluya el método **Main**.

Vamos a continuar el ejemplo aplicando el punto tercero, porque es lo más práctico y lo que más se ajusta a lo que hemos denominado programación orientada a objetos, ya que de esta forma cada fichero fuente se corresponde con una clase de objetos. Por lo tanto, abra el fichero fuente *Test.cs* y añada a la clase *Test* el código que se muestra a continuación:

```
public static void Main(string[] args)
{
    CCuenta cuenta01 = new CCuenta();

    cuenta01.Nombre = "Un nombre";
    cuenta01.Cuenta = "Una cuenta";
    cuenta01.TipoDeInterés = 2.5;

    cuenta01.ingreso(10000);
    cuenta01.reintegro(5000);

    System.Console.WriteLine(cuenta01.Nombre);
    System.Console.WriteLine(cuenta01.Cuenta);
    System.Console.WriteLine(cuenta01.Saldo);
    System.Console.WriteLine(cuenta01.TipoDeInterés);
}
```

El método **Main** generalmente se declara público, no devuelve un resultado y tiene un parámetro *args* que es una matriz de una dimensión de tipo **string**, pero puede devolver un entero al sistema operativo y puede no tener parámetros. Analicemos el método **Main** del ejemplo anterior para que tenga una idea clara de lo que hace:

- La primera línea crea un objeto de la clase *CCuenta* y almacena una referencia al mismo en la variable *cuenta01* (observe el empleo del operador **new**). Esta variable la utilizaremos para acceder a ese objeto en las líneas siguientes. Ahora quizás empiece a entender por qué anteriormente decíamos que un programa orientado a objetos se compone solamente de objetos.
- Las cinco líneas siguientes establecen un determinado estado para el objeto referenciado por *cuenta01*; de éstas, las tres primeras fijan las propiedades *Nombre*, *Cuenta* y *TipoDeInterés*, y las dos siguientes envían los mensajes: *ingreso* y *reintegro*; las respuestas a estos mensajes, como ya sabe, es la ejecución de los métodos respectivos. Se puede observar que para acceder a un miembro del objeto se utiliza el operador punto (.).

- En las cuatro últimas líneas se muestra el estado del objeto; en nuestro caso mostramos el valor de las propiedades *Nombre*, *Cuenta*, *Saldo* y *TipoDeInterés* del objeto referenciado por *cuenta01*.

En general, para acceder a un miembro de un objeto (atributo o método), se utiliza la sintaxis siguiente:

*nombre\_objeto.nombre\_miembro*

De esta forma quedan eliminadas las ambigüedades que surgirían si hubiéramos creado más de un objeto. Esto es, supongamos que se hubieran creado dos objetos *CCuenta*: *cuenta01* y *cuenta02* y que para asignar el nombre a uno de ellos se hubiera utilizado la sintaxis *Nombre = "Un nombre"*; en este caso surgiría la pregunta: ¿la propiedad nombre a qué objeto corresponde?

Una vez escrito el programa, podemos comprobar que tenemos dos ficheros fuente: *CCuenta.cs* y *Test.cs*. Compile el programa ejecutando la orden *Generar Clase* del menú *Generar* de *Visual Studio* y después, ejecútelo.

Para ejecutar el programa que acabamos de compilar, seleccione la orden *Iniciar sin depurar* o *Iniciar* del menú *Depurar*. Observará los siguientes resultados:

```
Un nombre
Una cuenta
5000
2,5
```

Para finalizar, he aquí algunas notas que no debe olvidar:

- Cualquier método de una clase tiene acceso (puede invocar) a todos los otros miembros (atributos y métodos) de su clase.
- Un objeto de una clase sólo puede invocar a métodos de su clase; dicho de otra forma, únicamente puede responder a los mensajes para los que ha sido programado.

## Utilizando sólo métodos

Por compatibilidad con otros lenguajes, como C++, quizás prefiera implementar el comportamiento del objeto sólo con métodos. Desde este punto de vista, la clase *CCuenta* quedaría así:

```
class CCuenta
{
    // Atributos
```

```
private string nombre;
private string cuenta;
private double saldo;
private double tipoDeInterés;
```

```
// Métodos
```

```
public void asignarNombre(string nom)
```

```
{
    if (nom.Length == 0)
    {
        System.Console.WriteLine("Error: cadena vacía");
        return;
    }
    nombre = nom;
}
```

```
public string obtenerNombre()
```

```
{
    return nombre;
}
```

```
public void asignarCuenta(string cue)
```

```
{
    if (cue.Length == 0)
    {
        System.Console.WriteLine("Error: cuenta no válida");
        return;
    }
    cuenta = cue;
}
```

```
public string obtenerCuenta()
```

```
{
    return cuenta;
}
```

```
public double estado()
```

```
{
    return saldo;
}
```

```
public void ingreso(double cantidad)
```

```
{
    if (cantidad < 0)
    {
        System.Console.WriteLine("Error: cantidad negativa");
        return;
    }
    saldo = saldo + cantidad;
}
```

```

public void reintegro(double cantidad)
{
    if (saldo - cantidad < 0)
    {
        System.Console.WriteLine("Error: no dispone de saldo");
        return;
    }
    saldo = saldo - cantidad;
}

public void asignarTipoDeInterés(double tipo)
{
    if (tipo < 0)
    {
        System.Console.WriteLine("Error: tipo no válido");
        return;
    }
    tipoDeInterés = tipo;
}

public double obtenerTipoDeInterés()
{
    return tipoDeInterés;
}
}

```

## Constructores

Un *constructor* es un método especial de una clase que es llamado automáticamente siempre que se crea un objeto de esa clase. Su función es iniciar el objeto.

Un *constructor* se distingue fácilmente porque tiene el mismo nombre que la clase a la que pertenece y no puede retornar un valor (ni siquiera se puede especificar la palabra reservada **void**). Por ejemplo, si añadiéramos a la clase *CCuenta* un constructor, tendríamos que llamarlo también *CCuenta*. Ahora bien, cuando en una clase no escribimos explícitamente un constructor, C# asume uno por omisión. Por ejemplo, la clase *CCuenta* que hemos escrito anteriormente tiene por omisión un constructor definido así:

```
public CCuenta() {}
```

Un *constructor por omisión* de una clase *C* es un constructor sin parámetros que no hace nada. Sin embargo, es necesario porque, según lo que acabamos de exponer, será invocado cada vez que se construya un objeto sin especificar ningún argumento, en cuyo caso el objeto será iniciado con los valores predeterminados por el sistema (los atributos numéricos a ceros, los alfanuméricos a nulos, y las referencias a objetos a **null**).



Si usted quiere comprobar que un *constructor* es un método especial de una clase que es llamado automáticamente cada vez que se crea un objeto de esa clase, añada el siguiente método a la clase *CCuenta* del programa anterior y podrá verificar que cuando **Main** crea *cuenta01* se visualiza el mensaje “Objeto CCuenta creado”, señal inequívoca de que el constructor ha sido invocado.

```
public CCuenta() { System.Console.WriteLine("Objeto CCuenta creado"); }
```

Como ejemplo, vamos a añadir un constructor a la clase *CCuenta* con el fin de poder iniciar los atributos de cada nuevo objeto con unos valores determinados pasados como argumentos en el instante en el que se solicita crearlo:

```
class CCuenta
{
    // Atributos
    private string nombre;
    private string cuenta;
    private double saldo;
    private double tipoDeInterés;

    // Métodos
    public CCuenta() { }
    public CCuenta(string nom, string cue, double sal, double tipo)
    {
        asignarNombre(nom);
        asignarCuenta(cue);
        ingreso(sal);
        asignarTipoDeInterés(tipo);
    }

    // ...
}
```

Siempre que en una clase se define explícitamente un constructor, el constructor implícito (constructor por omisión) es reemplazado por éste. Por eso, hemos tenido que definirlo también explícitamente; de lo contrario, intentar crear un objeto sin especificar parámetros daría lugar a un error.

Observe que los constructores, salvo en casos excepcionales, deben declararse siempre públicos para que puedan ser invocados desde cualquier parte.

Una línea como la siguiente invocará al constructor sin parámetros:

```
CCuenta cuenta01 = new CCuenta(); // invoca al constructor CCuenta
```

El operador **new** crea un nuevo objeto, en este caso de la clase *CCuenta*, y a continuación se invoca al constructor de su clase para realizar las operaciones de

iniciación que estén programadas. Y una línea como la siguiente invocará al constructor con cuatro parámetros de la misma clase:

```
CCuenta cuenta02 = new CCuenta("Un nombre", "Una cuenta",
                                6000, 3.5);
```

Puede probar lo expuesto hasta ahora escribiendo la clase *Test* como se muestra a continuación. Puede también realizar una segunda prueba eliminando el constructor sin parámetros de la clase *CCuenta* y podrá comprobar que el compilador le muestra un error en la línea que invoca al constructor sin parámetros.

```
class Test
{
    public static void Main(string[] args)
    {
        CCuenta cuenta01 = new CCuenta();
        CCuenta cuenta02 =
            new CCuenta("Un nombre", "Una cuenta", 6000, 3.5);

        cuenta01.asignarNombre("Un nombre");
        cuenta01.asignarCuenta("Una cuenta");
        cuenta01.asignarTipoDeInterés(2.5);

        cuenta01.ingreso(10000);
        cuenta01.reintegro(5000);

        System.Console.WriteLine(cuenta01.obtenerNombre());
        System.Console.WriteLine(cuenta01.obtenerCuenta());
        System.Console.WriteLine(cuenta01.estado());
        System.Console.WriteLine(cuenta01.obtenerTipoDeInterés());
        System.Console.WriteLine();
        System.Console.WriteLine(cuenta02.obtenerNombre());
        System.Console.WriteLine(cuenta02.obtenerCuenta());
        System.Console.WriteLine(cuenta02.estado());
        System.Console.WriteLine(cuenta02.obtenerTipoDeInterés());
    }
}
```

## Destructor

Cuando se crea un nuevo objeto utilizando el operador **new**, C# asigna automáticamente la cantidad de memoria necesaria para ubicar ese objeto (en realidad, nos estamos refiriendo a la máquina virtual de .NET). Si no hubiera suficiente espacio de memoria disponible, el operador **new** lanzará una excepción **System.OutOfMemoryException** cuyo estudio posponemos. Después de saber esto, quizás se pregunte: ¿quién libera esa memoria y cuándo lo hace? La respuesta es otra vez la misma: C# se encarga de hacerlo en cuanto el objeto no se utilice, cosa que

ocurre cuando ya no existe ninguna referencia al mismo. Por ejemplo, en el código que se muestra a continuación, la memoria asignada al objeto referenciado por *cuenta01* será liberada cuando finalice la ejecución de **Main**, ya que esa variable se destruirá cuando el flujo de ejecución salga fuera de su ámbito, momento en el que el objeto *CCuenta* quedará desreferenciado.

```
public static void Main(string[] args)
{
    CCuenta cuenta01 = new CCuenta();
    // ...
}
```

Ahora es suficiente con que sepa que C# cuenta con una herramienta denominada *recolector de basura* que busca objetos que no se utilizan con el fin de destruirlos, liberando así la memoria que ocupan.

Antes de que un objeto sea destruido, C# invoca automáticamente al método destructor, lo que nos permitirá realizar operaciones como, por ejemplo, guardar información de estado, cerrar ficheros o bases de datos u operaciones de limpieza. El siguiente ejemplo muestra cómo añadir el destructor a una clase, en nuestro caso a la clase *CCuenta*, en el supuesto de que fuera necesario realizar operaciones de finalización justo antes de que un objeto *CCuenta* vaya a ser destruido:

```
~CCuenta()
{
    // Operaciones de limpieza
}
```

Una clase sólo puede tener un destructor. Un destructor no permite modificadores de acceso ni tiene parámetros. Se invoca automáticamente. No se puede llamar explícitamente. Los destructores no se pueden heredar ni sobrecargar.

El destructor llama implícitamente al método **Finalize** de la clase **Object** en la clase base del objeto.

## Métodos sobrecargados

Quizás le haya llamado la atención que ahora en la clase *CCuenta* hay un mismo método definido dos veces, nos referimos al constructor *CCuenta*. Pues bien, cuando en una clase un mismo método se define varias veces con distinto número de parámetros o bien con el mismo número de parámetros pero diferenciándose una definición de otra en que al menos un parámetro es de un tipo diferente, se dice que el método está *sobrecargado*.