

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

La sentencia más común en C# es la sentencia de asignación. Su forma general es:

$$\text{variable} = \text{expresión}$$

que indica que el valor que resulte de evaluar la *expresión* tiene que ser almacenado en la *variable* especificada. Por ejemplo:

```
int cont = 0;
double intereses, capital;
float tantoPorCiento;
string mensaje;
//...
cont = cont + 1;
intereses = capital * tantoPorCiento / 100;
mensaje = "La operación es correcta";
```

Una sentencia compuesta o bloque es una colección de sentencias simples incluidas entre llaves - `{ }` -. Un bloque puede contener a otros bloques.

Una sentencia C# puede escribirse en varias líneas físicas:

```
PagoMensual = CantidadPrest * (Interés / (1 -
              (1 / (System.Math.Pow((1 + Interés), Meses)))));
```

MÉTODOS

Un método es una colección de sentencias que ejecutan una tarea específica. En C#, un método siempre pertenece a una clase y su definición nunca puede contener a la definición de otro método; esto es, C# no permite métodos anidados.

Definición de un método

La definición de un método consta de una *cabecera* y del *cuerpo* del método encerrado entre llaves. La sintaxis para escribir un método es la siguiente:

```
[modificador] tipo-resultado nombre-método ([lista de parámetros])
{
    declaraciones de variables locales;
    sentencias;
    [return ([expresión])];
}
```

Las variables declaradas en el cuerpo del método son locales a dicho método y por definición solamente son accesibles dentro del mismo.

Un *modificador* es una palabra clave que modifica el nivel de protección pre-determinado del método. Véase el capítulo *Programación orientada a objetos* expuesto un poco más adelante.

El *tipo del resultado* especifica qué tipo de expresión retorna el método. Éste puede ser cualquier tipo primitivo o referenciado. Para indicar que no se devuelve nada, se utiliza la palabra reservada **void**. El resultado de un método es devuelto por medio de la siguiente sentencia:

```
return [(]expresión[)];
```

La sentencia **return** puede ser o no la última y puede aparecer más de una vez en el cuerpo del método. En el caso de que el método no retorne un valor (**void**), se puede omitir o especificar simplemente **return**. Por ejemplo:

```
void mEscribir()  
{  
    // ...  
    return;  
}
```

La *lista de parámetros* de un método son las variables que reciben los valores de los argumentos especificados cuando se invoca al mismo. Consiste en una lista de cero, uno o más identificadores con sus tipos, separados por comas. A continuación se muestra un ejemplo:

```
public double Sumar(double x, double y)  
{  
    // ...  
}
```

Método Main

Todo programa C# tiene un método denominado **Main**, y sólo uno. Este método es el punto de entrada al programa y también el punto de salida. Su definición es la que se muestra a continuación:

```
public static void Main(string[] args)  
{  
    // Cuerpo del método  
}
```

Como se puede observar, el método **Main** es público (**public**), estático (**static**), no devuelve nada (**void**) y tiene un argumento de tipo **string** que almacenará los argumentos pasados en la línea de órdenes cuando se invoca al programa para su ejecución.

Pasando argumentos a los métodos

Cuando se invoca a un método, el primer argumento es pasado al primer parámetro, el segundo argumento es pasado al segundo parámetro y así sucesivamente. Por ejemplo, supongamos una clase *CRacional* con un método **static** *Sumar* (en el capítulo *Programación orientada a objetos* expuesto un poco más adelante repasaremos todo lo relacionado con clases de objetos).

```
CRacional r1, r2, r3;
r1 = new CRacional(3, 7);      // crear un objeto CRacional 3/7
r2 = new CRacional(2, 5);      // crear un objeto CRacional 2/5
r3 = CRacional.Sumar(r1, r2);  // r3 = 3/7 + 2/5
```

Analicemos el método *Sumar*. Este método tiene dos parámetros *a* y *b* de tipo *CRacional*. Después de que el método ha sido invocado, *a* y *b* señalan a los mismos objetos que *r1* y *r2*. Esto significa que los objetos pasados a los parámetros de un método son siempre referencias a dichos objetos, lo cual quiere decir que cualquier modificación que realice el método sobre esos objetos la está haciendo sobre los objetos originales. En cambio, las variables de alguno de los tipos primitivos (**int**, **float**, **double**, etc.) pasan por valor por tratarse de estructuras, lo cual significa que se pasa una copia, por lo que cualquier modificación que se haga a esas variables dentro del método no afecta a la variable original; no obstante, C# también permite pasar este tipo de variables por referencia, cuestión que estudiaremos a continuación.

```
public static CRacional Sumar(CRacional a, CRacional b)
{
    CRacional r = new CRacional(); // crear un objeto CRacional
    int num = a.Numerador * b.Denominador +
              a.Denominador * b.Numerador;
    int den = a.Denominador * b.Denominador;
    r.AsignarDatos(num, den);
    return r;
}
```

A continuación, el método *Sumar* utiliza **new** para crear un nuevo objeto *r* (se invoca al constructor sin parámetros) al que asigna el resultado de la suma de los objetos *a* y *b*. Finalmente devuelve *r*. Otra vez más, lo que se devuelve es una referencia que se copia en *r3*. Finalizado este proceso, la variable *r* desaparece por ser local; no sucede lo mismo con el objeto que señalaba, ya que ahora está señalado por *r3*.

El recolector de basura de C# sólo eliminará un objeto cuando no exista ninguna referencia al mismo.

Pasar un argumento de un tipo primitivo

Cuando un método C# invoca a otro método y le pasa un argumento de un tipo primitivo, pasa una copia de ese argumento. Por ejemplo:

```
public static void Incrementar10(int param)
{
    param += 10;
}

public static void Main(string[] args)
{
    int arg = 1234;
    Incrementar10(arg);
    System.Console.WriteLine(arg);
}
```

La línea sombreada del ejemplo anterior invoca al método *Incrementar10* y copia el valor del argumento *arg* en el parámetro *param* del método. Esto significa que el argumento ha sido pasado por valor. Por lo tanto, cualquier modificación que haga el método sobre *param* no afectará a la variable original. Según lo expuesto el método **Main** mostrará el resultado 1234, valor original de *arg*.

¿Qué hay que hacer para que un método pueda modificar el valor original del argumento que se le pasa? Pasar dicho argumento por referencia.

Según lo estudiado hasta ahora, cuando se pasa un argumento que es un objeto (tipo referenciado), C# no hace una copia del objeto sobre el parámetro correspondiente del método, sino que informa al método acerca del lugar de la memoria donde está ese objeto para que pueda acceder al mismo, lo que se denomina pasar un argumento por referencia. Esto es, lo que se copia en el parámetro del método es una referencia al objeto. Para proceder en el mismo sentido con una variable de un tipo primitivo hay que utilizar la palabra reservada **ref** tanto en la definición del método como en la llamada al mismo, detalles que se muestran en el ejemplo siguiente:

```
public static void Incrementar10(ref int param)
{
    param += 10;
}

public static void Main(string[] args)
{
    int arg = 1234;
    Incrementar10(ref arg);
    System.Console.WriteLine(arg);
}
```

En el ejemplo anterior, el método **Main** define una variable *arg* de tipo **int** iniciada con el valor 1234. Después invoca al método *Incrementar10* pasándole como argumento una referencia (**ref**) a esa variable, lo que supone declarar su parámetro *param* como una referencia (**ref**) a un **int**. Ahora *param* hace referencia al mismo entero que *arg*. Por lo tanto, todos los cambios realizados por el método afectarán a la variable original. Como consecuencia, el resultado mostrado por **Main** será ahora 1244.

Argumentos que son matrices

Recuerde que los objetos pasados como argumentos a un método son siempre referencias a dichos objetos, lo cual significa que cualquier modificación que se haga sobre esos objetos dentro del método afecta a los objetos originales, y las matrices son objetos. En cambio, las variables de un tipo primitivo se pasan por valor, lo cual significa que se pasa una copia, por lo que cualquier modificación que se haga sobre esas variables dentro del método no afecta a la variable original.

Para aclarar lo expuesto, el siguiente ejemplo implementa un método con un parámetro de tipo **double[,]**, que permite multiplicar por 2 los elementos de una matriz numérica de dos dimensiones pasada como argumento.

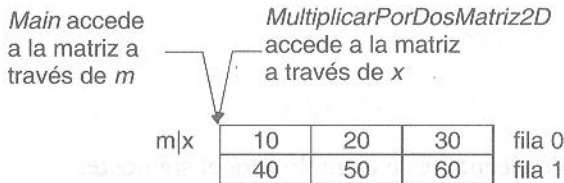
```
public class Test
{
    public static void MultiplicarPorDosMatriz2D(double[,] x)
    {
        for (int f = 0; f < x.GetLength(0); f++)
        {
            for (int c = 0; c < x.GetLength(1); c++)
            {
                x[f,c] *= 2;
            }
        }
    }

    public static void Main(string[] args)
    {
        double[,] m = {{10, 20, 30}, {40, 50, 60}};

        MultiplicarPorDosMatriz2D(m);
        // Visualizar la matriz por filas
        for (int f = 0; f < m.GetLength(0); f++)
        {
            for (int c = 0; c < m.GetLength(1); c++)
            {
                System.Console.Write(m[f,c] + " ");
                System.Console.WriteLine();
            }
        }
    }
}
```

El método **GetLength** de la clase **Array** permite obtener el valor de la dimensión pasada como argumento del objeto matriz que recibe este mensaje. La primera dimensión es la 0.

La aplicación anterior se ejecuta de la forma siguiente: el método **Main** crea e inicia una matriz *m* de dos dimensiones de tipo **double**. Después invoca al método *MultiplicarPorDosMatriz2D* pasando como argumento la matriz *m*; esto implica que el método tenga un parámetro declarado así: *double[,] x*. Por ser *m* un objeto, el parámetro *x* recibe una referencia a la matriz *m*; esto es, *x* almacenará la posición de memoria donde se encuentra la matriz, no una copia de su contenido. Por lo tanto, ahora el método *MultiplicarPorDosMatriz2D* tiene acceso a la misma matriz que el método **Main**. Gráficamente puede imaginárselo así:



¿Cuál es el resultado? Que cuando el método **Main** visualice los elementos de la matriz *m*, éstos aparecerán con los cambios introducidos por el método *MultiplicarPorDosMatriz2D*. Esto es, ambos métodos trabajan sobre la misma matriz.

Así mismo, un método puede retornar un valor de cualquier tipo primitivo, o bien una referencia a cualquier clase de objetos. Por lo tanto, en el caso de que un método devuelva una matriz, lo que realmente devuelve es una referencia a la matriz.

Argumentos opcionales

A diferencia de C++ o de Visual Basic, C# no soporta métodos con parámetros que tengan asignados valores por omisión.

Número indefinido de argumentos

La palabra clave **params** permite especificar que un procedimiento aceptará un número arbitrario de argumentos. Eso permite escribir procedimientos como el siguiente:

```
class Test
{
```

```

public static void Visualizar(params object[] matriz)
{
    int i = 0;
    foreach (object x in matriz)
    {
        i++;
        System.Console.WriteLine("{0} {1} {2} {3}",
                                   "Parámetro ", i, "=", x);
    }
    System.Console.WriteLine();
}

static void Main(string[] args)
{
    Test.Visualizar(2);
    Test.Visualizar(2, 3.7);
    Test.Visualizar(2, 3.7, 8.125);
}

```

El resultado de ejecutar este ejemplo será el siguiente:

Parámetro 1 = 2

Parámetro 1 = 2

Parámetro 2 = 3,7

Parámetro 1 = 2

Parámetro 2 = 3,7

Parámetro 3 = 8,125

Métodos recursivos

Se dice que un método es recursivo si se llama a sí mismo. El compilador C# permite cualquier número de llamadas recursivas a un método. Cada vez que el método es llamado, sus parámetros y sus variables locales son iniciados.

¿Cuándo es eficaz escribir un método recursivo? La respuesta es sencilla, cuando el proceso a programar sea por definición recursivo. Por ejemplo, el cálculo del factorial de un número, $n! = n(n-1)!$, es por definición un proceso recursivo que se enuncia así: $factorial(n) = n * factorial(n-1)$.

```
using System;
```

```
using MisClases.ES; // Espacio De Nombres de la clase Leer
```

```
public class Test
```

```
{
```

```
    // Cálculo del factorial de un número
```



```
public static long factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return checked(n * factorial(n-1));
}
```

```
public static void Main(string[] args)
{
    int numero;
    long fac;

    Console.Write("¿Número? ");
    numero = Leer.dataInt();

    try
    {
        fac = factorial(numero);
        Console.WriteLine("\nEl factorial de " + numero + " es: " + fac);
    }
    catch(OverflowException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Por defecto, un programa C# no comprueba la posibilidad de desbordamiento aritmético. Para forzar a su comprobación podemos utilizar la sentencia **checked** de cualquiera de las dos formas siguientes:

`checked(expresión);`

```
checked
{
    sentencias;
}
```

Una expresión o un conjunto de sentencias **checked** se revisa para ver si hay desbordamiento aritmético (**unchecked** no comprueba) y en el caso de que lo haya, se lanzará una excepción del tipo **System.OverflowException**. En el ejemplo anterior, si la expresión $n * \text{factorial}(n-1)$ produce un desbordamiento, se lanzará una excepción que será atrapada por **Main**.

Propiedades y métodos

El comportamiento de un objeto queda definido por las acciones que puede emprender. Por ejemplo, pensando acerca de un objeto de la clase *CCuenta*, esto es, de una cuenta de un cliente de un determinado banco, algunas acciones que se pueden realizar sobre ella son:

- ◇ Asignar el nombre de un cliente del banco a una cuenta.
- ◇ Obtener el nombre del cliente de una cuenta.
- ◇ Asignar el número de la cuenta.
- ◇ Obtener el número de la cuenta.
- ◇ Realizar un ingreso.
- ◇ Realizar un reintegro.
- ◇ Asignar el tipo de interés.
- ◇ Obtener el tipo de interés.
- ◇ Obtener el saldo, esto es, el estado de la cuenta.

Para definir este comportamiento se pueden utilizar tanto métodos como propiedades. En general, los métodos representan acciones que un objeto puede realizar, mientras que las propiedades representan información sobre un objeto. Según esto, nombre, número de la cuenta, tipo de interés y saldo podrían considerarse propiedades, e ingreso y reintegro acciones. No obstante, desde el punto de vista de compatibilidad con otros lenguajes (por ejemplo, con C++) todas las acciones enumeradas anteriormente podrían implementarse como métodos.

Los métodos son rutinas de código, definidas dentro del cuerpo de la clase, que se ejecutan en respuesta a alguna acción tomada desde dentro de un objeto de esa clase o bien desde otro objeto de la misma o de otra clase. Recuerde que los objetos se comunican mediante mensajes. El conjunto de mensajes a los que un objeto puede responder se corresponde con el conjunto de métodos que implementa su clase.

Como ejemplo, vamos a agregar a la clase *CCuenta* un método que responda a la acción de ingresar una cantidad en una cuenta:

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

Observe que el método ha sido declarado público (**public**). Un miembro declarado público está accesible para cualquier otra clase o subclase que necesite utilizarlo. La interfaz pública de una clase, o simplemente interfaz, está formada por todos los miembros públicos de la misma.

Como se puede observar, un método consta de su nombre precedido por el tipo del valor que devuelve cuando finaliza su ejecución (la palabra reservada **void** indica que el método no devuelve ningún valor) y seguido por una lista de parámetros separados por comas y encerrados entre paréntesis (en el ejemplo, hay un parámetro *cantidad*). Los paréntesis indican a C# que el identificador, *ingreso*, se refiere a un método y no a un atributo. A continuación se escribe el cuerpo del método encerrado entre { y }.

El método *ingreso* asegura que la cantidad a asignar no sea negativa. Si la cantidad fuera negativa, simplemente visualizará un mensaje indicándolo; en otro caso, asignará la *cantidad* pasada como argumento al atributo *saldo* del objeto que reciba el mensaje "ingreso".

Cuando decimos que un objeto recibe un mensaje, debemos entender que el mensaje es un concepto que subyace en nuestra mente; la acción real es invocar al método que responde a ese mensaje con el fin de modificar el estado del objeto. Según esto, podemos decir que los nombres de los métodos de una clase forman el conjunto de mensajes a los que un objeto de esa clase puede responder.

Las propiedades tienen un aspecto análogo a un método. Utilizan descriptores de acceso para controlar cómo se establecen y devuelven valores de los atributos a los que se refieren. Estos descriptores de acceso son rutinas de código declaradas dentro de la propiedad para recuperar (**get**) o establecer (**set**) el valor de la misma. Su sintaxis es la siguiente:

```
public tipo nombre_propiedad
{
    get
    {
        // Aquí se devuelve "return" el valor del atributo
    }

    set
    {
        // Aquí se asigna el valor "value" al atributo
    }
}
```

El cuerpo del descriptor de acceso **get** debe devolver el valor del tipo de la propiedad. La ejecución del descriptor de acceso **get** equivale a leer el valor del

atributo al que se refiere la propiedad. El descriptor de acceso **set** utiliza un parámetro implícito denominado **value**, que tiene el mismo tipo que la propiedad.

Como ejemplo, vamos a agregar a la clase *CCuenta* una propiedad que permita obtener o asignar el nombre de un cliente a una cuenta:

```
public string Nombre
{
    get
    {
        return nombre;
    }
    set
    {
        if (value.Length == 0)
        {
            System.Console.WriteLine("Error: cadena vacía");
            return;
        }
        nombre = value;
    }
}
```

Observe que la propiedad ha sido declarada pública (**public**). La explicación es la misma que hemos dado para los métodos. Fíjese también que a través de **set** aseguramos que el nombre a asignar no sea una cadena vacía (el atributo **Length** de la clase **String** contiene el número de caracteres que hay almacenados en el objeto **String** que recibe ese mensaje); si el nombre fuera una cadena vacía, simplemente visualizará un mensaje indicándolo; en otro caso, asignará la cadena **value** al atributo *nombre* del objeto. A través de **get** simplemente devolvemos el valor del atributo.

Normalmente, los descriptores de acceso de una propiedad se definen por parejas, utilizando **get** y **set**. Ahora bien, si la propiedad es de sólo lectura (sólo **get**) o de sólo escritura (sólo **set**) puede definirse cada descriptor de acceso de forma individual. Por ejemplo, vamos a agregar a la clase *CCuenta* una propiedad que permita obtener el saldo de una cuenta. Ésta será una propiedad de sólo lectura:

```
public double Saldo
{
    get { return saldo; }
}
```

Cuando terminemos de escribir todos los métodos previstos, tendremos creada la clase *CCuenta* que guardaremos en un fichero denominado *CCuenta.cs*.


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

    // Métodos
    public string Nombre
    {
        get { return nombre; }

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

    public string Cuenta
    {
        get { return cuenta; }

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

    public double Saldo
    {
        get { 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 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**.