Métodos y parámetros

Contenido

Notas para el instructor

Este módulo explica el uso de métodos, parámetros y métodos sobrecargados. También explica que es recomendable descomponer la lógica del programa en unidades funcionales, ya que facilita la reutilización del código.

El módulo comienza explicando qué son los métodos, cómo se crean y cómo se pueden hacer llamadas a métodos. A continuación se discuten brevemente los modificadores de acceso **private** y **public** (que se tratarán con más detalle en módulos posteriores). El módulo concluye con una explicación de la sobrecarga de métodos y las reglas sobre signaturas de métodos.

Este módulo sienta también las bases para los temas sobre encapsulación, que se tratará en un módulo posterior.

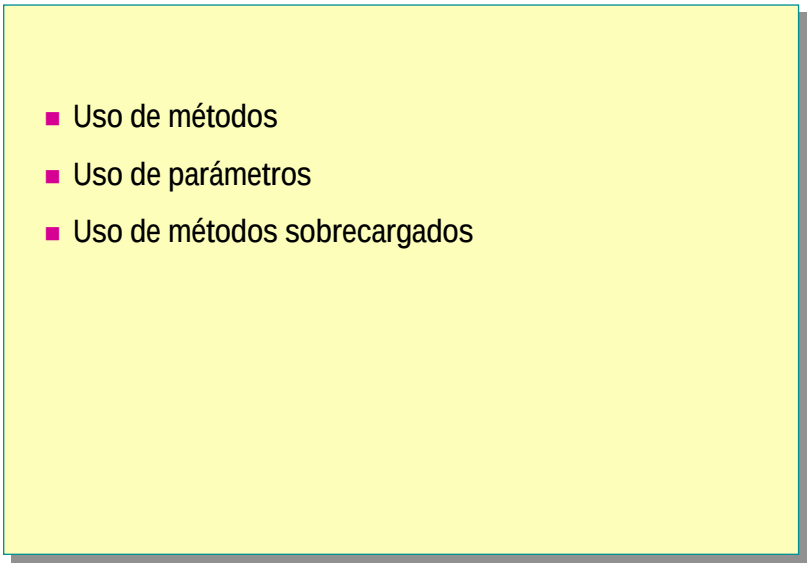
Al final de este módulo, los estudiantes serán capaces de:

- Crear métodos estáticos que aceptan parámetros y devuelven valores.
- Pasar parámetros a métodos utilizando distintos mecanismos.
- Seguir reglas de ámbito para variables y métodos.

Notas generales

Objetivo del tema
Ofrecer una introducción a los contenidos y objetivos del módulo.

Explicación previa
En este módulo aprenderá cómo usar métodos en C#, cómo se pasan parámetros y cómo se devuelven valores.



Para su información
Este módulo describe la sintaxis básica para el uso de métodos en C#, pero no incluye una discusión completa de conceptos orientados a objetos como la encapsulación y el ocultamiento de información. En particular, este módulo discute los módulos estáticos pero no los métodos de instancia. Para parámetros y valores devueltos se utilizarán únicamente tipos valor, ya que todavía no se han tratado los tipos de referencia.

Uno de los principios básicos del diseño de aplicaciones es que deben estar divididas en unidades funcionales, ya que las secciones pequeñas de código son más fáciles de entender, diseñar, desarrollar y depurar. La división de una aplicación en unidades funcionales permite también la reutilización de componentes funcionales en toda la aplicación.

Una aplicación C# está estructurada en clases que contienen bloques de código con nombre llamados métodos. Un *método* es un miembro de una clase que lleva a cabo una acción o calcula un valor.

Al final de este módulo, usted será capaz de:

- Crear métodos estáticos que aceptan parámetros y devuelven valores.
- Pasar parámetros a métodos de distintas maneras.
- Declarar y usar métodos sobrecargados.

Uso de métodos

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
En esta sección estudiará el uso de métodos en C#.

- Definición de métodos
- Llamadas a métodos
- Uso de la instrucción return
- Uso de variables locales
- Devolución de valores

En esta sección estudiará el uso de métodos en C#. Los métodos son un mecanismo importante para estructurar el código de un programa. Aprenderá cómo se crean y cómo se pueden hacer llamadas a métodos desde una sola clase y desde una clase a otra.

También aprenderá a usar variables locales, así como a asignarlas y destruirlas.

Finalmente, aprenderá a devolver un valor desde un método y a utilizar parámetros para transferir datos a un método o extraerlos de él.

Definición de métodos

Objetivo del tema
Mostrar la sintaxis para la definición de métodos simples.

Explicación previa
Esta transparencia muestra cómo definir un método simple.

- Main es un método
  - Para definir métodos propios se usa la misma sintaxis

```
using System;

class ExampleClass
{
    static void ExampleMethod( )
    {
        Console.WriteLine("ExampleMethod");
    }
    static void Main( )
    {
        // ...
    }
}
```

Recomendación al profesor
No entre por el momento en explicaciones detalladas sobre el significado de static, void o la lista de parámetros vacía. Lo más importante que hay que recordar es que un método pertenece a una clase, que tiene un nombre y que contiene un bloque de código.

Un método es una serie de instrucciones C# que han sido agrupadas bajo un nombre determinado. La mayor parte de los lenguajes de programación modernos incluyen un concepto similar; se puede considerar que un método es equivalente a una función, una subrutina, un procedimiento o un subprograma.

Ejemplos de métodos

El código mostrado en la transparencia contiene tres métodos:

- El método **Main**
- El método **WriteLine**
- El método **ExampleMethod**

El método **Main** es el punto de entrada de la aplicación, mientras que **WriteLine** forma parte de Microsoft® .NET Framework y se puede llamar desde un programa. El método **WriteLine** es un método estático de la clase **System.Console**. El método **ExampleMethod** pertenece a **ExampleMethod** y contiene una llamada al método **WriteLine**.

Todos los métodos en C# pertenecen a una clase, al contrario de lo que ocurre en otros lenguajes de programación como C, C++ y Microsoft Visual Basic®, que permiten subrutinas y funciones globales.

Creación de métodos

Cuando se crea un método hay que especificar lo siguiente:

- Nombre

  Un método no puede tener el mismo nombre que una variable, una constante o cualquier otro elemento que no sea un código y haya sido declarado en la clase. El nombre del método puede ser cualquier identificador permitido de C# y distingue entre mayúsculas y minúsculas.

- Lista de parámetros

  A continuación del nombre del método viene una lista de parámetros para el método. Esta lista aparece entre paréntesis que deben estar presentes aunque no hay ningún parámetro, como se ve en los ejemplos de la transparencia.

- Cuerpo del método

  Después de los paréntesis viene el cuerpo del método, que debe estar entre llaves ({ y }) aunque no contenga más que una instrucción.

Sintaxis para la definición de métodos

Para crear un método se utiliza la siguiente sintaxis:

```
static void MethodName( )
{
  cuerpo del método
}
```

Recomendación al profesor
Por el momento, no dedique demasiado tiempo a hablar de las palabras clave static y void. La palabra clave void se tratará más adelante en esta sección, mientras que static se discutirá en módulos posteriores dedicados a conceptos orientados a objetos.

El siguiente ejemplo muestra cómo crear un método llamado **ExampleMethod** en la clase **ExampleClass**:

```
using System;
class ExampleClass
{
   static void ExampleMethod( )
   {
      Console.WriteLine("Example method");
   }

   static void Main( )
   {
      Console.WriteLine("Main method");
   }
}
```

Nota C# distingue entre mayúsculas y minúsculas en los nombres de métodos, lo que permite declarar y usar métodos cuyos nombres sólo se diferencian en letras mayúsculas o minúsculas. Por ejemplo, es posible declarar métodos llamados **print** y **PRINT** en la misma clase. Sin embargo, el runtime de lenguaje común requiere que los nombres de métodos dentro de una clase se diferencien también en otros aspectos para garantizar la compatibilidad con otros lenguajes en los que los nombres de los métodos no distinguen entre mayúsculas y minúsculas. Esto es importante si se desea que una aplicación interactúe con otras aplicaciones escritas en lenguajes que no sean C#.

Llamadas a métodos

Objetivo del tema
Explicar cómo hacer una llamada a un método simple.

Explicación previa
Una vez definido un método, hay que saber cómo llamarlo.

- Una vez definido un método, se puede:
  - Llamar a un método desde dentro de la misma clase
    Se usa el nombre del método seguido de una lista de parámetros entre paréntesis
  - Llamar a un método que está en una clase diferente
    Hay que indicar al compilador cuál es la clase que contiene el método que se desea llamar
    El método llamado se debe declarar con la palabra clave **public**
  - Usar llamadas anidadas
    Unos métodos pueden hacer llamadas a otros, que a su vez pueden llamar a otros métodos, y así sucesivamente

---

Después de definir un método, es posible hacer llamadas a ese método desde dentro de la misma clase y desde otras clases.

Llamadas a métodos

Recomendación al profesor
La sintaxis resultará familiar para los desarrolladores en C y C++. A los desarrolladores en Visual Basic puede indicarles que en C# no existe la instrucción **Call** y que hay que usar paréntesis en todas las llamadas a métodos.

Para hacer una llamada a un método hay que emplear el nombre del método seguido de una lista de parámetros entre paréntesis. Los paréntesis son obligatorios aunque el método llamado no contenga ningún parámetro, como se ve en el siguiente ejemplo.

```
MethodName( );
```

Nota para los desarrolladores en Visual Basic La instrucción **Call** no existe. Los paréntesis son obligatorios en todas las llamadas a métodos.

El programa del siguiente ejemplo comienza al principio del método **Main** de **ExampleClass**. La primera instrucción muestra "Comienza el programa." La segunda instrucción en **Main** es la llamada a **ExampleClass**. El flujo de control pasa a la primera instrucción en **ExampleClass** y aparece "Hola, mundo". Al final del método, el control pasa a la instrucción inmediatamente después de la llamada al método, que es la instrucción que muestra "Termina el programa."

```
using System;

class ExampleClass
{
   static void ExampleClass( )
   {
      Console.WriteLine("Hola, mundo");
   }

   static void Main( )
   {
      Console.WriteLine("Comienza el programa.");
      ExampleMethod( );
      Console.WriteLine("Termina el programa.");
   }
}
```

Llamadas a métodos desde otras clases

Recomendación al profesor
Los contenidos de este tema están simplificados. En un módulo posterior se explicarán con detalle las palabras clave public, private, internal y protected. Lo único que tiene que explicar por el momento es que los métodos llamados desde una clase diferente tienen que estar declarados como public.

Para que los métodos de una clase puedan hacer llamadas a métodos en otra clase hay que:

- Especificar la clase que contiene el método que se desea llamar.

  Para especificar la clase que contiene el método se utiliza la siguiente sintaxis:

  ```
  ClassName.MethodName( );
  ```

- Declarar el método llamado con la palabra clave **public**.

El siguiente ejemplo muestra cómo hacer una llamada al método **TestMethod**, que está definido en la clase **A**, desde **Main** en la clase **B**:

```
using System;

class A
{
   public static void TestMethod( )
   {
      Console.WriteLine("Esto es TestMethod en clase A");
   }
}

class B
{
   static void Main( )
   {
      A.TestMethod ( );
   }
}
```

Si en el ejemplo anterior se eliminara el nombre de la clase, el compilador buscaría un método llamado **TestMethod** en la clase **B**. Puesto que en esa clase no hay ningún método con ese nombre, el compilador mostrará el siguiente error: "El nombre 'TestMethod' no existe en la clase o espacio de nombres 'B.'"

Un método que no se declare como público es privado por defecto en la clase. Por ejemplo, si en la definición de **TestMethod** se omite la palabra clave **public**, el compilador mostrará el siguiente error: "'A.TestMethod()' no es accesible debido a su nivel de protección."

También se puede emplear la palabra clave **private** para indicar que sólo es posible hacer llamadas al método desde dentro de la clase. Estas dos líneas de código tienen exactamente el mismo efecto, ya que los métodos son privados por defecto:

```
private static void MyMethod( );
static void MyMethod( );
```

Las palabras clave **public** y **private** especifican la *accesibilidad* del método. Estas palabras clave controlan si es posible hacer llamadas a un método desde fuera de la clase en la que está definido.

Llamadas anidadas a métodos

También es posible hacer llamadas a métodos desde dentro de métodos. El siguiente ejemplo muestra cómo anidar llamadas a métodos:

```
using System;
class NestExample
{
   static void Method1( )
   {
      Console.WriteLine("Method1");
   }
   static void Method2( )
   {
      Method1( );
      Console.WriteLine("Method2");
      Method1( );
   }
   static void Main( )
   {
      Method2( );
      Method1( );
   }
}
```

La salida de este programa es la siguiente:

```
Method1
Method2
Method1
Method1
```

La anidación permite hacer llamadas a un número ilimitado de métodos. No hay ningún límite predefinido para el nivel de anidación. Sin embargo, el entorno de tiempo de ejecución puede imponer límites, normalmente debido a la cantidad de RAM disponible para ejecutar el proceso. Cada llamada a un método necesita memoria para almacenar direcciones de retorno y otra información.

Como regla general, si se agota la memoria para llamadas anidadas a métodos se debe probablemente a problemas en el diseño de clases.

Uso de la instrucción return

Objetivo del tema
Presentar la instrucción **return** y explicar cómo usarla al final de un método.

Explicación previa
Normalmente, la ejecución de un método se devuelve al llamador cuando se llega a la última instrucción del método, excepto si se utiliza la instrucción return.

- Return inmediato
- Return con una instrucción condicional

```
static void ExampleMethod( )
{
   int numBeans;
   //...

   Console.WriteLine("Hello");
   if (numBeans < 10)
      return;
   Console.WriteLine("World");
}
```

Recomendación al profesor
Esta resultará familiar para los programadores en C y C++. Para los programadores en Visual Basic, la instrucción **return** por sí misma es equivalente a **Exit Sub**, **Exit Function** o **Exit Property** en Visual Basic.

La instrucción **return** se puede emplear para hacer que un método se devuelva inmediatamente al llamador. Sin una instrucción **return**, lo normal es que la ejecución se devuelva al llamador cuando se alcance la última instrucción del método.

Return inmediato

Por defecto, un método es devuelto a su llamador cuando se llega al final de la última instrucción del bloque de código. La instrucción **return** se utiliza cuando se quiere que un método sea devuelto inmediatamente al llamador.

El método del siguiente ejemplo muestra “Hola,” y es inmediatamente devuelto a su llamador:

```
static void ExampleMethod( )
{
   Console.WriteLine("Hola");
   return;
   Console.WriteLine("mundo");
}
```

Este uso de la instrucción **return** no resulta muy útil, ya que la llamada final a **Console.WriteLine** no se ejecuta nunca. Si los avisos del compilador de C# están activados a nivel 2 o superior, el compilador mostrará el siguiente mensaje: “Detectado código inaccesible.”

Return con una instrucción condicional

Es mucho más habitual, y mucho más útil, utilizar la instrucción **return** como parte de una instrucción condicional como **if** o **switch**. Esto permite que un método sea devuelto al llamador si se cumple cierta condición.

El método del siguiente ejemplo será devuelto si la variable *numBeans* es menor que 10; en caso contrario, la ejecución continuará dentro de este módulo.

```
static void ExampleMethod( )
{
   int numBeans;
   //...
   Console.WriteLine("Hola");
   if (numBeans < 10)
      return;
   Console.WriteLine("mundo");
}
```

Consejo En general se considera una buena práctica de programación que un método tenga un punto de entrada y otro de salida. El diseño de C# garantiza que la ejecución de todos los métodos comienza por la primera instrucción. Un método sin ninguna instrucción **return** tiene un punto de salida al final del bloque de código. Un método con varias instrucciones **return** tiene múltiples puntos de salida, lo que en algunos casos puede hacer que el método sea difícil de entender y mantener.

Return con un valor

Si un método está definido con un tipo de datos en lugar de **void**, return se usa para asignar un valor a la función. Este punto se discutirá más adelante en este módulo.

Recomendación al profesor
Los valores devueltos se discutirán más adelante en este módulo. Este párrafo se ha incluido aquí para anticiparse a las preguntas de los estudiantes sobre el tema.

Uso de variables locales

Objetivo del tema
Describir las variables locales y explicar cómo se crean y destruyen.

Explicación previa
Cada método tiene su propio conjunto de variables locales.

- Variables locales
  - Se crean cuando comienza el método
  - Son privadas para el método
  - Se destruyen a la salida
- Variables compartidas
  - Para compartir se utilizan variables de clase
- Conflictos de ámbito
  - El compilador no avisa si hay conflictos entre nombres locales y de clase

Cada método tiene su propio conjunto de variables locales, que sólo se pueden utilizar dentro del método en el que están declaradas. No es posible acceder a las variables locales desde ninguna otra parte de la aplicación.

Variables locales

Se pueden incluir variables locales en el cuerpo de un método, como se ve en el siguiente ejemplo:

```
static void MethodWithLocals( )
{
   int x = 1; // Variable con valor inicial
   ulong y;
   string z;
      ...
}
```

Las variables locales pueden llevar asignado un valor inicial (como la variable *x* en el código del ejemplo anterior). La variable no estará inicializada si no se le asigna un valor o si no tiene una expresión inicial para determinar un valor.

Recomendación al profesor
Esta aplicación sobre asignación y liberación de memoria se refiere únicamente a tipos de valor. La administración de memoria para tipos de referencia es más compleja y se tratará en un módulo posterior.

Las variables que se declaran en un método son completamente independientes de las variables declaradas en otros métodos, aunque sus nombres sean iguales.

Cada vez que se hace una llamada a un método se asigna memoria para variables locales, que se libera cuando termina el método. Por lo tanto, los valores almacenados en estas variables no se conservan desde una llamada a un método y la siguiente.

Variables compartidas

El siguiente código intenta contar el número de veces que ha sido llamado un método:

Recomendación al profesor
Los desarrolladores experimentados en Visual Basic tal vez comenten que, en Visual Basic, poner static antes del nombre de un método hace que todas las variables locales de ese método tengan clase de almacenamiento estático. Recuerde este hecho, aunque normalmente no será necesario mencionarlo.

```
class CallCounter_Bad
{
   static void Init( )
   {
      int nCount = 0;
   }
   static void CountCalls( )
   {
      int nCount;
      ++nCount;
      Console.WriteLine("Método llamado {0} veces", nCount);
   }
   static void Main( )
   {
      Init( );
      CountCalls( );
      CountCalls( );
   }
}
```

Este programa no se puede compilar debido a dos problemas importantes. La variable *nCount* en **Inic** no es la misma que la variable *nCount* en **CountCalls**. Independientemente del número de veces que se llame al método **CountCalls**, el valor *nCount* se pierde cada vez que termina **CountCalls**.

Para su información
Éste es el comportamiento habitual de las variables locales en C, C++ y Visual Basic, por lo que no deberían surgir demasiados problemas.

La forma correcta de escribir este código es utilizando una variable de clase, como se muestra en el siguiente ejemplo:

```
class CallCounter_Good
{
   static int nCount;
   static void Init( )
   {
      nCount = 0;
   }
   static void CountCalls( )
   {
      ++ nCount;
      Console.Write("Método llamado " + nCuenta + " veces.");
   }
   static void Main( )
   {
      Init( );
      CountCalls( );
      CountCalls( );
   }
}
```

En este ejemplo, *nCount* se declara como clase en vez de método. Por lo tanto, *nCount* se comparte entre todos los métodos de la clase.

Conflictos de ámbito

En C# se puede declarar una variable local con el mismo nombre que una variable de clase, aunque eso puede producir resultados inesperados. En el siguiente ejemplo, *NumItems* está declarada como una variable de clase **ScopeDemo**, y también como una variable local en **Method1**. Las dos variables son completamente distintas. En **Method1**, *NumObjetos* se refiere a la variable local, mientras que en **Method2** se refiere a la variable de clase.

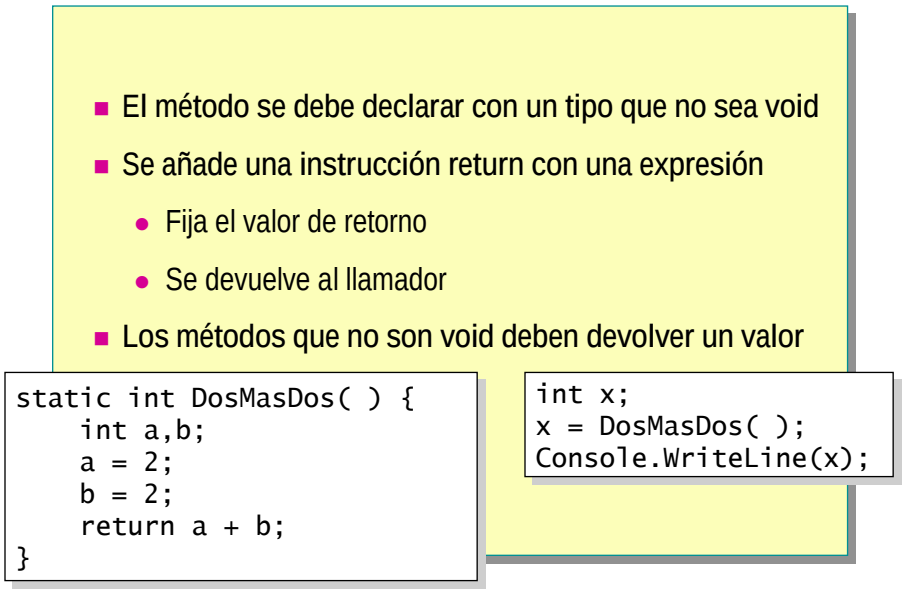
```
class ScopeDemo
{
   static int numItems = 0;
   static void Method1( )
   {
       int numItems = 42;
          ...
   }
   static void Method2( )
   {
       numItems = 61;
   }
}
```

Consejo El compilador de C# no indica si hay variables locales y variables de clase que tienen los mismos nombres, por lo que conviene adoptar una convención de nomenclatura para distinguirlas unas variables de otras.

Devolución de valores

Objetivo del tema
Explicar cómo utilizar la instrucción **return** para devolver valores.

Explicación previa
La instrucción **return** se puede utilizar para devolver un valor desde un método.



Ya hemos aprendido a usar la instrucción **return** para hacer que un método finalice inmediatamente. También se puede utilizar la instrucción **return** para devolver un valor desde un método. Para devolver un valor hay que:

1. Declarar el método con el tipo del valor que se desea devolver.
2. Añadir al método una instrucción **return**.
3. Incluir el valor que se desea devolver al llamador.

Declaración de métodos con un tipo que no sea void

Para declarar un método de forma que devuelva un valor al llamador hay que cambiar la palabra clave **void** por el tipo del valor que se desea devolver.

Uso de instrucciones return

Recomendación al profesor
La instrucción return devuelve un valor al llamador inmediatamente. Lo mismo ocurre en C y C++. En Visual Basic, las funciones miembro devuelven un valor por asignación al nombre de la función sin causar un return inmediato.

La palabra clave **return** seguida de una expresión hace que el método finalice inmediatamente y devuelve la expresión como valor de retorno del método.

El siguiente ejemplo muestra cómo declarar un método llamado **DosMasDos** que devuelve el valor 4 a **Main** cuando hace una llamada a **DosMasDos**:

```
class ExampleReturningValue
{
    static int DosMasDos( )
    {
        int a,b;
        a = 2;
        b = 2;
        return a + b;
    }

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

El valor devuelto es un **int**, ya que **int** es el tipo de retorno del método. Cuando se hace una llamada al método, éste devuelve el valor 4. En este ejemplo, el valor se almacena en la variable local *x* de **Main**.

Los métodos que no son void deben devolver valores

Si se declara un método con un tipo distinto de void, es obligatorio añadir al menos una instrucción **return**. El compilador intenta comprobar que cada uno de estos métodos devuelve siempre un valor al método de llamada. Si detecta que un método que no es void no incluye ninguna instrucción **return**, el compilador mostrará el siguiente mensaje de error: “No todas las rutas de código devuelven un valor.” Este mensaje también aparecerá si el compilador detecta que es posible ejecutar un método que no es void sin devolver un valor.

Consejo La instrucción **return** sólo se puede utilizar para devolver un valor desde cada llamada a un método. Si se desea devolver más de un valor desde una llamada a un método se pueden utilizar los parámetros **out** o **ref**, que se discutirán más adelante en este módulo. También es posible devolver una referencia a una tabla o clase o estructura, que puede contener varios valores. La norma general, según la cual hay que evitar el uso de varias instrucciones **return** en un solo método, es válida también para métodos que no son void.

Uso de parámetros

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
En esta sección aprenderá a utilizar parámetros en métodos.

- Declaración y llamadas a parámetros
- Mecanismos de paso de parámetros
- Paso por valor
- Paso por referencia
- Parámetros de salida
- Uso de listas de parámetros de longitud variable
- Normas para el paso de parámetros
- Uso de métodos recursivos

En esta sección aprenderá a declarar parámetros y a hacer llamadas a métodos con parámetros. Estudiará también como se pasan parámetros. Finalmente, aprenderá el uso de llamadas a métodos recursivos en C#.

En esta sección aprenderá a:

- Declarar y hacer llamadas a parámetros.
- Pasar parámetros utilizando los siguientes mecanismo:
  - Paso por valor
  - Paso por referencia
  - Parámetros de salida
- Usar llamadas a métodos recursivos.

Recomendación al profesor
Dependiendo de su formación, es posible que los estudiantes tengan distintas ideas sobre el paso de parámetros. C no permite directamente el paso de parámetros por referencia (en su lugar se pasan punteros). C++ incluye un mecanismo de paso por referencia, además de punteros similares a los de C. En Visual Basic, los parámetros se pasan por referencia de forma predeterminada, salvo que se desactive esta opción con la palabra clave ByVal.

Declaración y llamadas a parámetros

Objetivo del tema
Mostrar la sintaxis básica para la declaración y el uso de parámetros.

Explicación previa
En la declaración del método, los parámetros se colocan entre los paréntesis que siguen al nombre del método.

- Declaración de parámetros
  - Se ponen entre paréntesis después del nombre del método
  - Se definen el tipo y el nombre de cada parámetro
- Llamadas a métodos con parámetros
  - Un valor para cada parámetro

```
static void  MethodWithParameters(int n, string y)
{ ... }

MethodWithParameters(2, "Hola, mundo");
```

Los parámetros permiten pasar información hacia dentro y fuera de un método. La definición de un método puede incluir una lista de parámetros entre paréntesis a continuación del nombre del método. Las listas de parámetros de todos los ejemplos que hemos visto hasta ahora en este módulo estaban vacías.

Declaración de parámetros

Cada parámetro tiene un tipo y un nombre. Se pueden declarar parámetros poniendo sus declaraciones entre los paréntesis que siguen al nombre del método. La sintaxis que se utiliza es similar a la empleada para declarar variables locales, salvo que las declaraciones de parámetros se separan con una coma en lugar de un punto y coma.

El siguiente ejemplo muestra cómo declarar un método con parámetros:

```
static void MethodWithParameters(int n, string y)
{
    // ...
}
```

Este ejemplo declara el método **MethodWithParameters** con dos parámetros: *n* e *y*. El primer parámetro es de tipo **int**, mientras que el segundo es de tipo **string**. Los parámetros aparecen en la lista separados por comas.

Llamadas a métodos con parámetros

El código de llamada debe indicar los valores de los parámetros en la llamada al método.

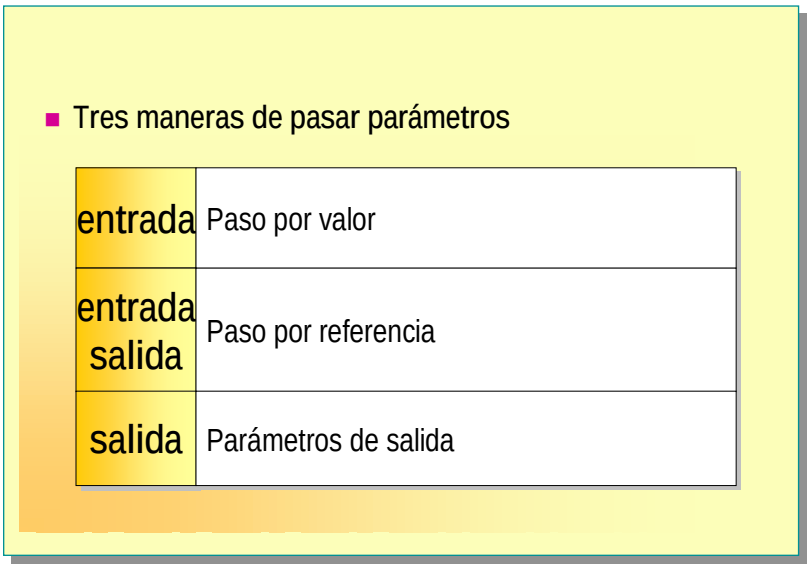
El siguiente código muestra dos ejemplos de llamadas a un método con parámetros. En ambos casos, los valores de los parámetros son encontrados y colocados en los parámetros *n* e *y* al principio de la ejecución de **MethodWithParameters**.

```
MethodWithParameters(2, "Hola, mundo");

int p = 7;
string s = "Mensaje de prueba";

MethodWithParameters(p, s);
```

Mecanismos de paso de parámetros

Objetivo del tema
Presentar tres formas de pasar parámetros.

Explicación previa
Los parámetros se pueden pasar de tres maneras distintas.



Para su información
En C y C++ se usa por defecto el mecanismo de paso por valor.

C no incluye ningún mecanismo de paso por referencia.

C++ utiliza el modificador & para indicar paso por referencia.

Visual Basic utiliza por defecto paso por referencia.

Los parámetros se pueden pasar de tres maneras distintas:

- Por valor

  Los parámetros valor se llaman a veces *parámetros de entrada*, ya que los datos se pueden transferir a un método pero no fuera de él.

- Por referencia

  Los parámetros referencia se llaman a veces *parámetros de entrada/salida*, ya que los datos se pueden transferir dentro y fuera de un método.

- Por salida

  Los parámetros de salida se llaman a veces *parámetros out*, ya que los datos se pueden transferir fuera de un método pero no dentro de él.

Paso por valor

Objetivo del tema
Explicar el mecanismo de paso por valor.

Explicación previa
El paso por valor es el mecanismo predeterminado.

- Mecanismo predeterminado para el paso de parámetros:
  - Se copia el valor del parámetro
  - Se puede cambiar la variable dentro del método
  - No afecta al valor fuera del método
  - El parámetro debe ser de un tipo igual o compatible

```
static void SumaUno(int x)
{
      x++; // Incrementar x
}
static void Main( )
{
      int k = 6;
      SumaUno(k);
      Console.WriteLine(k); // Muestra el valor 6, no 7
}
```

La mayor parte de los parámetros se utilizan en las aplicaciones para pasar información a un método, no fuera de él. Por eso el paso por valor es el mecanismo predeterminado para pasar parámetros en C#.

Definición de parámetros valor

La definición más sencilla de un parámetro es un nombre de tipo seguido de un nombre de variable, y se conoce como *parámetro valor*. Cuando se hace una llamada a un método, para cada parámetro valor se crea una nueva ubicación de almacenamiento donde se copian los valores de las expresiones correspondientes.

La expresión de cada parámetro valor debe ser del mismo tipo que la declaración del parámetro valor, o bien de un tipo que pueda ser convertido implícitamente a ese tipo. Dentro del método se puede escribir código que cambie el valor del parámetro. Esto no afectará a ninguna variable fuera de la llamada al método.

En el siguiente ejemplo, la variable *x* en **AddOne (SumaUno)** es completamente independiente de la variable *k* en **Main**. Se puede cambiar la variable *x* en **AddOne**, pero esto no afectará al valor de *k*.

```
static void AddOne(int x)
{
    x++;
}
static void Main( )
{
    int k = 6;
    AddOne (k);
    Console.WriteLine(k); // Muestra el valor 6, no 7
}
```

Paso por referencia

Objetivo del tema
Explicar el mecanismo de paso por referencia.

Explicación previa
Un parámetro referencia es una referencia a una posición de memoria.

- ¿Qué son los parámetros referencia?
  - Una referencia a una posición de memoria
- Uso de parámetros referencia
  - Se usa la palabra clave **ref** en la declaración y las llamadas al método
  - Los tipos y valores de variables deben coincidir
  - Los cambios hechos en el método afectan al llamador
  - Hay que asignar un valor al parámetro antes de la llamada al método

¿Qué son los parámetros referencia?

Un parámetro referencia es una referencia a una posición de memoria. A diferencia de un parámetro valor, un parámetro referencia no crea una nueva ubicación de almacenamiento. Por el contrario, un parámetro referencia representa la misma posición de memoria que la variable indicada en la llamada al método.

Recomendación al profesor
Puede explicar a los programadores en C y C++ que el operador **ref** es similar al operador de dirección (&) en C y C++, aunque no son iguales. Las direcciones de C y C++ permiten modificar posiciones de memoria arbitrarias. C# es más seguro.

Declaración de parámetros referencia

Un parámetro referencia se puede declarar poniendo la palabra clave **ref** antes del nombre del tipo, como se ve en el siguiente ejemplo:

```
static void ShowReference(ref int nId, ref long nCount)
{
    // ...
}
```

Uso de distintos tipos de parámetros

La palabra clave **ref** afecta únicamente al parámetro al que antecede, no a toda la lista de parámetros. En el siguiente método, *nId* se pasa por referencia pero *longVar* se pasa por valor:

```
static void OneRefOneVal(ref int nId, long longVar)
{
  // ...
}
```

Coincidencia de tipos y valores de parámetros

En la llamada a un método, los parámetros referencia se indican utilizando la palabra clave **ref** seguida de una variable. El valor indicado en la llamada al método debe ser exactamente igual al tipo en la definición del método, y además debe ser una variable, no una constante ni una expresión calculada.

```
int x;
long q;
ShowReference(ref x, ref q);
```

Si se omite la palabra clave **ref** o si se emplea una constante o una expresión calculada, el compilador rechazará la llamada y mostrará un mensaje de error parecido al siguiente: “Imposible convertir de ‘int’ a ‘ref int.’”

Cambio de valores de parámetros referencia

Si se cambia el valor de un parámetro referencia cambiará también la variable indicada por el llamador, ya que ambas son referencias a la misma posición de memoria. El siguiente ejemplo ilustra cómo el cambio del parámetro referencia afecta también a la variable:

```
static void AddOne(ref int x)
{
    x++;
}
static void Main( )
{
    int k = 6;
    AddOne(ref k);
    Console.WriteLine(k); // Muestra el valor 7
}
```

Este código funciona porque, cuando se hace la llamada a **AddOne**, su parámetro *x* apunta a la misma posición de memoria que la variable *k* en **Main**. Por lo tanto, al incrementar *x* se incrementará también *k*.

Asignación de parámetros antes de la llamada al método

Un parámetro **ref** tiene que estar definitivamente asignado al punto de llamada; es decir, el compilador debe asegurarse de que se asigna un valor antes de hacer la llamada. El siguiente ejemplo muestra cómo se pueden inicializar parámetros referencia antes de la llamada al método:

```
static void AddOne(ref int x)
{
    x++;
}

static void Main( )
{
    int k = 6;
    AddOne(ref k);
    Console.WriteLine(k); // 7
}
```

El siguiente ejemplo muestra lo que ocurre si no se inicializa un parámetro referencia *k* antes de la llamada al método **AddOne**:

```
int k;
AddOne(ref k);
Console.WriteLine(k);
```

El compilador de C# rechazará este código y mostrará el siguiente mensaje de error: “Uso de variable local ‘*k*’ no asignada.”

Parámetros de salida

Objetivo del tema
Explicar la palabra clave out y su relación con el paso por referencia.

Explicación previa
Los parámetros de salida son similares a los parámetros referencia, pero transfieren datos fuera del método en lugar de al método.

- ¿Qué son los parámetros de salida?
  - Pasan valores hacia fuera, pero no hacia dentro
- Uso de parámetros de salida
  - Como ref, pero no se pasan valores al método
  - Se usa la palabra clave out en la declaración y las llamadas al método

```
static void OutDemo(out int p)
{
      // ...
}
int n;
OutDemo(out n);
```

¿Qué son los parámetros de salida?

Los parámetros de salida son similares a los parámetros referencia, pero transfieren datos fuera del método en lugar de al método. Al igual que u parámetro referencia, un parámetro de salida es una referencia a una ubicación de almacenamiento indicada por el llamador. Sin embargo, no es necesario asignar un valor a la variable indicada para el parámetro **out** antes de hacer la llamada, y el método asume que el parámetro no ha sido inicializado.

Los parámetros de salida son útiles cuando se quiere devolver valores de un método por medio de un parámetro sin tener que asignar a éste un valor inicial.

Uso de parámetros de salida

Para declarar un parámetro de salida se utiliza la palabra clave **out** antes del tipo y el nombre, como se ve en el siguiente ejemplo:

```
static void DemoOut(out int p)
{
    // ...
}
```

Como ocurre con la palabra clave **ref**, **out** afecta únicamente a un parámetro y cada parámetro **out** debe estar separado de los demás.

En la llamada a un método con un parámetro **out** hay que poner la palabra clave **out** antes de la variable que se desea pasar, como en este ejemplo:

```
int n;
DemoOut(out n);
```

En el cuerpo del método al que se hace la llamada no hay ninguna indicación al contenido del parámetro de salida, por lo que éste se trata como si fuera una variable local sin asignar. El valor del parámetro out se tiene que asignar dentro del método.

Uso de listas de parámetros de longitud variable

Objetivo del tema
Presentar la palabra clave params.

Explicación previa
C# incluye un mecanismo para pasar listas de parámetros de longitud variable.

- Se usa la palabra clave params
- Se declara como tabla al final de la lista de parámetros
- Siempre paso por valor

```
static long AddList(params long[ ] v)
{
    long total, i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
static void Main( )
{
    long x = AddList(63,21,84);
}
```

Recomendación al profesor
No es necesario que discuta este tema con mucho detalle. Recuerde además que en este curso todavía no se han tratado las matrices.

C# incluye un mecanismo para pasar listas de parámetros de longitud variable.

Declaración de parámetros de longitud variable

En ocasiones resulta útil tener un método que pueda aceptar un número variable de parámetros. En C# se puede emplear la palabra clave **params** para indicar una lista de parámetros de longitud variable. Cuando se declara un parámetro de longitud variable hay que:

- Declarar sólo un parámetro **params** por método.
- Poner el parámetro al final de la lista de parámetros.
- Declarar el parámetro como de tipo tabla unidimensional.

Este ejemplo muestra cómo declarar una lista de parámetros de longitud variable:

```
static long AddList (params long[ ] v)
{
    long total;
    long i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
```

Como un parámetro **params** es siempre una tabla, todos los valores deben ser del mismo tipo.

Para su información
Visual Basic tiene una función parecida llamada ParamArray(). C y C++ tienen (...) y las macros va_*, que manejan listas de parámetros variables, como en printf.

Paso de valores

Hay dos maneras de pasar valores al parámetro **params** cuando se hace una llamada a un método con un parámetro de longitud variable:

- Como una lista de elementos separados por comas (la lista puede estar vacía)
- Como una tabla

Ambas técnicas se muestran en el siguiente código. El compilador trata las dos técnicas de la misma forma.

```
static void Main( )
{
    long x;
    x = AddList (63, 21, 84); // Lista
    x = AddList (new long[ ]{ 63, 21, 84 }); // Tabla
}
```

Independientemente del método empleado para hacer la llamada al método, el parámetro **params** se trata siempre como una tabla. Se puede usar la propiedad **Length** de la tabla para determinar el número de parámetros pasados a cada llamada.

Los datos se copian en el parámetro **params** y, aunque es posible modificar los valores dentro del método, los valores fuera del método no cambian.

Para su información
Para conservar los cambios realizados en el método se usa una tabla object.

Normas para el paso de parámetros

Objetivo del tema
Discutir los distintos mecanismos para el paso de parámetros.

Explicación previa
Aquí se ofrecen algunas normas para el paso de parámetros.

- Mecanismos
  - El paso por valor es el más habitual
  - El valor de retorno del método es útil para un solo valor
  - **ref** y/o **out** son útiles para más de un valor de retorno
  - **ref** sólo se usa si los datos se pasan en ambos sentidos
- Eficiencia
  - El paso por valor suele ser el más eficaz

La abundancia de opciones disponibles para pasar parámetros puede hacer que la elección no resulte obvia. Dos de los factores que hay que considerar a la hora de elegir la forma de pasar parámetros son el mecanismo y su eficiencia.

Mecanismos

Los parámetros valor ofrecen cierta protección frente a modificaciones no deseadas, ya que los cambios realizados dentro del método no tienen efecto fuera de él. Esto sugiere que los parámetros valor son la mejor opción, salvo que sea preciso pasar información fuera de un método.

Recomendación al profesor
Esto es una ligera simplificación. Un método puede devolver varios valores utilizando tipos de referencia (por ejemplo, una tabla).

Para pasar datos fuera de un método se puede utilizar la instrucción **return**, parámetros referencia o parámetros de salida. La instrucción **return** es fácil de usar, pero sólo puede devolver un resultado. Si se necesita devolver varios valores hay que usar parámetros referencia y de salida. Utilice **ref** si desea transferir datos en las dos direcciones, y **out** si sólo necesita transferir datos fuera del método.

Eficiencia

En general, los tipos simples como **int** y **long** se pasan más eficazmente por valor.

Estos aspectos de eficiencia no forman parte del lenguaje, por lo que no hay que tomárselos al pie de la letra. Aunque la eficiencia puede ser un punto a tener en cuenta en aplicaciones grandes y que consumen muchos recursos, normalmente es preferible preocuparse de la corrección, la estabilidad y la potencia del programa antes que de la eficiencia. El empleo de buenas prácticas de programación tiene prioridad sobre la eficiencia.

Uso de métodos recursivos

Objetivo del tema
Explicar que C# permite el uso recursivo de métodos.

Explicación previa
Como la mayor parte de los lenguajes de programación, C# permite el uso recursivo de métodos.

- Un método puede hacer una llamada a sí mismo
  - Directamente
  - Indirectamente
- Útil para resolver ciertos problemas

Recomendación al profesor
No todos los estudiantes estarán familiarizados con la recursión. En caso de problemas, explique que por el momento no es necesario comprender la recursión en detalle. Lo único que tienen que saber es que pueden utilizarla cuando sea necesario.

Un método puede hacer una llamada a sí mismo. Esta técnica se conoce como *recursión* y ofrece una solución para ciertos tipos de problemas. Los métodos recursivos suelen ser útiles para manipular estructuras de datos más complejas, como listas y árboles.

Los métodos en C# pueden ser mutuamente recursivos. Por ejemplo, puede darse una situación en la que un método A llame a un método B, y el método B llame al método A.

Ejemplo de método recursivo

La serie de Fibonacci aparece con cierta frecuencia en matemáticas y biología (por ejemplo, el índice de reproducción y la población de conejos). El término *n* de esta serie tiene el valor 1 si *n* es 1 ó 2, y en los demás casos es igual a la suma de los dos números que le preceden en la serie; es decir, el valor del término *n* para *n* mayor de 2 se obtiene a partir de dos valores anteriores de la serie. El hecho de que la definición del método incluya al propio método sugiere que puede ser recursivo.

El código para el método **Fibonacci** podría ser siguiente:

```
static ulong Fibonacci(ulong n)
{
    if (n <= 2)
        return 1;
    else
        return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Recomendación al profesor
El código del ejemplo usa **ulongs** para poder utilizar valores grandes.

Como puede verse, dentro del método se hacen dos llamadas al mismo método.

Un método recursivo debe tener una condición de fin para que devuelva un valor sin hacer más llamadas. En el caso del método **Fibonacci**, la condición de fin es `n <= 2`.

Uso de métodos sobrecargados

Objetivo del tema
Ofrecer una introducción a los temas tratados en esta sección.

Explicación previa
En esta sección aprenderá a usar métodos sobrecargados.

- Declaración de métodos sobrecargados
- Signaturas de métodos
- Uso de métodos sobrecargados

Los métodos no pueden tener el mismo nombre que otros elementos en una clase. Sin embargo, dos o más métodos en una clase sí pueden compartir el mismo nombre. A esto se le da el nombre de sobrecarga.

En esta sección aprenderá a:

- Declarar métodos sobrecargados.
- Utilizar signaturas en C# para distinguir métodos que tienen el mismo nombre.
- Saber cuándo usar métodos sobrecargados.

Declaración de métodos sobrecargados

Objetivo del tema
Explicar cómo sobrecargar nombres de métodos.

Explicación previa
Es posible dar el mismo nombre a más de un método en una clase.

- Métodos que comparten un nombre en una clase
  - Se distinguen examinando la lista de parámetros

```
class OverloadingExample
{
    static int Suma(int a, int b)
    {
        return a + b;
    }
    static int Suma(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        Console.WriteLine(Suma(1,2) + Suma(1,2,3));
    }
}
```

Para su información
C++ admite la sobrecarga, pero C no.
Visual Basic .NET es la primera versión de Visual Basic que permite métodos sobrecargados.

Los métodos sobrecargados son métodos que tienen el mismo nombre dentro de una clase. El compilador de C# distingue métodos sobrecargados comparando las listas de parámetros.

Ejemplos de métodos sobrecargados

El siguiente código muestra cómo se pueden utilizar distintos métodos con el mismo nombre en una clase:

```
class OverloadingExample
{
    static int Add(int a, int b)
    {
        return a + b;
    }
    static int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        Console.WriteLine(Add(1,2) + Add(1,2,3));
    }
}
```

Recomendación al profesor
Las signaturas se discutirán con más detalle en el siguiente tema.

El compilador de C# encuentra en la clase dos métodos llamados **Add**, y dos llamadas a métodos llamados **Add** en **Main**. Aunque los nombres de los métodos son iguales, el compilador puede distinguir entre los dos métodos **Add** comparando las listas de parámetros.

El primer método **Add** acepta dos parámetros, ambos de tipo **int**. Por su parte, el segundo método **Add** lleva tres parámetros que también son de tipo **int**. Como las listas de parámetros son diferentes, el compilador permite definir ambos métodos dentro de la misma clase.

La primera instrucción en **Main** contiene una llamada a **Add** con dos parámetros **int**, por lo que el compilador la interpreta como una llamada al primer método **Add**. La segunda llamada a **Add** lleva tres parámetros **int** y el compilador la interpreta como una llamada al segundo método **Add**.

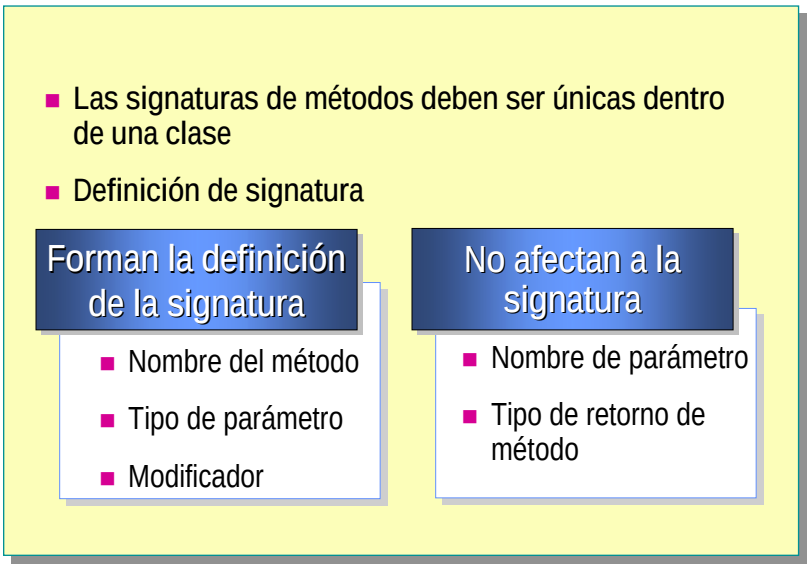
No está permitido compartir nombres entre métodos y variables, constantes o enumeraciones en la misma clase. El siguiente código no se compilará porque se ha utilizado el nombre *k* para un método y una variable:

```
class BadMethodNames
{
  static int k;
  static void k( ) {
    // ...
  }
}
```

Signaturas de métodos

Objetivo del tema
Explicar la función de las signaturas de métodos.

Explicación previa
El compilador de C# puede distinguir entre métodos en una clase examinando la signatura de cada método.



El compilador de C# utiliza signaturas para distinguir entre métodos en una clase. Cada método dentro de una clase debe tener una signatura diferente de la de todos los demás métodos declarados en esa clase.

Definición de signatura

La signatura de un método consta del nombre del método, el número de parámetros del método y el tipo y modificador (como **out** o **ref**) de cada parámetro.

Para su información
No está permitido definir métodos sobrecargados que se diferencien únicamente en ref y out.

Los tres métodos siguientes tienen distintas signaturas, por lo que pueden estar declarados en la misma clase.

```
static int LastErrorCode( )
{

}
static int LastErrorCode(int n)
{

}

static int LastErrorCode(int n, int p)
{

}
```

Elementos que no afectan a la signatura

La signatura de un método *no* incluye el tipo retorno. Los dos métodos siguientes tienen la misma signatura, por lo que no pueden estar declarados en la misma clase.

```
static int LastErrorCode(int n)
{
}
static string LastErrorCode(int n)
{
}
```

La signatura de un método *no* incluye los nombres de los parámetros. Los dos métodos siguientes tienen la misma signatura, aunque los nombres de los parámetros sean diferentes.

```
static int LastErrorCode(int n)
{
}
static int LastErrorCode(int x)
{
}
```

Uso de métodos sobrecargados

Objetivo del tema
Explicar cuándo usar métodos sobrecargados.

Explicación previa
Este tema describe algunas situaciones en las que es conveniente sobrecargar métodos.

- Conviene usar métodos sobrecargados si:
  - Hay métodos similares que requieren parámetros diferentes
  - Se quiere añadir funcionalidad al código existente
- No hay que abusar, ya que:
  - Son difíciles de depurar
  - Son difíciles de mantener

Los métodos sobrecargados son útiles si se tienen dos métodos similares que requieren distinto número o tipo de parámetros.

Métodos similares que requieren parámetros diferentes

Imaginemos que una clase contiene un método que envía un mensaje de saludo al usuario. A veces se sabe el nombre del usuario, pero no siempre. Se pueden definir dos métodos diferentes llamados **Greet** y **GreetUser**, como muestra el siguiente código:

```
class GreetDemo
{
    static void Greet( )
    {
        Console.WriteLine("Hola");
    }
    static void GreetUser(string Name)
    {
        Console.WriteLine("Hola " + Name);
    }
    static void Main( )
    {
        Greet( );
        GreetUser("Alex");
    }
}
```

El programa funciona, pero en la clase hay dos métodos que realizan una tarea prácticamente idéntica aunque tengan nombres diferentes. Esta clase se puede reescribir con sobrecarga de métodos como se muestra a continuación:

```
class GreetDemo
{
    static void Greet( )
    {
        Console.WriteLine("Hola");
    }
    static void Greet(string Name)
    {
        Console.WriteLine("Hola " + Name);
    }
    static void Main( )
    {
        Greet( );
        Greet("Alex");
    }
}
```

Adición de funcionalidad al código existente

La sobrecarga de métodos también es útil si se desea añadir nuevas funciones a una aplicación sin tener que hacer demasiados cambios al código existente. Por ejemplo, se podría ampliar el código anterior añadiendo otro método que salude a un usuario de forma diferente según la hora del día, como se muestra a continuación:

```
class GreetDemo
{
    enum TimeOfDay { Manana, Tarde, Noche }

    static void Greet( )
    {
        Console.WriteLine("Hola");
    }
    static void Greet(string Name)
    {
        Console.WriteLine("Hola " + Name);
    }
    static void Greet(string Name, TimeOfDay td)
    {
        string Message = "";

        switch(td)
        {
        case TimeOfDay.Manana:
            Message="Buenos días";
            break;
        case HoraDelDia.Tarde:
            Message="Buenas tardes";
            break;
        case HoraDelDia.Noche:
            Message="Buenas noches";
            break;
        }
        Console.WriteLine(Mensaje + " " + Name);
    }
    static void Main( )
    {
        Greet( );
        Greet("Alex");
        Greet("Sandra", HoraDelDia.Manana);
    }
}
```

Cuándo utilizar sobrecarga

El uso excesivo de la sobrecarga de métodos puede hacer que las clases resulten difíciles de depurar y mantener. En general, sólo se debe sobrecargar métodos que tengan funciones muy estrechamente relacionadas pero que difieran en la cantidad o el tipo de datos que necesitan.