

Más conceptos fundamentales de POO

Variables de instancia vs. miembros estáticos

Por omisión, los miembros de una clase son variables de instancia (existe una copia de los datos por cada instancia de la clase y los métodos se aplican sobre los datos de una instancia concreta).

Se pueden definir miembros estáticos que son comunes a todas las instancias de la clase. Lógicamente, los métodos estáticos no pueden acceder a variables de instancia, ni a la variable `this` que hace referencia al objeto actual.

De cualquier forma, no conviene abusar de los miembros estáticos, ya que son básicamente datos y funciones globales en entornos orientados a objetos.

En la sección laboratorio de esta clase, puede ejercitar con el ejercicio correspondiente.

Más miembros de una clase

Constantes (const)

Una constante es un dato cuyo valor se evalúa en tiempo de compilación y, por tanto, es implícitamente estático (p.ej. `Math.PI`).

```
public class MiClase
{
    public const string version = "1.0.0"    ;
    public const string s1 = "abc" + "def"   ;
    public const int i3 = 1 + 2;
    public const double PI_I3 = i3 * Math.PI;

    public const double s = Math.Sin(Math.PI);    //ERROR
... }
```

Campos (fields)

Propiedades

Las propiedades son campos virtuales, al estilo de Delphi o C++Builder. Su aspecto es el de un campo (desde el exterior de la clase no se diferencian) pero están implementadas con código como los métodos. Pueden ser de sólo lectura, de sólo escritura o de lectura y escritura.

Observe el siguiente código de ejemplo:

La clase tiene:

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

namespace Propiedades
{
    public class Button : Control
    {
        private string caption;
        public string Caption
        {
            get { return caption; }
            set
            {
                caption = value; Repaint();
            }
        }

        public void Repaint()
        {
            Console.WriteLine("Repinto el control");
        }
    }
}
```

El programa principal usa la clase:

```
static void Main(string[] args)
{
    Button b = new Button();
    b.Caption = "OK";
    String s = b.Caption;
}
```

Indexadores (indexers)

C# no permite, hablado con rigor, la sobrecarga del operador de acceso a tablas []. Si permite, no obstante, definir lo que llama un indexador para una clase que permite la misma funcionalidad.

Los indexadores permiten definir código a ejecutar cada vez que se acceda a un objeto del tipo del que son miembros usando la sintaxis propia de las tablas, ya sea para leer o escribir. Esto es especialmente útil para hacer más clara la sintaxis de acceso a elementos de objetos que puedan contener colecciones de elementos, pues permite tratarlos como si fuesen tablas normales.

A diferencia de las tablas, los índices que se les pase entre corchetes no tiene porqué ser enteros, pudiéndose definir varios indexadores en un mismo tipo siempre y cuando cada uno tome un número o tipo de índices diferente.

La sintaxis empleada para la definición de un indexador es similar a la de la definición de una propiedad.

```
public class MiClase
{
    ...
    public string this[int x]
    {
        get
        {
            // Obtener un elemento
        }
        set
        {
            // Fijar un elemento
        }
    }
}
```

```

    }
} ...
} ...
MiClase MiObjeto = new MiClase();

```

El código que aparece en el bloque get se ejecuta cuando la expresión **MiObjeto[x]** aparece en la parte derecha de una expresión mientras que el código que aparece en el bloque set se ejecuta cuando **MiObjeto[x]** aparece en la parte izquierda de una expresión.

Algunas consideraciones finales:

- Igual que las propiedades, pueden ser de sólo lectura, de sólo escritura o de lectura y escritura.
- El nombre dado a un indexador siempre ha de ser **this**.

Lo que diferenciará a unos indexadores de otros será el número y tipo de sus índices.

En la sección laboratorio de esta clase, puede ejercitar con el ejercicio correspondiente.

Métodos

Implementan las operaciones que se pueden realizar con los objetos de un tipo concreto. *Constructores, destructores y operadores* son casos particulares de métodos. Las *propiedades* y los *indexadores* se implementan con métodos (get y set).

Como en cualquier lenguaje de programación, los métodos pueden tener parámetros, contener órdenes y devolver un valor (con return).

Por omisión, los parámetros se pasan por valor (por lo que los tipos "valor" no podrían modificarse en la llamada a un método). El modificador **ref** permite que pasemos parámetros por referencia. Para evitar problemas de mantenimiento, el modificador **ref** hay que especificarlo tanto en la definición del método como en el código que realiza la llamada. Además, **la variable que se pase por referencia ha de estar inicializada previamente**.

```

void RefFunction(ref int p)
{
    p++;
}

```

```
} .....
```

```
int x = 10;
```

```
RefFunction(ref x); // x vale ahora 11
```

El modificador **out** permite devolver valores a través de los argumentos de un método. De esta forma, se permite que el método inicialice el valor de una variable. En cualquier caso, la variable ha de tener un valor antes de terminar la ejecución del método. Igual que antes, para evitar problemas de mantenimiento, el modificador **out** hay que especificarlo tanto en la definición del método como en el código que realiza la llamada.

```
void OutFunction(out int p)
```

```
{  
    p = 22;  
} .....
```

```
int x; // x aún no está inicializada
```

```
OutFunction(out x);
```

```
Console.WriteLine(x); // x vale ahora 22
```

Recordamos la sobrecarga de métodos mencionada en la clase anterior: Como en otros lenguajes, el identificador de un método puede estar sobrecargado siempre y cuando las firmas de las distintas implementaciones del método sean únicas (la firma tiene en cuenta los argumentos, no el tipo de valor que devuelven).

```
void Print(int i);
```

```
void Print(string s);
```

```
void Print(char c);
```

```
void Print(float f);
```

```
int Print(float f); // Error: Firma duplicada
```

Arrays de parámetros

Como en C, un método puede tener un número variable de argumentos.

```

public static int Suma(params int[] intArr)
{
    int sum = 0;
    foreach (int i in intArr) sum += i;
    return sum;
}
.....
int sum1 = Suma(13, 87, 34); // sum1 vale 134
Console.WriteLine(sum1);
int sum2 = Suma(13, 87, 34, 6); // sum2 vale 140
Console.WriteLine(sum2);

```

produce el siguiente resultado:

```

134
140

```

Más sobre constructores

Métodos especiales que son invocados cuando se instancia una clase (o un struct).

- Se emplean habitualmente para inicializar correctamente un objeto.
- Como cualquier otro método, pueden sobrecargarse.
- Si una clase no define ningún constructor se crea un constructor sin parámetros (*implícito*).
- No se permite un constructor sin parámetros para los struct.

C# permite especificar código para inicializar una clase mediante un **constructor estático**. El constructor estático se invoca una única vez, antes de llamar al constructor de una instancia particular de la clase o a cualquier método estático de la clase. Sólo puede haber un constructor estático por tipo y éste no puede tener parámetros.

Más sobre destructores

Se utilizan para liberar los recursos reservados por una instancia (justo antes de que el recolector de basura libere la memoria que ocupe la instancia).

A diferencia de C++, la llamada al destructor no está garantizada por lo que tendremos que utilizar una orden *using* e implementar el interfaz *IDisposable* para asegurarnos de que se

liberan los recursos asociados a un objeto). Sólo las clases pueden tener destructores (no los struct).

Para más información consulte este link:

<https://blogs.msdn.microsoft.com/daniem/2009/08/18/cosas-que-deberas-saber-sobre-los-de-structores-en-net/>

En la sección laboratorio de esta clase, puede ejercitar con el ejercicio correspondiente.

Sobrecarga de operadores

Como en C++, se pueden sobrecargar (siempre con un método *static*) algunos operadores *unarios* (+, -, !, ~, ++, --, **true**, **false**) y *binarios* (+, -, *, /, %, &, |, ^, ==, !=, <, >, <=, >=, <, >).

- No se puede sobrecargar el acceso a miembros, la invocación de métodos, el operador de asignación ni los operadores **sizeof**, **new**, **is**, **as**, **typeof**, **checked**, **unchecked**, **&**, y **?:**.
- Los operadores **&** y **||** se evalúan directamente a partir de los operadores **&** y **|**.
- La sobrecarga de un operador binario (por ejemplo *****), sobrecarga implícitamente el operador de asignación correspondiente (***=**).

En la sección laboratorio de esta clase, puede ejercitar con el ejercicio correspondiente.

Conversiones de tipo

Dado que C# tiene tipos estáticos en tiempo de compilación, después de declarar una variable, no se puede volver a declarar ni se le puede asignar un valor de otro tipo a menos que ese tipo sea convertible de forma implícita al tipo de la variable. Por ejemplo, **string** no se puede convertir de forma implícita a **int**. Por tanto, después de declarar **i** como un valor **int**, no se le puede asignar la cadena "Hola", como se muestra en el código siguiente:

```
int i; i = "Hola"; // error CS0029: Cannot implicitly
convert type 'string' to 'int'
```

Pero es posible que en ocasiones sea necesario copiar un valor en una variable o parámetro de otro tipo. Por ejemplo, una variable entera que se necesita pasar a un método cuyo parámetro es de tipo `double`. O es posible que tenga que asignar una variable de clase a una variable de tipo de interfaz. Estos tipos de operaciones se denominan *conversiones de tipos*. En C#, se pueden realizar las siguientes conversiones de tipos:

- **Conversiones implícitas:** no se requiere ninguna sintaxis especial porque la conversión tiene seguridad de tipos y no se perderá ningún dato. Los ejemplos incluyen conversiones de tipos enteros más pequeños a más grandes, y conversiones de clases derivadas a clases base.
- **Conversiones explícitas:** las conversiones explícitas requieren un operador de conversión. La conversión es necesaria si es posible que se pierda información en la conversión, o cuando es posible que la conversión no sea correcta por otros motivos. Entre los ejemplos típicos están la conversión numérica a un tipo que tiene menos precisión o un intervalo más pequeño, y la conversión de una instancia de clase base a una clase derivada.
- **Conversiones definidas por el usuario:** las conversiones definidas por el usuario se realizan por medio de métodos especiales que se pueden definir para habilitar las conversiones explícitas e implícitas entre tipos personalizados que no tienen una relación de clase base-clase derivada. Para obtener más información, vea [Operadores de conversión](#).

C# no permite definir conversiones entre clases que se relacionan mediante herencia. Dichas conversiones están ya disponibles: de manera implícita desde una clase derivada a una clase base y de manera explícita a la inversa.

En la sección laboratorio de esta clase, puede ejercitar con el ejercicio correspondiente.