educación IT
CODE YOUR FUTURE

Programación Web .NET Core

Módulo 1

educación IT
CODE YOUR FUTURE

Más conceptos fundamentales de POO

Variables de instancia vs. miembros estáticos

Por defecto, **los miembros de una clase son variables de instancia** (hay una copia de los datos por cada instancia de la clase y los métodos se aplican en los datos de una instancia concreta).

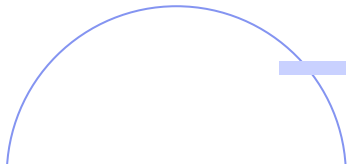
Se pueden definir **miembros estáticos que son comunes a todas las instancias de la clase.** Los métodos estáticos no pueden acceder a variables de instancia, ni a la variable **`this`** que hace referencia al objeto actual. **No es conveniente abusar de los miembros estáticos**, ya que son básicamente datos y funciones globales en entornos orientados a objetos.

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 (por ejemplo: `Math.PI`).

```
publicclassMiClase
{
public const  stringversion = "1.0.0";
  public const  string s1 = "abc" + "def";
publicconstint i3 = 1 + 2;
publicconstdouble PI_I3 = i3 * Math.PI;

publicconstdouble s = Math.Sin(Math.PI);      //ERROR
.
.
.

}
```



Campos (fields)

Un ***campo*** es una variable que almacena datos en una clase (referencias a instancias de clases, estructuras o `arrays`).

Campos de sólo lectura (`readonly`): Son similares a las constantes. Su valor se inicia en tiempo de ejecución (en su declaración o en el constructor). A diferencia de las constantes, si cambiamos su valor, no hay que recompilar los clientes de la clase. Y los campos de sólo lectura pueden ser variables de instancia, o variables estáticas.

```
publicclassMiClase
{
publicstaticreadonlydouble d1 = Math.Sin(Math.PI);
publicreadonlystring s1;

publicMiClase(string s)
    {
        s1 = s;
    }
}
   ......
MiClasemio = newMiClase("Prueba");
Console.WriteLine(mio.s1);
Console.WriteLine(MiClase.d1);

Produce como resultado:
1,22460635382238E-16
```

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.

La clase tiene:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Linq;
usingSystem.Text;
usingSystem.Threading.Tasks;
usingSystem.Windows.Forms;

namespace Propiedades
{
publicclassButton : Control
    {
private      string     caption;
publicstring      Caption
        {
get { returncaption; }
```

...

```
set
            {
caption = value; Repaint();
            }
        }

publicvoidRepaint()
        {
Console.WriteLine("Repinto el control");
        }
    }

}
```

El programa principal usa la clase:

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

Indexadores *(indexers)*

C# no permite la sobrecarga del operador de acceso a tablas [ ]. Sí se puede definir 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 pasen entre corchetes no tienen porqué ser enteros. Pudiéndose definir varios indexadores en un mismo tipo; esto siempre y cuando cada uno tome un número o tipo de índices diferente.

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

```
publicclassMiClase
{
.
.
.
publicstringthis[int x]
    {
get
        {
// Obtener un elemento
        }
set
        {
// Fijar un elemento

}
}
```

...

```
.
.
.
}
.
.
.
MiClaseMiObjeto = newMiClase();
```

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 en el momento en que **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.

Métodos

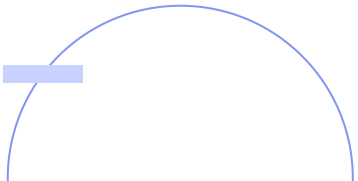
Implementan las **operaciones que pueden hacer los objetos de un tipo concreto**.
Constructores, destructores y operadores son casos particulares de métodos. Las propiedades y los indexadores se implementan con métodos (`get` y `set`).

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 se pasen parámetros por referencia.

Para evitar problemas de mantenimiento, se debe especificar el modificador **ref** 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**.

```
voidRefFunction(refint p)
{
p
+
+
;
}

int x = 10;

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



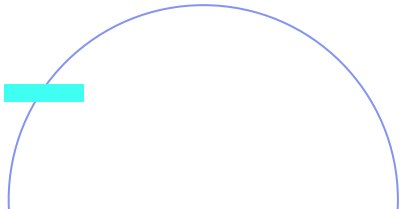
El modificador **out** permite devolver valores a través de los argumentos de un método. De esta forma, se permite que el método inicialice el valor de una variable. En cualquier caso, la variable ha de tener un valor antes de terminar la ejecución del método. Igual que antes, para evitar problemas de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

```
voidOutFunction(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:

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 parámetros de entrada, no el tipo de valor que devuelven).

```
voidPrint(int i);
voidPrint(string s);
voidPrint(char c);
voidPrint(float f);
intPrint(float d);  // Error: Firma duplicada
```



Arrays de parámetros

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

```
publicstaticint Suma(paramsint[]intArr)
{
int sum = 0;
foreach (int   i  inintArr) 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

Son **métodos especiales que se invocan cuando se instancia una clase.**

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

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 del momento en 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 se tendrá que utilizar una orden **`using`** e implementar el interfaz llamada **`IDisposable`** para asegurar que se liberen los recursos asociados a un objeto). Sólo las clases pueden tener destructores (no los `struct`).

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 (`*=`).

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 podemos ver en el código que está a continuación:

```
int i;  i = "Hola"; // error CS0029:
Cannotimplicitlyconverttype '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 conversiones de tipos que veremos en la pantalla siguiente.

Conversiones de tipos posibles en C#

- **Conversiones implícitas:** no se requiere ninguna sintaxis especial porque la conversión tiene seguridad de tipos y no se perderá ningún dato. Por ejemplo, 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. Ejemplos típicos: 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 hacen 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. Más información en el siguiente link: 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.

educación IT
CODE YOUR FUTURE

¡Sigamos trabajando!