Introducción a C#

4.3.- DEFINIENDO MIEMBROS DE UNA CLASE

Dentro de la definición de la clase, se definen cada uno de los miembros :

- CAMPOS

- PROPIEDADES

- MÉTODOS

Todos ellos comparten una serie de modificadores de acceso proporcionando así diferentes niveles de accesibilidad

Introducción a C#

Niveles de accesibilidad → modificadores de acceso :

· **public** → miembros son accesibles desde cualquier parte

· **internal** → miembros son accesibles desde cualquier parte del fichero donde está definido

· **private** → miembros son sólo accesibles desde el código dentro de la misma clase

· **protected** → miembros son sólo accesibles desde el código dentro de la clase y las clases derivadas
· **protected internal** → miembros son sólo accesibles desde el código de las clases derivadas o desde cualquier parte del fichero donde está definido.
· **protected private** → miembros son accesibles desde dentro de la misma clase o desde clases derivadas en el mismo fichero

También se pueden definir como **static**

Introducción a C#

DEFINIENDO CAMPOS :

Se definen según el formato de definición estándar de variables, junto con los modificadores vistos anteriormente.

<modifier(s)> <type> <FieldName> [= value];

Los campos pueden ser además :

- **readonly** → Los campos sólo pueden ser asignados inicialmente o durante la ejecución del constructor.

- **const** → Los campos actúan como constantes, asignándose inicialmente.
Son estáticos aunque no se ponga la palabra clave **static.**
(da un error si se pone !)

Introducción a C#

DEFINIENDO MÉTODOS :

- Los métodos se definen usando el formato estándar de definición de funciones, junto con los modificadores vistos anteriormente.

<modifier(s)> <returnType> <methodName>(<type1><param1>, ...) ;

Se pueden utilizar las siguientes palabras claves con los métodos :

· **virtual** → El método puede ser sobrescrito e invalidado

· **sealed** → El método no puede ser sobreescrito

· **abstract** → El método ha de ser sobrescrito en la clase derivada si la clase derivada no es abstracta

· **override** → El método sobrescribe un método de la clase base.
Si se utiliza override se puede especificar también sealed (**override sealed**) para indicar que no se pueden hacer más modificaciones en las clases derivadas.

Introducción a C#

DEFINIENDO PROPIEDADES :

Las propiedades se definen de forma similar a los campos.

Mediante las propiedades se puede realizar un procesamiento adicional antes de modificar el estado de la propiedad (incluso no modificarlo)

Las propiedades pueden incorporar dos métodos, llamados “accessors”:

- un método get para obtener el valor de la propiedad
  → cuando se accede a la propiedad

- un método set para poner el valor de la propiedad
  → cuando se asigna un valor

Estos métodos se pueden también utilizar para controlar el nivel de acceso a la propiedad.

Introducción a C#

Se puede omitir uno u otro de los métodos para crear propiedades de sólo lectura o propiedades de sólo escritura.

(naturalmente esto sólo es aplicable a código externo; dentro de la clase el acceso al valor de la propiedad es siempre posible)

Omitir los dos métodos no tiene sentido, ya que dejaríamos la propiedad sin ningún uso posible.

Los métodos get y set pueden tener modificadores de accesibilidad, pero sólo pueden ser más restrictivos que el modificador de acceso de la propiedad

La sintaxis básica de una propiedad está compuesto por los modificadores de acceso, un tipo, un nombre y uno o ambos métodos accessor que contiene el procesamiento de la propiedad según el siguiente esquema :

Introducción a C#

```
public int MyIntProp
{
    get
    {
        // Property get code.
    }
    set
    {
        // Property set code.
    }
}
```

Las propiedades se utilizan a menudo con un campo definido como privado controlando el acceso al campo desde el código externo.

El código externo no puede acceder directamente al campo porque es privado, pero puede hacer uso de la propiedad para acceder a él mediante los métodos get and set.

Introducción a C#

```
// Field used by property.
private int myInt;
// Property.
public int MyIntProp
{
   get { return myInt; }
   set { myInt = value; }
}
```

Se utiliza la palabra clave **value** dentro del método set para referirse al valor recibido por el usuario de la propiedad. (debe ser del tipo definido para la propiedad)

Los métodos accessors pueden tener sus propios modificadores de acceso con la única limitación que no está permitido que los métodos accessor tengan más accesibilidad que la propiedad a la que pertenecen.

Las propiedades pueden utilizar las palabras claves **virtual, override (override sealed) o abstract** al igual que los métodos, cosa que no es posible con los campos.

Introducción a C#

C# 6 introduce una nueva característica llamada propiedades basadas en expresiones (expression based properties) al igual que introducía los métodos basados en expresiones.

→ De esta forma, se reduce la extensión de la propiedad a una única línea de código

- El siguiente código crearía una propiedad de sólo lectura que nos devolvería el doble del contenido del campo myDoubleInt

```
// Field used by property.
private int myDoubledInt = 5;
// Property.
public int MyDoubledIntProp => (myDoubledInt * 2);
```

Ejemplo 1

Introducción a C#

PROPIEDADES AUTOMÁTICAS

- La idea es definir una propiedad con una sintaxis simplificada y el compilador C# rellena el resto

- Para crear una propiedad automática se utiliza el siguiente código :

<modificador(es)><tipo><NombrePropiedad> { get; set; }
(los métodos get y set pueden ir precedidos de modificadores de acceso)

C# completa el código de la siguiente forma :
- crea un campo privado (no conocemos su nombre, se define durante la compilación)
- completa el código de get y set para obtener y poner el valor al campo privado a través de la propiedad (no podemos cambiar el código por defecto, porque desconocemos donde se guarda el dato)
- sí podemos modificar el modificador de acceso de get y set

Introducción a C#

OJO ! → Al campo privado no hay manera de acceder si no es con la propiedad, ya que en el momento de programación de código no conocemos el nombre (ni siquiera para el código interno de la clase)

Limitación : Se deben definir los métodos get y set, aunque si los hacemos privados tendremos propiedades de sólo lectura y sólo escritura para código externo.

Sin embargo, C# 6 proporciona las llamadas propiedades automáticas de sólo lectura y los inicializadores para propiedades automáticas.

<modificador(es)> <tipo> <NombrePropiedad> {get; } = value;

→ Estaremos generando un tipo de dato inmutable !

Introducción a C#

SOBRECARGA DE METODOS

a) Métodos virtual → Métodos override / override sealed √
Métodos sin override

b) Independientemente que el método sea virtual o no, el método se puede sobrecargar en la clase derivada, ocultando el método de la clase base para la clase hija (utilizar new para evitar un warning)

- Si no es virtual no se puede invalidar, solo ocultar!

- Si es virtual se puede ocultar la implementación sobrescribiendo el método sin la palabra clave override precedido de new
  - Y se puede invalidar sobrescribiendo el método con la palabra clave override

Invalidar se refiere a que el método override reemplaza el método de la clase base, y la clase base accedería al método de la clase hija si estamos utilizando polimorfismo.

Introducción a C#

Diferencia entre poner override o no :

- La sobrecarga de un método virtual mediante un método override / override sealed reemplaza la implementación en la clase base de forma que la llamada al método desde la clase base cuando se utiliza polimorfismo ejecuta el método sobreescrito.

- La sobrecarga de un método virtual o no, mediante un método sin la palabra override no reemplaza la implementación en la clase base de forma que la llamada al método desde la clase base cuando se utiliza polimorfismo ejecuta el método de la clase base

Ejemplo 2

Introducción a C#

Independientemente de uno u otro método el código del método de la clase base está disponible desde la clase derivada.

Se utiliza cuando :

- Se desea ocultar un método heredado public de usuarios de una clase derivada, pero se quiere acceder a su funcionalidad desde la clase derivada
- Cuando se quiere añadir funcionalidad a un método de la clase base en vez de reemplazarlo con una nueva implementación

→ Entonces es útil utilizar la palabra clave **base**

→ hace referencia a la implementación de la clase base desde la clase derivada

Limitación : No se puede acceder a las métodos **private**.

base actúa utilizando instancias de objeto, por lo tanto es un error utilizarlo desde un método **static**

Ejemplo3

Introducción a C#

Palabra clave **this**

- Se puede utilizar desde dentro de un método de la clase y hace referencia a la instancia del objeto desde el que se está llamando.

- Se utiliza para pasar una referencia a la instancia del objeto actual a un método.

El método debe aceptar un parámetro de un tipo compatible con la clase, es decir :

- tipo de la misma clase
- tipo de una clase de la que esta clase derive
- tipo de la clase System.Object

- Se utiliza también para nombrar de forma completa los miembros de la clase (por si se creara una variable con el mismo tipo local)

Ejemplo 4, Ejemplo 5

Introducción a C#

CLASES ANIDADAS

- Las clases se pueden definir dentro de un namespace

- Las clases se pueden definir también dentro de otras clases
  → clases anidadas

- Para las clases anidadas se pueden utilizar todos los modificadores de acceso disponibles no sólo internal y public

- Las clases anidadas son vistas como tipos

- Se podria incluso sobreescribir un tipo de definición de clase desde una clase heredada

  → Puedes utilizar la palabra clave new para ocultar un tipo de definición de clase heredada de una clase base

Introducción a C#

- Para instanciar una clase anidada desde fuera de la clase contenedora se ha de utilizar el nombre completamente cualificado

MyClass.MyNestedClass myObj = new MyClass.MynestedClass();

- Una razón para utilizar clases anidadas es para definir clases que son privadas a la clase contenedora de forma que ningún código del namespace tenga acceso a ella, pero sí dentro de la clase

- Otra razón sería que las clases anidadas tienen acceso a los métodos private y protected de la clase contenedora

```
public class MyClass
{
    public class MyNestedClass
    {
        public int NestedClassField;
    }
}
```

Ejemplo 6

Introducción a C#

IMPLEMENTACIÓN DE INTERFACES

```
interface IMyInterface
{
    // Interface members.
}
```

- No se permiten modificadores de acceso
    - Todos los interfaces son public

- Los interfaces no pueden tener código alguno

- Los interfaces no pueden definir miembros de campo

- Los miembros de un interfaz no pueden definirse con las palabras claves static, virtual, abstract o sealed

- No se pueden definir tipo alguno dentro de un interfaz

Introducción a C#

- Se pueden definir miembros usando la palabra clave new, si se quieren ocultar miembros heredados de interfaces base.

- Las propiedades definidas dentro de interfaces definen uno o ambos de los métodos accessors
  → no especifica como se almacena el dato de la propiedad (los interfaces no pueden definir campos)

- Los interfaces se pueden definir como miembros de clases, pero no de interfaces
  → No se pueden definir tipo alguno dentro de un interfaz

Introducción a C#

CLASES QUE IMPLEMENTAN INTERFACES :

- Deben contener implementaciones para todos los miembros de esa interfaz ajustándose a la firma y debe ser public

- Es posible implementar miembros de los interfaces utilizando la palabra clave virtual o abstract, pero no static o const

- La implementación de un miembro del interfaz puede proceder de la clase base

- Heredar de una clase base que implementa un interfaz significa que ese interfaz es soportado por la clase derivada, pudiendo ser sobrescrito por la clase derivada.

→ Si se oculta un miembro de la clase base se referirá al de la clase base si se utiliza desde un tipo del interfaz

Introducción a C#

IMPLEMENTACIÓN EXPLICITA DE MIEMBROS DE INTERFACE

- Las variables de tipo interfaz son posibles

- Pueden contener cualquier objeto de cualquier clase que implemente el interfaz

- Los miembros de un interfaz pueden ser definidos implicitamente (definirlo tal cual) o explicitamente (prefijado por el nombre del interfaz)

- Si una clase implementa un miembro de forma explicita, entonces ese miembro sólo puede ser accedido por variables del tipo del interfaz

Introducción a C#

- La implementación de propiedades de un interfaz no necesita corresponder exactamente la definición en el interfaz si son implementados de forma IMPLICITA

Por ejemplo, sería posible definir un accessor que no estuviera en la definición

→ se puede utilizar también para modificar el acceso public por defecto de un accessor dentro de un interfaz para que tenga un tipo más restrictivo

Ejemplo 7

Introducción a C#

CLASES PARCIALES

- Una clase puede definirse en más de un fichero

→ Hay que definirla como partial class

→ La palabra clave partial es necesario utilizarla en todas las clases que implementen una porción de la clase.

- Una clase parcial que implemente un interface, el interface se aplica a la clase completa

- Una clase parcial puede incluir una clase base en una o en varias clases parciales, pero siempre la misma clase base

→ las clases C# sólo pueden heredar de una clase base.

Introducción a C#

DEFINICIÓN DE METODOS PARCIALES

- Un método parcial se definen en una clase parcial sin el cuerpo del método, pero se implementa en otra clase parcial

→ en ambos casos se necesita poner la palabra clave partial

- Los métodos parciales pueden ser static, pero siempre son private y no pueden devolver ningún valor

- No pueden utilizar parámetro out, pero sí un parámetro ref

- No pueden ser virtual, abstract, override, new o sealed

→ La utilización de metodos parciales está ligada a la eficiencia del código

Ejemplo 8

Introducción a C#

- Cuando se compila código que contiene un método parcial sin una implementación, el compilador elimina el método de forma completa incluida la llamada al método.

→ Utilización : Se define un método que el desarrollador elije implentarlo o no dependiendo de la situación no incurriendo en una pérdida de rendimiento de la aplicación

→ Mejora del rendimiento de la aplicación

- Si el método se implementara sin código pero no partial, la llamada existiría y se ejecutaría aunque no ejecutara código alguno.