

## Modificadores de acceso

Los modificadores de acceso nos permiten especificar quién puede usar un tipo o un miembro del tipo, de forma que nos permiten gestionar la encapsulación de los objetos en nuestras aplicaciones:

- Una clase definida en un *namespace* puede ser **public** o **internal**:
  - **public**: la clase se podrá utilizar por todos.
  - **internal**: la clase solo se podrá utilizar por los componentes del *assembly* que la contengan (un *assembly* es básicamente una dll que esta compilada a código intermedio).
- Los miembros de una *clase* pueden ser **public**, **private**, **protected**, **internal** o **protected internal**
- Los miembros de un *struct* pueden ser **public** , **private** o **internal**

Modificador de acceso	Un miembro del tipo T (por ejemplo T es una clase) definido en el assembly A es accesible...
public	desde cualquier lugar
private (por omisión)	sólo desde dentro de T (por omisión)
protected	desde T y los tipos derivados de T (es decir visible desde una clase y sus clases derivadas)
internal	desde los tipos incluidos en A (es decir visible desde todas las clases del mismo assembly)
protected internal	desde T, los tipos derivados de T y los tipos incluidos en A (es decir visible desde la clase, desde sus clases derivadas y desde todas las clases del mismo assembly)

## Herencia

Hemos tenido una breve introducción sobre el concepto de herencia en POO. Es momento de profundizar ahora ese concepto.
---

El mecanismo de **herencia** es uno de los pilares fundamentales en los que se basa la programación orientada a objetos. Es un mecanismo que *permite definir nuevas clases a partir de otras ya definidas*. Si en la definición de una clase indicamos que ésta deriva de otra, entonces la primera -a la que se le suele llamar **clase hija** o **clase derivada**- será tratada por el compilador automáticamente como si su definición incluyese la definición de la segunda -a la que se le suele llamar **clase padre** o **clase base**. Las clases que derivan de otras se definen usando la siguiente sintaxis:

```
class <claseHija> : <clasePadre>
{
    <miembrosHija>
}
```

A los miembros definidos en la clase hija se le añadirán los que hubiésemos definido en la clase padre: la clase derivada "hereda" de la clase base.

La palabra clave **base** se utiliza para obtener acceso a los miembros de la clase base desde una clase derivada.

C# sólo permite **herencia simple**.

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

## Redefinición de métodos

Siempre que se redefina un método que aparecía en la clase base, hay que utilizar explícitamente la palabra reservada **override** y, de esta forma, se evitan redefiniciones accidentales (una fuente de errores en lenguajes como Java o C++).

Si en la clase **Point** agregamos el método **ToString**, es decir lo heredamos de *System.Object* y lo redefinimos:

```
public class Point
{
    public override string ToString()
    { return ( "[" + this.X + ", " + this.Y + "]" ); } ...
}
```

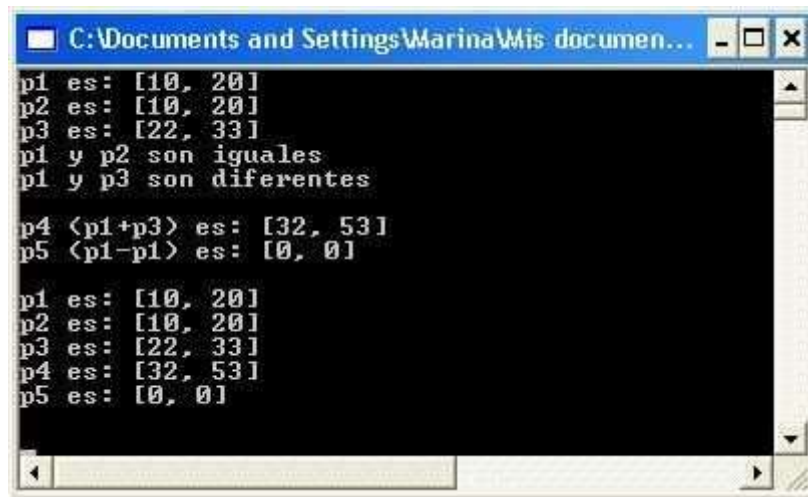
y sustituimos las instrucciones del tipo:

```
Console.WriteLine("p1 es: ({0},{1})", p1.X, p1.Y);
```

por:

```
Console.WriteLine("p1 es: " + p1.ToString());
```

el resultado de la ejecución es:



```
p1 es: [10, 20]
p2 es: [10, 20]
p3 es: [22, 33]
p1 y p2 son iguales
p1 y p3 son diferentes

p4 <p1+p3> es: [32, 53]
p5 <p1-p1> es: [0, 0]

p1 es: [10, 20]
p2 es: [10, 20]
p3 es: [22, 33]
p4 es: [32, 53]
p5 es: [0, 0]
```

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

## Métodos virtuales

Un método es **virtual** si puede redefinirse en una clase derivada. **Los métodos son no virtuales por omisión.**

- Los métodos *no virtuales* no son polimórficos (no pueden reemplazarse) ni pueden ser abstractos.
- Los métodos *virtuales* se definen en una clase base (empleando la palabra reservada **virtual**) y pueden ser reemplazados (empleando la palabra reservada **override**) en las subclases (éstas proporcionan su propia -específica- implementación).

Generalmente, contendrán una implementación por omisión del método (si no, se deberían utilizar *métodos abstractos*).

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