

# Tema 9.2

Descargar estos apuntes en [pdf](#) o [html](#)

## Índice

- ▼ Diferentes Polimorfismos Diferentes Formas
  - Definiciones de polimorfismo
  - Parámetros opcionales o por defecto
- ▼ Propiedades en CSharp
  - Recordando que era la encapsulación
  - ▼ Encapsulación a través de propiedades
    - Usando las propiedades definidas
    - Evolución del uso de propiedades a lo largo del tiempo
    - Propiedades '*autoimplementadas*'
    - Miembros con cuerpo de expresión

# Diferentes Polimorfismos Diferentes Formas

## Definiciones de polimorfismo

### 1. Polimorfismo de datos o inclusión

- **Ya lo hemos visto** cuando estudiamos el concepto de herencia y downcasting. Se basa en el Upcasting o **Principio de Sustitución de Liskov**
- Es de datos porque, tenemos un objeto o dato con diferentes formas dependiendo del tipo con que lo referenciamos.
- Además de este, tendremos otras formas de polimorfismo...

### 2. Polimorfismo paramétrico o tipos genéricos

- Lo vamos a tratar más adelante en este tema 9.

### 3. Polimorfismo funcional o sobrecarga

- Al igual que el de datos, **ya lo hemos visto** y usado con anterioridad. Pero ahora es cuando lo definiremos formalmente como una característica de los lenguajes OO.
- Es la **capacidad de definir operaciones o métodos con el mismo identificador** o nombre. Siempre y cuando, la signatura cambie.
- Recuerda que en C# dos métodos tienen diferente signatura si:
  - Tienen diferente tipo de retorno.
  - Tienen diferente número de parámetros.
  - Teniendo el mismo número de parámetros, algún tipo es diferente.
  - Teniendo el mismo número de parámetros y el mismo tipo alguno tiene el modificador ref o out.

```
Class SobrecargaValida
{
    public void MetodoA(int x) { ; }
    public void MetodoA(ref int x) { ; }
}
```

```
Class SobrecargaInvalida
{
    public void MetodoA(out int x) { ; }
    public void MetodoA(ref int x) { ; }
}
```

- ¿Para qué se usa el polimorfismo funcional?
  - Por ejemplo, lo vimos cuando tuvimos que **definir varios constructores** en una clase.
  - Para **evitar el uso de parámetros opcionales o por defecto** en los métodos.

## Parámetros opcionales o por defecto

Una llamada a un método debe proporcionar los argumentos reales para todos los parámetros, sin embargo **se pueden omitir aquellos argumentos** de [parámetros opcionales](#).

👉 **Importante:** Los parámetros opcionales **se definen al final de la lista de parámetros**, después de los parámetros necesarios. Si el autor de la llamada proporciona un argumento para algún parámetro de una sucesión de parámetros opcionales, **debe proporcionar argumentos para todos los parámetros opcionales anteriores** o en su lugar indicar el identificador del parámetro formal.

Veamos esto último a través de un **ejemplo**, de sintaxis:

```

static class Ejemplo
{
    static void Metodo(
        string cadenaRequerida, int enteroRequerido,
        // No puedo definir ningún opcional antes del último requerido.
        string cadenaOpcional = "", int enteroOpcional = 10)
    { ... }

    static void Main()
    {
        // Correcto
        Metodo("Cadena obligatoria", 3, "Cadena Opcional", 33);

        // Correcto enteroOpcional = 10
        Metodo("Cadena obligatoria", 3, "Cadena Opcional");

        // Correcto cadenaOpcional = "" y enteroOpcional = 10
        Metodo("Cadena obligatoria", 3);

        // Si sabemos el nombre del identificador del parámetro en el método...
        Metodo("Cadena obligatoria", 3, enteroOpcional: 10);
        // Correcto cadenaOpcional = ""

        Metodo("Cadena obligatoria", 3, 10);    // Incorrecto
        Metodo("Cadena obligatoria", 3, , 10); // Incorrecto
    }
}

```

Se pueden definir en multitud de lenguajes como C#, Python, PHP, Javascript, Kotlin, etc. Sin embargo, **Java no los permite** porque tienen inconvenientes:

- 🧠 Mal usados, **pueden dar lugar a baja cohesión** (métodos '*navaja suiza*' o que hacen muchas cosas según los parámetros que le lleguen).
- Ralentizan la ejecución.
- Lleva a confusión a los usuarios de una clase.

👉 **Importante:** Por las razones anteriores. **No deberíamos usarlos en métodos públicos.**  
 (👁️ *fíjate que Microsoft apenas los usa en sus BCL y sí la sobrecarga*)

### 🎓 Caso de estudio:

Vamos a tratar un ejemplo de como evitar parámetros opcionales en los métodos públicos o **en lenguajes que no nos los permitan como Java**, a través de C#.

Si recordamos de temas anteriores, definimos una estructura **Punto2D** que ahora va a tener el método **Desplaza** con el valor del ángulo a 0 de forma opcional.

```
struct Punto2D
{
    public readonly double X;
    public readonly double Y;

    public Punto2D(in double x, in double y) { Y = y; X = x; }

    public Punto2D Desplaza(in double distancia, double anguloGrados = 0D)
    {
        double anguloRadianes = anguloGrados * Math.PI / 180D;
        double fila = Y + distancia * Math.Sin(anguloRadianes);
        double columna = X + distancia * Math.Cos(anguloRadianes);
        return new Punto2D(fila, columna);
    }

    public override string ToString() { return $"({X:G2} - {Y:G2})"; }
}
```

Ahora en un programa podríamos instancias un objeto valor Punto2D

```
Punto2D p = new Punto2D(2D, 4D);
```

y continuación hacer ...

```
p.Desplaza(4D);
```

Como el parámetro formal **anguloGrados** es opcional, podremos llamar al método **Desplaza** sin especificarlo y en ese caso tomará su valor por defecto **0D** grados, desplazando el punto 4 unidades a la derecha.

## 📖 Caso de estudio (continuación...)

💡 ¿Cómo deberíamos refactorizar el código anterior **usando polimorfismo funcional** o sobrecarga?

La forma más común sería la siguiente...

```
struct Punto2D
{
    // Definimos como privado el método a sobrecargar para no repetir el código
    // además para que no haya conflicto de nombres le ponemos un _
    private Punto2D _Desplaza(in double distancia, double anguloGrados)
    {
        double anguloRadianes = anguloGrados * Math.PI / 180d;
        double columna = X + distancia * Math.Cos(anguloRadianes);
        double fila = Y + distancia * Math.Sin(anguloRadianes);
        return new Punto2D(columna, fila);
    }

    // Definimos las sobrecargas públicas, con los parámetros posibles.
    public Punto2D Desplaza(ushort numPosiciones)
    {
        // Aquí decidiremos el valor por defecto.
        return _Desplaza(numPosiciones, 0d);
    }

    public Punto2D Desplaza(ushort numPosiciones, double angulo)
    {
        return _Desplaza(numPosiciones, angulo);
    }
}
```

”

*Controlling complexity is the  
essence of computer programming.*

- Brian Kernighan.

”

# Propiedades en CSharp

Una propiedad es una mezcla entre el concepto de campo y el concepto de método. Externamente **es accedida como un campo**, pero internamente es posible **asociar código a ejecutar** en cada asignación o lectura de su valor.

👉 Pero la idea principal es que ... **'es la forma que tiene C# de implementar los métodos accedidos y mutadores.'**

## Recordando que era la encapsulación

Antes de hablar de las propiedades **vamos a repasar el concepto de encapsulación** que vimos en temas anteriores y que **va asociado al uso de propiedades**.

🗣️ Recapitulación de los **Objetivos**:

- Evitar que un cliente de mis clases puedan dejar objetos instanciados de las mismas en un estado inadecuado.
- **Ocultar detalles de la implementación** de una clase.
- **Disminuir el acoplamiento**, esto es, realizar cambios o actualizaciones en la clases sin preocuparnos cómo están siendo usadas.

🗣️ Recapitulación de **modificadores de acceso** que hay para clases, tipos, campos y métodos:

- **private** : Accesible solo desde la clase. Es lo que deberíamos poner por defecto a los mutadores y si tenemos alguna duda.
- **public** : Accesible por todos. Puede ser una accesibilidad por defecto para nuestros accedidos.
- **protected** : Accesible solo desde la clase o las subclases.
- **internal** : Accesible solo desde clases del artefacto actual.
- **protected internal new** : Accesible solo desde clases del artefacto actual y además sean subclases de la clase donde se ha definido.

🗣️ Recapitulación de cómo las hemos implementado hasta ahora:

- Hemos utilizado la sintaxis de otros lenguajes como **Java** o **C++** para definir los accedidos y mutadores que era, básicamente, usar métodos con una notación especial.

```

class Clase
{
    private <Tipo> idCampo;

    private <Tipo> Get<IdCampo>()
    {
        return <idCampo>
    }

    public void Set<idCampo>(<Tipo> <idCampo>)
    {
        this.<idCampo> = <idCampo>
    }
}

```

## Encapsulación a través de propiedades


Como ya hemos comentado, **las propiedades son** un '*azúcar sintáctico*' incluido por el lenguaje C# para abreviar la manera de implementar la encapsulación. De tal manera que la forma básica de implementar los getter y los setter en C# será...

```

class Clase
{
    private <Tipo> idCampo;

    <Tipo> <IdCampo> {
        set
        {
            <idCampo> = value;
        }
        get
        {
            return <idCampo>;
        }
    }
}

```

 **Nota:** La sintaxis anterior tiene adaptaciones conforme ha ido evolucionando en lenguaje, para **simplificar al máximo su uso**, según casos de uso. Más adelante, una vez tratemos el concepto de propiedad, abordaremos dichas simplificaciones.



Veamos cómo aplicar y usar la nueva sintáxis a través de la clase **Escritor** que implementamos en el tema 5 y a la que hemos añadido ciertas actualizaciones fáciles de apreciar a primera vista.

A través de ella vamos a ver la forma de usar propiedades y cómo llevar el concepto a las versiones más modernas del lenguaje. Para ello, recordemos pues el código implementando los *Getters* y *Setters* como métodos.

```

class Escritor
{
    private readonly string nombre;
    private readonly DateTime nacimiento;
    private int publicaciones;

    public string GetNombre()
    {
        return nombre;
    }
    public DateTime GetNacimiento()
    {
        return nacimiento;
    }
    public int GetEdad()
    {
        return DateTime.Now.Year - nacimiento.Year;
    }
    public int GetPublicaciones()
    {
        return publicaciones;
    }
    private void SetPublicaciones(in int publicaciones)
    {
        this.publicaciones = publicaciones;
    }

    public Escritor(string nombre, in DateTime nacimiento)
    {
        this.nombre = nombre;
        this.nacimiento = nacimiento;
        SetPublicaciones(0);
    }

    public override string ToString()
    {
        return $"Nombre: {GetNombre()}\n" +
            $"Nacimiento: {GetNacimiento().ToShortDateString()}\n" +
            $"Edad: {GetEdad()}\n" +
            $"Publicaciones: {GetPublicaciones()}";
    }
}

```

La sintaxis equivalente **comentada** usando propiedades sería la siguiente...

```

class Escritor
{
    private readonly string nombre;
    private readonly DateTime nacimiento;
    private int publicaciones;

    // Fíjate que de la línea 9 a la 19 hemos implementado las propiedades
    // solo con el get porque los campos asociados son readonly.
    public string Nombre // El id de la propiedad debe ser el del campo pero en
    { // PascalCasing. Siendo el tipo el mismo que el campo.
        get
        {
            return nombre; // Como es un getter hacemos un return.
        }
    }
    public DateTime Nacimiento // Estamos haciendo public lo definido en la
    { // propiedad. En este caso únicamente el getter
        get
        {
            return nacimiento;
        }
    }
    public int Edad // Propiedad calculada a partir de otras.
    {
        get // Fíjate que es este getter estamos ya accediendo
        { // a la propiedad del campo nacimiento (su get).
            return DateTime.Now.Year - Nacimiento.Year;
        }
    }

    // Dentro de la propiedad Publicaciones definiremos tanto el get como el set.
    public int Publicaciones // Son public get y set en principio,
    { // aunque luego concretaremos el set.
        get
        {
            return publicaciones;
        }
        private set // Concretamos la accesibilidad general de la
        { // propiedad para el set, pero siempre debe ser más
            // restrictiva que la general para la propiedad.
            publicaciones = value;
        }
    }
}

```

45

46

}

...

Del código anterior podemos resumir que:

- Podremos definir solo uno de los dos, `set` o `get` y podrán estar afectados por los modificadores de accesibilidad como el resto de métodos.
- Puedo especificar un modificador de accesibilidad más restrictivo que el de la propiedad a una de las definiciones de `get` y `set`.

## Usando las propiedades definidas

Para nosotros sintácticamente será como si estuviéramos accediendo directamente al campo, pero con el nombre en PascalCasing. Sin embargo, se estará ejecutando el código definido en el cuerpo de la propiedad, como sucedía al definir los métodos.

Por ejemplo, si completamos el código de la clase escritor usando la propiedades definidas. Tendremos el siguiente código...

```
1  class Escritor
2  {
3      // ... código omitido por abreviar
4
5      public Escritor(string nombre, in DateTime nacimiento)
6      {
7          this.nombre = nombre;
8          this.nacimiento = nacimiento;
9          Publicaciones = 0;           // Se estará ejecutando el el set de
10                                     // la propiedad Publicaciones
11     }
12     public override string ToString()
13     {
14         // Estaremos accediendo al get de cada una de las propiedades definidas.
15         return $"Nombre: {Nombre}\n" +
16             $"Nacimiento: {Nacimiento.ToShortDateString()}\n" +
17             $"Edad: {Edad}\n" +
18             $"Publicaciones: {Publicaciones}";
19     }
20 }
```

# Evolución del uso de propiedades a lo largo del tiempo

## Propiedades 'autoimplementadas'

En **C# 3.0** y versiones posteriores, aparecen las **propiedades autoimplementadas** que hacen que la declaración de propiedad sea más concisa **cuando no se requiere ninguna lógica adicional** en los descriptores de acceso de la propiedad o estemos creando **clases 'ligeras'**.

Al declarar una propiedad, el compilador **crea un campo de respaldo privado y anónimo** al que solamente puede obtenerse acceso a través de los descriptores de acceso get y set de la propiedad.

Veamos a través de un ejemplo de definición de una clase **Persona** simple, cómo este '*syntactic sugar*' hace que la definición de nuestra clase sea más concisa. Supongamos pues el siguiente código ...

```
1  class Persona
2  {
3      private readonly string dni;
4      private string nombre;
5
6      public string Dni
7      {
8          get
9          {
10             return dni;
11          }
12     }
13
14     public string Nombre
15     {
16         get
17         {
18             return nombre;
19         }
20         private set
21         {
22             nombre = value;
23         }
24     }
25 }
```

el código anterior con las propiedades **autoimplementadas** pasaría **de 25 a 5 líneas** y tendríamos la **misma funcionalidad...**

```
1 | class Persona // Estará creando un campo de llamado supongamos _dni
2 | { // que es privado y readonly como en el caso de arriba
3 |     public string Dni { get; } // y del que solo vemos su propiedad Dni.
4 |     public string Nombre { get; private set; }
5 | }
```

Vamos ahora la implementación de nuestra clase escritor usando estas propiedades autoimplementadas.


```
1 | class Escritor
2 | {
3 |     public string Nombre { get; }
4 |     public DateTime Nacimiento { get; }
5 |     public int Publicaciones { get; private set; }
6 |
7 |     public int Edad // La propiedad calculada no se puede autoimplementar.
8 |     {
9 |         get
10 |         {
11 |             return DateTime.Now.Year - Nacimiento.Year;
12 |         }
13 |     }
14 |
15 |     public Escritor(string nombre, in DateTime nacimiento)
16 |     {
17 |         Nombre = nombre; // Las propiedades autoimplementadas con solo get
18 |         Nacimiento = nacimiento; // en el constructor será el único sitio donde se
19 |         Publicaciones = 0; // puedan asignar.
20 |     }
21 |
22 |     public override string ToString()
23 |     {
24 |         return $"Nombre: {Nombre}\n" +
25 |             $"Nacimiento: {Nacimiento.ToShortDateString()}\n" +
26 |             $"Edad: {Edad}\n" +
27 |             $"Publicaciones: {Publicaciones}";
28 |     }
29 | }
```



Esta sintaxis será la más concisa, **si no vamos a realizar comprobaciones o transformaciones** en los setter y getters.

## Miembros con cuerpo de expresión

En el caso de que no usemos propiedades autoimplementadas. Otra forma de realizar un '*syntactic sugar*' al definir las propiedades son los **miembros con cuerpo de expresión** se introdujeron en **C# 6 y 7**. No vamos a profundizar mucho en ellos (usos y significados) hasta el final del curso. Pero son una característica que también podemos encontrar en otros lenguajes como JavaScript, Kotlin, etc.

 **Definición:** Podemos simplificar diciendo que son aplicables en aquellos métodos que estén formados por una única expresión o instrucción en forma de expresión independientemente de si se evalúan algo o a void.

Veamos un ejemplo con dos métodos donde se cumple la definición anterior.


```
public void SetNombre(string nombre)
{
    this.nombre = nombre;
}
public override string ToString()
{
    return $"Dni: {Dni}" + $"Nombre: {Nombre}";
}
```

se pueden representar de una forma más abreviada como la siguiente...

```
public void SetNombre(string nombre) => Nombre = nombre;

public override string ToString() => $"Dni: {Dni}" + $"Nombre: {Nombre}";
```

Si nos fijamos, eliminaremos el cuerpo del método y dejaremos la expresión precedida del operador `=>` y quitando `return` si hubiera.

 **Nota:** A partir de este momento lo iremos usando en los ejemplos para ir quedándonos con este nuevo '*azúcar sintáctico*'.

Veamos cómo sería la sintaxis **si el miembro es una propiedad**. Para ello, vamos a partir de una clase que tiene dos campos y sus respectivas propiedades de acceso que no usan '*cuerpo de expresión*'.

```
class MiClase
{
    private readonly int campo1;
    private string campo2;

    public int Campo1
    {
        get
        {
            return campo1;
        }
    }
    public string Campo2
    {
        get
        {
            return campo2;
        }
        private set
        {
            campo2 = value;
        }
    }
}
```

Si usamos **cuerpos de expresión** en el código anterior quedará...

```
1  class MiClase
2  {
3      private readonly int campo1;
4      private string campo2;
5
6      public int Campo1 => campo1;
7      public string Campo2
8      {
9          get => campo2;
10         private set => campo2 = value;
11     }
12 }
```

👉 **Importante:** Fíjate que la sintaxis es más simple aún si usamos cuerpos de expresión para la propiedad de un campo de **solo lectura** ( **línea 6** ) que si lo usamos para implementar la propiedad de un campo con su getter y setter ( **líneas 7 a 11** ).

Si refactorizamos la última versión de **Escritor** para usar un cuerpo de expresión en la propiedad **Edad** . Podremos refactorizarla así por **tratarse únicamente de una expresión**.

```
class Escritor
{
    public string Nombre { get; }
    public DateTime Nacimiento { get; }
    public int Publicaciones { get; private set; }

    public int Edad => DateTime.Now.Year - Nacimiento.Year;

    // ... código omitido por abreviar.
}
```

### Resumen de casos de uso de propiedades:

1. **Autoimplementadas:** No vamos a necesitar conocer el campo ni vamos a añadir ningún control en los mutadores o accesoros.
2. **Cuerpos de expresión:**  
Es una propiedad calculada que se puede codificar en una única expresión.  
Necesitamos un identificador accesible para el campo.  
Vamos a realizar una transformación o comprobación del campo que se puede codificar en una única expresión.
3. **'Tradicional':** Además de necesitar acceder al identificador del campo, el cuerpo del getter o setter está formado por más de una instrucción.

”

*Most papers in computer science  
describe how their author learned  
what someone else already knew.*

”

- Peter Landin. (He coined 'Syntactic Sugar' term.)