

Tema 7 parte 1

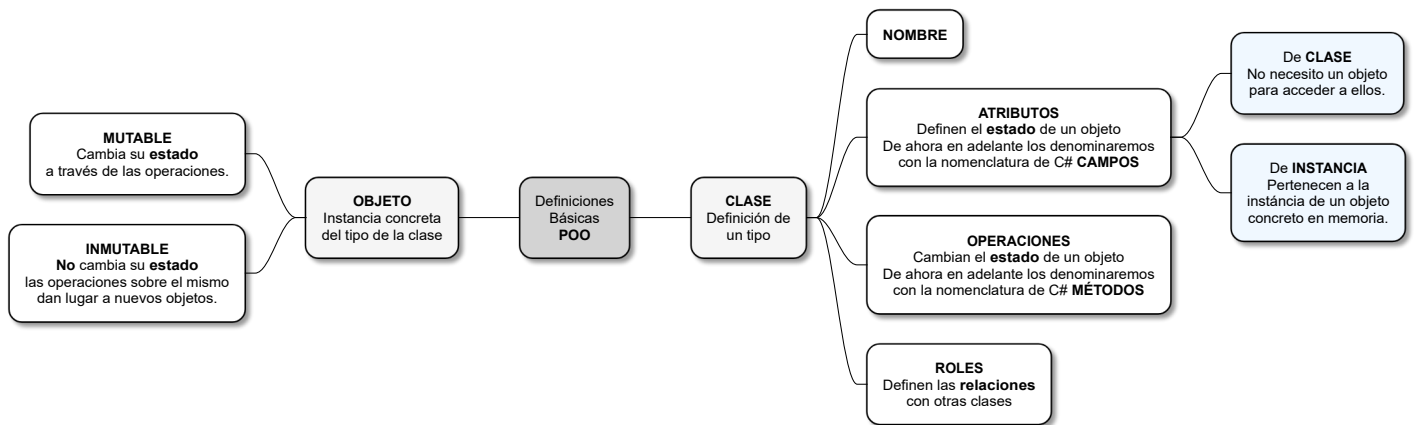
[Descargar estos apuntes](#)

Índice

1. Índice
2. Repaso Tema 5
3. Diseño Orientada a Objetos Básico
 1. Definición de operación o método
 1. Tipos de métodos
 2. Enfoque de los métodos desde la teoría de POO
 2. Definición de Encapsulación
 3. Definición de Constructor Y Destructor
4. Definiendo nuestra primera clase a través de 'C#'
 1. Paso 1: Definir los campos (atributos en POO clásica)
 2. Paso 2: Definir los constructores / destructores de objetos
 1. Referencia implícita this
 2. Constructor por defecto
 3. Constructor copia
 4. Destructor/es
 3. Paso 3: Definir los accesores y mutadores (Propiedades en C#)
 1. 🗨️ Tips de diseño de 'getters' y 'setters'
 4. Paso 4: Definir métodos u operaciones de la clase
 5. Paso 5: Crear un pequeño test para nuestra clase (Opcional)
 6. Resumen de directrices generales de definición de clases
5. Definiendo tipos y objetos 'valor'
 1. Un poco de historia
 2. 🗨️ Tips para definir un tipo usando struct
 3. Definir objetos valor a través de 'C#'
 1. Creando objetos valor
 2. Definiendo operaciones inmutables en el objeto valor
 4. Ejemplo de objeto valor definido en las BCL
 1. Formas de instanciar objetos valor fecha
 2. Operaciones más comunes con fechas
 3. Fechas y cadenas
6. Roles entre clases
 1. Relaciones Todo-Parte
 1. Agregación o referencia
 1. Ventajas de la agregación 👍
 2. Desventajas de la agregación 🗨️
 2. Composición o subobjetos
 1. Ventajas de la composición 👍
 2. Desventajas de la agregación 🗨️
 3. Ejemplo de Agregación
 4. Ejemplo de Composición

Repaso Tema 5

Repasemos a través del siguiente diagrama las definiciones y conceptos iniciales de POO que vimos en el tema 5.



Diseño Orientada a Objetos Básico

Aunque en el Tema 5 hablamos de las definiciones básicas, vamos a profundizar un poco más en algunos conceptos de la programación orientado a objetos para poder hacer nuestros **diseños o modelos** de forma correcta.

Definición de operación o método

- Definen el **Comportamiento** y las **Operaciones** que se pueden realizar con los objetos.
- Permiten interactuar y relacionarse a los objetos.

Tipos de métodos

- **Métodos de instancia o también (de objeto)**
 - Necesito tener un objeto instanciado en memoria para acceder a ellos.
 - Pueden acceder tanto a atributos de **instancia** como de **clase**.
 - Pueden modificar el **estado de un objeto** concreto en memoria si este es mutable.
- **Métodos de clase o también estáticos**
 - No necesito tener un objeto instanciado en memoria para acceder a ellos.
 - Solo pueden acceder a los atributos de clase (static) y no a los de instancia.
- **Métodos de acceso y actualización**
 - También se les conoce como **Accesores - Mutadores** en general, **Propiedades** (C# y Kotlin) o **Setters - Getters** (Java).

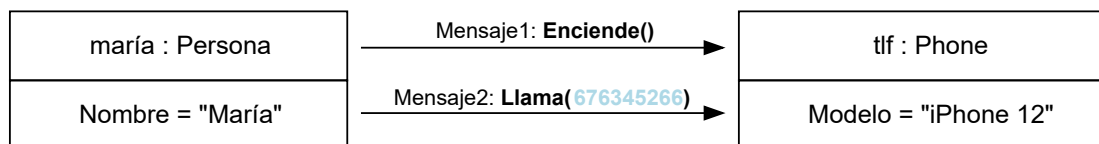
Enfoque de los métodos desde la teoría de POO

- En la teoría tradicional de la POO, los objetos se comunican entre ellos a través de un mecanismo de **paso de mensajes**. ([Alan Kay](#), [Bertrand Meyer](#))
- En el fondo cuando desde un método de un objeto de instancia llamamos o invocamos a un método de otro objetos, estaremos haciendo este paso de mensajes y por tanto comunicando ambos objetos.
- Hay más formas de pasar estos mensajes, pero la más básica es esta.
- Vamos a verlo a través de un ejemplo '**simplificado**' de código para entender el concepto '**abstracto**' de mensaje y comunicación entre objetos:

```
class Persona
{
    public void MétodoDeInstancia()
    {
        Phone tlf = new Phone("iPhone 12");

        // Mensaje 1
        // Un objeto concreto de Persona (en el ejemplo 'María')
        // Se comunica con e objetos tlf a través de un
        // mensaje, esto es llamando a su método Enciende()
        // Esto cambia el estado del objeto tlf a encendido.
        tlf.Enciende();

        // Mensaje 2
        // María pasa el mensaje a tlf de que llame a un número
        // cambiando su estado a llamando...
        tlf.Llama(676345266);
    }
}
```



Definición de Encapsulación

- En POO, se denomina encapsulación al la **ocultación del estado**, es decir, de los atributos, de un objeto. De tal manera que, solo se puede cambiar mediante las operaciones definidas para ese objeto o sus accesores - mutadores.
- De esta forma el usuario de la clase solo interacciona con los objetos abstrayéndose de como están implementados (**no sabe nada de la implementación**).
- Se evita que el usuario pueda cambiar su estado de maneras imprevistas e incontroladas.

Definición de Constructor Y Destructor

CONSTRUCTOR

- Método o métodos especiales que me servirán para instanciar e inicializar el estado de un objeto en memoria.
- Toda clase debe tener al menos un constructor.

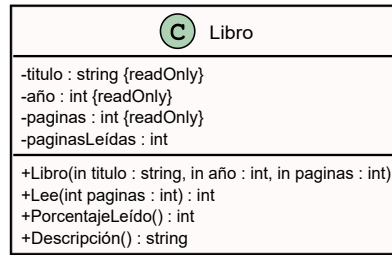
DESTRUCTOR

- Un único método especial encargado de eliminar una instancia en memoria de un objeto.
- En la gran mayoría de lenguajes OO modernos no hace falta definirlos y llamarlos, ya que de esta labor de eliminación de instancias de objetos en memoria, se encarga el denominado '**recolector de basura**' (GC), cuando un objeto ya no es referenciado por nadie.

Definiendo nuestra primera clase a través de 'C#'

Supongamos que queremos definir un tipo que represente libros.

Una posible representación UML del mismo podría ser:



En la mayoría de lenguajes OO, para definir nuestra clase, seguiremos una plantilla similar a esta:

```
class <NombreDeLaClase>
{
    <campos>
    <constructor/es>
    <accesores/mutadores>( o <propiedades en C#>)
    <métodos>
}
```

Paso 1: Definir los campos (atributos en POO clásica)

No deben ser accedidos desde fuera de clase. Para ello, antepondremos la cláusula `private` siempre.

Fíjate que hemos marcado dos campos con la [propiedad de atributo de clase](#) `{readOnly}` esto significa que, **una vez creado el objeto**, ya no se podrán modificar los valores de **título**, **año** y **paginas**.

```
class Libro
{
    // <campos>
    private readonly string titulo;
    private readonly int año;
    private readonly int paginas;
    private int paginasLeídas;
}
```

Paso 2: Definir los constructores / destructores de objetos

En C# el método constructor tiene el mismo nombre que la clase y no lleva tipo de retorno (Es implícito).

Tip: Si justo después del nombre de la clase pulsamos `Ctrl+.` en VSCode.

El VSCode **nos ofrecerá crear un constructor** en la refactorización de código.

```
class Libro
{
    // <campos>
    private readonly string titulo;
    private readonly int año;
    private readonly int paginas;
    private int paginasLeídas;

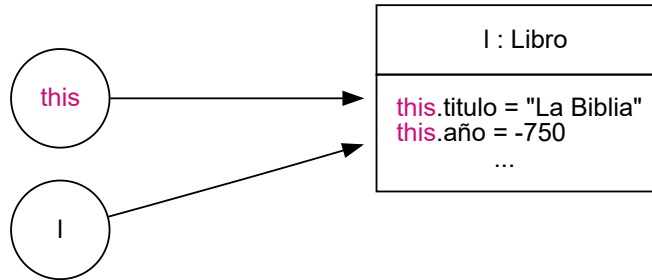
    // <constructor/es>
    public Libro(string titulo, in int año, in int paginas)
    {
        this.titulo = titulo;
        this.año = año;
        this.paginas = paginas;
        paginasLeídas = 0; // Al crear cualquier libro, llevaré leídas 0 páginas.
    }
}
```

👉 **Importante:** Fíjate que aunque hemos marcado `título` y `año` como `readonly` (solo lectura) los podemos asignar a un valor inicial en el constructor. Esta es la única vez que los podremos asignar. En el resto de métodos únicamente podremos acceder a su valor.

Referencia implícita `this`

- `this` es una **referencia implícita** a la instancia en memoria del objeto que en ese momento estamos creando o está accediendo a un método de la clase.
- En este caso nos ayuda a **diferenciar entre los identificadores** de los campos y los parámetros de entrada del constructor.
En el constructor el identificador `año` es del parámetro de entrada del mismo. Pero `this.año` es el campo `año` del objeto que estamos construyendo en ese momento.

```
Libro l = new Libro("La Biblia",-750, ...);
```



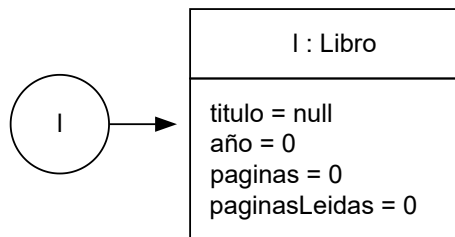
Constructor por defecto

- Si no definimos ningún constructor, **se define uno por defecto** que nos permitirá crear instancias de libro de la siguiente manera.

```
// El constructor por defecto, no recibe ningún parámetro que defina el estado inicial del objeto  
Libro l = new Libro();
```

- Sin embargo, si definimos un constructor concreto, **dejará de estar disponible el constructor por defecto**, a no ser que lo definamos explícitamente nosotros.

Sin embargo esto permitirá crear instancias del objeto libro a los valores default.



👉 Estaríamos dando la oportunidad de **crear objetos libro sin un estado apropiado**. Por tanto, **no es conveniente utilizar constructores por defecto**, a no ser que por alguna razón específica, lo definamos nosotros explícitamente y dispongamos de otras formas de crear nuestro objetos.

Constructor copia

- Se trata de una **aproximación** inicial al **clonado de objetos mutables**.
- Es un constructor **optativo**, que copiará el estado de una instancia de un objeto de la clase que lo define.
- No tiene sentido su implementación en objetos inmutables.
- En un principio vamos a implementarlo como un constructor más desde el punto de vista de POO tradicional o lenguajes como C++. Pero más adelante, veremos que para realizar copias en C# y en Java usaremos un método especial llamado **Clone()**
- Una posible implementación del mismo para nuestro ejemplo que podemos llevar a cualquier lenguaje OO puede ser....

```
// Entra un objeto libro referencia do por l que queremos copiar.
public Libro(Libro l)
{
    // No usamos this porque no hay posibilidad de confusión.
    titulo = l.titulo;
    año = l.año;
    paginas = l.paginas;
    paginasLeidas = l.paginasLeidas;
}
```

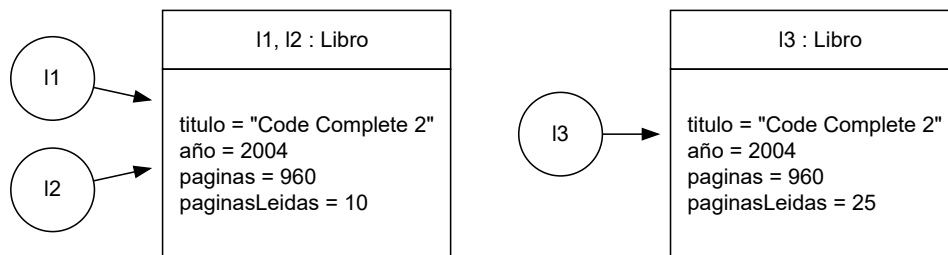
- Recordemos del tema 5 que si hiciésemos el siguiente código...

```
Libro l1 = new Libro("Code Complete 2", 2004, 960);

// l2 y l1 son el mismo objeto
l2 = l1;

// l3 y l1 son objetos diferentes.
l3 = new Libro(l1);

l2.Lee(10); // Modifica l2 y por ende l1 dejando leídas 10 páginas del libro.
l3.Lee(25); // Modifica l3 que es una copia en memoria de l1, no afectando por tanto al estado del objeto l1.
```



Destructor/es

- Cuando un objeto **deja de ser referenciado** por algún identificador, al cabo de tiempo es eliminado de la memoria por el **Recolector de Basura** (Garbage Collector o GC).
- Esta eliminación la hará llamando a su **destructor por defecto**, el cual nosotros podremos redefinir de tal manera que deje el estado del objeto a unos valores que no permitan usarlo más.

Nota: Si lo redefinimos, no le aplicaremos ningún modificador de acceso (public, private).

```
class Libro
{
    // <campos>
    // <constructor/es>

    // <destructor>
    ~Libro()
    {
        // Los otros valores son readonly y no podemos modificarlos.
        paginas = int.MinValue;
        paginasLeidas = int.MinValue;
    }
}
```

👉 **Importante:** Este paso de definición de nuestra clase, nos lo saltaremos en todos los **lenguajes gestionados**, esto es, que dispongan de un GC para la eliminación de objetos. Solo se redefinirá en ocasiones en lenguajes no gestionados como C++

Paso 3: Definir los accesores y mutadores (Propiedades en C#)

👉 **Nota:** Usaremos en principio la forma de hacerlo de **Java**. Aunque más adelante veremos que en C# existe un '**Syntactic Sugar**' para definirlos y usarlos.

- Para el **accesor**, crearemos un método con el prefijo **Get<idCampo>** seguido del nombre del campo en PascalCasing por ser C#.
- Para el **mutador**, crearemos un método con el prefijo **Set<idCampo>** seguido del nombre del campo en PascalCasing por ser C#.

💡 Tips de diseño de 'getters' y 'setters'

1. No tengo porqué definirlos **todos**, solo si los necesito o me los piden.
2. Si el campo es **readonly** solo podrá haber **accesor** (getter).
3. Los accesores (getters) llevarán el modificador **public** por defecto.
4. Los mutadores (setters) llevarán el modificador **private** por defecto para asegurar la encapsulación y solo serán **public** si fuese necesario, siempre y cuando nos aseguremos que el objeto queda en buen estado.
5. En lugar de usar directamente los campos dentro de los **métodos** y constructores, intentaremos usar los getters y setters definidos.

```
class Libro
{
    // <campos>

    // <constructor/es>
    public Libro(string titulo, in int año, in int paginas)
    {
        this.titulo = titulo;
        this.año = año;
        this.paginas = paginas;
        SetPaginasLeidas(0);
    }
    public Libro(Libro l)
    {
        titulo = l.GetTitulo();
        año = l.GetAño();
        paginas = l.GetPaginas();
        SetPaginasLeidas(l.GetPaginasLeidas());
    }

    // <accesores/mutadores>
    public string GetTitulo()
    {
        return titulo;
    }

    public int GetAño()
    {
        return año;
    }

    public int GetPaginas()
    {
        return paginas;
    }

    public int GetPaginasLeidas()
    {
        return paginasLeidas;
    }

    private void SetPaginasLeidas(int paginas)
    {
        paginasLeidas = paginas;
    }
}
```

Paso 4: Definir métodos u operaciones de la clase

- Desde cualquier método de instancia, podremos acceder a los campos y accesorios/mutadores de la clase. Ya sea a través de `this` o `'inferido'` si no hay un parámetro formal con el mismo id.
 - Sin embargo, trataremos de usar los accesorios/mutadores en lugar de los campos.
- Para nuestro ejemplo tendremos...

```
class Libro
{
    // <campos>
    // <constructor/es>
    // <accesores/mutadores>

    // <métodos>
    public int Lee(in int paginas)
    {
        int leídas = Math.Clamp(paginas, 0, GetPaginas() - GetPaginasLeidas());
        SetPaginasLeidas(GetPaginasLeidas() + leídas);
        return leídas;
    }

    public int PorcentajeLeido()
    {
        return Convert.ToInt32(GetPaginasLeidas() * 1000 / GetPaginas());
    }

    public string Descripcion()
    {
        return $"Título: {GetTitulo()}\n" +
            $"Año: {GetAño()}\n" +
            $"Páginas: {GetPaginas()}";
    }
}
```

Paso 5: Crear un pequeño test para nuestra clase (Opcional)

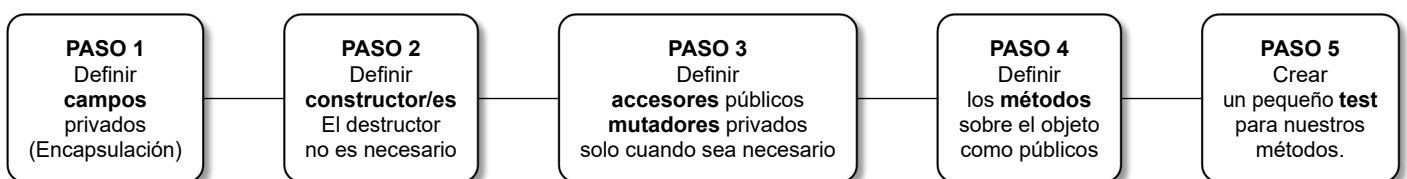
- **Nota:** Más adelante veremos cómo crear nuestros test de forma más apropiada.
- En este caso, vamos a hacer un pequeño programa que cree una instancia de un objeto libro y posteriormente lo lea de 300 en 300 páginas.

```
static class Programa
{
    static void Main()
    {
        // Creo el libro.
        Libro libro = new Libro("Code Complete 2", 2004, 960);

        // Muestro la descripción de mi libro
        Console.WriteLine(libro.Descripcion());

        // Mientras pueda leer alguna página (máximo 300)
        // muestro las páginas leídas y el porcentaje de lo que llevo leído del libro.
        const int MAXIMO_PAGINAS_A_LEER = 300;
        int leídas;
        while((leídas = libro.Lee(MAXIMO_PAGINAS_A_LEER)) > 0)
            Console.WriteLine($"Leídas: {leídas} {libro.PorcentajeLeido()}%");
    }
}
```

Resumen de directrices generales de definición de clases




Caso de estudio

Junto a la clase Libro anterior, vamos a modelar la clase **Escritor** suponiendo que un escritor tiene un **nombre**, un año de **nacimiento** y un número de **publicaciones**.


Además del constructor y los accesores y mutadores para sus campos, vamos a definir el método descripción que me muestre sus datos y método **Libro Escribe(string titulo)** donde el escritor creará un libro con el título recibido de entre 400 y 800 páginas.

 **Intenta modelarla e implementarla tu mismo, antes de ver la propuesta de solución.**

Un posible modelado de la clase, podría ser el siguiente...

 Escritor
-nombre : string {readOnly} -nacimiento : int {readOnly} -publicaciones : int
+Escritor(in nombre : string, in nacimiento : int) +Escritor(in e : Escritor) +GetNombre() : string +GetNacimiento() : int +GetPublicaciones() : int -SetPublicaciones(in publicaciones : int) : void +Escribe(in titulo : string) : Libro +Descripción() : string

La implementación propuesta siguiendo los pasos y criterios descritos sería...

 **Fíjate en la implementación y comentarios del método `Libro Escribe(string titulo)`**

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

    public string GetNombre()
    {
        return nombre;
    }
    public int GetNacimiento()
    {
        return nacimiento;
    }
    public int GetPublicaciones()
    {
        return publicaciones;
    }
    private void SetPublicaciones(in int publicaciones)
    {
        this.publicaciones = publicaciones;
    }
    public Escritor(string nombre, in int nacimiento)
    {
        this.nombre = nombre;
        this.nacimiento = nacimiento;
        SetPublicaciones(0);
    }
    // Hemos definido un constructor copia para escritores, aunque aquí tendría menos sentido hacerlo.
    public Escritor(Escritor e)
    {
        nombre = e.nombre;
        nacimiento = e.nacimiento;
        SetPublicaciones(e.publicaciones);
    }
    public string Descripción()
    {
        return $"Nombre: {GetNombre()}\n" +
            $"Nacimiento: {GetNacimiento()}\n" +
            $"Publicaciones: {GetPublicaciones()}";
    }
}
```

```
// A este tipo de objetos que sin ser constructores, devuelven una instancia de un objeto
// se les denomina 'Métodos factoria'.
// En cierto modo tienes sentido, ya que un escritor produce libros.
public Libro Escribe(string titulo)
{
    // Para establecer el rango de páginas de sus libros, hemos usado un nuevo tipo
    // añadido en C# denominado rango (por ver posibilidades del lenguaje).
    // Aunque podríamos haber definido simplemente 2 enteros.
    // Este rango podría ser incluso un campo que defina una característica de nuestros
    // objetos escritor.
    Range r = 400..800;
    SetPublicaciones(GetPublicaciones() + 1); // Incremento el número de publicaciones del escritor.
    // Creo un libro, con el año actual y un número de páginas aleatoria en el rango.
    return new Libro(titulo, DateTime.Now.Year, new Random().Next(r.Start.Value, r.End.Value + 1));
}
}
```

Definiendo tipos y objetos 'valor'

Ya vimos en el **Tema 5** que todos los objetos, ya sean creados a través de nuestras propias clases o a través de las definidas en la BCL son **tipos referencia**. Pero, ¿Hay alguna forma de definir **tipos valor**? La respuesta es **SÍ**. Utilizando la palabra reservada **struct**.

Nota: La palabra reservada **struct** viene de los antecesores de C# que son C y C++, pero en ellos su [interpretación](#) está más orientada a la definición de tipos compuestos heterogéneos.

Un poco de historia

A principios de la década del 2000 cuando [Anders Hejlsberg](#) definió el lenguaje, siguiendo la estela de Java. En ese momento, todos los tipos debían ser tratados como objetos en memoria y por tanto los tipos básicos como por ejemplo **int** también eran objetos. Pero, si todas las clases definen tipos referencia. ¿Cómo hacer que estos tipos básicos fueran objetos valor?.

Por ese motivo se incluyó la palabra reservada **struct** que permitía definir tipos como **class** pero con ciertas restricciones. Además, estos tipos serían **valor**. Por ejemplo, si nos fijamos **int** es una alias para el tipo **System.Int32** y si nos fijamos en su [definición](#), este es un **struct**.

En diseño orientado a objetos a través de diagramas UML **también se les conoce como Data Type**.

En la POO moderna, la aproximación para el uso de estructuras que más consenso produce, es la definición de [Value Object](#) que [Martin Fowler](#) da sobre los mismos. Las características que deberían tener los tipos para definirlos a través **struct** según Martin la describiremos a continuación.

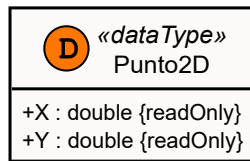
💡 Tips para definir un tipo usando struct

- Deberían ser **inmutables**, esto es, cualquier operación sobre un objeto struct debería producir un nuevo objeto struct.
- Sus campos, a su vez, serán tipos inmutables.
- No deberían tener ningún campo marcado como **{id}** que haga a los objetos únicos.
- **Los objetos representan entidades completamente intercambiables**. Por ejemplo...
 - Un objeto **billete de 5€** es intercambiable por otro **billete de 5€**. Sigo teniendo la misma cosa. El mismo 'valor'.
 - Un objeto naipes con el **7 de picas** es intercambiable por otro **7 de picas**.
 - Un objeto IP con el valor **192.168.0.1** es intercambiable por otro **192.168.0.1**.
 - Una objeto coordenada de la consola con los valores **X (columna) = 40** y **Y (fila) = 10** es intercambiable con otro objetos con los mismos valores.

Definir objetos valor a través de 'C#'

Vamos a definir un tipo valor denominado `Punto2D` que contendrá 2 campos que serán las coordenadas `x` e `y` respectivas al punto.

Si la definiéramos en UML usaríamos el estereotipo `<<dataType>>` de una forma similar a esta...



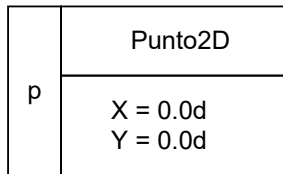
```
struct Punto2D
{
    public readonly double X;
    public readonly double Y;
}
```

En lugar de usar el palabra reservada `class`, hemos usado `struct`. Además, como queremos que sea **immutable**, los campos los hemos definido `readonly` y como una vez definidos ya no podremos modificarlos, los hemos definido cómo públicos para que desde fuera se pueda consultar su valor.

Creando objetos valor

La forma más simple, es hacerlo de forma análoga a cómo declarábamos un entero (recordemos que en el fondo un `int` es un `struct`).

```
Punto2D p;
```



Nada más declarar `p`, como sucede cuando declaramos un entero, ya tendremos una instancia valor de nuestro punto con los valores por defecto para `double` que es `0.0d`.

Pero al hacer el objeto inmutable, ya no podremos modificar su estado. Por esta razón, lo más apropiado es **definir un constructor** que me permita crear objetos punto.

Nota: Añadimos además el método `string ATexto()` para poder representar los puntos definidos.

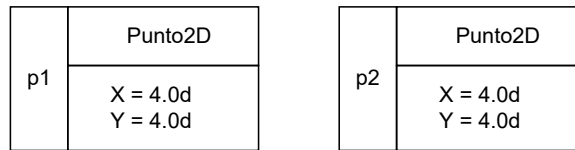
```
struct Punto2D
{
    public readonly double X;
    public readonly double Y;

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

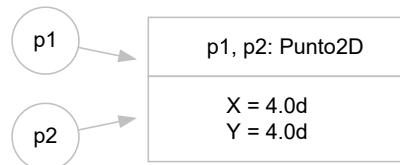
    public string ATexto()
    {
        return $"({X:G2} - {Y:G2})";
    }
}
```

Ahora si hacemos una asignación de un punto sobre otro se generará una copia en memoria, el objeto **Punto2D** ...

```
Punto2D p1 = new Punto2D(4d, 4d);
Punto2D p2 = p1;
Console.WriteLine("p1 = " + p1.ATexto());
Console.WriteLine("p2 = " + p2.ATexto());
```



Si **Punto2D** lo hubiésemos definido con **class** tendríamos un **tipo referencia** y lo siguiente en memoria...

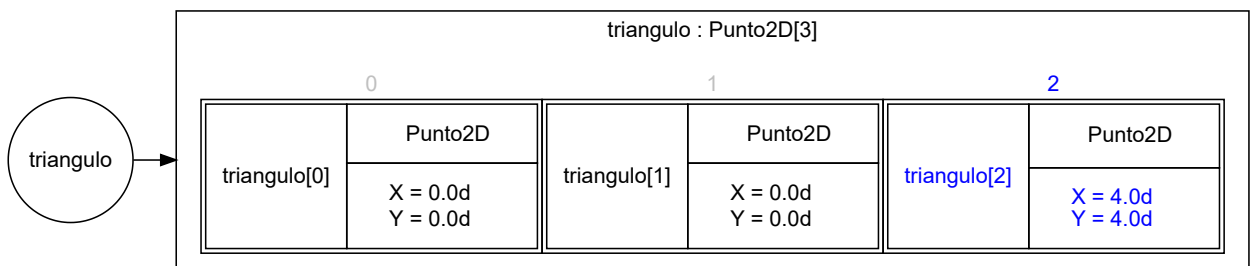


Si planteamos lo mismo pero en términos de colecciones homogéneas. Si hacemos ...

```
Punto2D triangulo[] = new Punto2D[3];
// Aquí los puntos ya estarían creados por ser Punto2D un tipo valor.

triangulo[2] = new Punto2D(4D, 4D);
Console.WriteLine("triangulo[2] = " + triangulo[2].ATexto());
```

Se creará un array con los tres objetos valor inicializados por '*defecto*' y posteriormente hacemos una **copia** del un nuevo objeto valor con coordenadas (4 - 4) en la posición de índice 2 en array.



Recordemos del **tema 5**, que si **Punto2D** fuese un tipo referencia, después de ejecutar **Punto2D triangulo[] = new Punto2D[3];** tendríamos un array con tres referencias a objetos **Punto2D** a **null**.

Definiendo operaciones inmutables en el objeto valor

Supongamos que queremos definir una operación que desplace un punto una distancia y un ángulo en el plano.

Puesto que el método no puede cambiar el estado del punto, devolverá un nuevo punto donde se habrá aplicado el desplazamiento.

```
struct Punto2D
{
    // Código omitido para abreviar...
    public Punto2D Desplaza(in double distancia, in double anguloGrados)
    {
        double anguloRadianes = anguloGrados * Math.PI / 180d;
        double fila = Y + distancia * Math.Sin(angulo.Radianes);
        double columna = X + distancia * Math.Cos(angulo.Radianes);


        // Creamos la estructura y devolveremos una COPIA.
        return new Punto2D(fila, columna);
    }
}
```

Caso de estudio

Supongamos que queremos definir un tipo valor para los ángulos y así no tener que especificar las unidades.

- Internamente guardará el valor del ángulo en grados (entero) y en radianes (double).
- Definiré un constructor que me los cree a partir de un valor en grados.
- Un método **Suma** que a partir de un valor en grados lo suma y me devuelva un nuevo valor de ángulo.
- Finalmente usa el tipo en **Punto2D** y has un pequeño programa de prueba.

Nota: Intenta realizar la implementación antes de ver la siguiente propuesta de solución...

	«DataType» Angulo
+Grados : int {readOnly}	
+Radianes : double {readOnly}	
+Angulo(in grados : int)	
+Suma(in grados : int) : Angulo	

```
namespace ObjetosValor {
    struct Angulo {
        public readonly int Grados;
        public readonly double Radianes;

        public Angulo(in int grados)
        {
            const int GRADOS_TOTALES = 360;
            int gr = grados % GRADOS_TOTALES;
            gr = gr < 0 ? gr + GRADOS_TOTALES : gr;
            Grados = gr;
            Radianes = Grados * Math.PI / 180d;
        }
        public Angulo Suma(in int grados)
        {
            return new Angulo(Grados + grados);
        }
    }
    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, in Angulo angulo)
        {
            double fila = Y + distancia * Math.Sin(angulo.Radianes);
            double columna = X + distancia * Math.Cos(angulo.Radianes);
            return new Punto2D(fila, columna);
        }
        public string ATexto()
        {
            return $"({X:G2} - {Y:G2})";
        }
    }
    public static class Principal {
        public static void Main() {
            Angulo a = new Angulo(0);
            Punto2D p1 = new Punto2D(4d, 4d);
            Punto2D p2 = p1;
            Punto2D p3 = p2.Desplaza(4d, a.Suma(45));
            Console.WriteLine($"a = {a.Grados} grados");
            Console.WriteLine("p1 = " + p1.ATexto());
            Console.WriteLine("p2 = " + p2.ATexto());
            Console.WriteLine("p3 = " + p3.ATexto());
        }
    }
}
```

Caso de estudio


Vamos a definir una clase para modelar un **ISBN**

- Un ISBN es un código normalizado internacional para libros (***International Standard Book Number***).
- Los ISBN tuvieron 10 dígitos hasta diciembre de 2006 pero, desde enero de 2007, **tienen siempre 13 dígitos** que se corresponden con los números del código de barras **EAN13** (*código de barras de 13 dígitos*).
- Se calculan utilizando una fórmula matemática específica e incluyen un dígito de control que valida el código.
- Cada ISBN **se compone de 5 elementos separados entre sí** por un espacio o un guion. Tres de los cinco elementos pueden variar en longitud:
 1. **Prefijo**: Actualmente sólo pueden ser **978** o **979**. Siempre tiene 3 dígitos de longitud.
 2. **Grupo de registro**: Identifica a un determinado país, una región geográfica o un área lingüística que participan en el sistema ISBN. Este elemento **puede tener entre 1 y 5 dígitos** de longitud.
 3. **Titular**: Identifica a un determinado editor o a un sello editorial. Puede tener **hasta 7 dígitos** de longitud.
 4. **Publicación**: Identifica una determinada edición y formato de un determinado título. Puede ser de **hasta 6 dígitos** de longitud. Además, este elemento se rellenará con ceros por la izquierda si el ISBN no alcanzase los 13 dígitos.
 5. **Dígito de control**: Es siempre el último y único dígito que valida matemáticamente al resto del número. Se calcula utilizando **el sistema de módulo 10 con pesos alternativos de 1 y 3**.

Ejemplo: Para el ISBN **978-1-78528-144** el DC se calculará.

```
suma = 9 + 7*3 + 8 + 1*3 + 7 + 8*3 + 5 + 2*3 + 8 + 1*3 + 4 + 4*3 =  
      = 9 + 21 + 8 + 3 + 7 + 24 + 5 + 6 + 8 + 3 + 4 + 12 = 110  
resto = suma % 10 = 110 % 10 = 0  
dc = resto == 0 : 0 ? 10 - resto;
```

- A la hora de modelar una clase para el ISBN, podríamos definirla como un tipo referencia mediante class (siempre es buena opción) o como un tipo valor mediante un struct pues cumple con los requisitos para ello, sobre todo, que una vez establecido no va a cambiar de estado. En nuestro caso, los campos que definirán los elementos del ISBN los vamos a definir como cadenas y va a ser un tipo referencia.
- **Nota:** Vamos a obviar definir los accesores o getters para simplificar.
- Una aproximación preliminar al diseño de nuestra clase podría ser la siguiente:

 Isbn13
-prefijo : string {readOnly} -grupoDeRegistro : string {readOnly} -titular : string {readOnly} -publicacion : string {readOnly}
+Isbn13(in prefijo : int, in grupoDeRegistro : int, in titular : int, on publicacion : int) +Isbn13(in isbn13 : string) +Isbn13(in isbn : Isbn13) +DigitoDeControl() : int +ATexto(in separador : string) : string

Definiremos **tres** constructores:

1. Uno que reciba los cuatro elementos que definen el ISBN en forma de enteros sin el DC.

```
Isbn13 isbn = new Isbn13(978, 1, 78528, 14);
```

2. Otro que reciba el ISBN como cadena.

```
Isbn13 isbn = new Isbn13("978-1-78528-144-0");  
Isbn13 isbn = new Isbn13("978 1 78528 144 0");  
Isbn13 isbn = new Isbn13("9781785281440");
```

3. Por último un constructor copia.

¿Serías capaz de implementar la clase propuesta?

Si no se te ocurre nada. Puedes consultar la **propuesta de solución comentada** en la siguiente página y después volver a intentarlo

```

class Isbn13
{
    // Definimos como constantes privadas y de clase todos aquellos valores que vamos a utilizar para
    // comprobaciones de los rangos de valores de los campos y así no utilizar 'Números Mágicos'
    private static readonly int[] PREFIJOS = new int[] { 978, 979 };
    private const int MAX_LONGITUD_GRUPO = 5;
    private const int MAX_LONGITUD_TITULAR = 7;
    private const int MAX_LONGITUD_PUBLICACION = 6;
    private const int LONGITUD_ISBN = 13;

    // Definición de los campos que guardan los elementos del ISBN.
    // Son de solo lectura y string, para preservar ceros por la izquierda y evitar desbordamientos de tipo.
    private readonly string prefijo;
    private readonly string grupoDeRegistro;
    private readonly string titular;
    private readonly string publicacion;

    public Isbn13(in int prefijo, in int grupoDeRegistro, in int titular, in int publicacion)
    {
        // Comprobamos que el prefijo esté entre los prefijos válidos,
        // en caso contrario generamos un error y finalizamos la creación del objeto.
        this.prefijo = prefijo.ToString();
        if (Array.IndexOf(PREFIJOS, prefijo) < 0)
            throw new ArgumentException($"El prefijo {prefijo} no es válido.");

        // Comprobamos que la longitud de los campos no supere la especificada,
        // en caso contrario generamos un error y finalizamos la creación del objeto.
        this.grupoDeRegistro = grupoDeRegistro.ToString();
        if (this.grupoDeRegistro.Length > MAX_LONGITUD_PUBLICACION)
            throw new ArgumentException($"El grupo de registro {grupoDeRegistro} es demasiado largo.");

        this.titular = titular.ToString();
        if (this.titular.Length > MAX_LONGITUD_PUBLICACION)
            throw new ArgumentException($"El titular {titular} es demasiado largo.");

        this.publicacion = publicacion.ToString();
        if (this.publicacion.Length > MAX_LONGITUD_PUBLICACION)
            throw new ArgumentException($"La publicacion {publicacion} es demasiado larga.");

        // Obtenemos la diferencia de longitud con un código de 13 dígitos sin contar el dígito de control.
        string isbn = string.Join("", prefijo, grupoDeRegistro, titular, publicacion);
        int excesoLongitud = isbn.Length - (LONGITUD_ISBN - 1);

        // Si no llega a 12 rellenamos con ceros por la izquierda.
        if (excesoLongitud < 0)
            this.publicacion = this.publicacion.PadLeft(Math.Abs(excesoLongitud) + this.publicacion.Length, '0');

        // Si es más largo de 12 dígitos generamos un error y finalizamos la creación del objeto.
        if (excesoLongitud > 0)
            throw new ArgumentException($"El ISBN {isbn} es demasiado largo.");
    }

    public Isbn13(string isbn13)
    {
        // Separadores posibles ' ', '-' o nada.
        string s = "[ -]?";
        // Defino el grupo "prefijo", con alternancia de los que haya en el array 978|979
        // fíjate que el código es robusto pues está preparado para permitir futuros prefijos.
        string erPrefijo = $"(<prefijo>{string.Join("|", PREFIJOS)})";
        // Defino el grupo "grupoDeRegistro", un valor entre 1 y MAX_LONGITUD_GRUPO dígitos.
        string erGrupoDeRegistro = @"(<grupoDeRegistro>\d{1," + MAX_LONGITUD_GRUPO + "})";
        // Defino el grupo "titular", un valor entre 1 y MAX_LONGITUD_TITULAR dígitos.
        string erTitular = @"(<titular>\d{1," + MAX_LONGITUD_TITULAR + "})";
        // Defino el grupo "publicacion", un valor entre 1 y MAX_LONGITUD_PUBLICACION dígitos.
        string erPublicacion = @"(<publicacion>\d{1," + MAX_LONGITUD_PUBLICACION + "})";
        // Defino el grupo "dc", que contendrá el dígito de control.
        string erDC = @"(<dc>\d)";

        // Intento ver si la entrada se corresponde con la expresión para comprobar ISBN
        // al dividir la expresión esta queda más legible y es por tanto hay menos posibilidad
        // de cometer errores o ampliarla.
        Match m = Regex.Match(isbn13, $"^({erPrefijo}{s}{erGrupoDeRegistro}{s}{erTitular}{s}{erPublicacion}{s}{erDC})$");

        // Si no hay correspondencia generamos un error y finalizamos la creación del objeto.
        if (!m.Success)
    }
}

```

```

        throw new ArgumentException($"{isbn13} no es un valor válido para un ISBN");

// Si hay correspondencia ya podremos extraer el valor de los grupos y rellenar los campos
// que definirán el objeto y el dc para comprobarlo.
prefijo = m.Groups["prefijo"].Value;
grupoDeRegistro = m.Groups["grupoDeRegistro"].Value;
titular = m.Groups["titular"].Value;
publicacion = m.Groups["publicacion"].Value;
int dc = int.Parse(m.Groups["dc"].Value);

// Si el ISBN cumple el formato pero más el dc es menor que 13
// generamos un error y finalizamos la creación del objeto.
// Nota: La llamada a this.ATexto("").Length no la podremos hacer si no hemos
// definido el valor de los campos.
if (ATexto("").Length != LONGITUD_ISBN)
    throw new ArgumentException($"El dígito de control para {isbn13} debería ser un EAN13");

// Calculamos cual debería ser el dc según el valor de los campos y lo comprobamos con el
// introducido. Si no coinciden, generamos un error y finalizamos la creación del objeto.
int dcCorrecto = DigitoDeControl();
if (dc != dcCorrecto)
    throw new ArgumentException($"El dígito de control para {isbn13} debería ser {dcCorrecto} en lugar de {dc}");
}

// Constructor copia
public Isbn13(Isbn13 isbn)
{
    prefijo = isbn.prefijo;
    grupoDeRegistro = isbn.grupoDeRegistro;
    titular = isbn.titular;
    publicacion = isbn.publicacion;
}

// Cálculo del DC según las especificaciones.
public int DigitoDeControl()
{
    string isbn = string.Join("", prefijo, grupoDeRegistro, titular, publicacion);
    double suma = 0;
    for (int i = 0; i < isbn.Length; i++)
        suma += ((i % 2 == 0) ? 1 : 3) * int.Parse(isbn[i].ToString());
    double resto = suma % 10;
    return resto == 0 ? 0 : Convert.ToInt32(10 - resto);
}

// Devuelvo el ISBN en formato de cadena, con el separador que me indiquen por parámetro.
public string ATexto(string separador)
{
    return string.Join(separador, prefijo, grupoDeRegistro, titular, publicacion, DigitoDeControl().ToString());
}
}

```


Ejemplo de objeto valor definido en las BCL

Uno de los objetos valor predefinidos en las BCL y que más vamos usar, es el tipo de dato **fecha**: [System.DateTime](#)

Formas de instanciar objetos valor fecha

```
DateTime f1 = new DateTime(2020, 3, 19);           // 19/03/2020 a las 00:00:00
DateTime f2 = new DateTime(2020, 3, 19, 8, 30, 00); // 19/03/2020 a las 8:30:00
DateTime f3 = DateTime.Today;                     // Hoy a las 00:00:00
DateTime f4 = DateTime.Now;                       // Hoy a a la hora actual en mi zona horaria.
DateTime f5 = DateTime.UtcNow;                    // Hoy a a la hora actual en zona UTC.
```

Operaciones más comunes con fechas

Podemos usar todos los operadores de comparación `==`, `!=`, `>`, `<`, `>=` y `<=`

En las operaciones de suma `+` y resta `-`, interviene otro tipo de dato valor denominado [TimeSpan](#) que representará un periodo de tiempo que podremos representar en las siguiente magnitudes `años`, `meses`, `días`, `horas`, `minutos`, `segundos` según la precisión que necesitemos.

```
DateTime f1 = new DateTime(2020, 3, 19);
TimeSpan periodo = new TimeSpan(25, 0, 0, 0); // Periodo de tiempo con precisión de 25 días

4 DateTime f2 = f1 + periodo;
  Console.WriteLine(f2.ToShortDateString()); // Muestra 13/04/2020

// Muestra Del 19/03/2020 al 13/04/2020 hay 25 días
8 Console.WriteLine($"Del {f1.ToShortDateString()} al {f2.ToShortDateString()} hay {(f2 - f1).Days} días");
```

Estas operaciones también las podremos hacer, a través de diferentes métodos `f.AddHours(...)`, `f.AddDays(...)`, etc. y así evitarnos usar `TimeSpan`

```
DateTime f1 = new DateTime(2020, 3, 19);
DateTime f2 = new DateTime(2020, 8, 1);

// Recuerda el método no cambia el estado de f1, sino que devuelve un objeto valor fecha que copiamos en f3
5 DateTime f3 = f1.AddDays(25);

Console.WriteLine(f1.ToShortDateString()); // Muestra 19/03/2020 (En España)
Console.WriteLine(f2.ToLongDateString()); // Muestra sábado, 1 de agosto de 2020 (En España)
Console.WriteLine(f3.ToLongDateString()); // Muestra lunes, 13 de abril de 2020 (En España)
```

Fechas y cadenas

Una aproximación sencilla es usar las propiedades del objeto para dar el formato:

```
DateTime f = new DateTime(2020, 3, 19);
Console.WriteLine($"{f.Day:D2}-{f.Month:D2}-{f.Year:D4}"); // Muestra 19-03-2020
```

Pero el lenguaje provee una sintaxis especial para cadenas de fecha y hora personalizadas de `DateTime` a `string` y viceversa. Usaremos los siguientes [formatos definidos por el lenguaje](#).

Aquí dispones un **cuadro resumen** de los principales formatos descritos en el enlace:

Formato	Descripción	Formato	Descripción
d	Día del mes con el mínimo de dígitos	/	Separador fecha según el país
dd	Día del mes con 2 dígitos	gg	Indicar la era según el país AC/DC
ddd	Día de la semana abreviado según el país	h	Hora de 0 a 12 horas
dddd	Día de la semana completo según el país	hh	Hora de 00 a 12 horas
M	Mes con el mínimo de dígitos	z	Indicador Zona Horaria
MM	Mes con 2 dígitos	H	Hora de 0 a 23 horas
MMM	Nombre del mes abreviado según el país	HH	Hora de 00 a 23 horas
MMMM	Nombre del mes completo según el país	mm	Minutos con 2 dígitos
yyyy	Año con 4 dígitos	ss	Segundos con 2 dígitos

👁 **Nota:** Aquellos caracteres que tienen un significado especial como **d**, **h**, **M**, etc. si queremos que se representen tal cual y no sean sustituidos por un valor de DateTime deberemos escaparlos con `\\`. Por ejemplo, `\\d` es el caracter **d** y no el día del mes en formato numérico.

```
DateTime f = new DateTime(2020, 3, 19, 19, 0, 0, DateTimeKind.Utc);
Console.WriteLine(f.ToString("dd-MM-yyyy")); // Muestra: 19-03-2020
Console.WriteLine(f.ToString(@"dddd d \de MMMM \de yyyy a la\s H \hora\s UTCz")); // Muestra: jueves 19 de marzo de 2020 a las 19 horas U
```

Tendremos también la conversión en sentido inverso si hacemos un Parse. Aunque en este último caso también podemos utilizar ER.

```
public static void Main()
{
    while (true)
    {
        Console.Write("Introduce una fecha con formato (dd/MM/yyyy): ");
        string texto = Console.ReadLine();
        if (DateTime.TryParseExact(texto, "dd/MM/yyyy", null,
            System.Globalization.DateTimeStyles.AllowWhiteSpaces |
            System.Globalization.DateTimeStyles.AdjustToUniversal,
            out DateTime fecha))
        {
            Console.WriteLine($"Has introducido {fecha.ToShortDateString()}");
        }
        else
        {
            Console.WriteLine($"{texto} es un fecha incorrecta.");
        }
    }
}
```

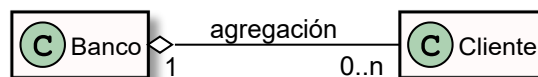
Roles entre clases

Relaciones Todo-Parte

- Suelen responder a la pregunta ¿**Tiene un ...?**
- Este tipo de relaciones también pueden tener **cardinalidad**.

Agregación o referencia

- Almacenaremos una **referencia** al objeto '*original*'.
 - **Es la relación todo-parte más común.**
 - La agregación indica independencia de los objetos, esto significa que si desaparece el contenedor, no desaparece el agregado.
 - En el siguiente ejemplo diremos que: '*1 clase **Banco** tendrá de 0 a N **Clientes***'.
- Pero al tratarse de una agregación, al desaparecer un objeto banco, no desaparecerán con él sus clientes.



Ventajas de la agregación 👍

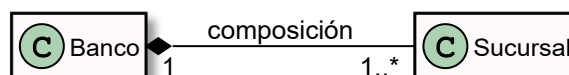
- Ahorramos memoria.
- Al compartirse la referencia al mismo objeto, mantendremos la integridad referencial.
- Mejor manejo de objetos complejos.
- Los objetos solo se crean cuando se necesitan.

Desventajas de la agregación 🗨

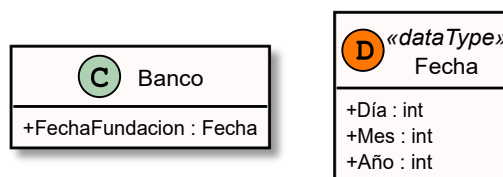
- Sobre todo en lenguajes no gestionados como C++. Se puede producir un efecto de **Aliasing**, si destruimos el objeto agregado.

Composición o subobjetos

- Almacenaremos un **objeto valor** o una referencia a una **copia** del objeto '*original*'.
 - La composición indica dependencia de los objetos, esto significa que si desaparece el contenedor, desaparece el subobjeto.
 - En el siguiente ejemplo diremos que: '*1 clase **Banco** tendrá de 1 a N **Sucursales***'.
- Pero al tratarse de una composición, al desaparecer un objeto banco, desaparecerán con él sus sucursales.



- Además, cualquier **campo** que defina nuestra clase que sea un tipo valor, **value object**, **data type**, etc. será considerado como una composición.
- La relación no hará falta expresarla, pues no puede ser de otra forma que no sea una composición por la naturaleza valor del objeto fecha. Simplemente indicaremos el tipo de datos en la definición.



Ventajas de la composición 👍

- Evitaremos efectos de **Aliasing**.

Desventajas de la composición 🗨

- Consume mucha memoria.
- No compartiremos. Por tanto, si necesitamos mantener la integridad referencial, deberemos hacerlo manualmente.

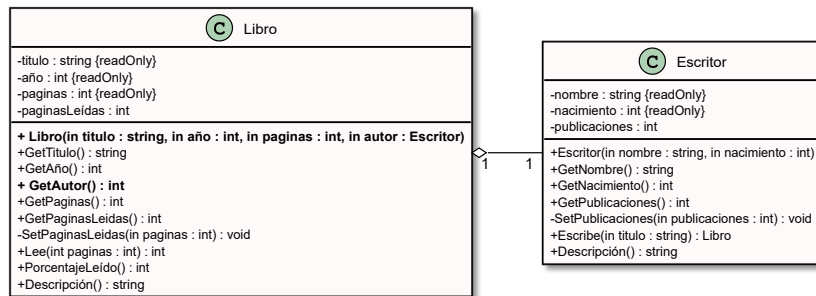
- Se pueden crear objetos que puede que no necesitemos.

Ejemplo de Agregación

Vamos a crear una relación entre nuestras clases de ejemplo **Escritor** y **Libro** de tal manera que el **Libro** va a tener información del **Escritor** a través de una agregación.

Cumple con la máxima de que si desaparece el libro no tiene porque desaparecer el escritor.

Una posible forma de expresarlo en UML es la siguiente.



De la relación **se debe deducir** que en la clase libro tendremos un campo nuevo que podemos denominar **autor** y que será de solo lectura.

Veamos los cambios que implicará esta relación en la implementación de nuestra clase **Libro**.

```
class Libro
{
    // ... código omitido por brevedad.
    // Añadiremos el campo que establece la relación de agregación indicada.
    private readonly Escritor autor;

    // Añadiremos un 'getter' para el campo
    public Escritor GetAutor()
    {
        return autor;
    }

    // Recibimos la referencia al escritor en el constructor de libro.
    public Libro(string titulo, in int año, in int paginas, Escritor autor)
    {
        // ... código omitido por brevedad.
        // Nos quedamos con la referencia y en ningún caso deberemos hacer un copia de escritor.
        this.autor = autor;
    }

    public Libro(Libro l)
    {
        // ... código omitido por brevedad.
        // Si tenemos constructor copia o método de clonado.
        // Copiaremos la referencia al escritor y en ningún caso haremos
        // copia en profundidad o llamaremos al constructor copia de escritor..
        autor = l.autor;
    }

    public string Descripción()
    {
        // Podemos añadir la información del escritor a la descripción del libro.
        return $"Libro\n" +
            "-----\n" +
            $"Título: {GetTitulo()}\n" +
            $"Año: {GetAño()}\n" +
            $"Páginas: {GetPaginas()}\n" +
            $"Autor -----\n" +
            $"{autor.Descripción()}\n";
    }
}
```

En cuanto a la clase **Escritor** la única modificación la deberíamos hacer en el método **Escribe()** que ahora debe pasar el escritor al constructor de **Libro**.

```
public Libro Escribe(string titulo)
{
    Range r = 400..800;
    SetPublicaciones(GetPublicaciones() + 1);
    // Fíjate que ahora pasamos this que es la referencia implícita al objeto
    // escritor que está escribiendo el libro.
    return new Libro(titulo, DateTime.Now.Year, new Random().Next(r.Start.Value, r.End.Value + 1), this);
}
```

Si implementamos y ejecutamos el siguiente código de test...

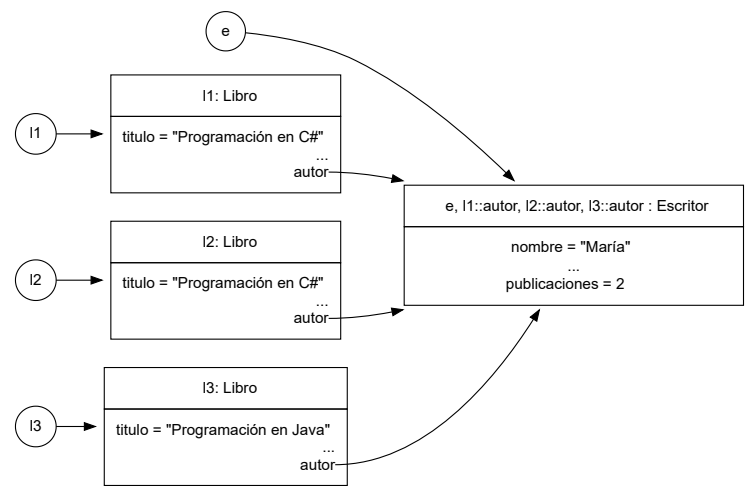
```
static void Main()
{
    Escritor e = new Escritor("María", 1980);
    Libro l1 = e.Escribe("Programación en C#");
    Libro l2 = new Libro(l1);
    // Console.WriteLine(l1.Descripcion());
    Console.WriteLine(l2.Descripcion());
    Libro l3 = e.Escribe("Programación en Java");
    Console.WriteLine(l3.Descripcion());
    // Console.WriteLine(l3.Descripcion());
}
```

Al ejecutarlo obtendremos una salida similar a esta...

```
Libro
-----
Título: Programación en C#
Año: 2020
Páginas: 517
Autor -----
Nombre: María
Nacimiento: 1980
Publicaciones: 1

Libro
-----
Título: Programación en C#
Año: 2020
Páginas: 517
Autor -----
Nombre: María
Nacimiento: 1980
Publicaciones: 2
```

Los tres objetos **Libro 11**, **12** y **13** referenciarán al mismo objeto **Escritor e**. Es más, si te fijas en la salida, el escritor al ver la primera descripción en **12** solo ha publicado un libro y después de escribir el segundo libro la descripción de **12** mantiene la integridad referencial indicándonos que ha publicado 2.

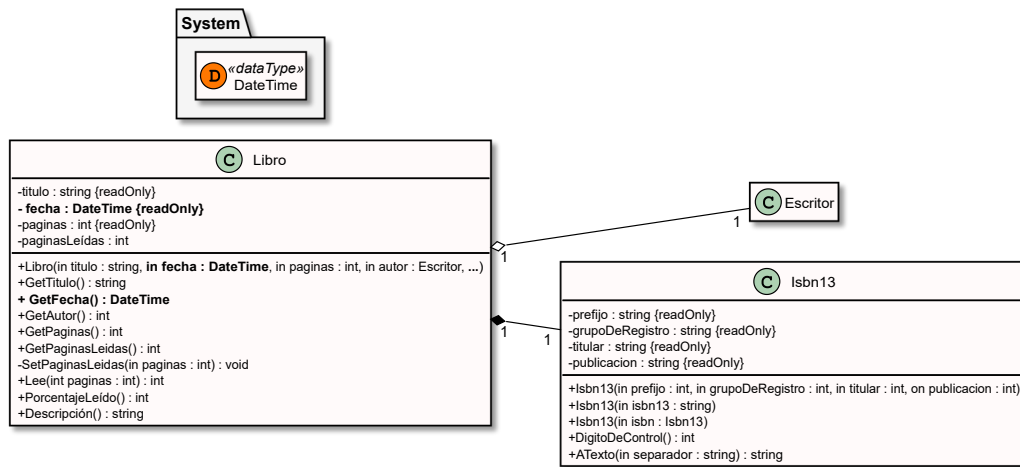


Ejemplo de Composición

Vamos ampliar nuestro ejemplo con la clase **Libro** y vamos a crear una relación entre **Libro** y la clase **Isbn3** y además la fecha de publicación va a ser un **DateTime**.

En ambos casos se cumple con la máxima de que si desaparece el libro, debería desaparecer su ISBN (tipo referencia) y su fecha de publicación (tipo valor).

Una posible forma de expresarlo en UML es la siguiente.



Fíjate que con **DateTime** no hace falta indicar la relación porque al ser un tipo valor, forzosamente ya es una composición. Además, de esta relación también **se debe deducir** que en la clase libro tendremos un campo nuevo que podemos denominar **isbn** y que será de solo lectura.

Veamos los cambios que implicará esta relación en la implementación de nuestra clase **Libro**.

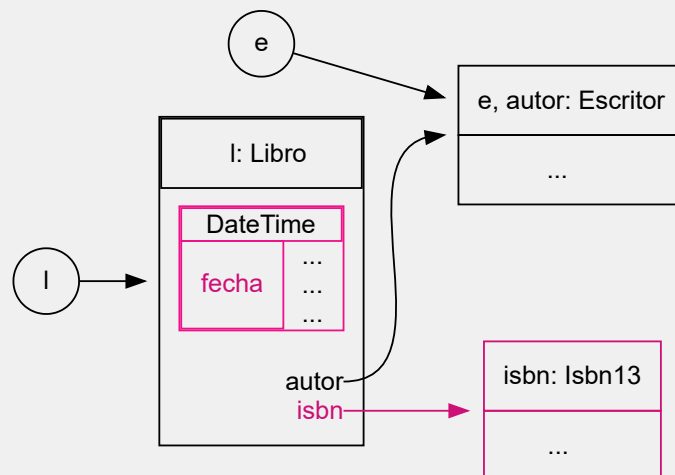
Lo mas lógico es que llegue el **ISBN** al constructor de **Libro**, puesto que no es **responsabilidad** del libro definirlo, sino más bien de un posible objeto de tipo **Editorial** de existir el mismo.

👉 **Importante:** Deberemos asegurarnos que la **referencia** al objeto **Isbn13** que tenga nuestro objeto **Libro**, solo la tenga él y no esté referenciado por nadie más, como puede suceder con el **Escriitor** por ser una agregación.

De esta manera al ser '*Destruido*' el libro, el el objeto **Isbn13** dejará de estar referenciado y también será destruido por el GC.

👁 Fíjate que por esa misma razón no definimos ningún **accesor publico** del tipo `GetISBN() : Isbn13`. Si lo hiciéramos, este debería devolver una **copia**.

¿Qué sentido tiene que haya un objeto isbn referenciado por alguien, si su libro asociado ya no existe?



Sabiendo que nuestro libro debe ser el único en referenciar el objeto ISBN. Una de las formas de abordarlo es la siguiente (Existen otras que veremos más adelante al ver POO Avanzada):

Pasaremos los datos necesarios para construir un objeto ISBN al constructor de Libro (en nuestro caso una cadena con el isbn) y será este quien lo cree. En el fondo esto es lo que sucede con un tipo valor como `DateTime` por su *'naturaleza'*.

De esta manera, además, el objeto `Isbn13` que estoy creando desaparecerá junto con el libro.

👁 **Nota:** Fíjate que con el `Escritor`, no sucede lo mismo.

```
class Libro
{
    private readonly DateTime fecha;
    private readonly Isbn13 isbn;
    // ... código omitido por brevedad
    public Libro(
        string titulo, in DateTime fecha, in int paginas, Escritor autor,
        string isbn)
    {
        this.titulo = titulo;
        this.fecha = fecha; // Aquí estamos haciendo una copia por ser un tipo valor.
        this.paginas = paginas;
        this.autor = autor;
        this.isbn = new Isbn13(isbn);
        SetPaginasLeidas(0);
    }
    public Libro(Libro l)
    {
        titulo = l.titulo;
        fecha = l.fecha; // Aquí estamos haciendo una copia por ser un tipo valor.
        paginas = l.paginas;
        autor = l.autor;
        // En el constructor copia haremos una copia en profundidad.
        // duplicando el objeto referenciado por isbn, para que cada libro tenga el suyo
        // y así desaparezca con él.
        isbn = new Isbn13(l.isbn);
        SetPaginasLeidas(l.paginasLeidas);
    }
    public string Descripcion()
    {
        return $"Libro\n" + "-----\n" +
            $"Título: {GetTitulo()}\n" +
            $"Fecha: {GetFecha().ToShortTimeString()}\n" +
            $"Páginas: {GetPaginas()}\n" +
            $"ISBN: {isbn.ATexto("-")}\n" +
            $"Autor -----\n" + $"{autor.Descripcion()}\n";
    }
}
class Escritor
{
    // ... código omitido por brevedad
    // Tampoco es responsabilidad del escritor crear el ISBN.
    public Libro Escribe(string titulo, string isbn)
    {
        Range r = 400..800;
        SetPublicaciones(GetPublicaciones() + 1);
        return new Libro(titulo, DateTime.Now, new Random().Next(r.Start.Value, r.End.Value + 1), this, isbn);
    }
}
static class Programa
{
    static void Main()
    {
        Escritor e = new Escritor("María", 1980);
        // Nos hemos asegurado que el libro l tenga su propia referencia al objeto isbn.
        // Lo que hagamos con el objeto isbn referenciado en el main, ya no importa porque no afecta a l
        Libro l = e.Escribe("Programación en C#", "9788420454665");
        Console.WriteLine(l.Descripcion());
    }
}
```