

Tema 7

[Descargar estos apuntes](#)

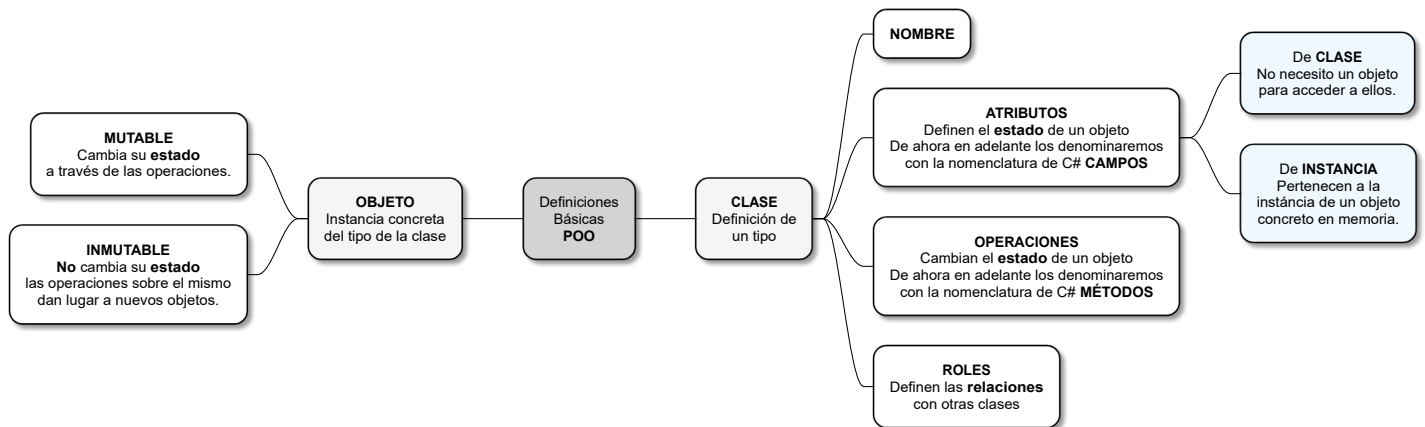
Índice

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
 2. Relación de Herencia (Generalización)
 1. Tipos de Herencia
 1. Herencia Simple
 2. Herencia Múltiple
 2. Implementando la herencia en C#
 1. Palabra reservada base
 3. Ocultación e Invalidación
 1. Ocultación o reemplazo en C#

- 2. Invalidación o refinamiento en C#
- 4. Polimorfismo de datos o inclusión
- 5. Principio de sustitución de Liskov (Upcasting)
- 6. Downcasting
 - 1. Formas de realizar el Downcasting
- 7. Ligadura Dinámica
 - 1. Ejemplo de uso del Enlace Dinámico
- 8. Utilidad del polimorfismo de datos (sustitución) y el enlace dinámico
- 9. El caso especial de la clase Object en C#
- 3. Abstracción
 - 1. Concepto de clase abstracta
 - 2. Clases abstractas en C#
- 7. Gestión de errores en POO
 - 1. Programación defensiva
 - 2. Excepciones
 - 1. Excepciones en C#
 - 1. Ejemplo de generación de excepciones en C#

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 orientada 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).

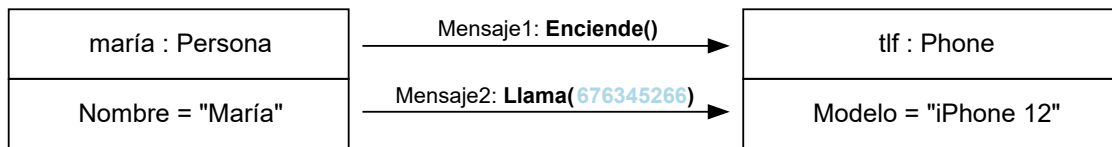
Enfoque de los métodos desde la teoría de POO

- 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 accesorios - 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.

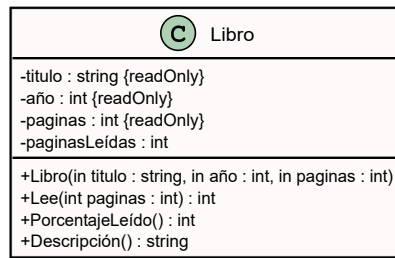
DESTRUCTOR

- 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;

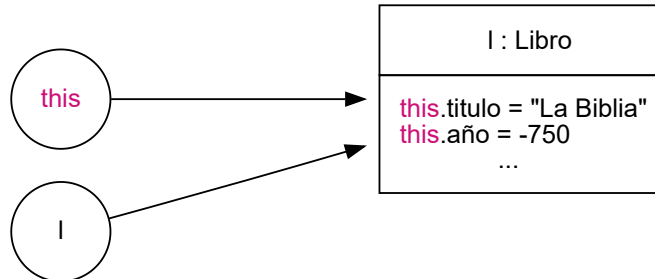
    // <constructor/es>
    public Libro(string titulo, in int año, in int paginas)
    {
        this.titulo = titulo;
        this.año = año;
        this.paginas = paginas;
        paginasLeídas = 0; // Al crear cualquier libro, llevaré leídas 0 páginas.
    }
}
```

👉 **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, ...);
```

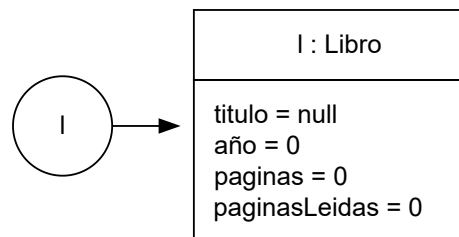


Constructor por defecto

- Si no definimos ningún constructor, **se define uno por defecto** que nos permitirá crear instancias de libro de la siguiente manera.
- Sin embargo, si definimos un constructor concreto, **dejará de estar disponible el constructor por defecto**, a no ser que lo definamos explícitamente nosotros.

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

Sin embargo esto permitirá crear instancias del objeto libro a los valores default para los tipos con e se definen los campos.



👉 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 nuestros 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;
}
```

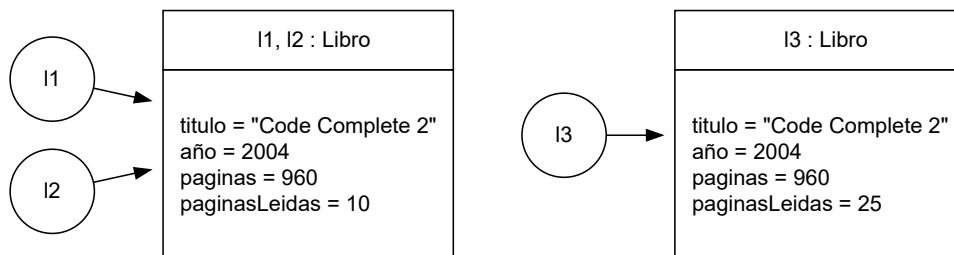
- Recordemos del tema 5 que si hiriéremos el siguiente código...

```
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() * 100D / GetPaginas());
    }

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

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

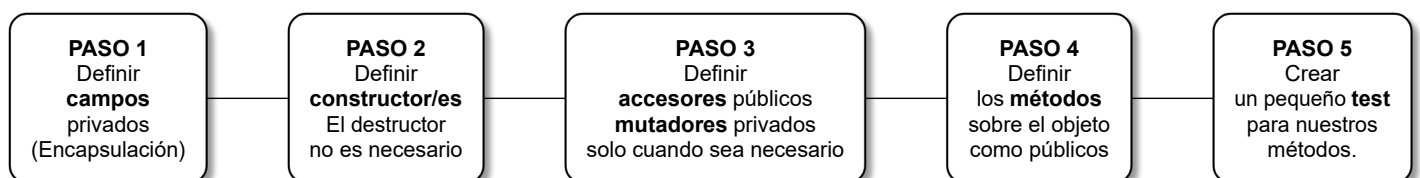
- **Nota:** Más adelante veremos cómo crear nuestros test e forma más apropiada.
- En este caso, vamos a hacer un pequeño programa que cree un instancia de un objeto libro y posteriormente lo lea de 300 en 300 páginas.

```
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 factoría'.
// 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#8 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 el 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 vamos describiremos a continuación.

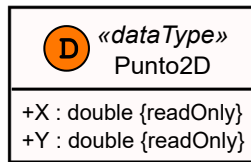
💡 Tips para definir un tipo usando struct

- 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**.
 - Un 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...



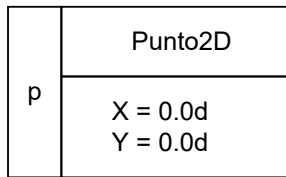
```
struct Punto2D
{
    public readonly double X;
    public readonly double Y;
}
```

En lugar de usar el palabra reservada `class`, hemos usado `struct`. Además, como que remos 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 `esto`, 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.

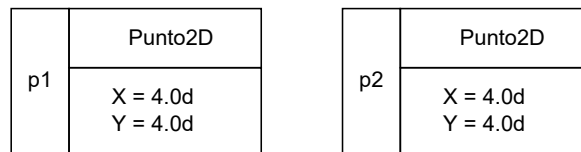
```
struct Punto2D
{
    public readonly double X;
    public readonly double Y;

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

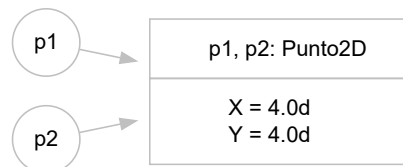
    public string ATexto()
    {
        return $"({X:G2} - {Y:G2})";
    }
}
```

Ahora si hacemos una asignación de un punto sobre otro, tendremos 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...

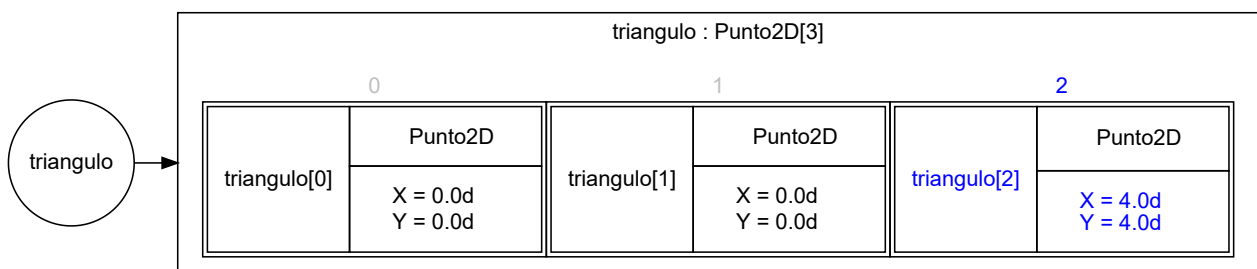


Si planteamos lo mismo pero en términos de colecciones homogéneas. Si hacemos ...

```
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...

	«Data Type» 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:

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

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

- Otro que reciba el ISBN como cadena.

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

- 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
        // fijate 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(f5.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 I
```

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.");
        }
    }
}
```

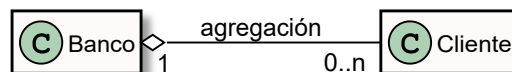
Roles entre clases

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 📌

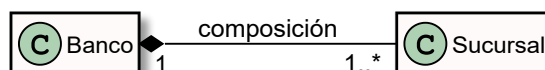
- Ahorramos memoria.
- Al compartirse la referencia al mismo objeto, mantendremos la integridad referencial.
- Mejor manejo de objetos complejos.
- Los objetos solo se crean cuando se necesitan.

Desventajas de la agregación 🗑️

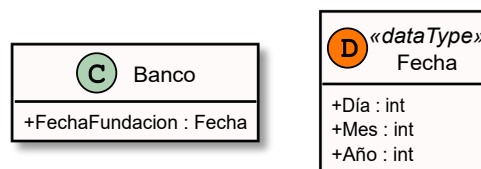
- Sobre todo en lenguajes no gestionados como C++. Se puede producir un efecto de **Aliasing**, si destruimos el objeto agregado.

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 un 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 agregación 🗑️

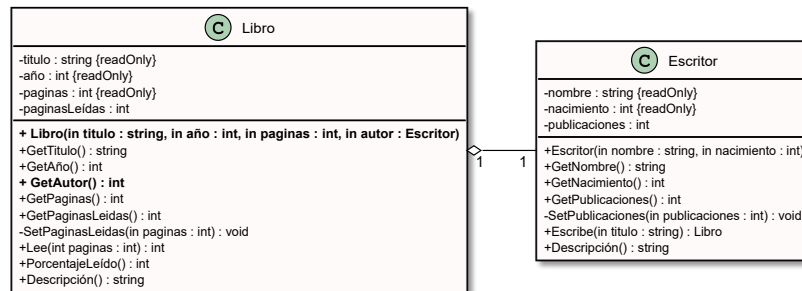
- Consume mucha memoria.
- No compartiremos. Por tanto, si necesitamos mantener la integridad referencial, deberemos hacerlo manualmente.
- Se pueden crear objetos que puede que no necesitamos.

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 Descripcion()
    {
        // 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.Descripcion()}\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...

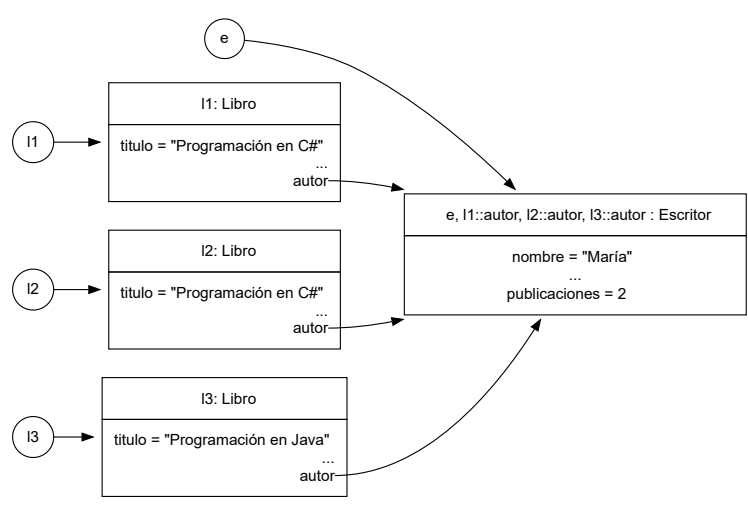
```
static void Main()
{
    Escritor e = new Escritor("María", 1980);
    Libro l1 = e.Escribe("Programación en C#");
    Libro l2 = new Libro(l1);
    // Console.WriteLine(l1.Descripcion());
    Console.WriteLine(l2.Descripcion());
    Libro l3 = e.Escribe("Programación en Java");
    Console.WriteLine(l3.Descripcion());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

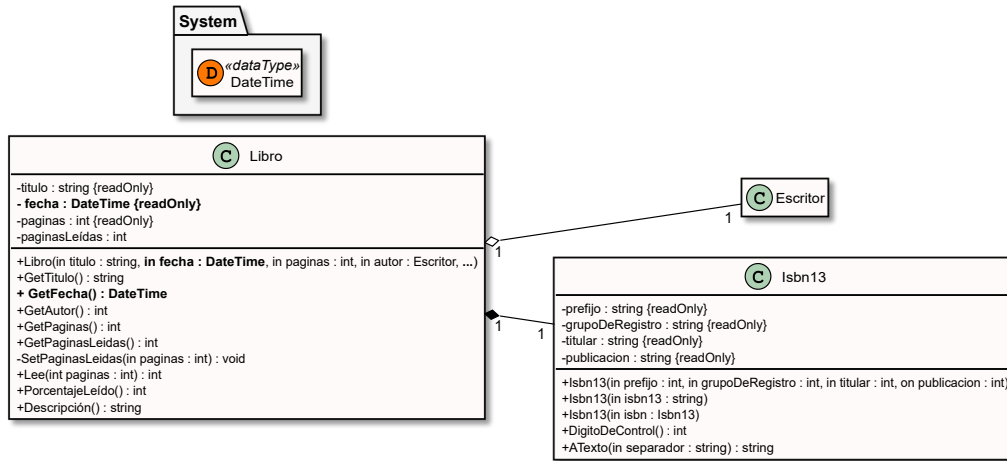


Ejemplo de Composición

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**.

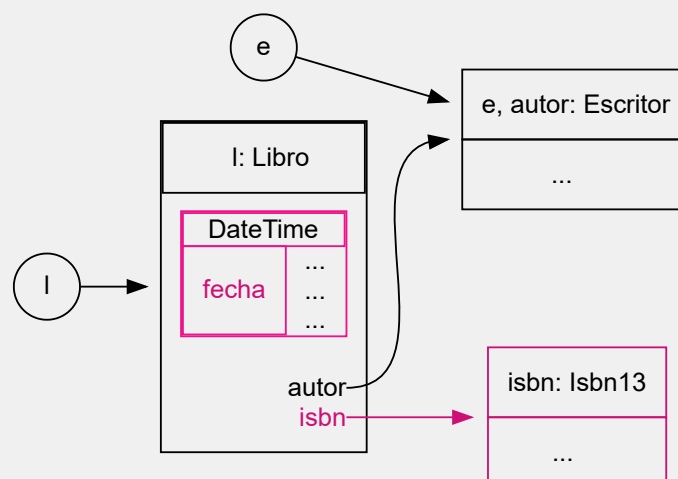
Lo mas lógico es que llegue el **ISBN** al constructor de **Libro**, puesto que no es **responsabilidad** del libro definirlo, sino más bien de un posible objeto de tipo **Editorial** de existir el mismo.

Importante: Debemos 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 **Escritor** 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 tenemos que recibir el ISBN como parámetro, la forma **más común** (*existen otras*) de abordarlo es la siguiente:

Si el objeto compuesto que recibimos como parámetros es un tipo referencia, independientemente de si es mutable o no, nos quedemos con **una copia del mismo**.

En el fondo esto es lo que sucede con un tipo valor como `DateTime` por su '*naturaleza*'.

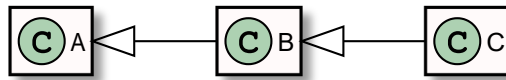
De esta manera el objeto `Isbn13` que estoy copiando desaparecerá junto con el libro.

🔗 **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,
        Isbn13 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;
        // Lo mismo sucede con el constructor copia donde 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, Isbn13 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);
        Isbn13 isbn = new Isbn13("9788420454665");
        // 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#", isbn);
        Console.WriteLine(l.Descripcion());
    }
}
```


Relación de Herencia (Generalización)

- Una de las características principales de la POO. Formalmente podemos definirla como....
 - Un tipo de relación entre clases, en la cual una clase denominada **subclase** (o también *clase hija*), comparte la estructura y/o comportamiento definidos en una o más clases, llamadas **superclases** (o también *clase padre*, *clase base*).
 - En otras palabras, una subclase añade sus propios atributos y métodos a los de la superclase, por lo que generalmente es mayor que esta y representará a un grupo **menor** de objetos.
- De una forma menos formal podemos resaltar que...
 - La subclase es un **concreción** de la superclase que representará una **generalización**.
 - Representará el tipo de relación '**Es un/a**'. Ej. '*Un coche es un vehículo.*'
 - La **herencia** nos servirá para reutilizar código y por tanto no repetir funcionalidades.
- Representaremos el rol a través de una flecha de punta hueca de la subclase a la superclase y se podrán producir relaciones **transitivas**.



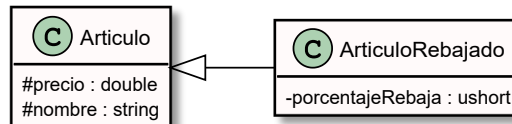
- B hereda de A
- B es una concreción A
- A es una generalización B
- A es la superclase y B la subclase
- C hereda de B y A
- B y C son subclases de A

Tipos de Herencia

Herencia Simple

- Cuando la subclase hereda de **una sola** superclase.
- Será la que nosotros vamos a usar.
- Por **ejemplo**, supongamos Tenemos la superclase **Articulo** con un **precio** y un **nombre**.

Una subclase de **Articulo** denominada **ArticuloRebajado** que además añade al articulo un campo con el porcentaje de rebaja denominado **porcentajeRebaja**.

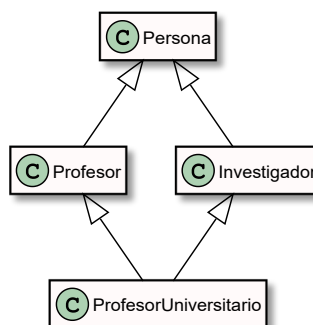


👁 La instancia de un objeto en memoria de **ArticuloRebajado** además de tener un campo **porcentajeRebaja**, tendrá los campos **precio** y **nombre** por ser también un **Articulo** y todos ellos definirán su estado.

Herencia Múltiple

- Se dará cuando una subclase hereda características de varias superclases.
- **Tiene más desventajas que ventajas. Por eso C# y Java NO la permiten.** Aunque otros lenguajes como Python o C++ sí.
- Entre las **desventajas** que hace que C# no la permita podemos destacar:
 - **Menor velocidad** de ejecución.
 - **Herencia repetida (Transitividad).**

En el ejemplo **ProfesorUniversitario** hereda **2 veces** o a través de dos clases diferentes los atributos de **Persona**.



- **Diseños más complejos** y más difíciles de aprender y utilizar por el programador.

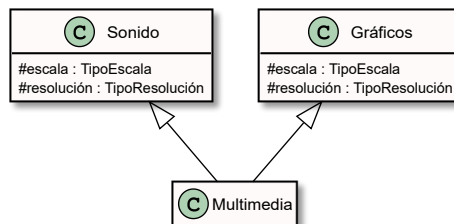
Nota: Siempre podremos **rediseñar** utilizando herencia simple.

- **Colisiones de Nombres**

En el ejemplo la subclase **Multimedia** hereda campos **con el mismo nombre** de las clases base **Sonido** y **Gráficos**.

Cuando hagamos referencia al campo **escala** en **Multimedia**.

¿Cómo podemos saber si estamos haciendo referencia a la escala de **Sonido** o la 3de **Gráficos** ?



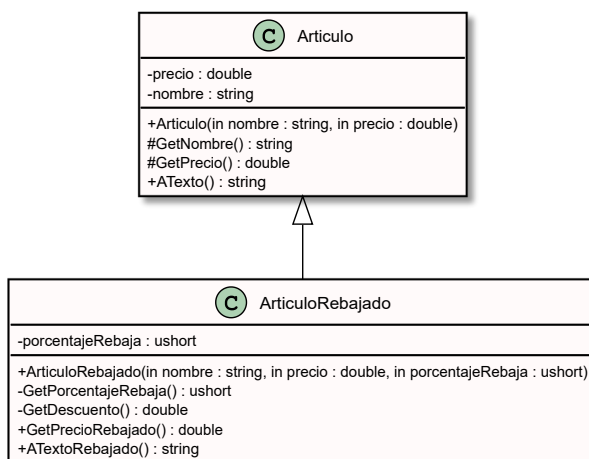
Implementando la herencia en C#

Separaremos el nombre de la subclase y la superclase por el carácter **':'** como en C++

```

class <NombreSubClase> : <NombreSuperClase>
{
    // Definiremos solo las concreciones de la subclase respecto la superclase
}
  
```

Partamos del ejemplo que hemos visto en la herencia simple, expresado en el siguiente diagrama de clases UML...



👉 Fíjate que en el diagrama de clases, aparece el símbolo **#**.

- Es un modificador que **solo tiene sentido aplicarlo a una superclase**.
- Se representará por la palabra reservada **protected**, y significará que el **campo** o el **método** al que modifica, no puede ser accedido desde fuera de la clase como en el caso de **private**, pero si desde las **subclases** de la misma.

```

class Artículo
{
    private double precio;
    private string nombre;

    public Artículo(string nombre, in double precio)
    {
        this.nombre = nombre;
        this.precio = precio;
    }
    public string ATexto() {
        return $"Nombre: {nombre}\n Precio: {precio:F2}€";
    }
    protected string GetNombre() {
        return nombre;
    }
    protected double GetPrecio() {
        return precio;
    }
}

```

```

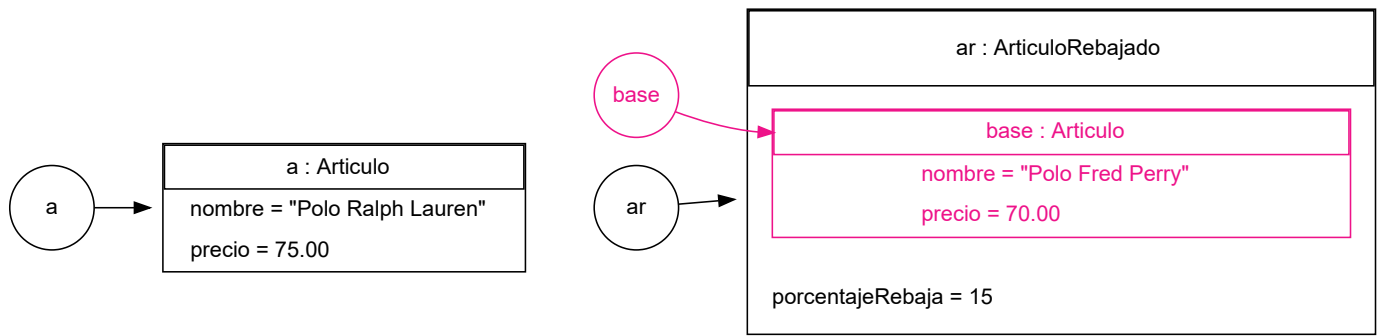
1 class ArtículoRebajado : Artículo
{
    private ushort porcentajeRebaja;

    public ArtículoRebajado(
        string nombre, in double precio,
7         in ushort porcentajeRebaja) : base(nombre, precio)
    {
        this.porcentajeRebaja = porcentajeRebaja;
    }
    private ushort GetPorcentajeRebaja()
    {
        return porcentajeRebaja;
    }
    private double GetDescuento()
    {
        return GetPrecio() * GetPorcentajeRebaja() / 100d;
    }
    public double GetPrecioRebajado()
    {
        return GetPrecio() - GetDescuento();
    }
    public string ATextoRebajado()
    {
        return $"Nombre: {GetNombre()}\nRebaja: {GetPorcentajeRebaja()}%\n" +
            $"Antes: {GetPrecio():F2}€\nAhora: {GetPrecioRebajado():F2}€";
    }
}

static class EjemploHerencia
{
    static void Main()
    {
        Artículo a = new Artículo("Polo Ralph Lauren", 75d);
        Console.WriteLine(a.ATexto());

        ArtículoRebajado ar = new ArtículoRebajado("Polo Fred Perry", 70d, 15);
        Console.WriteLine(ar.ATexto());
        Console.WriteLine(ar.ATextoRebajado());
    }
}

```



Palabra reservada **base**

- Si nos fijamos (**línea 7**) el constructor de `ArticuloRebajado` solo se encarga de inicializar y crear los atributos específicos de la subclase, para crear los de la clase, llamaremos al constructor que deseemos de la clase base o superclase, utilizando la palabra reservada `:base(<parámetrosBase>)` a continuación de la declaración del constructor de la subclase.
- Si hay un **constructor por defecto** en la superclase **no haría falta poner nada**, puesto que automáticamente sería llamado al llamar al de la subclase.
- Al igual que `this` era una referencia implícita al objeto de la propia clase, en las subclases tenemos la palabra reservada `base` que también es una referencia implícita a un objeto de la superclase, para la subclase actual.

Nota: Me servirá en los casos en los que en la subclase y en la superclase tengamos un **método con el mismo nombre**.

Ocultación e Invalidación

Si te has fijado, en `Articulo` el getter de `precio` lo hemos llamado `double GetPrecio()` y en `ArticuloRebajado` le hemos llamado `double GetPrecioRebajado()`. Esto lo hemos hecho para no producir colisiones de signatures (*recuerda que la subclase puede acceder a los métodos de la superclase*).

Sin embargo, en el fondo es una **redundancia**, porque si a un objeto `ArticuloRebajado articuloRebajado;` le aplico un método llamado `articuloRebajado.GetPrecio();` ya se que estoy obteniendo el precio rebajado.

Si le ponemos el mismo nombre, **tendríamos métodos con signatures idénticas en la superclase y la subclase** y posiblemente recibamos algún tipo de aviso del compilador. Pero ...

- ¿Se puede hacer esto?
- ¿Cómo resolvemos la ambigüedad que se produce?

👉 En la **POO tradicional** hay **dos estrategias** posibles:

- **Reemplazo:** Se sustituye completamente la implementación del método heredado manteniendo la signature. Comúnmente se le conoce como **Ocultación** (*hiding*)
- **Refinamiento:** Se añade nueva funcionalidad al comportamiento heredado. Comúnmente se le conoce como **Invalidación** (*overriding*)

Ocultación o reemplazo en C#

- Será la **estrategia** que aplica **por defecto de C#**, aunque como hemos comentado, el compilador **nos avisará** por si nos hemos 'despistado' y realmente queríamos hacer otra cosa. Por ejemplo, VSCode generará el siguiente mensaje.

```
'ArticuloRebajado.GetPrecio()' oculta el miembro heredado 'Articulo.GetPrecio()'. Use la palabra clave new si su intención era ocultarlo. CS0108
```

- Puesto que con el **reemplazo** lo que buscamos es definir una nueva funcionalidad para una operación heredada, antepondremos la palabra reservada **new** a la operación o método de la clase hija o subclase con la misma signatura que queremos ocultar en la clase base o superclase.
- Supongamos la misma relación de herencia anterior donde ahora queremos hacer una ocultación de los métodos **GetPrecio** y **ATexto**.

C ArticuloRebajado	
-porcentajeRebaja : ushort	
+ArticuloRebajado(in nombre : string, in precio : double, in porcentajeRebaja : ushort)	
-GetPorcentajeRebaja() : ushort	
-GetDescuento() : double	
+ GetPrecio() : double	
+ ATexto() : string	

La implementación en C# quedaría como sigue...

👉 **Importante (línea 17):**

- **base.GetPrecio();** me devuelve el precio del original del artículo (sin descuento).
- **this.GetPrecio();** o **GetPrecio();** me devuelve el precio rebajado (con descuento).

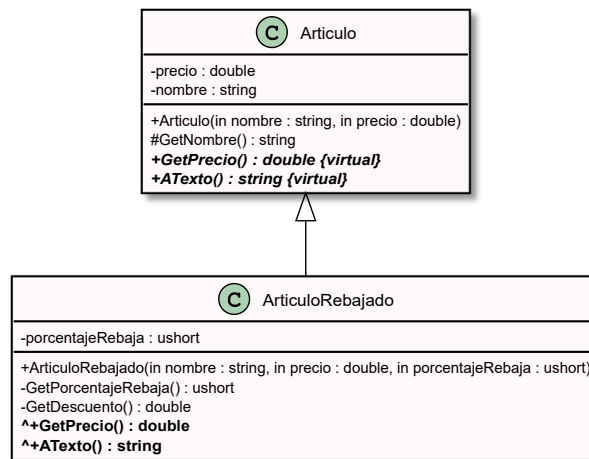
```
class ArticuloRebajado : Articulo
{
    // ... código omitido para abreviar.
    private double GetDescuento()
    {
        return base.GetPrecio() * GetPorcentajeRebaja() / 100d;
    }
    8 public new double GetPrecio() // Añado el modificador new para confirmar que quiero hacer una ocultación.
    {
        // Aquí si en lugar de llamar a base.GetPrecio(), llamase a GetPrecio() tendría una llamada
        // recursiva y se produciría una 'Stack Overflow'.
        return base.GetPrecio() - GetDescuento();
    }
    14 public new string ATexto() // Añado el modificador new para confirmar que quiero hacer una ocultación.
    {
        return $"Nombre: {GetNombre()}\nRebaja: {GetPorcentajeRebaja()}%\n" +
        17      $"Antes: {base.GetPrecio():F2}€\nAhora: {this.GetPrecio():F2}€";
    }
}
```

Invalidación o refinamiento en C#

- Será la opción que tomaremos en el 99% de los casos, pues es más flexible que la ocultación y me permitirá realizar los **enlaces dinámicos** que veremos mas adelante.
- Haremos lo que hacía la clase padre, más nueva funcionalidad.
- Llamaremos o marcaremos como métodos '**virtuales**' a los métodos '**invalidables**' en la superclase que sean **públicos** o **protegidos**.
- Para ello utilizaremos la palabra reservada **virtual** precediendo a la declaración del método invalidable y la palabra **override** precediendo la declaración de un método que invalida a uno invalidable o virtual en la superclase.
- 👁 A diferencia con la ocultación, ambos **métodos deberán tener la misma accesibilidad**.

Nota: Para representar lo que queremos hacer. En '**nuestros**' diagramas de clases UML...

- Pondremos el modificador **{virtual}** al nombre del método invalidable. También, marcaremos en '**cursiva**' aquellos métodos virtuales o virtuales puros (que trataremos más adelante) ya que aunque dejó de usarse a partir de la versión 2.5 de UML, sigue siendo una notación ampliamente usada .
- Aquellos métodos que **invaliden** un método en su superclase los marcaremos con el caracter **^** precediendo al nombre del método, para tenerlo claro.
- Si no ponemos nada, supondremos por convenio que estamos haciendo una ocultación.



La implementación en C# quedaría como sigue...

1. Modifico los métodos como invalidables con **virtual** en **Articulo** y los hago public como va a ser en las subclases.

```
class Articulo
{
    // ... código omitido para abreviar.
    4 public virtual double GetPrecio() {
        return precio;
    }
    7 public virtual string ATexto() {
        return $"Nombre: {nombre} Precio: {precio:F2}€" +
    }
}
```

2. Invalido el método con la misma signatura en la subclase con el modificador **override**

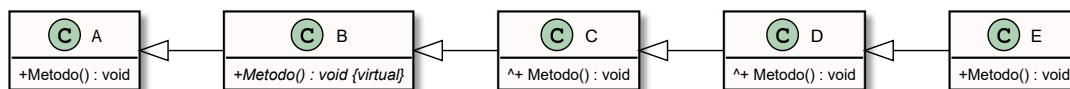
💡 Tips:

- 1 - Si escribo **public override** `Ctrl + <espacio>` el '**intellisense**' me ofrecerá permitirá escoger entre los métodos invalidables.
- 2 - Si refactorizamos sobre el nombre de la clase con `Ctrl + .` una de ellas será '**Generar invalidaciones...**'

```
class ArtículoRebajado : Artículo
{
    // ... código omitido para abreviar.
    public override double GetPrecio() // Añado el modificador override para confirmar que quiero hacer una invalidación.
    {
        return base.GetPrecio() - GetDescuento();
    }
    public override string ATexto() // Añado el modificador override para confirmar que quiero hacer una invalidación.
    {
        return $"Nombre: {GetNombre()}\nRebaja: {GetPorcentajeRebaja()}%\n" +
            $"Antes: {base.GetPrecio():F2}€\nAhora: {GetPrecio():F2}€";
    }
}
```

- Lo normal es que en el momento que modifiquemos algún método en la jerarquía con **virtual** (invalidable), los métodos con la misma signatura en las subclases se modificarán con **override**.
- Sin embargo, podemos hacer diseños más complejos como el del ejemplo siguiente donde volvemos a ocultar como sucede en la clase **E**.

Deberemos evitar diseños complejos.



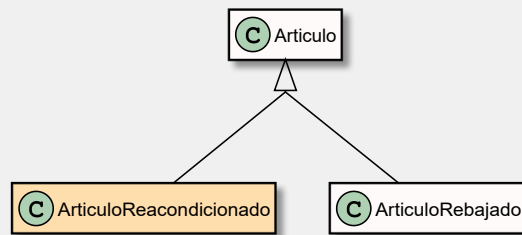
```

class A
{
    public void Metodo() { ... }
}
class B : A
{
    // Oculta el de A y lo marco como virtual o invalidable.
    public new virtual void Metodo() { ... }
}
class C : B
{
    public override void Metodo() // Invalido Metodo() en B
    {
        ...
        base.Metodo(); // Llamada a la implementación de B
    }
}
class D : C
{
    public override void Metodo() // Invalido Metodo() en B y C
    {
        ...
        base.Metodo(); // Llamada a la implementación de C
    }
}
class E : D
{
    public new void Metodo() // Corto la secuencia de invalidaciones ocultando el método.
    {
        ...
    }
}
  
```


Caso de estudio

Vamos a definir otra concreción más de la clase **Articulo**. En este caso, va a representar artículos reacondicionados, de los que vamos a añadir una **fecha de reacondicionamiento**, la **empresa** que lo realiza y una **descripción** del trabajo realizado.

- Solo vamos a invalidar el método **string ATexto()**.
- Instancia un objeto de la nueva clase.



Nota: Antes de ver la la propuesta de implementación, intenta pensar cómo sería la misma.

```
class ArticuloReacondicionado : Articulo
{
    // Una vez reacondicionado, los datos no van a cambiar.
    private readonly DateTime fechaReacondicionamiento;
    private readonly string empresa;
    private readonly string descripcion;

    public ArticuloReacondicionado(
        string nombre,
        double precio,
        DateTime fechaReacondicionamiento,
        string empresa,
        string descripcion) : base(nombre, precio)
    {
        this.fechaReacondicionamiento = fechaReacondicionamiento;
        this.empresa = empresa;
        this.descripcion = descripcion;
    }
    public string GetEmpresa()
    {
        return empresa;
    }
    public override string ATexto()
    {
        // Aprovechamos para no repetir el código que muestra los datos que son de artículo llamando a base.ATexto()
        return base.ATexto() +
            $"\nFecha reacondicionamiento: {fechaReacondicionamiento.ToShortDateString()}\n" +
            $"Empresa: {GetEmpresa()}\n" +
            $"Descripción: {descripcion}";
    }
}

static class EjemploHerencia
{
    static void Main()
    {
        ArticuloReacondicionado ac = new ArticuloReacondicionado(
            "Fuente TFX", 50d,
            DateTime.Now, "Balmis S.A",
            "Se cambia condensador electrolítico");
        Console.WriteLine(ac.ATexto());
    }
}
```

Polimorfismo de datos o inclusión

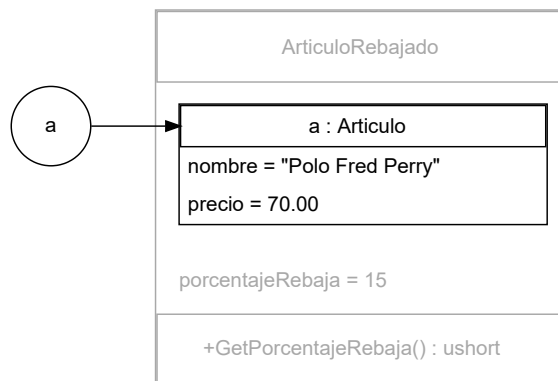
- Es la capacidad de un identificador de hacer referencia a instancias de distintas clases durante su ejecución.
- Se logra a través del **principio de sustitución**.

Principio de sustitución de **Liskov** (Upcasting)

- Podemos decir que es, cuando un identificador que hemos declarado del tipo la superclase, referencia a un objeto de la subclase.
- También se le conoce como **upcasting**
- La **conversión** o 'cast' se hace de forma **implícita**.
- Creamos un objeto de la subclase y lo asignamos a la superclase.

```
ArticuloRebajado ar = new ArticuloRebajado("Polo Fred Perry", 70d, 15);
Articulo a = ar;

// También podemos hacerlo directamente.
Articulo a = new ArticuloRebajado("Polo Fred Perry", 70d, 15);
```



👉 **Importante:** Aunque creamos un objeto **ArticuloRebajado** **completo** en memoria. A través de **a** solo podremos acceder a la parte de **Articulo** que hay dentro del **ArticuloRebajado** .

Por ejemplo, si tuviéramos un accesor público en **ArticuloRebajado** denominado **ushort GetPorcentajeRebaja()** , no podríamos hacer **a.GetPorcentajeRebaja()** .

Downcasting

- Se tratará de la **operación contraria a la sustitución** o upcasting.
- Solo podremos hacerla si realmente la referencia que tenemos es del tipo al que queremos hacer el downcasting, en caso contrario obtendremos un **error en tiempo de ejecución**.



Formas de realizar el Downcasting

1. Mediante **cast explícito**:

```
Articulo a = new ArticuloRebajado("Polo Fred Perry", 70d, 15);
ArticuloRebajado ar = (ArticuloRebajado)a; // realmente a es un ArticuloRebajado
```

Sin embargo el siguiente código **produciría un error al ejecutar** 😬

```
Articulo a = new Articulo("Polo Ralph Lauren", 75f);
ArticuloRebajado ar = (ArticuloRebajado)a;
```

2. Mediante el operador **is**

Nos sirve para preguntarle a un objeto si es de un determinado tipo y saber así con seguridad si podemos hacer el downcasting.

```
Articulo a = new Articulo("Polo Ralph Lauren", 75f);

3 if (a is ArticuloRebajado)
{
    ArticuloRebajado ar = (ArticuloRebajado)a;
    Console.WriteLine(ar);
}
```

3. Mediante el operador **as**

Realiza directamente el downcasting y si no puede asigna **null**.

```
Articulo a = new Articulo("Polo Ralph Lauren", 75f);

2 ArticuloRebajado ar = a as ArticuloRebajado;
Console.WriteLine(ar);

// Equivaldría a hacer...
ArticuloRebajado ar = a is ArticuloRebajado ? (ArticuloRebajado)a : null;
```

4. Apoyándonos en el operador de uso combinado null **??** que vimos en el tema 2

Se podría usar en el downcasting de la siguiente forma:

```
Articulo a = new ArticuloReacondicionado("Samsung s10", 300d, DateTime.Now, "FOXCOM", "Cambio de batería");

ArticuloRebajado ar = a as ArticuloRebajado
    ??
    new ArticuloRebajado( a.GetNombre(), a.GetPrecio(), 0d);
```

5. Usando la instrucción **switch** con patrones de tipo

```
Articulo a = new ArticuloReacondicionado("Samsung s11", 300d, DateTime.Now, "FOXCOM", "Cambio de batería");

switch (a)
{
    case null:
        break;
    case ArticuloRebajado _:
        Console.WriteLine("Es rebajado");
        break;
    case ArticuloReacondicionado ac when ac.GetEmpresa() == "FOXCOM":
        Console.WriteLine("Es reacondicionado de FOXCOM");
        break;
    case ArticuloReacondicionado ac:
        Console.WriteLine(ac.ATexto());
        break;
    default:
        Console.WriteLine("No es un tipo de objeto contemplado");
        break;
}
```

6. Usando la expresión **switch**

```
Articulo a = new ArticuloReacondicionado("Samsung s11", 300d, DateTime.Now, "FOXCOM", "Cambio de batería");

Console.WriteLine((a switch
{
    ArticuloRebajado ar => ar,
    _ => new ArticuloRebajado(a.GetNombre(), a.GetPrecio(), 0)
}).ATexto());
```

Ligadura Dinámica

También se le conoce como **enlace dinámico**.

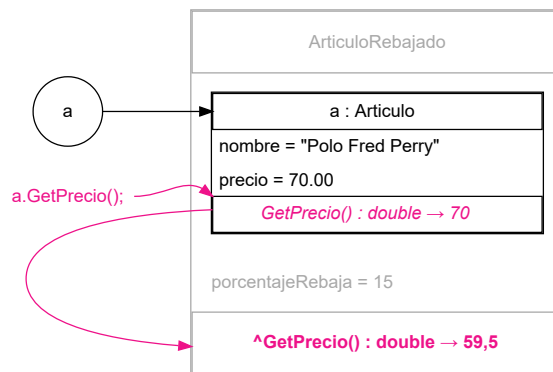
Una de las principales 'ventajas' de la **invalidación**, es que al hacer una **sustitución** como la que hemos visto del tipo...

```
Articulo a = new ArticuloRebajado("Polo Fred Perry", 70d, 15);
Console.WriteLine(a.GetPrecio());
```

`Console.WriteLine(a.GetPrecio());` mostrará **59,5** y no **70**. Pero,... ¿Cómo puede suceder esto si **a** está referenciando a la parte de **Articulo** que hay en el objeto **ArticuloRebajado** instanciado y **GetPrecio()** de **Articulo** me devuelve el precio sin el descuento?

Si nos fijamos en la figura siguiente, lo que realmente sucede es que al hacer **a.GetPrecio()** y ver que **GetPrecio()** está marcado como invalidable o **virtual**. Buscará posibles invalidaciones de ese método en el objeto realmente instanciado (**ArticuloRebajado**) y si existen lo que hará es llamar a la invalidación.

A este enlace entre el método virtual y su invalidación, lo denominaremos **ligadura dinámica** y se denomina '**dinámica**' puesto que **se decide en tiempo de ejecución**, dependiendo del objeto que realmente tengamos instanciado y esté referenciado por la sustitución.



Ejemplo de uso del Enlace Dinámico

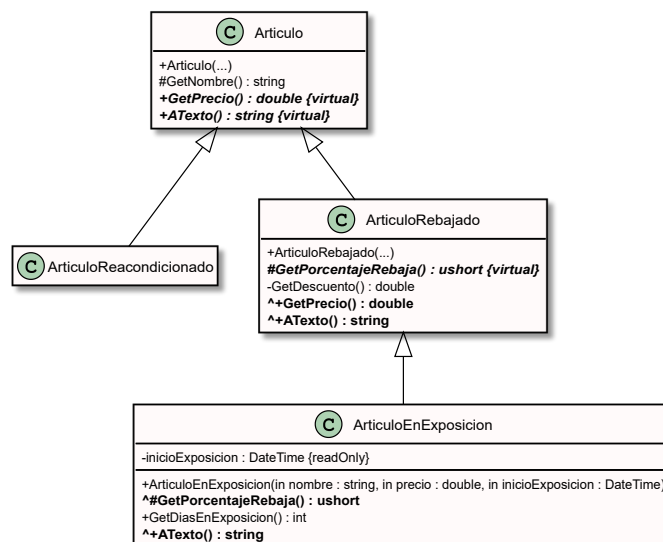
Veamos un ejemplo más elaborado a través de otro ejemplo en el que vamos a ampliar nuestra jerarquía de artículos.

Supongamos que nos piden hacer una concreción más de artículos para aquellos que están en exposición. Nos comentan que **los artículos en exposición siempre tienen algún tipo de rebaja**. Por lo tanto podemos decir que **un artículo en exposición - es un - artículo rebajado**.

Además, se nos especifica que el porcentaje de rebaja coincidirá con los días que el artículo esté en exposición siendo un mínimo de un 1% y un máximo de 75% de su valor. De esta manera si un artículo lleva 20 días en exposición su descuento será del 20% pero si lleva 100 días su descuento será del 75%.

De lo expuesto, al crear un artículo en exposición, nos interesará saber la fecha en que se inició la misma.

Una posible modelación del diagrama de clases para implementación UML siguiendo nuestro convenio de nomenclatura sería...



Vamos a realizar una propuesta de implementación, comentada, de la especificación anterior.

```
class ArtículoRebajado : Artículo
{
    // ... <Código omitido para abreviar>

    // Modificamos la accesibilidad a protected porque a los métodos privados no se les
    // puede aplicar el modificador virtual.
    protected virtual ushort GetPorcentajeRebaja()
    {
        return porcentajeRebaja;
    }
    private double GetDescuento()
    {
        return base.GetPrecio() * GetPorcentajeRebaja() / 100d;
    }
    public override double GetPrecio()
    {
        return base.GetPrecio() - GetDescuento();
    }
}
class ArtículoEnExposicion : ArtículoRebajado
{
    private readonly DateTime inicioExposicion;


    // En un principio el descuento es 0 y lo calculeramos dinámicamente.
    public ArtículoEnExposicion(string nombre, in double precio, in DateTime inicioExposicion) : base(nombre, precio, 0)
    {
        this.inicioExposicion = inicioExposicion;
    }
    // Invalidamo el porcentaje para calcularlo en función de los días en exposición.
    protected override ushort GetPorcentajeRebaja()
    {
        return Convert.ToInt16(Math.Clamp(GetDiasEnExposicion(), 1, 75));
    }
    // Los días en exposición se calculan en el momento actual, desde el inicio de la exposición.
    public int GetDiasEnExposicion()
    {
        return (DateTime.Now - inicioExposicion).Days;
    }
    // Invalidamo el método ATexto() de Artículo y ArtículoRebajado para que añada la nueva información.
    public override string ATexto()
    {
        return base.ATexto() +
            $"{nEn exposición desde: {inicioExposicion.ToShortDateString()} total {GetDiasEnExposicion()} días";
    }
}
```

Si ahora en el programa principal ejecutamos el siguiente código...

```
// Creamos un ArtículoEnExposicion con fecha de inicio hace 10 días.
// Hacemos una sustitución hacia el tipo más genérico Artículo
Artículo a = new ArtículoEnExposicion("TV Samsung OLED 50'", 999d, DateTime.Now.AddDays(-10));
Console.WriteLine(a.ATexto());
```

Mostrará el siguiente código...

```
Nombre: TV Samsung OLED 50'
Rebaja: 10%
Antes: 999,00€
Ahora: 899,10€
En exposición desde: <fecha actual - 10 días> total 10 días
```

 **Reflexión:** Si lo meditamos, se están realizando **dos enlaces o ligaduras dinámicas** ➡ ...

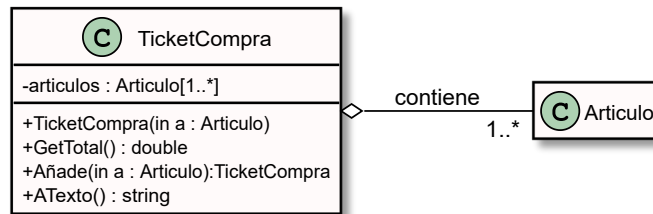
1. `a.ATexto()` ➡ `^ArtículoRebajado.ATexto()` ➡ `^ArtículoEnExposicion.ATexto()`
2. Al ejecutar `^ArtículoEnExposicion.ATexto()` se llamará a `base.ATexto()` el cual llamará a `^ArtículoRebajado.GetPrecio()` este al no estar invalidado llama a `ArtículoRebajado.GetDescuento()` que a su vez llama a `ArtículoRebajado.GetPorcentajeRebaja()` ➡ `^ArtículoEnExposicion.GetPorcentajeRebaja()` produciéndose el segundo enlace.

Utilidad del polimorfismo de datos (sustitución) y el enlace dinámico

- En ocasiones el software cambia y se añaden nuevas especificaciones, como pudieran ser nuevos tipos de artículos en la jerarquía.
- Con el polimorfismo de datos, podremos adaptarnos a futuros cambios (Nuevas formas de un objeto), **sin realizar cambios** traumáticos y costosos en nuestros objetos ni en nuestra implementación.

Veamos esto a través de un **ejemplo simplificado**:

- Supongamos que queremos modelar una clase **TicketCompra** que contenga al menos un artículo y que me permita añadir artículos al mismo.
- Además, vamos a añadir un método para mostrar el ticket y otro para calcular el total del mismo.
- Un posible diseño simplificado para expresar esto podría ser ...



- Una posible implementación de la clase **TicketCompra** podría ser:

```
class TicketCompra
{
    private Articulo[] articulos;

    // Me aseguro de que un ticket al menos tenga un artículo.
    public TicketCompra(Articulo a)
    {
        articulos = new Articulo[] { a };
    }
    public double GetTotal()
    {
        double total = 0d;
        foreach (Articulo a in articulos)
            total += a.GetPrecio();
        return total;
    }
    // Fluent interface pattern para añadir artículo a mi ticket
    public TicketCompra Añade(Articulo a)
    {
        Array.Resize(ref articulos, articulos.Length + 1);
        articulos[articulos.Length-1] = a;
        return this;
    }
    public string ATexto()
    {
        StringBuilder ticket = new StringBuilder();
        foreach (Articulo a in articulos)
            ticket.Append($"{a.GetNombre(),-22} {a.GetPrecio(),10:F2}€\n");
        ticket.Append($"Total:",22) {GetTotal(),10:F2}€\n");
        return ticket.ToString();
    }
}
```

- Si implementamos el siguiente programa principal de test...

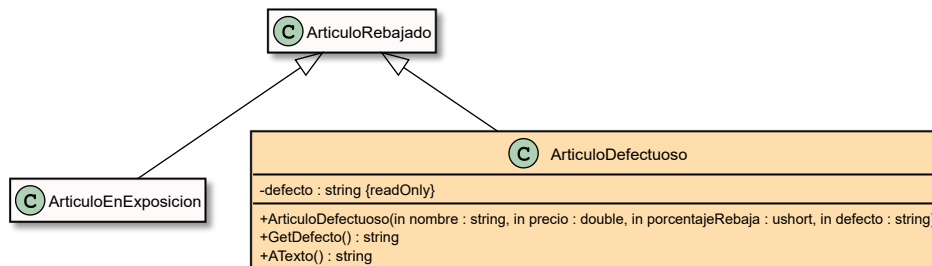
```
static void Main()
{
    TicketCompra t = new TicketCompra(new Articulo("Airpods Apple", 100d));
    t.Añade(new ArticuloEnExposicion("TV Samsung OLED 50'", 999d, DateTime.Now.AddDays(-10)))
        .Añade(new ArticuloRebajado("Honor Band 4", 35d, 15))
        .Añade(new ArticuloReacondicionado("Samsung s10", 300d, DateTime.Now.AddDays(-20), "TechnoBalmis", "Cambio de batería"));
    Console.WriteLine(t.ATexto());
}
```

- Obtendremos una salida del tipo...

Airpods Apple	100,00€
TV Samsung OLED 50''	899,10€
Honor Band 4	29,75€
Samsung s10	300,00€
Total:	1328,85€

Si nos fijamos aunque el ticket solo maneje artículos genéricos, ha sabido calcular correctamente a través del enlace dinámico el precio de cada uno de los artículos y el total.

- Si ahora añadiésemos otro tipo de artículo como por ejemplo **artículos defectuosos**, que funcionan bien y no han sido reacondicionados, pero tienen algún defecto menor, como una rozadura, pixel muerto, etc. y por tanto se les aplica una rebaja, pero también nos interesa guardar información del defecto.
- A la hora de modelizar podemos decir que **un artículo defectuoso - es un - artículo rebajado** y modelizarlo de la siguiente manera.



- Donde una posible implementación podría ser ...

```

class ArticuloDefectuoso : ArticuloRebajado
{
    private readonly string defecto;

    public ArticuloDefectuoso(string nombre, in double precio,
                              in ushort porcentajeRebaja, string defecto)
        : base(nombre, precio, porcentajeRebaja)
    {
        this.defecto = defecto;
    }
    public string GetDefecto()
    {
        return defecto;
    }
    public override string ATexto()
    {
        return base.ATexto() +
            $"{\nDefecto: {defecto}}";
    }
}
  
```

- 🖱 Si ahora añadimos un nuevo artículo de este tipo a nuestro ticket.

```

static void Main()
{
    TicketCompra t = new TicketCompra(new Articulo("Airpods Apple", 100d));
    t.Añade(new ArticuloEnExposicion("TV Samsung OLED 50''", 999d, DateTime.Now.AddDays(-10)))
    .Añade(new ArticuloRebajado("Honor Band 4", 35d, 15))
    .Añade(new ArticuloReacondicionado("Samsung s10", 300d, DateTime.Now.AddDays(-20), "TechnoBalmis", "Cambio de batería"))
    .Añade(new ArticuloDefectuoso("Woffler Bang & Olufsen", 300d, 5, "Pequeño picado en caja."));
    Console.WriteLine(t.ATexto());
}
  
```

No tendremos que modificar nuestra clase TicketCompra y esto será gracias al polimorfismo de datos.

El caso especial de la clase **Object** en C#

- La Clase **Object** definida en **System**, es una clase especial de la cual heredan de forma **implícita** todos los objetos, tanto valor como referencia, creados en C#.
- Por tanto, podemos decir que un objeto de **la clase Object** puede sustituir a cualquier objeto definido por nosotros o en las BCL.
- Historicamente, en los principios de C#, esta clase se utilizaba para tratar objetos de forma genérica como en colecciones, antes de que el lenguaje implementara la genericidad a través de genéricos o clases parametrizadas, cuyo uso es más recomendable.
- Podemos encontrar esta clase en otros lenguajes como Java.
- Define una serie de métodos invalidables o virtuales que podremos redefinir** en cualquiera de las clases que nosotros creemos. Entre ellos podemos destacar:

1. **public virtual string ToString()**

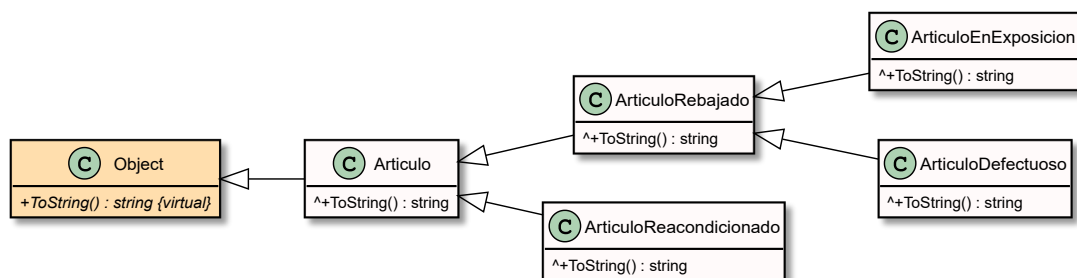
Este método, es llamado **automáticamente** cada vez que un objeto se intenta formatear cómo cadena y equivaldría en cierta manera al método **string ATexto()** que hemos definido en nuestra jerarquía de artículos.

En el fondo, aunque no se especifique, **Articulo** hereda implícitamente de **Object** y por tanto las clases **Articulo**, **ArticuloRebajado**, **ArticuloReacondicionado**, etc. heredan el método **ToString()** y además pueden invalidarlo.

De hecho si no lo invalidamos y hacemos el siguiente código...

```
namespace EjemploErencia
{
    class Articulo { ... }

    class Principal
    {
        static void Main()
        {
            Articulo a = new Articulo("Falda", 30f);
            Console.WriteLine(a);
            // Mostrará "EjemploErencia.Articulo" que es el NCC de la clase y
            // es lo que hace la implementación de ToString en la clase Object si no la invalidamos.
        }
    }
}
```



Por tanto podríamos haber hecho...

```
class Articulo
{
    ...
    public override string ToString() // Invalido el ToString de Object
    {
        return $"Nombre: {nombre}\nPrecio: {precio:F2}€";
    }
}
class ArticuloReacondicionado : Articulo
{
    ...
    public override string ToString() // Invalido el ToString de Articulo
    {
        return base +
            $"{nFecha reacondicionamiento: {fechaReacondicionamiento.ToShortDateString()}}\n" +
            $"Empresa: {GetEmpresa()}\n" +
            $"Descripción: {descripcion}";
    }
}
```



```

class Principal
{
    static void Main()
    {
        Artículo a = new Artículo("Falda", 30f);
        Console.WriteLine(a);
        // Mostrará sin necesidad de hacer a.ToString().
        // Nombre: Falda
        // Precio: 30.00€

        Artículo arc = new ArtículoReacondicionado("Samsung s10", 300d, DateTime.Now, "FOXCOM", "Cambio de batería");
        Console.WriteLine($"Datos -----\\n{arc}");
        // Mostrará sin necesidad de hacer arc.ToString().
        // Datos -----
        // Nombre: Samsung s10
        // Precio: 300.00€
        // Fecha reacondicionamiento: <dd/mm/aaaa actual>
        // Empresa: FOXCOM
        // Descripción: Cambio de batería
    }
}

```

2. public virtual bool Equals(object obj)

Se usa para comparar el objeto sobre el que se aplica con cualquier otro que se le pase como parámetro en **profundidad**. Devuelve true si ambos objetos son iguales y false en caso contrario.

Veamos un ejemplo en el cual lo invalidaremos en las clases `Artículo` y `ArtículoRebajado`.

```

class Artículo
{
    ...
    public override bool Equals(object obj)
    {
        bool iguales = obj is Artículo a // Si la entrada es un artículo y es distinto de null
                        && a.nombre.CompareTo(nombre) == 0 // y el nombre de los artículos son iguales.
                        && Math.Abs(a.precio - precio) < 1e-5; // y los precios son iguales con una precisión de 0.00001
        return iguales;
    }
}
class ArtículoRebajado : Artículo
{
    ...
    public override bool Equals(object obj)
    {
        bool iguales = obj is ArtículoRebajado a // Si la entrada es un artículo rebajado
                        && base.Equals((Artículo)obj) // y sus partes de artículo son iguales
                        && a.porcentajeRebaja == porcentajeRebaja; // y tienen el mismo porcentaje de rebaja.
        return iguales;
    }
}
class Principal
{
    static void Main()
    {
        Artículo a = new ArtículoRebajado("Honor Band 5", 35d, 15);

        // Aunque a sea de tipo Artículo, se hace un enlace dinámico al Equals de
        // ArtículoRebajado al invalidar el de artículo y realmente serlo.
        // Por lo cual en todos los casos se llama al Equals de ArtículoRebajado que a su vez
        // llama al de artículo.
        Console.WriteLine(a.Equals(new ArtículoRebajado("Honor Band 4", 30d, 20))); // False
        Console.WriteLine(a.Equals(new ArtículoRebajado("Honor Band 5", 35d, 15))); // True
        Console.WriteLine(a.Equals(new ArtículoRebajado("Honor Band 5", 35d, 10))); // False

        // En este caso el downcasting obj as ArtículoRebajado se evaluará a null.
        Console.WriteLine(a.Equals(new Artículo("Honor Band 5", 35d))); // False
    }
}

```

El resto de funciones invalidables de object las puedes consultar en la documentación oficial y ya se explicará más adelante en caso de necesitarse.

Caso de estudio

Veamos un ejemplo simplificado de diseño '*similar*' al seguido con los artículos.

En este ejemplo vamos a suponer que el propietario de una campa de aparcamiento vehículos a largo plazo. Decide instalar un sistema automatizado de entrada y salida de vehículos.

Para ello, decide poner un sistema de cámaras y una IA que trata de identificar información de los vehículos que entran y salen.

El sistema es capaz de identificar de un **vehículo** al pasar la siguiente información común:

- **Matrícula** que creará con formato ' **DDDD LLL** ' donde **D** será un dígito de 0 a 9 y **L** una letra mayúscula excluidas las vocales.
- **Color** que será un conjunto de valores con las tonalidades básicas. Devolviendo la IA la predominante en el vehículo. Estas podrán ser uno de los siguiente valores **Blanco** , **Morado** , **Cian** , **Azul** , **Rojo** , **Verde** , **Negro** , **Naranja** y **Gris** .
- **Marca** que será un conjunto de valores de logos que la IA es capaz de identificar en las imágenes como **DESCONOCIDA** , **BMW** , **SEAT** , **AUDI** , **RENAULT** , **MAN** , **DAF** , **CITROEN** , **TOYOTA** , **SUZUKI** , **YAMAHA** , **MERCEDES** , **PEGASO** .
- **Número de ocupantes** que la IA cree que hay en el interior del vehículo.

Además, de todo esto la IA de momento sabe clasificar los **vehículos** en estas especificaciones **Coche** , **Moto** y **Camión** . Cada especificación, a su vez debe estar categorizada a través de un conjunto de valores y en algún caso añadiremos información extra.

- **Coche**: Podrá tomar uno de los valores del siguiente conjunto de datos **SinIdentificar** , **Berlina** , **Coupe** , **Sedan** , **Cabrio** , **TodoTerreno** , **MonoVolumen** y **Crossover** .
- **Moto**: Podrá tomar uno de los valores del siguiente conjunto de datos **SinIdentificar** , **Scooter** , **Motocross** , **Naked** , **Trail** y **Supermotard**
- **Camión**: Podrá tomar uno de los valores del siguiente conjunto de datos **SinIdentificar** , **Articulado** , **Frigorífico** , **Cisterna** y **Trailer** . Pero además en los camiones es capaz de añadir el número de **ejes** y **carga máxima** en kilos.


Nuestra IA, que será el programa principal, pasará un objeto **Vehículo** , con la información que ha podido recolectar, a un objeto **CampaVehiculos** que tendrá una **capacidad de plazas determinada** (para nuestro caso de estudio **5 vehículos** independientemente de su tamaño para simplificar). Estas plazas se numerarán de la **1** a la **5**.

La clase **CampaVehiculos** tendrá pues las siguientes **operaciones básicas**:


- **Privadas**
 - **int Busca(Vehiculo v)** : Me retornará el **índice** en un array de vehículos que ocupa el vehículo **v** o **-1** si no lo encuentra.
 - **int BuscaPlazaVacía()** : Me retornará el **primer índice vacío** en un array de vehículos o **-1** si no lo encuentra ninguno vacío.
- **Públicas**
 - **int GetPlazasOcupadas()** : Me retornará el número de plazas ocupadas en la campa.
 - **bool Entra(Vehiculo v, out int plaza, out string aviso)** : Si la campa no está llena y si la matrícula del vehículo no se encuentra ya registrada, aparcará el vehículo en la primera plaza disponible retornando **true** y en **plaza** donde lo ha aparcado (1 a Numero Plazas).
En caso de no poder entrar el vehículo por las causas ya mencionadas retornaremos **false** y en **aviso** uno de los siguientes mensajes:
 1. 'Ya se encuentra en el aparcamiento el vehículo DDDD LL '
 2. 'Aparcamiento Lleno '
 - **bool Sale(Vehiculo v, out int plaza, out string aviso)** : Si la campa no está vacía y si la matrícula del vehículo se encuentra registrada, sacará el vehículo en la plaza donde se encuentra retornando **true** y en **plaza** donde se encontraba aparcado.
En caso de sacar el vehículo por las causas ya mencionadas retornaremos **false** y en **aviso** uno de los siguientes mensajes:
 1. 'No se registró la entrada del vehículo DDDD LL '
 2. 'Aparcamiento vacío '
 - **string ToString()** invalidado para que nuestro las plazas y los datos que se recibiesen de la IA de cada vehículo.

 Intenta realizar primero por tu cuenta la implementación de la especificación anterior. Posteriormente comparala con la propuesta comentada de solución que encontrarás más adelante.

Además, en las siguiente páginas puedes encontrar un programa principal de prueba y la salida que debe producir.

 Fijate bien en la definición de clases del programa principal, pues puede ayudarte a resolver dudas sobre cómo implementar la especificación anterior.

Propuesta de programa de prueba:

-  **Tip:** Para este programa de test, vamos a utilizar el método **public Type GetType();** que poseen todos los objetos por heredar de **object** y que como vemos devuelve un objeto de tipo **Type** perteneciente al **API de reflexión** de .NET. Fíjate en la **(línea 5)** como usamos este método en **v.GetType().Name** aunque **v** es de tipo **Vehiculo**, **Name** me devolverá el nombre de la especificación que realmente se 'esconda' en el mismo.
- Nota:** Si quieres saber un poco más sobre el concepto de reflexión puedes consultar [este enlace](#).

```
static class Principal
{
    static void Entra(CampaVehiculos camp, Vehiculo v)
    {
        5      string t1 = $"Entrando {v.GetType().Name} {v.GetCategoria()} {v.GetMatricula()}";
        string t2 = camp.Entra(v, out int plaza, out string aviso) ? $"Aparcado en la plaza {plaza}" : $"{aviso}";
        Console.WriteLine($"{t1,-42}-> {t2}");
    }

    static void Sale(CampaVehiculos camp, Vehiculo v)
    {
        string t1 = $"Saliendo {v.GetType().Name} {v.GetCategoria()} {v.GetMatricula()}";
        string t2 = camp.Sale(v, out int plaza, out string aviso) ? $"Deja libre la plaza {plaza}" : $"{aviso}";
        Console.WriteLine($"{t1,-42}-> {t2}");
    }

    static void Main()
    {
        CampaVehiculos camp = new CampaVehiculos();

        Entra(camp, new Coche(new Matricula("1020 DRG"), Vehiculo.Color.Azul, Vehiculo.Marca.SEAT, 3, Coche.Categoria.Coupe));
        Entra(camp, new Camion(new Matricula("8798 JWR"), Vehiculo.Color.Blanco, Vehiculo.Marca.DAF, 1, 2, 6000, Camion.Categoria.Fr));
        Entra(camp, new Coche(new Matricula("7643 LRF"), Vehiculo.Color.Rojo, Vehiculo.Marca.BMW, 4, Coche.Categoria.TodoTerreno));
        Entra(camp, new Coche(new Matricula("1020 DRG"), Vehiculo.Color.Azul, Vehiculo.Marca.SEAT, 3, Coche.Categoria.Coupe));
        Entra(camp, new Vehiculo(new Matricula("0000 DGP"), Vehiculo.Color.Negro, Vehiculo.Marca.DESCONOCIDA, 2));
        Entra(camp, new Moto(new Matricula("1111 GRF"), Vehiculo.Color.Rojo, Vehiculo.Marca.YAMAHA, 2, Moto.Categoria.Naked));
        Entra(camp, new Coche(new Matricula("1020 DRG"), Vehiculo.Color.Azul, Vehiculo.Marca.SEAT, 3, Coche.Categoria.Coupe));
        Sale(camp, new Camion(new Matricula("8798 JWR"), Vehiculo.Color.Blanco, Vehiculo.Marca.DAF, 1, 2, 6000, Camion.Categoria.Fr));
        Sale(camp, new Camion(new Matricula("8798 JWR"), Vehiculo.Color.Blanco, Vehiculo.Marca.DAF, 1, 2, 6000, Camion.Categoria.Fr));
        Sale(camp, new Moto(new Matricula("1111 GRF"), Vehiculo.Color.Rojo, Vehiculo.Marca.YAMAHA, 2, Moto.Categoria.Naked));
        Entra(camp, new Moto(new Matricula("1111 GRF"), Vehiculo.Color.Rojo, Vehiculo.Marca.YAMAHA, 2, Moto.Categoria.Naked));
        Entra(camp, new Camion(new Matricula("8798 JWR"), Vehiculo.Color.Blanco, Vehiculo.Marca.DAF, 1, 2, 6000, Camion.Categoria.S));

        Console.WriteLine(camp);
    }
}
```

```
Entrando Coche Coupe 1020 DRG      -> Aparcado en la plaza 1
Entrando Camion Frigorífico 8798 JWR -> Aparcado en la plaza 2
Entrando Coche TodoTerreno 7643 LRF -> Aparcado en la plaza 3
Entrando Coche Coupe 1020 DRG      -> Ya se encuentra en el aparcamiento el vehículo 1020 DRG
Entrando Vehiculo SinIdentificar 0000 DGP -> Aparcado en la plaza 4
Entrando Moto Naked 1111 GRF       -> Aparcado en la plaza 5
Entrando Coche Coupe 1020 DRG      -> Aparcamiento lleno
Saliendo Camion Frigorífico 8798 JWR -> Deja libre la plaza 2
Saliendo Camion Frigorífico 8798 JWR -> No se registró la entrada del vehículo 8798 JWR
Saliendo Moto Naked 1111 GRF       -> Deja libre la plaza 5
Entrando Moto Naked 1111 GRF       -> Aparcado en la plaza 2
Entrando Camion SinIdentificar 8798 JWR -> Aparcado en la plaza 5
```

Vehículos en el aparcamiento...

Plaza 1:
Matricula: 1020 DRG
Color: Azul
Marca: SEAT
Ocupantes: 3
Categoria: Coupe

Plaza 2:
Matricula: 1111 GRF
Color: Rojo
Marca: YAMAHA
Ocupantes: 2
Categoria: Naked

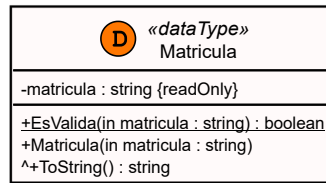
Plaza 3:
Matricula: 7643 LRF
Color: Rojo
Marca: BMW
Ocupantes: 4
Categoria: TodoTerreno

Plaza 4:
Matricula: 0000 DGP
Color: Negro
Marca: DESCONOCIDA
Ocupntes: 2
Categoria: SinIdentificar

Plaza 5:
Matricula: 8798 JWR
Color: Blanco
Marca: DAF
Ocupantes: 1
Categoria: SinIdentificar
Número de ejes: 2
Carga Máxima: 6000 Kg

1. Lo primero sería definir un **tipo para guardar matrículas válidas**. Básicamente contendrá un string, pero nos aseguraremos que lo que contiene es una matrícula.

Podríamos definirlo como clase, pero también cumple los requisitos para ser un tipo valor, puesto que un string es un tipo referencia **immutable**. Así pues, por repasar conceptos, vamos a definirlo como un tipo valor **struct**.



```
public struct Matricula
{
    private readonly string matricula;

    // De momento lo dejamos privado, pues podemos obtener la matrícula a través del ToString()
    private string GetMatricula()
    {
        return matricula;
    }

    // Dejamos la comprobación de matrícula como un método de clase publica de utilidad.
    public static bool EsValida(string matricula)
    {
        // [^a-z\W\s\dAEIOU] Cualquier letra que no sea: Minúscula, Caracter no alfanumerico, Espacios o Vocales en mayúscula.
        // Lo cual deja solo las mayúsculas que no sean vocales.
        return Regex.IsMatch(matricula, @"^[0-9]{4} [^a-z\W\s\dAEIOU]{3}$");
    }

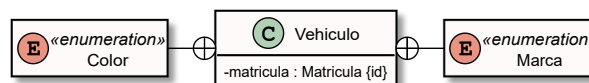
    public Matricula(string matricula)
    {
        // Solo permitimos crear matrículas válidas.
        if (!EsValida(matricula))
            throw new ArgumentException($"La matrícula {matricula} no es correcta.");
        this.matricula = matricula;
    }

    // Las estructuras como las clases, también heredan de Object y por tanto pueden
    // invalidar ToString.
    public override string ToString()
    {
        return GetMatricula();
    }
}
```

2. En segundo lugar definiremos la clase **Vehículo**. Fíjate que al usar el tipo valor Matrícula existe una **composición implícita**.

Además, hemos definido de forma **anidada** dos tipos enumerados publicos dentro de Vehículo. De esta manera que ara hacer referencia a ellos desde fuera de la clase usaremos la notación **Vehiculo.Color** y **Vehiculo.Marca** de esta manera no tendremos redundancia en los nombre y no habrá conflicto de nombres con otros tipos **Color**.

Esta relación de **anidación** en un diagrama de clases UML la podremos expresar de la siguiente manera:



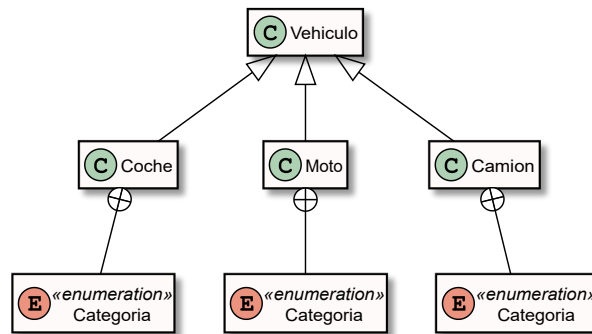
```
public class Vehiculo
{
    // Definimos los tipos enumerados de la especificación
    public enum Color { Blanco, Morado, Cian, Azul, Rojo, Verde, Negro, Naranja, Gris };
    public enum Marca { DESCONOCIDA, BMW, SEAT, AUDI, RENAULT, MAN, DAF, CITROEN, TOYOTA, SUZUKI, YAMAHA, MERCEDES, PEGASO };

    // Las características comunes a todos los vehículos, registrables por la IA
    // Serán de solo lectura y por tanto solo definiremos accesores.
    // Además, una vez creados los objetos vehículos ya no podrán cambiar de estado (Serán inmutables.).
    private readonly Matricula matricula;
    private readonly Color color;
    private readonly Marca marca;
    private readonly ushort ocupantes;
}
```

Fíjate en el los comentarios de la implementación de `object GetCategoria()`

```
public Matricula GetMatricula()
{
    return matricula;
}
public Color GetColor()
{
    return color;
}
public Marca GetMarca()
{
    return marca;
}
public ushort GetOcupantes()
{
    return ocupantes;
}
17 // Los objetos vehículos creados como tal no tienen categoría, y por tanto al preguntársela
// retornaremos SinIdentificar y daremos la posibilidad de invalidarla en futuras especificaciones.
// Además, fíjate que la categoría se devolverá en su forma de object con lo que, independientemente
// del tipo con el que la codifiquemos, de ella únicamente podremos saber la cadena que
// la representa llamando a ToString() (Ver línea 39).
public virtual object GetCategoria()
{
    return "SinIdentificar";
}
public Vehiculo(Matricula matricula, Color color, Marca marca, ushort ocupantes)
{
    this.matricula = matricula;
    this.color = color;
    this.marca = marca;
    this.ocupantes = ocupantes;
}
public override string ToString()
{
    return $"Matricula: {GetMatricula()}\n"
        + $"Color: {GetColor()}\n"
        + $"Marca: {GetMarca()}\n"
        + $"Ocupantes: {GetOcupantes()}\n"
39 + $"Categoria: {GetCategoria()}";
    // Aquí, aunque llamemos a GetCategoria() no mostrará siempre SinIdentificar puesto
    // que si hay una subclase que invalide el método, hará un enlace dinámico a la
    // implementación del mismo haga esa subclase.
}
public override bool Equals(object obj)
{
    // Invalidación de Equals de onject, para comparar dos objetos vehiculo por matricula.
    return obj is Vehiculo v && GetMatricula().ToString().CompareTo(v.GetMatricula().ToString()) == 0;
}
public override int GetHashCode()
{
    // En ocasiones la invalidación de Equals implica la invalidación de GetHashCode
    // El valos de hash de un vehiculo será el hash de la cadena de su matrícula.
    return GetMatricula().ToString().GetHashCode();
}
}
```

1. Vamos a realizar las **tres especificaciones de Vehículo** y usaremos **herencia** para ello. Además, como comentamos en **Vehículo**, definiremos un **tipo enum anidado Categoría** para cada una de las subclasificaciones de vehículos.
- Es fácil deducir las clasificaciones y los parámetros necesarios para definirlos, viendo el programa de ejemplo propuesto en el enunciado y estás serán **Coche**, **Moto** y **Camion**
- En todas la implementaciones invalidaremos el método **object GetCategoria()**, que al devolver diferentes tipos enumerados según la clasificación lo haremos haciendo una sustitución hacia **object**.



```

public class Coche : Vehiculo
{
    public enum Categoria { SinIdentificar, Berlina, Coupe, Sedan, Cabrio, TodoTerreno, MonoVolumen, Crossover };

    private readonly Categoria categoria;

    public override object GetCategoria()
    {
        return categoria;
    }
    public Coche(Matricula matricula, Color color, Marca marca, ushort ocupantes, Categoria categoria)
        : base(matricula, color, marca, ocupantes)
    {
        this.categoria = categoria;
    }
}

```

```

class Moto : Vehiculo
{
    public enum Categoria { SinIdentificar, Scooter, Motocross, Naked, Trail, Supermotard }

    private readonly Categoria categoria;

    public override object GetCategoria()
    {
        return categoria;
    }
    public Moto(Matricula matricula, Color color, Marca marca, ushort ocupantes, Categoria categoria)
        : base(matricula, color, marca, ocupantes)
    {
        this.categoria = categoria;
    }
}

```

Además en la especificación de camión, añadiremos los atributos adicionales que puede identificar nuestra IA según las especificaciones.

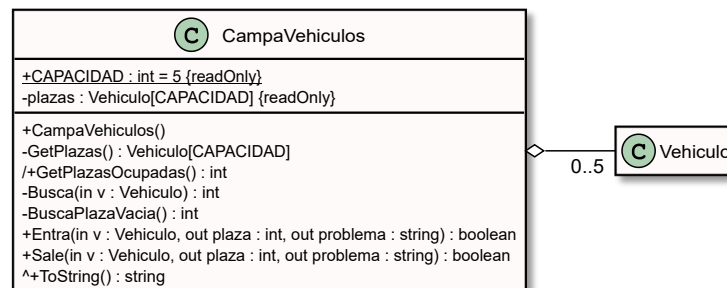
```
class Camion : Vehiculo
{
    public enum Categoria { SinIdentificar, Articulado, Frigorífico, Cisterna, Trailer };

    private readonly ushort ejes;
    private readonly ushort cargaMaximaKg;
    private readonly Categoria categoria;

    public Camion(Matricula matricula, Color color, Marca marca, ushort ocupantes, ushort ejes, ushort cargaMaximaKg, Categoria cate
        : base(matricula, color, marca, ocupantes)
    {
        this.ejes = ejes;
        this.cargaMaximaKg = cargaMaximaKg;
        this.categoria = categoria;
    }
    public ushort GetEjes()
    {
        return ejes;
    }
    public ushort GetCargaMaximaKg()
    {
        return cargaMaximaKg;
    }
    public override object GetCategoria()
    {
        return categoria;
    }

    // En este caso el ToString sí deberemos invalidarlo, pues deberá mostrar información adicional.
    public override string ToString()
    {
        return $"{base.ToString()}\n"
            + $"Número de ejes: {GetEjes()}\n"
            + $"Carga Máxima: {GetCargaMaximaKg()} Kg";
    }
}
```

2. Por último vamos a modelar una clase para nuestra campa de vehículos. Una posible modelización para las especificaciones del enunciado y donde se ha seguido la signatura propuesta para las operaciones del objeto campa en el programa de ejemplo, podría ser la siguiente...



Veamos la implementación comentada

```
class CampaVehiculos
{
    // campo de clase y de solo lectura con un valor literal en C# se define con un const
    public const int CAPACIDAD = 5;
    // El array aunque sea del tipo Vehiculo, realmente va a representar las 5 plazas
    // disponibles en la campa. Si hay un null significará que la plaza está vacía y
    // si hay un objeto Vehiculo la plaza estará ocupada.
    private readonly Vehiculo[] plazas;

    public CampaVehiculos()
    {
        plazas = new Vehiculo[CAPACIDAD];
    }
    private Vehiculo[] GetPlazas()
    {
        return plazas;
    }
}
```



```

// Es un accesor calculado que me indica el número de plazas que contienen un vehículo.
// Nota: Por eso lo hemos marcado con la / delante del nombre del método en el diagrama UML.
public int GetPlazasOcupadas()
{
    int ocupadas = 0;
    foreach (Vehiculo v in GetPlazas())
        if (v != null) ocupadas++;
    return ocupadas;
}
// Implementación de la especificación del Busca
private int Busca(Vehiculo v)
{
    for (int i = 0; i < GetPlazas().Length; i++)
        if (v.Equals(GetPlazas()[i]))
            return i;
    return -1;
}
// Implementación de la especificación del BuscaPlazaVacía
private int BuscaPlazaVacía()
{
    for (int i = 0; i < GetPlazas().Length; i++)
        if (GetPlazas()[i] == null)
            return i;
    return -1;
}

// Operación de entrada de un vehículo en un objeto campa.
// Devolverá un booleano indicándome si el vehículo ha podido entrar o no
// y dos valores subordinados al mismo:
// - Si retorna true en plaza nos indicará la plaza de 1 a 5 donde a entrado.
// - Si retorna false en problema nos indicará el problema por el que no ha podido entrar.
public bool Entra(Vehiculo v, out int plaza, out string problema)
{
    // Compruebo que haya espacio en la campa.
    bool entra = GetPlazasOcupadas() < CAPACIDAD;
    plaza = -1;
    problema = null;

    if (entra)
    {
        // Si hay espacio compruebo que el vehículo no esté ya registrado.
        int i = Busca(v);
        entra = i < 0;
        if (entra)
        {
            // Si no está registrado busco la primera plaza vacía para aparcarlo
            plaza = BuscaPlazaVacía() + 1;
            GetPlazas()[plaza - 1] = v;
        }
        else
            problema = $"Ya se encuentra en el aparcamiento el vehículo {v.GetMatricula()}";
    }
    else
        problema = "Aparcamiento lleno";

    return entra;
}

```

```

// Operación análoga y contraria a la de Entra.
public bool Sale(Vehiculo v, out int plaza, out string problema)
{
    // Compruebo que al menos haya un vehículo aparcado.
    bool sale = GetPlazasOcupadas() > 0;
    problema = null;
    plaza = -1;
    if (sale)
    {
        // Compruebo que el vehículo que deseamos que salga estaba registrado como aparcado.
        int i = Busca(v);
        sale = i >= 0;
        if (sale)
        {
            // Dejo su plaza vacía asignándole null.
            plaza = i + 1;
            GetPlazas()[i] = null;
        }
        else
            problema = $"No se registró la entrada del vehículo {v.GetMatricula()}";
    }
    else
        problema = "Aparcamiento vacío";

    return sale;
}

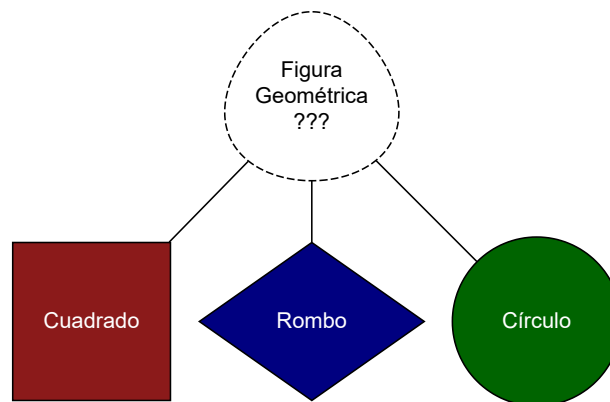
public override string ToString()
{
    StringBuilder texto = new StringBuilder("\nVehículos en el aparcamiento...\n\n");
    for (int i = 0; i < GetPlazas().Length; i++)
    {
        Vehiculo v = GetPlazas()[i];
        // Es importante tener en cuenta que en el array puede haber plazas sin vehículo (a null)
        texto.Append($"Plaza {i + 1}:\n{(v != null ? GetPlazas()[i].ToString() : "Vacía")}\n\n");
    }
    return texto.ToString();
}
}

```

Abstracción

Concepto de clase abstracta

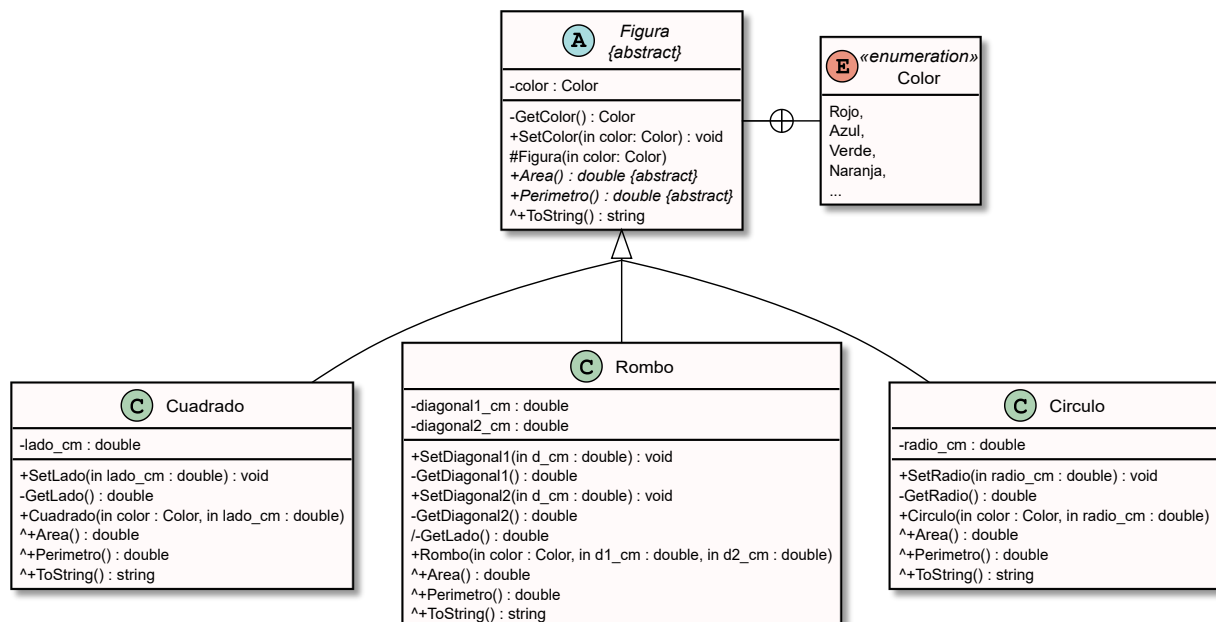
- En la mayoría de los casos, al buscar el polimorfismo con la herencia se nos darán **superclases que no tienen sentido como objetos**.
- A este tipo de clases se les denominará **Clases Abstractas** y de las mismas **no podremos definir objetos**, y sí objetos para sus subclases.



👉 En la POO tradicional diremos que:

- Cómo hemos comentado, no podremos **instanciar objetos** de una clase abstracta, pues no tiene sentido la existencia de dicha abstracción sin una especificación.
- Deberían, pero no es necesario, tener **al menos un campo** (atributo) común a todas las subclases.
- Deberían, pero no es necesario, tener **al menos un método abstracto o virtual puro**.
 - Un **método abstracto** o virtual puro no definirá o tendrá un '*cuerpo de método*' y por tanto dejaremos su implementación en manos de las subclases o especificaciones.
 - Este método abstracto deberá ser redefinido **obligatoriamente** en la subclases.
- A las clases abstractas con **todos sus métodos abstractos** se les denomina '**clases abstractas puras**'.

Un posible modelo de clases para representar la jerarquía de figuras representada en el diagrama anterior podría ser el siguiente:



Nota: Fíjate que en el diagrama UML hemos indicado que la clase es abstracta, además de usando el modificador **{abstract}**, hemos puesto en *cursiva* el nombre y en los métodos virtuales puros o abstractos además de la cursiva hemos puesto el modificador **{abstract}**. Aunque, la cursiva ya no es necesaria a partir de la versión 2.5 de UML aún sigue siendo ampliamente usada.

Clases asbtractas en C#

La sintáxis será muy similar a la de la herencia, pero ampliando las condiciones que hemos descrito con anterioridad.

Vamos a ver la sintaxis a través de la implementación de nuestro ejemplo con figuras.

1. En primer lugar vamos a implementar la clase abstracta **Figura**.

- **(línea 1)**: Definimos la clase anteponiendo el modificador **abstract**
- **(línea 4)**: Podemos añadir como con en la herencia un campo común a todas las especificaciones que en este caso es el **color**.
- **(líneas 19 y 20)**: Aquellos métodos que no podemos implementar hasta que no sepamos la especificación, les antepone el modificador **abstract** y los dejaremos sin implementar.
- **(línea 14)**: Los constructores que definamos pueden ser **protected** pues solo tiene sentido usarlos desde las subclases. Nunca podré hacer un **Figura f = new Figura(Figura.Rojo);**
- **(línea 24)**: Los datos disponibles en **Figura**, serán responsabilidad de la clase, para así no repetir código en las especificaciones y mantener al máximo la encapsulación (Ej. **GetColor()** es privado porque es **Figura** quien se encarga de componer el dato).

👉 **Importante:** Pero, ¿Cómo puede ser que desde aquí se llama a **Area()** o **Perimetro()** si no están implementados? Son métodos de **Figura** y por tanto accesibles desde ella. Además, estoy seguro de que cuando se instancie un objeto, será una especificación y por tanto se producirá un enlace dinámico a un método de la especificación donde estaremos obligados a implementarlo (invalidar).

```
1  abstract class Figura
{
    public enum Color { Rojo, Azul, Verde, Naranja }
4  private Color color;

    private Color GetColor() { return color; }
    public void SetColor(in Color color) { this.color = color; }
8  protected Figura(in Color color)
    {
        SetColor(color);
    }
12  abstract public double Area();
13  abstract public double Perimetro();
    public override string ToString()
    {
16     return $"Color: {GetColor()}\nArea: {Area():F2} cm2\nPerimetro: {Perimetro()} cm";
    }
}
```

2. Implementación de las especificaciones (subclases)

- **(líneas 11 y 15)**: Estaremos **obligados** a invalidar los métodos **abstractos** o virtuales puros.
- **(línea 21)**: Le pedimos a figura la información común aunque en realidad para ello llame a los métodos que he implementado aquí.

```
class Circulo : Figura
{
    private double radio_cm;

    public void SetRadio(double radio_cm) { this.radio_cm = radio_cm; }
    private double GetRadio() { return radio_cm; }
    public Circulo(in Color color, double radio_cm) : base(color)
    {
        SetRadio(radio_cm);
    }
11  public override double Area()
    {
        return Math.PI * Math.Pow(GetRadio(), 2);
    }
15  public override double Perimetro()
    {
        return Math.PI * GetRadio() * 2;
    }
    public override string ToString()
    {
21     return $"Circulo\nRadio: {GetRadio()} cm\n{base.ToString()}";
    }
}
```

El resto de implementaciones serán análogas a **Circulo**

```
class Cuadrado : Figura
{
    private double lado_cm;

    public void SetLado(double lado_cm) { this.lado_cm = lado_cm; }
    private double GetLado() { return lado_cm; }
    public Cuadrado(in Color color, double lado_cm) : base(color)
    {
        SetLado(lado_cm);
    }
    public override double Area()
    {
        return GetLado() * GetLado();
    }
    public override double Perimetro()
    {
        return GetLado() * 4d;
    }
    public override string ToString()
    {
        return $"Cuadrado\nLado: {GetLado()} cm\n{base.ToString()}";
    }
}
```

```
class Rombo : Figura
{
    private double diagonal1_cm;
    private double diagonal2_cm;

    public void SetDiagonal1(double d_cm) { diagonal1_cm = d_cm; }
    private double GetDiagonal1() { return diagonal1_cm; }
    public void SetDiagonal2(double d_cm) { diagonal2_cm = d_cm; }
    private double GetDiagonal2() { return diagonal2_cm; }
    private double GetLado() // Accesor Calculado
    {
        return Math.Sqrt(Math.Pow(GetDiagonal1()/2d, 2d) + Math.Pow(GetDiagonal2()/2d, 2d));
    }
    public Rombo(in Color color, double d1_cm, double d2_cm) : base(color)
    {
        SetDiagonal1(d1_cm);
        SetDiagonal2(d2_cm);
    }
    public override double Area()
    {
        return GetDiagonal1() * GetDiagonal2() / 2d;
    }
    public override double Perimetro()
    {
        return GetLado() * 4d;
    }
    public override string ToString()
    {
        return $"Rombo\nDiagonal1: {GetDiagonal1()} cm\nDiagonal2: {GetDiagonal2()} cm\nLado: {GetLado()} cm\n{base.ToString()}";
    }
}
```

3. El polimorfismo de datos lo vamos a poder realizar igual que en el caso de **Articulo** pero no vamos a poder crear ningún artículo directamente.

```
static void Main()
{
    Figura[] figuras = new Figura[]
    {
        new Cuadrado(Figura.Color.Rojo, 2),
        new Rombo(Figura.Color.Azul, 2, 2),
        new Circulo(Figura.Color.Verde, 2)
    };
    foreach (Figura f in figuras)
        Console.WriteLine(f);
}
```

Caso de estudio

Supongamos el siguiente modelo simplificado, en el cual, un **Sicólogo** crea un gabinete o **consulta** psicológica, donde cada día atiende a los **pacientes** que le llegan.

Los pacientes **irán entrado a la consulta** y en un momento dado el sicólogo les **pasará consulta** por orden de llegada.

Los pacientes tendrán un **nombre** y **de momento** nuestro sicólogo solo sabe atender a dos tipos de pacientes:

1. Pacientes **alegres**
2. Pacientes **tristes**

En el momento en que el sicólogo **atiende a un paciente** se producirá un diálogo con el mismo que empezará igual para todos los pacientes:

```
- Sicólogo: Buenos días!. ¿Cómo se llama?  
- Paciente: Soy <Nombre>  
- Sicólogo: Dígame <Nombre>!. ¿Qué siente?
```

Pero dependiendo del tipo de paciente obtendremos un tipo de **respuesta** diferente...

```
Para el alegre - Paciente: Pues... ahora estoy alegre.  
Para el triste - Paciente: Pues... ahora estoy triste.
```

El sicólogo realizará un **diagnóstico** diferente dependiendo del tipo de paciente.

```
Para el alegre - Sicólogo: <Nombre> le veo estupendamente. Enhorabuena!! no necesita más terapia.  
Para el triste - Sicólogo: <Nombre> tome fluoxetina 20mg y vuelva en un mes.
```

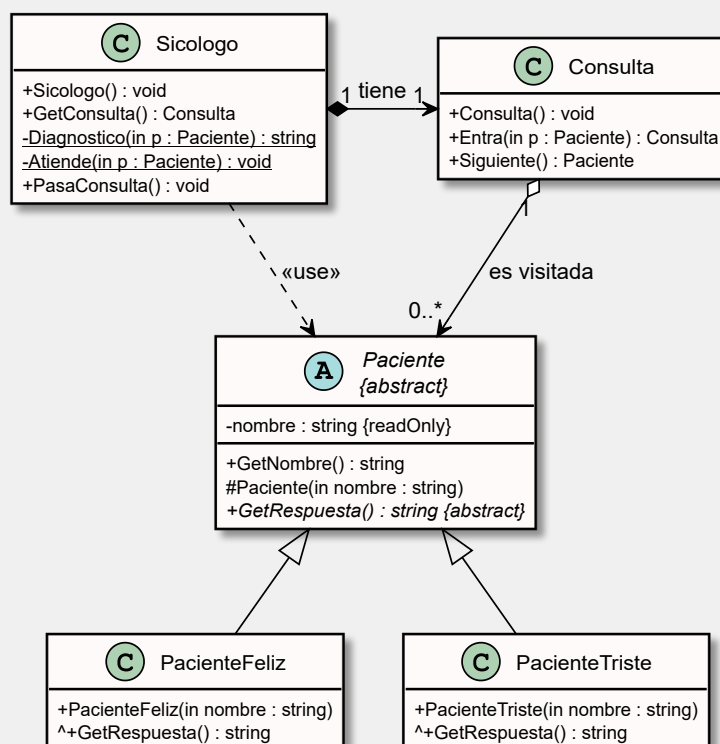
Tras realizar el diagnóstico el sicólogo dirá ...


```
- Sicólogo: Que pase el siguiente !!!
```

Atendiendo a otro paciente si hay aún pendientes en consulta.

 **Piensa en un posible modelado o diseño de clases para representar las especificaciones anteriores.**

Si no se te ocurre ninguno, aquí tienes una propuesta de implementación.



 **Antes de ver la implementación comentada de esta propuesta, intenta realizarla tú y así posteriormente puedes ver la propuesta de solución para lo que no has sabido resolver.**

Propuesta de solución al caso de estudio:

1. Crearemos la clase abstracta **Paciente**. Será abstracta porque los pacientes del sicólogo responden de forma diferente dependiendo de su especificación.

```
abstract class Paciente
{
    private readonly string nombre;

    public string GetNombre()
    {
        return nombre;
    }
    protected Paciente(string nombre)
    {
        this.nombre = nombre;
    }
    public abstract string GetRespuesta();
}
```

2. Definiremos las especificaciones de **Paciente** para que respondan según su estado de ánimo.

```
class PacienteAlegre : Paciente
{
    // Fíjate que los constructores no tienen cuerpo y lo único que hacen es pasar el nombre al constructor protected de Paciente
    public PacienteAlegre(string nombre) : base(nombre) { }
    public override string GetRespuesta()
    {
        return "Pues... ahora estoy alegre.";
    }
}
class PacienteTriste : Paciente
{
    public PacienteTriste(string nombre) : base(nombre) { }
    public override string GetRespuesta()
    {
        return "Pues... ahora estoy triste.";
    }
}
```

3. Definiremos la clase **Consulta** (👉 Lee atentamente los **comentarios**).

```
class Consulta
{
    private Paciente[] pacientes; // Array de pacientes que representa la agregación 0..* con la clase Paciente.
    public Consulta() { pacientes = null; } // En un principio la consulta estará vacía.
    // Patrón de interfaz 'fluido' para que los pacientes entren por orden en la consulta.
    public Consulta Entra(Paciente p)
    {
        // Redimensiono o creo el array y añado al paciente en la última posición.
        Array.Resize(ref pacientes, pacientes == null ? 1 : pacientes.Length + 1);
        pacientes[patientes.Length - 1] = p;
        return this;
    }
    // Retornará el siguiente paciente en la cola de la consulta o null si ya no quedan pacientes.
    // Nota: Más adelante veremos colecciones que nos permitirán gestionar esto de forma más simple.
    public Paciente Siguiente()
    {
        Paciente p = pacientes?[0]; // Asigno null o el paciente en el índice 0 ('primero')
        // Si hay más de 2 pacientes, eliminaré el primer elemento del array dejándolo con un elemento menos.
        if (pacientes != null && pacientes.Length > 1)
        {
            // Creo un array auxiliar copia con un elemento menos que el de pacientes.
            Paciente[] copia = new Paciente[patientes.Length - 1];
            // En este array copio desde el índice 1 del de pacientes en la posición 0 de la copia.
            // Nota: Ver en la documentación de C# esta función.
            Array.Copy(pacientes, 1, copia, 0, pacientes.Length - 1);
            pacientes = copia; // Pacientes pasa a ser ahora la copia donde he quitado el primer elemento.
        }
        else
        {
            pacientes = null;
        }
        return p;
    }
}
```

4. Definiremos la clase **Sicologo** (👉 Lee atentamente los **comentarios**).

Nota: Tiene una dependencia de uso con **Paciente** en el UML, ya que necesita conocer la clase. (En este caso hemos decidido representarla)

```
class Sicologo
{
    private readonly Consulta consulta; // Representa la composición con consulta 'Un Sicologo tiene una Consulta'
    // La consulta la crea el sicólogo y por tanto desaparece con él.
    // Es mejor práctica recibirla como parámetro y hacer una copia como indicamos, pero lo hemos hecho así por simplificar.
    public Sicologo() { consulta = new Consulta(); }
    // La consulta debe ser accesible por los pacientes.
    public Consulta GetConsulta() { return consulta; }
    // A partir de la especificación de paciente, damos un diagnóstico. Si es un paciente no conocido damos una respuesta ambigua.
    // El método lo hemos hecho de clase pues no necesitamos acceder a los datos de la instancia.
    // además nos permitirá a un sicólogo diagnosticar pacientes sin tener consulta si la hicieramos pública.
    private static string Diagnostico(Paciente p)
    {
        return p switch
        {
            PacienteAlegre _ => $"{p.GetNombre()} le veo estupendamente. Enhorabuena!! no necesita más terapia.",
            PacienteTriste _ => $"{p.GetNombre()} tome fluoxetina 20mg y vuelva en un mes.",
            _ => $"{p.GetNombre()} déjeme que estudie un poco más su caso y vuelva la semana que viene."
        };
    }
    // Atender al paciente es quien establece el diálogo.
    private static void Atiende(Paciente p)
    {
        Console.WriteLine("- Sicólogo: Buenos días!. ¿Cómo se llama?");
        Console.WriteLine($"- Paciente: Soy {p.GetNombre()}");
        Console.WriteLine($"- Sicólogo: Dígame {p.GetNombre()}!.. ¿Qué siente?");
        Console.WriteLine($"- Paciente: {p.GetRespuesta()}");
        StringBuilder accion = new StringBuilder("- Sicólogo: ");
        accion.Append(Diagnostico(p))
            .Append($"\\n- Sicólogo: Que pase el siguiente !!!\\n\\n");
        Console.WriteLine(accion);
    }
    // PasaConsulta atiende a los pacientes mientras exista alguno en la cola de la consulta.
    public void PasaConsulta()
    {
        Paciente p;
        while((p = consulta.Siguiente()) != null)
            Atiende(p);
    }
}
```

5. Vamos a crear un sencillo programa de test

```
Sicologo sicologo = new Sicologo();
sicologo.GetConsulta().Entra(new PacienteAlegre("Xusa")).Entra(new PacienteAlegre("Juanjo")).Entra(new PacienteTriste("Carmen"));
sicologo.PasaConsulta();
```

Cómo sucedía en otros casos podemos crear una nueva especificación de **Paciente** por ejemplo...

```
class PacienteSociopata : Paciente
{
    public PacienteSociopata(string nombre) : base(nombre) { }
    public override string GetRespuesta() { return "Vas a morir .. muuhaahahahaha !!"; }
}
```

El **consultorio** seguiría funcionando. Pero, esta vez el **Sicólogo** debería actualizarse para saber tratar a este tipo de pacientes. (Línea 5)

```
private static string Diagnostico(Paciente p)
{
    return p switch {
        ...
        PacienteSociopata _ => $"Lo siento!. Debo aplicarte una descarga de 10000V justo ahora.",
        _ => $"{p.GetNombre()} déjeme que estudie un poco más su caso y vuelva la semana que viene."
    };
}
```

⚡ **Retos:** ¿Se te ocurre cómo crear diferentes tipos de sicólogos que diagnostiquen de forma diferente o alguna forma de hacer que sepamos que tenemos que actualizar al sicólogo di hau nuevos pacientes?

Gestión de errores en POO

Programación defensiva

- En este tema vamos a explicar el

https://es.wikipedia.org/wiki/Steve_McConnell

Excepciones

- **Definición:** Es la forma en que los lenguajes orientados a objetos realizan el control de errores.
- **Frente al control de errores en la programación estructurada tradicional nos ofrecen:**
 - **Claridad**, ya que **evitamos lógica adicional** del caso de error.
 - Evitar que un método devuelva un error como parámetro a través de la pila de llamadas como sucedía en la programación estructurada.
 - **Tratamiento asegurado** de errores.

Excepciones en C#

- Todas derivan de la clase `System.Exception`
- Existen algunas ya muchas **predefinidas**.
- Podemos definir **excepciones propias** mediante el mecanismo de herencia.

Ejemplo de generación de excepciones en C#

Utilizaremos la instrucción **throw** → `* throw new <TipoExcepción>;`

Veamos un ejemplo en el que vamos a usar la excepción ya predefinida **ArgumentNullException** que me servirá para detectar aquellos tipos referencia que sean **null** antes de usarlos.

Por ejemplo, en el constructor copia que creamos para la clase **Articulo** podemos comprobar que el objeto **articulo** a copiar no sea **null**

```
class Sicoologo
{
    ...
    private static void Atiende(Paciente p)
    {
        Console.WriteLine("- Sicológico: Buenos días!. ¿Cómo se llama?");
        Console.WriteLine($"- Paciente: Soy {p.GetNombre()}");
        Console.WriteLine($"- Sicológico: Dígame {p.GetNombre()}!.. ¿Qué siente?");
        Console.WriteLine($"- Paciente: {p.GetRespuesta()}");
        StringBuilder accion = new StringBuilder("- Sicológico: ");
        accion.Append(Diagnostico(p))
            .Append($"\\n- Sicológico: Que pase el siguiente !!!\\n\\n");
        Console.WriteLine(accion);
    }
}
```