

Tema 7 parte 2

[Descargar estos apuntes](#)

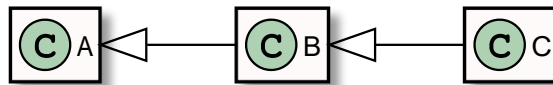
Índice

1. [Índice](#)
2. [Roles entre clases \(Continuación\)](#)
 1. [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#](#)
 2. [Abstracción](#)
 1. [Concepto de clase asbtracta](#)
 2. [Clases asbtractas en C#](#)

Roles entre clases (Continuación)

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 una **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**.



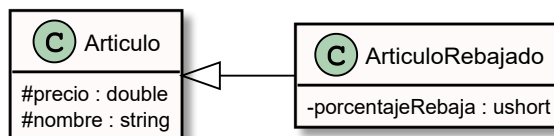
- **B** hereda de **A**
- **B** es una concreción **A**
- **A** es una generalización **B**
- **A** es la superclase y **B** la subclase
- **C** hereda de **B** y **A**
- **B** y **C** son subclases de **A**

Tipos de Herencia

Herencia Simple

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

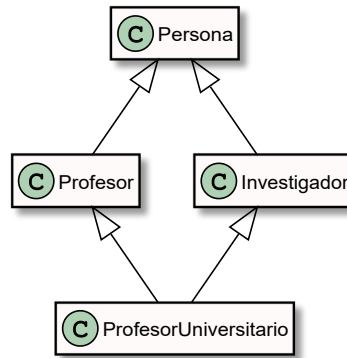
Una subclase de **Articulo** denominada **ArticuloRebajado** que además, añade al articulo un campo con el porcentaje de rebaja, denominado **porcentajeRebaja**.



👁 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**.



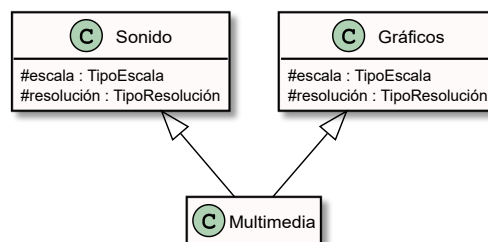
- **Diseños más complejos** y más difíciles de aprender y utilizar por el programador.

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 de **Gráficos** ?



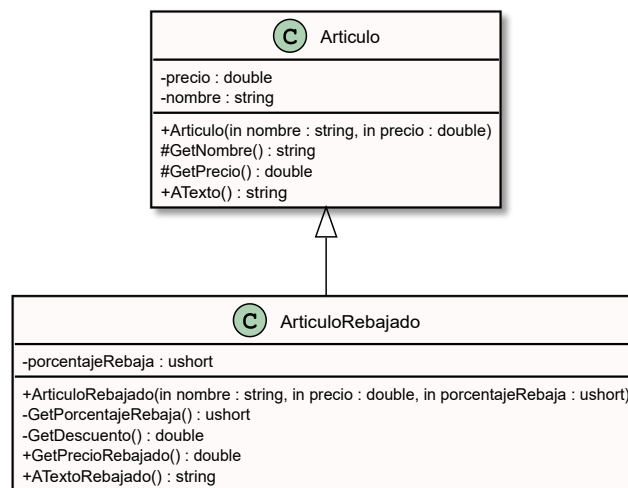
Implementando la herencia en C#

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;
    }
}

```

```

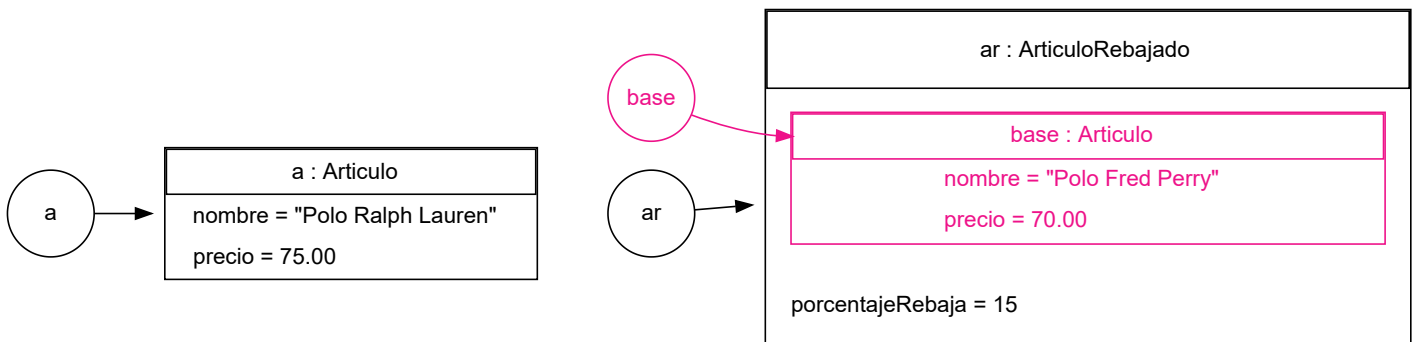
1 class ArtículoRebajado : Artículo
{
    private ushort porcentajeRebaja;

    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 `ArtículoRebajado` 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 `Artículo` el getter de `precio` lo hemos llamado `double GetPrecio()` y en `ArtículoRebajado` le hemos llamado `double GetPrecioRebajado()`. Esto lo hemos hecho para no producir colisiones de firmas (*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 `ArtículoRebajado 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 firmas 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:

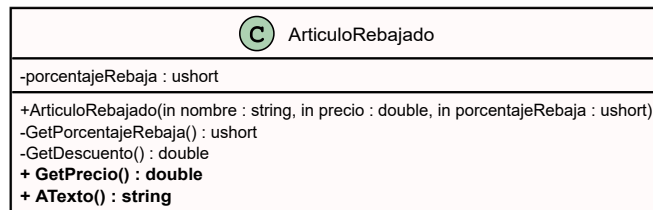
- **Reemplazo:** Se sustituye completamente la implementación del método heredado manteniendo la firma. Comúnmente se le conoce como **Ocultación** (*hiding*)
- **Refinamiento:** Se añade nueva funcionalidad al comportamiento heredado. Comúnmente se le conoce como **Invalidación** (*overriding*)

Ocultación o reemplazo en C#

- Será la **estrategia** que aplica **por defecto 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**.



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).

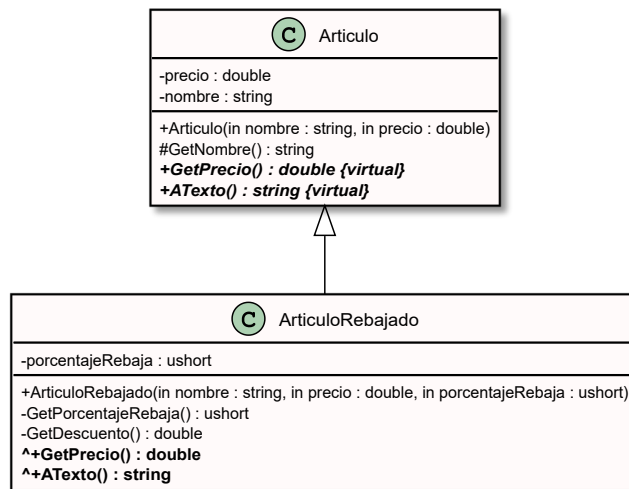
```
class ArticuloRebajado : Articulo
{
    // ... código omitido para abreviar.
    private double GetDescuento()
    {
        return base.GetPrecio() * GetPorcentajeRebaja() / 100d;
    }
    8 public new double GetPrecio() // Añado el modificador new para confirmar que quiero hacer una ocultación.
    {
        // 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 de 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 **Artículo** y los hago public como va a ser en las subclases.

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

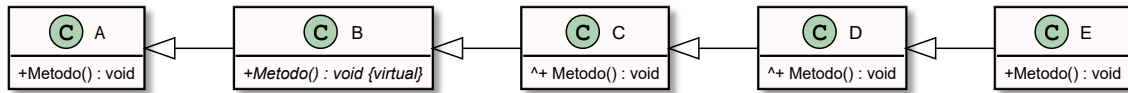
2. Invalido el método con la misma signatura en la subclase con el modificador **override**

💡 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.**



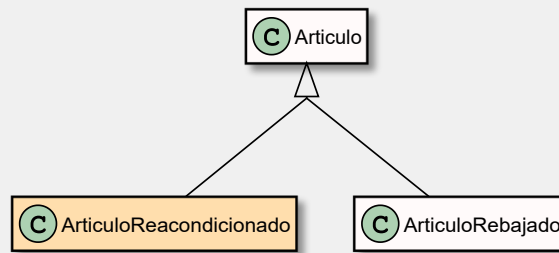
```

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

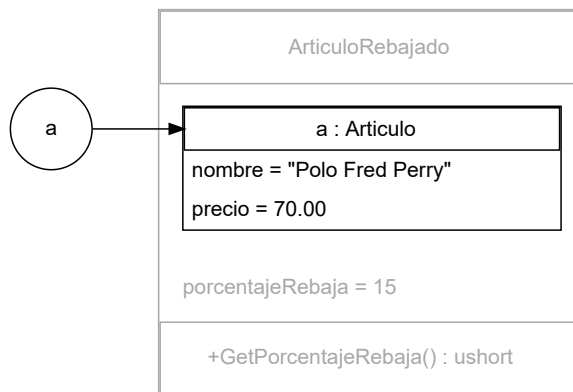
- Es la capacidad de un identificador de hacer referencia a instancias de distintas clases durante su ejecución.
- Se logra a través del **principio de sustitución**.

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

- Podemos decir que es, cuando un identificador que hemos declarado del tipo de 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 tuvieramos una accesos 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 ha 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**.

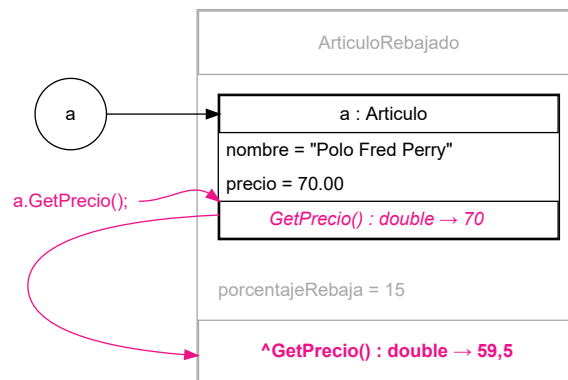
Una de las principales 'ventajas' de la **invalidación**, es que al hacer una **sustitución** como la que hemos visto del tipo...

```
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

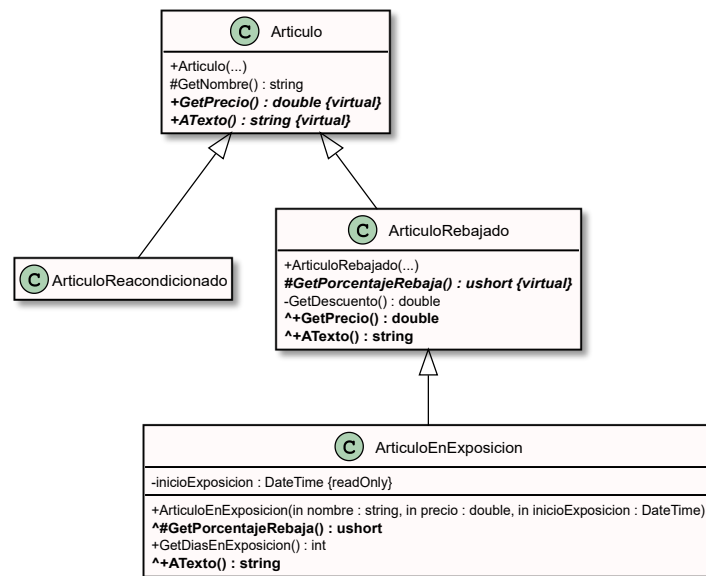
Veamos un ejemplo más elaborado a través de otro ejemplo en el que vamos a ampliar nuestra jerarquía de artículos.

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 ArticuloRebajado : Articulo
{
    // ... <Código omitido para abreviar>

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


    // En un principio el descuento es 0 y lo calculeramos dinámicamente.
    25 public ArticuloEnExposicion(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.
    30 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 Articulo y ArticuloRebajado 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 ArticuloEnExposicion con fecha de inicio hace 10 días.
// Hacemos una sustitución hacia el tipo más genérico Articulo
Articulo a = new ArticuloEnExposicion("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** ➡' ...

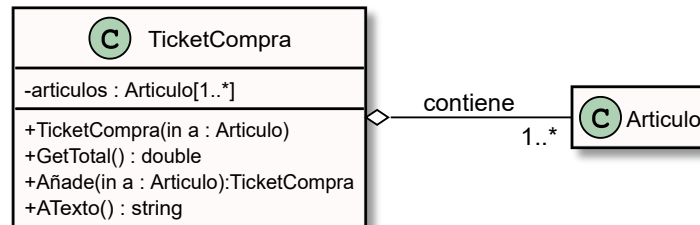
1. `a.ATexto()` ➡ `^ArticuloRebajado.ATexto()` ➡ `^ArticuloEnExposicion.ATexto()`
2. Al ejecutar `^ArticuloEnExposicion.ATexto()` se llamará a `base.ATexto()` el cual llamará a `^ArticuloRebajado.GetPrecio()` este al no estar invalidado llama a `ArticuloRebajado.GetDescuento()` que a su vez llama a `ArticuloRebajado.GetPorcentajeRebaja()`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

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...

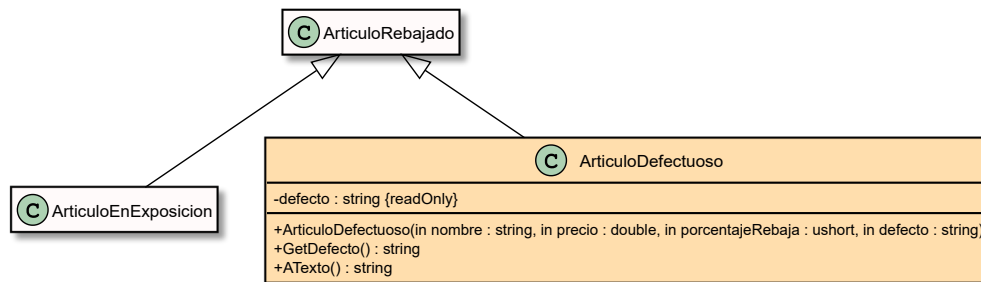
```
static void Main()
{
    TicketCompra t = new TicketCompra(new Articulo("Airpods Apple", 100d));
    t.Añade(new ArticuloEnExposicion("TV Samsung OLED 50'", 999d, DateTime.Now.AddDays(-10)))
    .Añade(new ArticuloRebajado("Honor Band 4", 35d, 15))
    .Añade(new ArticuloReacondicionado("Samsung 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()**

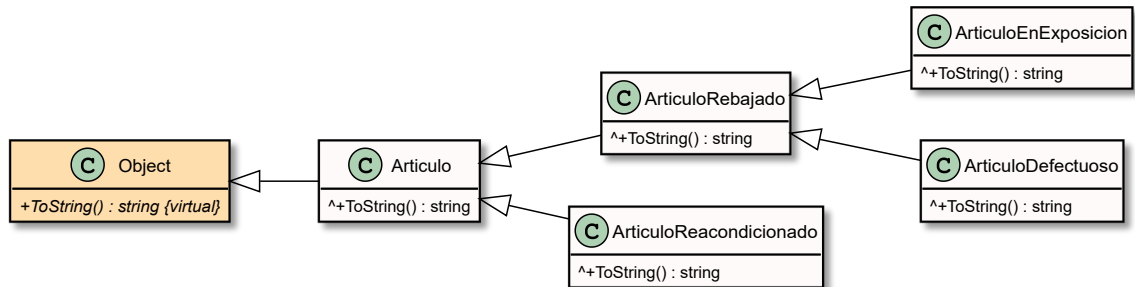
Este método, es llamado **automáticamente** cada vez que un objeto se intenta formatear cómo cadena y equivaldría en cierta manera al método **string ATexto()** que hemos definido en nuestra jerarquía de artículos.

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 +
            $"Fecha 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.

 Fíjate 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 'esconde' 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)
    {
        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.Camion));
        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.Camion));
        Sale(camp, new Camion(new Matricula("8798 JWR"), Vehiculo.Color.Blanco, Vehiculo.Marca.DAF, 1, 2, 6000, Camion.Categoria.Camion));
        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.Camion));

        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

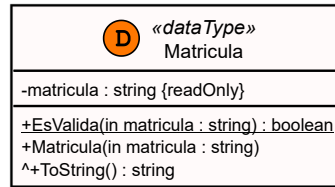
Plaza 3:
Matricula: 7643 LRF
Color: Rojo
Marca: BMW
Ocupantes: 4
Categoria: TodoTerreno

Plaza 4:
Matricula: 0000 DGP
Color: Negro
Marca: DESCONOCIDA
Ocupantes: 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 **inmutable**. Así pues, por repasar conceptos, vamos a definirlo como un tipo valor **struct**.

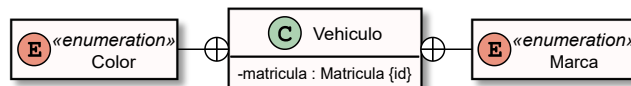


```
public struct Matricula
{
    private readonly string matricula;

    // 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 tañ manera que ara hacer referencia a ellos desde fuera de la clase usaremos la notación **Vehiculo.Color** y **Vahiculo.Marca** de esta manera no tendremos redundancia en los nombre y no habrá conflicto de nombres con otros tipos **Color**.

Esta relación de **anidación** el un diagrama de clases UML la podrmos expresar de la siguiente manera:



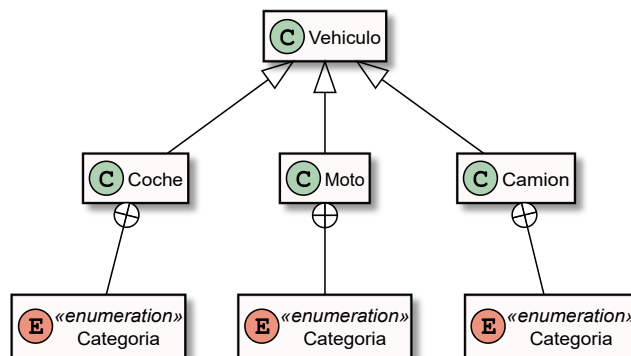
```
public class Vehiculo
{
    // Definimos los tipos enumerados de la especificación
    public enum Color { Blanco, Morado, Cian, Azul, Rojo, Verde, Negro, Naranja, Gris };
    public enum Marca { DESCONOCIDA, BMW, SEAT, AUDI, RENAULT, MAN, DAF, CITROEN, TOYOTA, SUZUKI, YAMAHA, MERCEDES, PEGASO };

    // 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;
}
// Los objetos vehiculos 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"
        + $"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 valor de hash de un vehiculo será el hash de la cadena de su matricula.
    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 GetCategoría()**, que al devolver diferentes tipos enumerados según la clasificación lo haremos haciendo una sustitución hacia **object**.



```

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

    private readonly Categoría categoria;

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

```

```

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

    private readonly Categoría categoria;

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

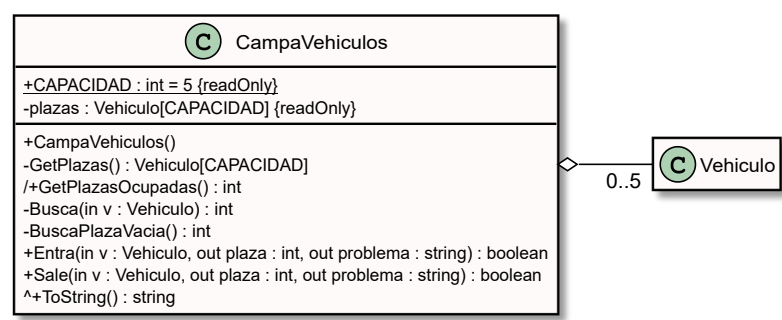
    public ushort GetEjes()
    {
        return ejes;
    }

    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 campaña de vehículos. Una posible modelización para las especificaciones del enunciado y donde se ha seguido la signatura propuesta para las operaciones del objeto campaña 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;
}

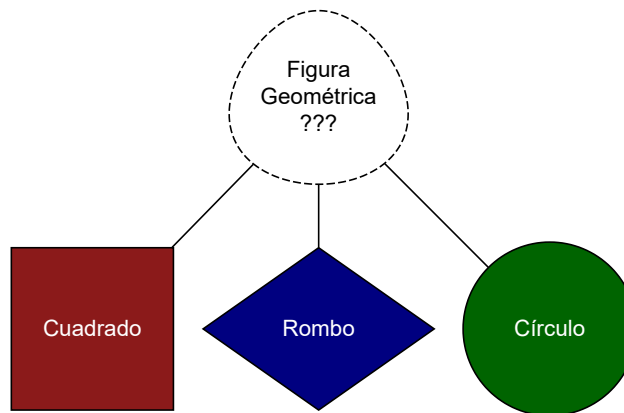
public override string ToString()
{
    StringBuilder texto = new StringBuilder("\nVehículos en el aparcamiento...\n\n");
    for (int i = 0; i < GetPlazas().Length; i++)
    {
        Vehiculo v = GetPlazas()[i];
        // Es importante tener en cuenta que en el array puede haber plazas sin vehículo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

```

Abstracción

Concepto de clase asbtracta

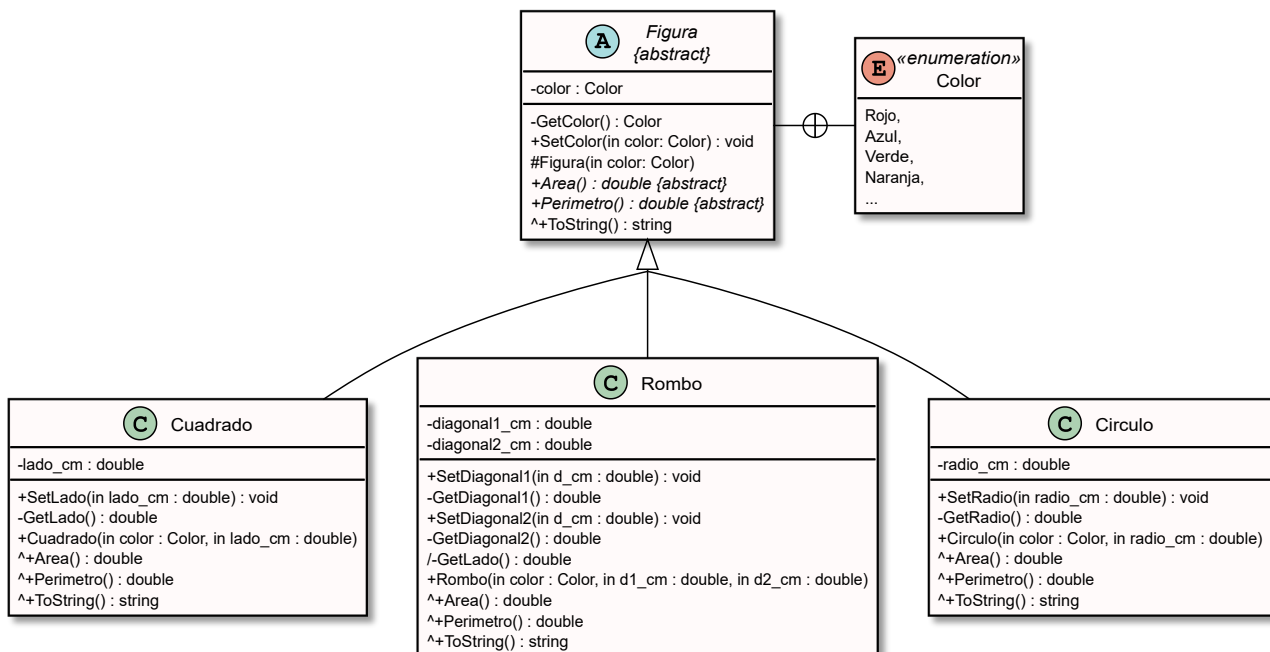
- En la mayoría de los casos, al buscar el polimorfismo con la herencia se nos darán **superclases que no tienen sentido como objetos**.
- A este tipo de clases se les denominará **Clases Abstractas** y de las mismas **no podremos definir objetos**, y sí objetos para sus subclases.



👉 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 ámpliamente 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 anteponemos 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 poque 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 instáncie 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\nPerímetro: {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 relidad 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);
    }
16 public override double Perimetro()
    {
        return Math.PI * GetRadio() * 2;
    }
    public override string ToString()
    {
21 return $"Círculo\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 entrando a la consulta** y en un momento dado el sicologo 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 diagnostico el sicólogo dira ...

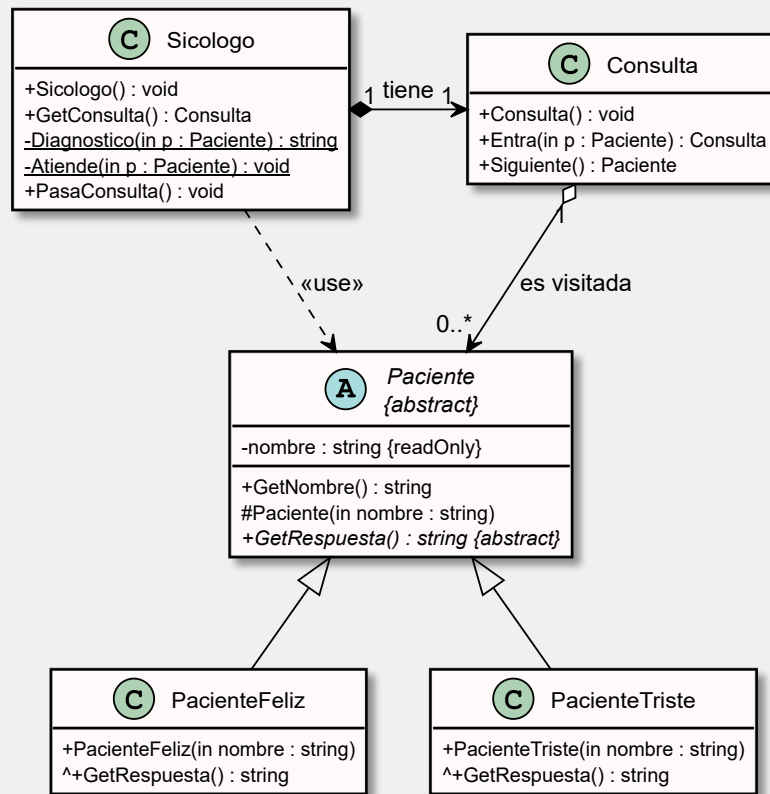
```
- Sicólogo: Que pase el siguiente !!!
```

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 prupuesta, 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 psicólogo 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 Pac
    4 public PacienteAlegre(string nombre) : base(nombre) { }
    public override string GetRespuesta()
    {
        return "Pues... ahora estoy alegre.";
    }
}
class PacienteTriste : Paciente
{
    12 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[pacientes.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 .. muuhaaahahahaha !!"; }
}
```

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 psicólogos que diagnostiquen de forma diferente o alguna forma de hacer que sepamos que tenemos que actualizar al psicólogo si hay nuevos pacientes?