Unidad 15

Descargar estos apunte en pdf o html

Índice

Herencia

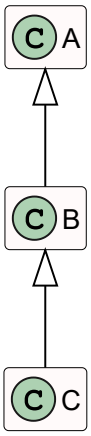
Es una de las características principales de la POO. Formalmente podemos definirla como....

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:

1. La **subclase** es una **concreción** de la superclase que representará una **generalización**.
2. Representará el tipo de relación '**Es un/a**'. Ej. '*Un coche **es un** vehículo.*'
3. La **herencia** nos servirá para reutilizar código y por tanto no repetir funcionalidades.

En UML representaremos el rol a través de una flecha de punta hueca de la subclase a la superclase y se podrán producir relaciones **transitivas**.



Interpretaciones del diagrama:

- **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**

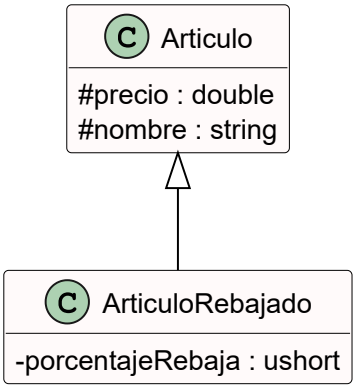
Tipos de Herencia

Herencia Simple

Se dará cuando la subclase hereda de **una sola** superclase y será la única que nosotros utilizaremos en C#.

Por **ejemplo**, supongamos que tenemos la superclase `Articulo` con un `id`, `precio` y `nombre`. Una subclase de `Articulo` denominada `ArticuloRebajado` que además,

añade al articulo un campo con el porcentaje de rebaja, denominado `porcentajeRebaja`. Claramente la relación se entiende como herencia pues: "***Un*** `ArticuloRebajado` ***es un*** `Articulo`".



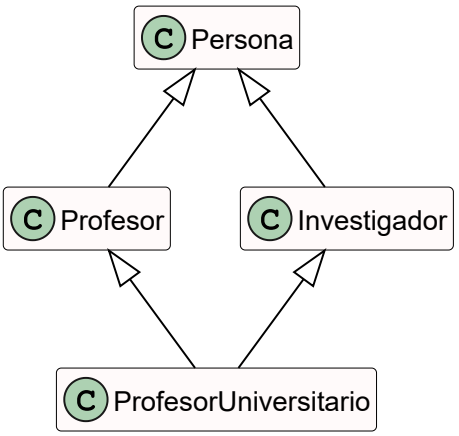
La instancia de un objeto en memoria de `ArticuloRebajado` además de tener una propiedad `porcentajeRebaja`, tendrá las propiedades `precio` y `nombre` por ser también un `Articulo` y todos ellas definirán su estado.

Herencia Múltiple

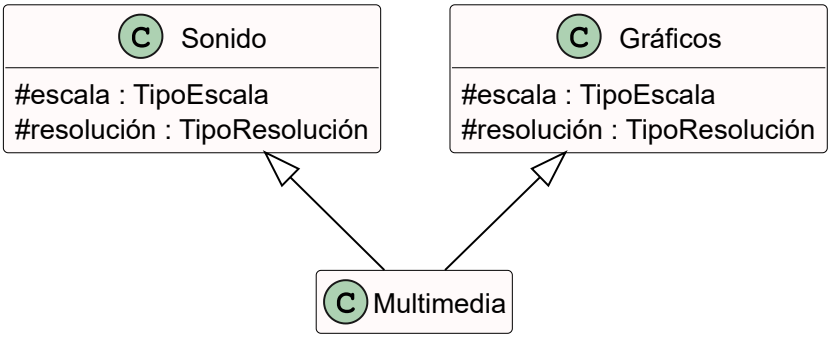
Se dará cuando una subclase **hereda características de varias superclases**. **Tiene más desventajas que ventajas. Por eso C#, Kotlin Java NO la permiten.** Aunque otros lenguajes como Python o C++ sí.

Entre las **desventajas** que hace que C# no la permita podemos destacar:

- **Menor velocidad** de ejecución.
- **Herencia repetida (Transitividad)**.
  Por ejemplo en el diagram `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. Además, siempre podremos **rediseñar** utilizando herencia simple o incluso composición.
- **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`?

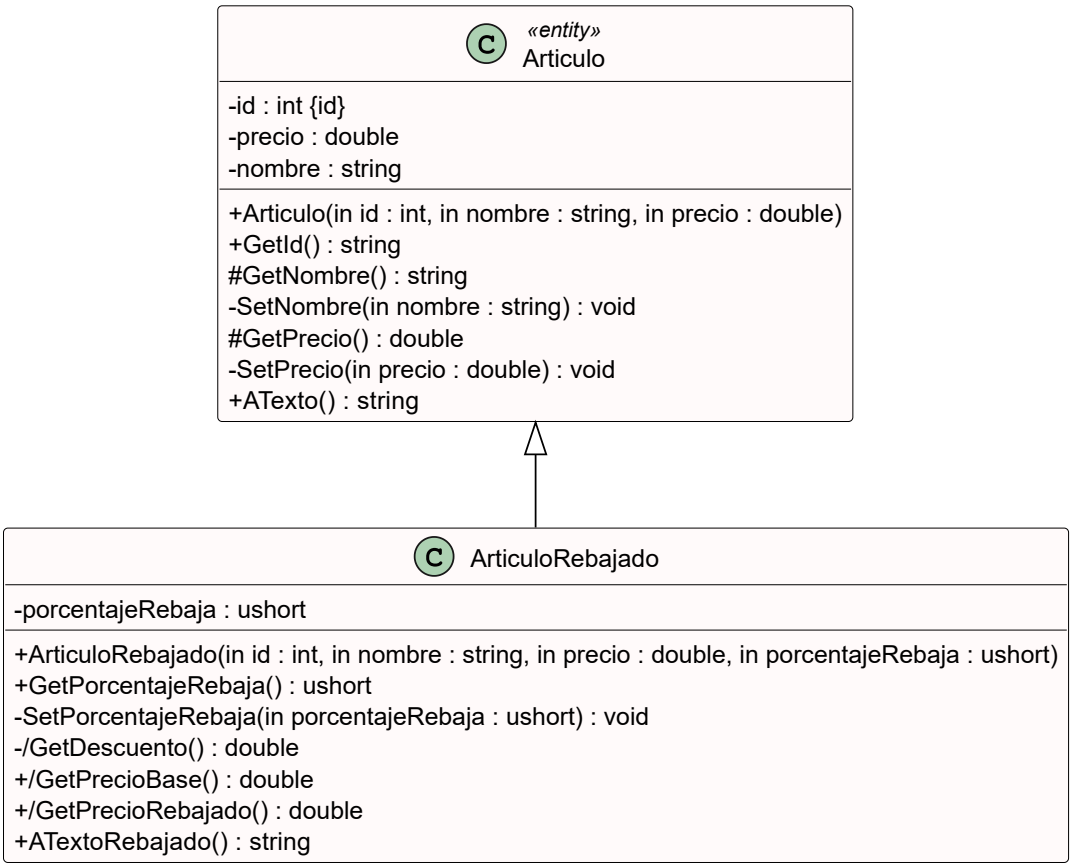


Implementando la herencia en CSharp

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**. Equivaldrá a indicar el modificador de acceso `protected` y significará que el **campo**, **propiedad** o el **método** al que modifica, **no puede ser accedido desde fuera de la clase** como en el caso de `private`, **pero sí desde las subclases de la misma**.

En nuestro diagrama se lo estaremos aplicando al `get` de la propiedades `Nombre` y `Precio`. De momento hemos sido restrictivo en el acceso a las subclases y si vemos que necesitamos acceder a un campo o propiedad desde fuera, lo cambiaremos a `public`.

Veamos cómo quedará la implementación de la **superclase** `Articulo`:

```
public class Articulo
{
    // Propiedade de sólo lectura por ser un Id y público el get por +GetId(): string
    public string Id { get; }
    // Propiedades privadas de modificación y protegidas para acceso
    // y así solo se pueda acceder desde la clase y sus subclases
    protected double Precio { get; private set; }

    public Articulo(
                string id,
                string nombre,
                double precio)
    {
        Id = id;
        Nombre = nombre;
        Precio = precio;
    }

    public string ATexto() => $"""
                Id: {Id}
                Nombre: {Nombre}
                Precio: {Precio:F2}€
                """;
}
```

Veamos cómo quedará la implementación de la **subclase** `ArticuloRebajado`:

```
public class ArticuloRebajado : Articulo
{
    public ushort PorcentajeRebaja { get; private set; }
    private double Descuento => base.Precio * PorcentajeRebaja / 100d;
    public double PrecioRebajado => base.Precio - Descuento;
    public double PrecioBase => base.Precio;

    public ArticuloRebajado(
                        string id,
                        string nombre,
                        double precio,
                        ushort porcentajeRebaja)
                        // Llamada al constructor de la clase base encargado
                        // de 'construir' la parte de Articulo del objeto
                        : base(id, nombre, precio)
    {
        PorcentajeRebaja = porcentajeRebaja;
    }
    public string ATextoRebajado() => $"""
                        Id: {base.Id}
                        Nombre: {base.Nombre}
                        Rebaja: {PorcentajeRebaja}%
                        Antes: {base.Precio:F2}€
                        Ahora: {PrecioRebajado:F2}€
                        """;
}
```

Podremos acceder desde la **subclase** a las propiedades públicas y protegidas de la **clase base o superclase** como `Id`, `Precio` y `Nombre`. Fíjate que hemos usado la palabra clave `base` para referirnos a la parte de `Articulo` del objeto `ArticuloRebajado` y así poder acceder a sus propiedades protegidas y públicas. Aunque en este caso, no sería en este caso necesario usarla pues no hay ambigüedad, **es una buena práctica para evitar errores** en futuras modificaciones.

Vamos a implementar un simple **programa principal de prueba**...

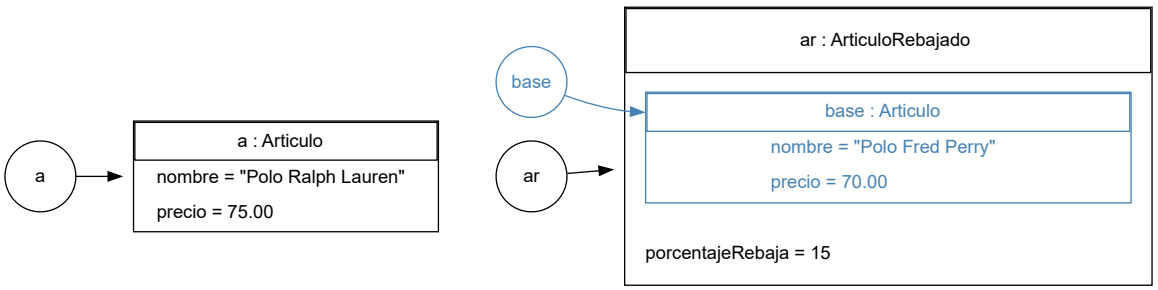
```
static class EjemploHerencia
{
    static void Main()
    {
        Articulo a = new (
            id: "A001",
            nombre: "Polo Ralph Lauren",
            precio: 75d);

        Console.WriteLine(new string('-', 20));
        Console.WriteLine(a.ATexto());

        ArticuloRebajado ar = new (
            id: "A002",
            nombre: "Polo Fred Perry",
            precio: 70d,
            porcentajeRebaja: 15);

        Console.WriteLine(new string('-', 20));
        Console.WriteLine(ar.ATexto());
        Console.WriteLine();
        Console.WriteLine(ar.ATextoRebajado());
        Console.WriteLine(new string('-', 20));
    }
}
```

Mostrará por consola:

```
--------------------
Id: A001
Nombre: Polo Ralph Lauren
Precio: 75,00?
--------------------
Id: A002
Nombre: Polo Fred Perry
Precio: 70,00?

Id: A002
Nombre: Polo Fred Perry
Rebaja: 15%
Antes: 70,00?
Ahora: 59,50?
--------------------
```

Puedes descargar el código de ejemplo de el siguiente enlace: **herencia_articulo1.cs**

Una aproximación a cómo sería el objeto instanciado de una subclase en memoria, sería la siguiente:



Recuerda que, podemos considerar la parte de la instancia de `ArticuloRebajado` resaltada, es una instancia de `Articulo` que es donde se guardan las propiedades `Id`, `Nombre` y `Precio` y la parte de objeto que corresponde a un `ArticuloRebajado` guarda solo la propiedad `PorcentajeRebaja` más las otras propiedades calculadas.

Palabra reservada `base`

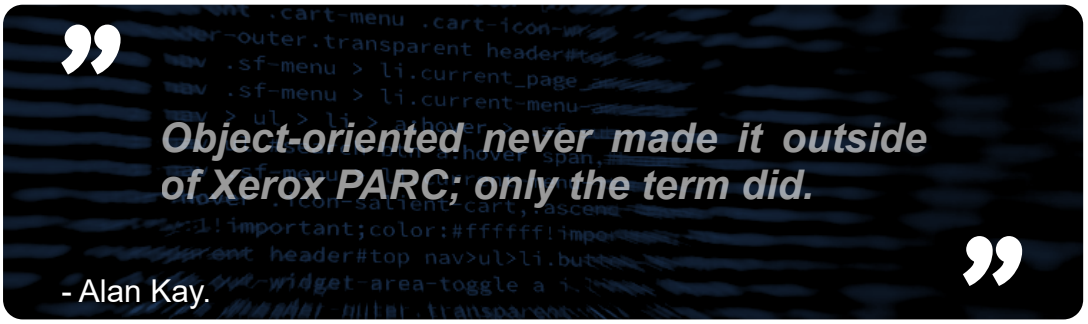
Vamos a ahondar un poco más en la palabra reservada `base` que hemos utilizado en el constructor de la subclase `ArticuloRebajado` y para acceder a las propiedades de la superclase `Articulo`.

Si nos fijamos en el constructor de `ArticuloRebajado` solo se encarga de **inicializar y crear las propiedades específicos de la subclase**, para crear los de la clase, hemos invocado a un constructor de la clase base, 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 seria 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 como se apreciaba en el diagrama anterior.

> **Nota**
>
> Ya veremos más adelante que será imprescindible su uso 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` hemos llamado a la propiedad `Precio` y en `ArticuloRebajado` hemos llamado a la propiedad `PrecioRebajado`.Esto lo hemos hecho para que no se solaparan los identificadores de las propiedades (*recuerda que la subclase puede ver las propiedades de la superclase*) y podríamos tener problemas de ambigüedad.

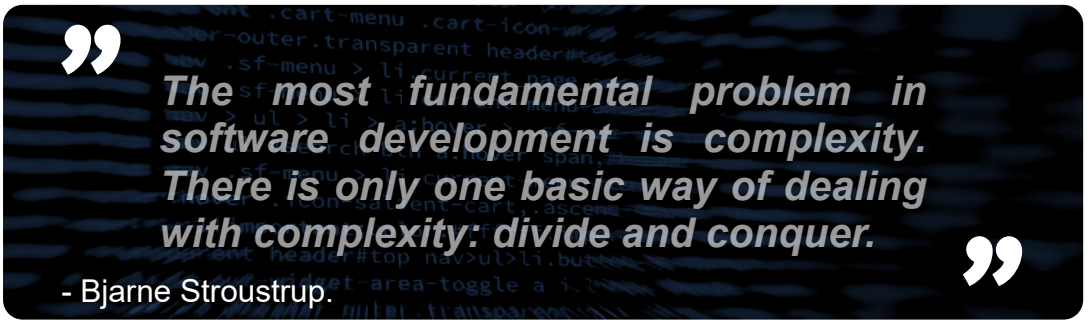
Sin embargo, en el fondo es una **redundancia**, porque si a un objeto `ArticuloRebajado articuloRebajado;` accedo a la propiedad `articuloRebajado.Precio;` ya se que estoy obteniendo el precio rebajado o debería estar obteniendo dicho precio.

Por tanto, si usamos nombres repetidos en ambas clases, **tendríamos propiedades o métodos con identificadores idénticos** 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:

1. **Reemplazo**: Se sustituye completamente la implementación del método o propiedad heredada manteniendo la signatura o tipo.
   Comúnmente se le conoce como **Ocultación** (*hiding*)
2. **Refinamiento**: Se añade nueva funcionalidad al comportamiento heredado. **Es la más común y también se le conoce como Invalidación (*overriding*)**



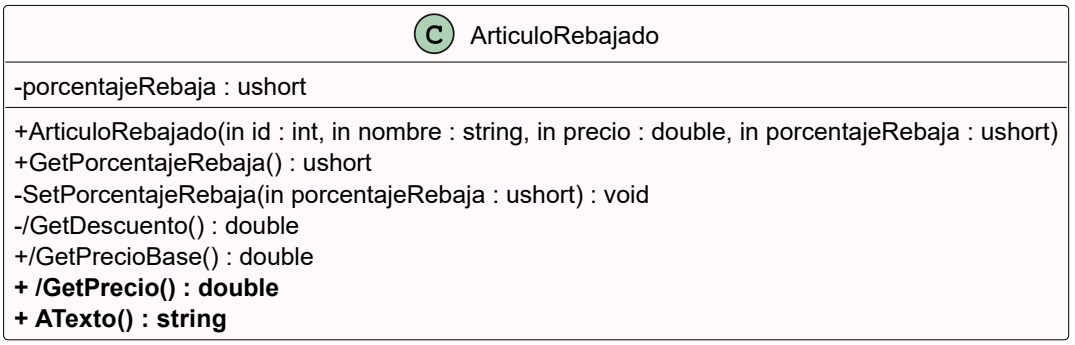
Ocultación o reemplazo en CSharp

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.Precio' oculta el miembro heredado 'Articulo.Precio'.
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** ✋
>
> Dentro del ámbito o alcance de definición de `ArticuloRebajado`, `base.Precio` me devuelve el precio del original del artículo (sin descuento) y `this.Precio` o simplemente `Precio` me devuelve el precio rebajado (con descuento).

La nueva implementación de `ArticuloRebajado` quedaría como sigue:

```
public class ArticuloRebajado : Articulo
{
    public ushort PorcentajeRebaja { get; private set; }
    private double Descuento => base.Precio * PorcentajeRebaja / 100d;
    // Añado el modificador new para confirmar que quiero hacer una ocultación.
    public new double Precio => base.Precio - Descuento;
    public double PrecioBase => base.Precio;

    public ArticuloRebajado(
                        string id,
                        string nombre,
                        double precio,
                        ushort porcentajeRebaja)
                        : base(id, nombre, precio)
    {
        PorcentajeRebaja = porcentajeRebaja;
    }
    // Añado el modificador new para confirmar que quiero hacer una ocultación.
    public new string ATexto() => $"""
                        Id: {base.Id}
                        Nombre: {base.Nombre}
                        Rebaja: {PorcentajeRebaja}%
                        Antes: {base.Precio:F2}€
                        Ahora: {this.Precio:F2}€
                        """;
}
```

> **Cuidado**
>
> Si en lugar de ...
>
> ```
> public new double Precio => base.Precio - Descuento;
> ```
>
> no usáramos la palabra `base` por error ...
>
> ```
> public new double Precio => Precio - Descuento;
> ```
>
> Estaríamos llamando a al `get` de `Precio` para calcular `Precio` y por tanto **entraríamos en un bucle infinito** que provocaría un **desbordamiento de pila** (*stack overflow*).

Invalidación o refinamiento en CSharp

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.

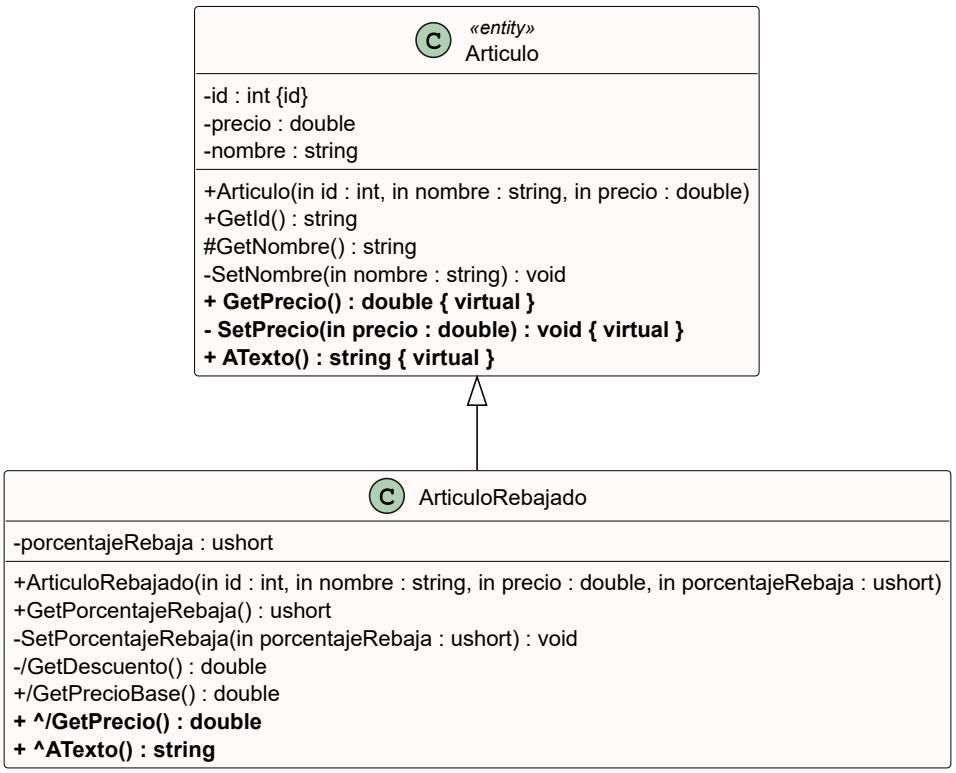
En ella, normalmente haremos lo que hacía la clase padre, más nueva funcionalidad y para ello, *'**marcaremos**'* como métodos *'**virtuales**'* a los métodos **'*invalidables*' en la superclase** que sean **públicos** o **protegidos**. Para ello utilizaremos la **palabra reservada** `virtual` **precediendo a la declaración del método invalidable** y la **palabra** `override` **precediendo la declaración de un método en la subclase que invalida a uno invalidable** o virtual en la superclase.

> ✋ **Importante**: A diferencia de la ocultación, ambos **métodos deberán tener la misma accesibilidad**.

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

Pondremos el modificador `{virtual}` al final 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 carácter `^` 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...

Modifico los métodos cómo invalidables con `virtual` en `Articulo` y los hago **public** el `get` como va a ser en las subclases. Recordemos que deben tener la misma accesibilidad.

```
class Articulo
{
    // ... código omitido para abreviar.
    public virtual double Precio { get; private set; }

    public virtual string ATexto() => $"""
                Id: {Id}
                Nombre: {Nombre}
                Precio: {Precio:F2}€
                """;
}
```

Invalido el método con la misma signatura en la subclase con el modificador `override`

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

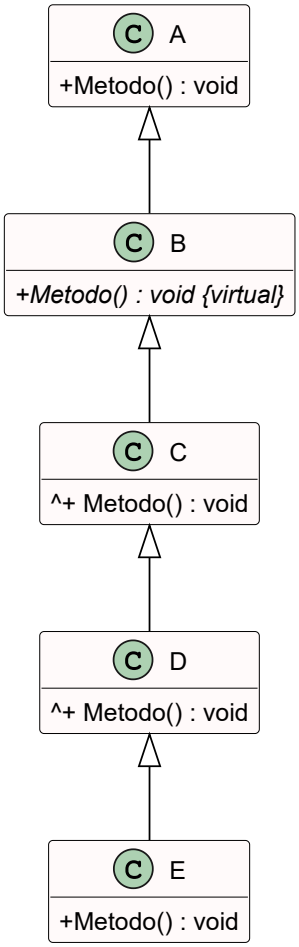
```
class ArticuloRebajado : Articulo
{
    // ... código omitido para abreviar.

    // Añado el modificador override para indicar que estoy invalidando la propiedad Precio.
    public override double Precio => base.Precio - Descuento;

    // Añado el modificador override para indicar que estoy invalidando la el método ATexto().
    public override string ATexto() => $"""
                        Id: {base.Id}
                        Nombre: {base.Nombre}
                        Rebaja: {PorcentajeRebaja}%
                        Antes: {base.Precio:F2}€
                        Ahora: {this.Precio:F2}€
                        """;
}
```

Si no has sabido seguir las modificaciones propuestas, puedes descargar el código del ejemplo anterior del siguiente enlace: **herencia_articulo_invalidacion.cs**

Combinando Invalidación y Ocultación

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
{
    // Corto la secuencia de invalidaciones ocultando el método.
    public new void Metodo()
    {
        ...
    }
}
```

🚀 Ampliación opcional:

¿Serías capaz de reconoces los elementos y equivalencias de de nuestra relación de herencia entre `Articulo` y `ArticuloRebajado` en C# en otros lenguajes cómo **JavaScript** o **Kotlin**?

JavaScript:

```
class Articulo {
    #nombre;
    #precio;

    constructor(id, nombre, precio) {
        this.id = id;
        this.#nombre = nombre;
        this.#precio = precio;
    }

    get precio() {
        return this.#precio;
    }
    get nombre() {
        return this.#nombre;
    }
    aTexto() {
        return `
        Id: ${this.id}
        Nombre: ${this.#nombre}
        Precio: ${this.precio.toFixed(2)}
        `.trim();
    }
}
```

```
class ArticuloRebajado extends Articulo {
    constructor(id, nombre, precio, porcentajeRebaja) {
        super(id, nombre, precio);
        this.porcentajeRebaja = porcentajeRebaja;
        this.precioBase = super.precio;
    }
    get #descuento() {
        return this.precioBase
               * this.porcentajeRebaja
               / 100.0;
    }
    get precio() {
        return this.precioBase - this.#descuento;
    }
    aTexto() {
        return `
        Id: ${this.id}
        Nombre: ${this.nombre}
        Rebaja: ${this.porcentajeRebaja}%
        Antes: ${this.precioBase.toFixed(2)}€
        Ahora: ${this.precio.toFixed(2)}€
        `.trim();
    }
}
```

Fíjate que en **JavaScript** usa la palabra clave `extends` para indicar que una clase hereda de otra y que los métodos y propiedades de la superclase se pueden llamar con `super` en lugar de `base` como en C#. Esto viene del lenguaje **Java**. Además no disponemos de las palabras reservadas `virtual` y `override` por lo que las características orientadas a objetos son más limitadas.

Kotlin:

```kotlin
open class Articulo(
    val id: String,
    protected var nombre: String,
    open var precio: Double
) {
    open fun aTexto(): String {
        return """
            Id: $id
            Nombre: $nombre
            Precio: ${"%.2f".format(preci
        """.trimIndent()
    }
}
```

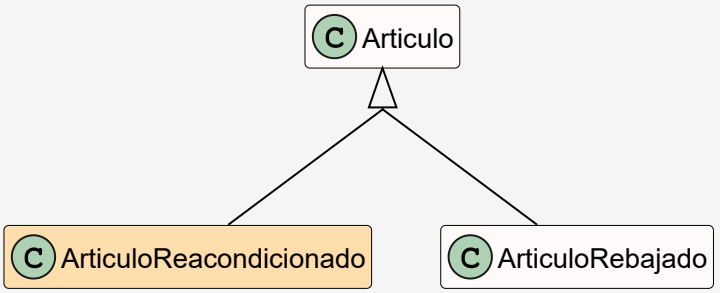
```kotlin
class ArticuloRebajado(
    id: String,
    nombre: String,
    precioBase: Double,
    var porcentajeRebaja: UShort
) : Articulo(id, nombre, precioBase) {

    private val descuento: Double
        get() = super.precio
                * porcentajeRebaja.toInt()
                / 100.0
    override var precio: Double
        get() = super.precio - descuento
        set(value) {
            super.precio = value
        }
    val precioBase: Double
        get() = super.precio

    override fun aTexto(): String {
        return """
            Id: $id
            Nombre: $nombre
            Rebaja: $porcentajeRebaja%
            Antes: ${"%.2f".format(precioBase)}€
            Ahora: ${"%.2f".format(precio)}€
        """.trimIndent()
    }
}
```

En Kolin, la forma de indicar la herencia es con la palabra clave `:` como en C# y para indicar que un método es **invalidable** usamos la palabra clave `open`, sin embargo, para indicar que un método **invalidado** en la subclase, usamos la palabra clave `override` como en C#. Además, para acceder a los métodos y propiedades de la superclase usamos la palabra clave `super` como en JavaScript.

Ejemplo:

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.



✋ **Alto**: Antes de ver la la propuesta de implementación, intenta pensar cómo sería la misma...

```
public class ArticuloReacondicionado : Articulo
{
    public DateOnly FechaReacondicionamiento { get; }
    public string Empresa { get; }
    public string Descripcion { get; }

    public ArticuloReacondicionado(
                        string id,
                        string nombre,
                        double precio,
                        DateOnly fechaReacondicionamiento,
                        string empresa,
                        string descripcion) : base(id, nombre, precio)
    {
        FechaReacondicionamiento = fechaReacondicionamiento;
        Empresa = empresa;
        Descripcion = descripcion;
    }
    public override string ATexto() => $"""
                        {base.ATexto()}
                        Fecha reacondicionamiento: {FechaReacondicionamiento.ToShortDateString()}
                        Empresa: {Empresa}
                        Descripción: {Descripcion}
                        """;
}
```

```
static void Main()
{
    ArticuloReacondicionado ac = new(
        id: "A003-R",
        nombre: "Fuente TFX",
        precio: 50d,
        fechaReacondicionamiento: DateOnly.FromDateTime(DateTime.Now),
        empresa: "Balmis S.A",
        descripcion: "Se cambia condensador electrolítico");

    Console.WriteLine(new string('-', 20));
    Console.WriteLine(ac.ATexto());
    Console.WriteLine(new string('-', 20));
}
```

Mostrará por consola:

```
--------------------
Id: A003-R
Nombre: Fuente TFX
Precio: 50,00?
Fecha reacondicionamiento: 17/08/2025
Empresa: Balmis S.A
Descripción: Se cambia condensador electro
--------------------
```

Fíjate que al invalidar a `ATexto()` hemos llamado al método de la superclase `base.ATexto()` para reutilizar su funcionalidad y así no repetir código. De esta manera podremos ver las características comunes de todos los artículos y las específicas de cada uno de ellos.

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**.

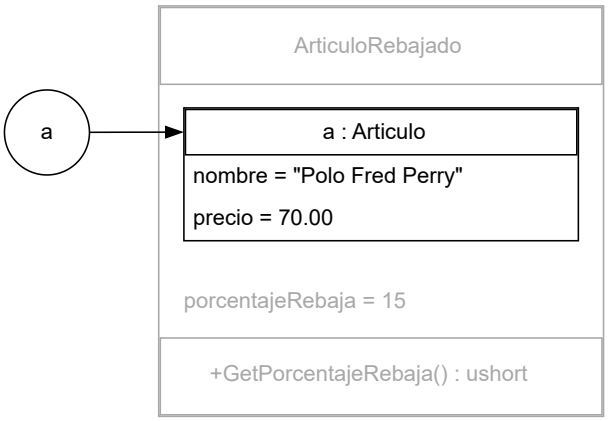
Principio de sustitución de Liskov (Upcasting)

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** y esta **conversión** o *'cast'* se hace de forma **implícita**.

La forma más simple de tenerlo es crear un objeto de la subclase y lo asignamos a la superclase. Por ejemplo...

```
ArticuloRebajado ar = new (
                    id: "A004",
                    nombre: "Polo Fred Perry",
                    precio: 70d,
                    porcentajeRebaja: 15);

Articulo a = ar;

// También podemos hacerlo directamente pero
// indicando el tipo en el new.
Articulo a = new ArticuloRebajado(
                    id: "A004",
                    nombre: "Polo Fred Perry",
                    precio: 70d,
                    porcentajeRebaja: 15);
```

Una posible **representación del objeto en memoria** sería la siguiente:



> **Nota**
>
> 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 propiedad público en `ArticuloRebajado` denominada `ushort PorcentajeRebaja`, no podríamos hacer `a.PorcentajeRebaja`.

Downcasting

Se tratará de la **operación contrária a la sustitución** o upcasting.
⚠️Pero ojo, solo podremos hacerla si realmente la referencia que tenemos es del tipo al que queremos hacer el downcasting, en caso contrário obtendremos un ❌**Error en tiempo de ejecución**.

Siguiendo con la representación en memoria es como si recuperaramos el acceso a la parte de `ArticuloRebajado` del objeto `Articulo` que hemos creado en el ejemplo anterior.

Formas de realizar el Downcasting

1. Mediante **cast explícito**:

   ```
   Articulo a = new ArticuloRebajado("A004", "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 ("A001", "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 ("A001", "Polo Ralph Lauren", 75f);

   if (a is ArticuloRebajado ar) // Preguntamos si admite la forma de ...
   {
       Console.WriteLine(ar);
   }
   ```

   Si se cumple la condición, `ar` se convierte en un objeto de tipo `ArticuloRebajado` y podemos acceder a sus propiedades y métodos. Si no se cumple, `ar` no se inicializa y no podemos usarlo.

3. Mediante el operador **as**

   Realiza directamente el downcasting y si no puede asigna **null**. EL problema es que vamos a tener que manejar tipos nulables.

```
Articulo a = new ("A001", "Polo Ralph Lauren", 75f);

// Generará un Warning porque se puede evaluar a null y ar no es anulable
// deberíamos declarar ArticuloRebajado como nulable
ArticuloRebajado? ar = a as ArticuloRebajado;
Console.WriteLine(ar?.ATexto() ?? "No hay datos de rebaja");
```

4. Apoyándonos en el operador de uso combinado null **??**

   ⚠️ Realmente **no sería un downcasting** sino una tranformación defensiva.

```
Articulo a = new ArticuloReacondicionado(id: "A005-R",
                                         nombre: "iPhone 16 Pro",
                                         precio: 950,
                                         fechaReacondicionamiento: new(2025, 8, 17),
                                         empresa: "Foxconn",
                                         descripcion: "Cambio de batería");
// Nos aseguramos de que en ar siempre hay un objeto instanciado y evitamos el aviso
// al no ser un tipo anulable 👍
ArticuloRebajado ar = a as ArticuloRebajado
                      ??
                      new (id: a.Id,
                          nombre: a.Nombre,
                          precio: a.Precio,
                          porcentajeRebaja: 15);
```

   Realmente **estamos creando un nuevo artículo** porque realmente no es del tipo que esperábamos. Por lo que no sería una opción muy recomendable al tratarse de **programación defensiva** 💀.

5. Usando un `switch` ya sea como instrucción o expresión sería la **forma más elegante y legible** de hacer un downcasting y además nos permite manejar el caso en que no se cumple la condición.

```
Articulo a = new ArticuloRebajado("A004", "Polo Fred Perry", 70d, 15);

string salida = a switch
{
    ArticuloRebajado ar when ar.PorcentajeRebaja > 30 => "Artículo rebajado con más del 30% de rebaja",
    ArticuloRebajado ar => $"Es un artículo rebajado con el {ar.PorcentajeRebaja}% de rebaja",
    ArticuloReacondicionado _ => "Es reacondicionado",
    _ => "No es un tipo de objeto contemplado",
};
Console.WriteLine(salida);
```

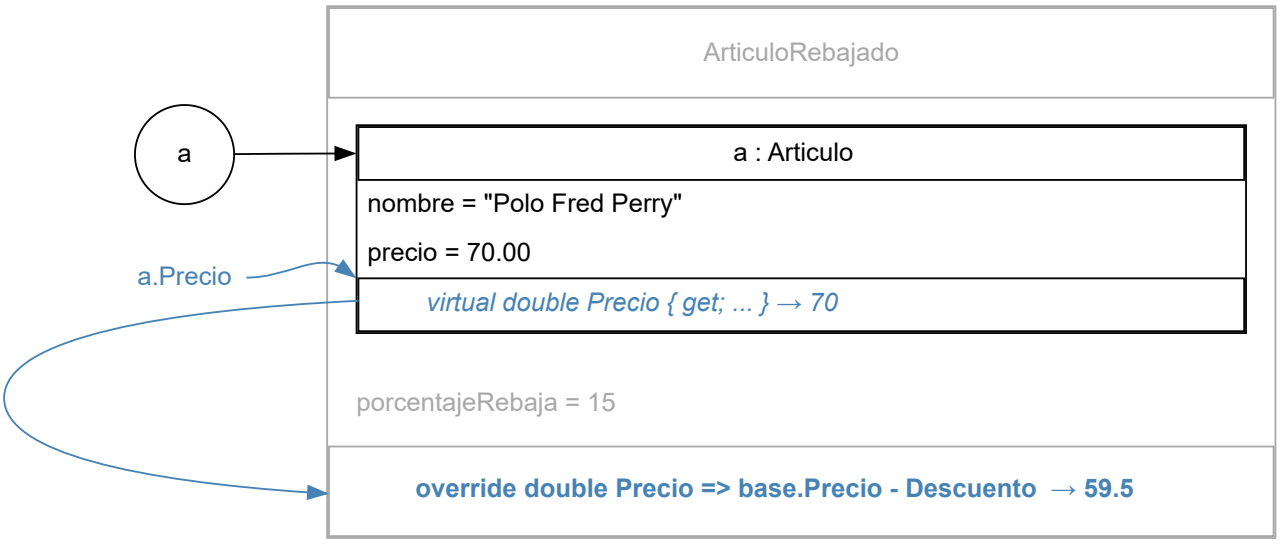
Ligadura Dinámica

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.Precio);
```

`Console.WriteLine(a.GetPrecio());` mostrará **59,5** y no **70**. Pero,... ¿Cómo puede suceder esto si `a` esta 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.Precio` y ver que la propiedad `virtual Precio { get; ... }` 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 o propiedad 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.

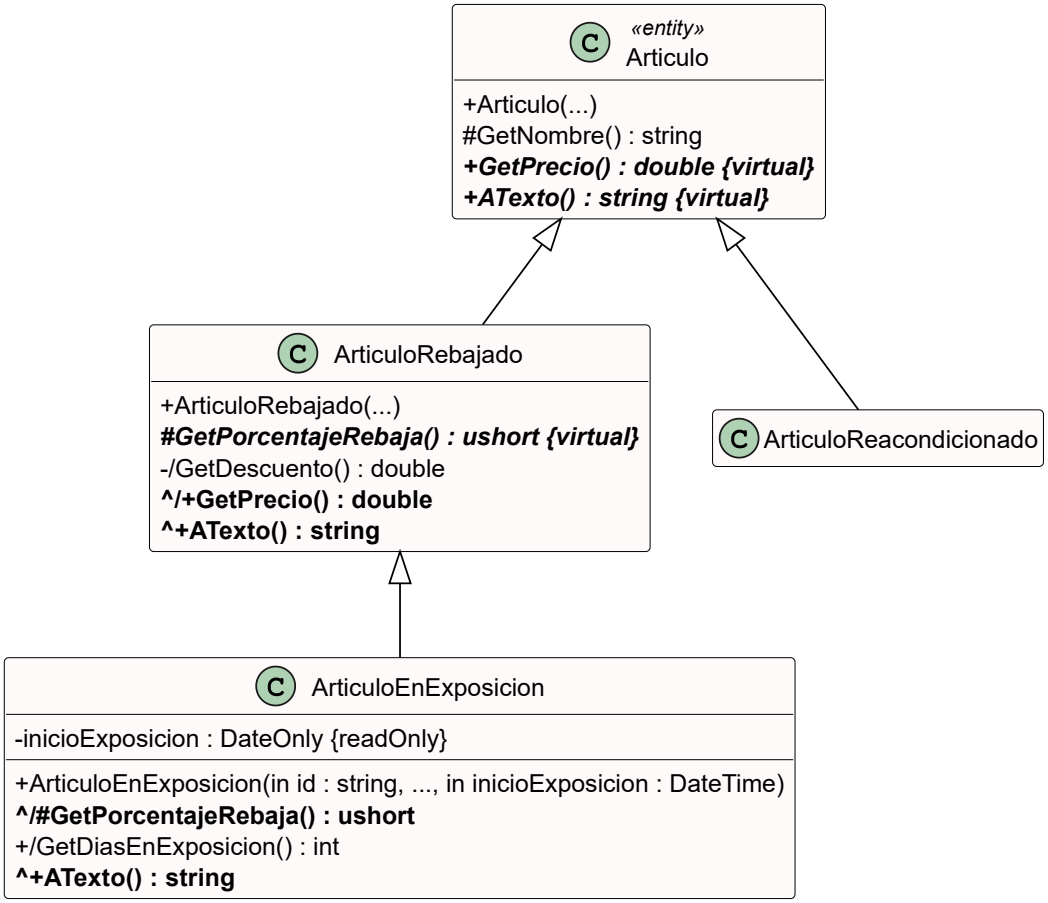
Ejemplo de uso del Enlace Dinámico

Veamos un ejemplo más elaborado a través de otro ejemplo donde 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.

En primer lugar marcando como `virtual` la propiedad `PorcentajeRebaja` de `ArticuloRebajado` para que pueda ser invalidada en la subclase `ArticuloEnExposicion`.

```
public class ArticuloRebajado : Articulo
{
    // ... código omitido para abreviar.
    public virtual ushort PorcentajeRebaja { get; private set; }

    private double Descuento => base.Precio * PorcentajeRebaja / 100d;

    public override double Precio => base.Precio - Descuento;
}
```

En `ArticuloEnExposicion`, vamos a invalidar la propiedad `PorcentajeRebaja` y el método `ATexto()` para que nos muestre la información de la exposición en el caso de que el objeto lo tengamos como un tipo de alguna de las superclases.

```
public class ArticuloEnExposicion : ArticuloRebajado
{
    public DateOnly InicioExposicion { get; }

    // En un principio el descuento es 0 y lo calculeramos dinámicamente.
    public ArticuloEnExposicion(
                        string id,
                        string nombre,
                        double precio,
                        DateOnly inicioExposicion) : base(id, nombre, precio, 0)
    {
        InicioExposicion = inicioExposicion;
    }

    // Invalidamo la obtención porcentaje para calcularlo en función de los días en exposición.
    public override ushort PorcentajeRebaja => Convert.ToUInt16(Math.Clamp(DiasEnExposicion, 1, 75));

    // Los días en esposición se calculan en el momento actual, desde el incio de la exposición.
    public int DiasEnExposicion => DateOnly.FromDateTime(DateTime.Now).DayNumber - InicioExposicion.DayNumber;

    // Invalidamos el método ATexto() de Articulo y ArticuloRebajado para que añada la nueva información.
    public override string ATexto() => $"""
                        {base.ATexto()}
                        En exposición desde: {InicioExposicion.ToShortDateString()} total {DiasEnExposicion} d
                        """;
}
```

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(
    id: "A006-E",
    nombre: "TV Samsung OLED 50''",
    precio: 999d,
    inicioExposicion: DateOnly.FromDateTime(DateTime.Now.AddDays(-10)));

Console.WriteLine(new string('-', 20));
Console.WriteLine(a.ATexto());
Console.WriteLine(new string('-', 20));
```

Mostrará por consola:

```
Id: A006-E
Nombre: TV Samsung OLED 50''
Rebaja: 10%
Antes: 999,00€
Ahora: 899,10€
En exposición desde: <fecha actual - 10 días> total 10 días
```

Si lo meditamos, se están realizando **dos enlaces o ligaduras dinámicas** '`→`' ...

1. `a.ATexto()` → `override ArticuloRebajado.Atexto()` → **`override ArticuloEnExposicion.Atexto()`**
2. Al ejecutar `override ArticuloEnExposicion.Atexto()` se llamará a `base.ATexto()` el cual llamará a `override ArticuloRebajado.Precio` este al no estar invalidado llama a `ArticuloRebajado.Descuento` que a su vez llama a *`virtual ArticuloRebajado.PorcentajeRebaja`* → **`override ArticuloEnExposicion.PorcentajeRebaja`** produciéndose el segundo enlace.

> **Resumen**
>
> En otras palabras, el **`PorcentajeRebaja`** que usamos en la propiedad **`Descuento`** en **`ArticuloRebajado`**, no es el de **`ArticuloRebajado`**, sino el de **`ArticuloEnExposicion`**, que es el que realmente se está ejecutando en ese momento por la ligadura dinámica.

Si has tenido algún problema, puedes descargar el código del ejemplo anterior del siguiente enlace: **herencia_articuloexposicion.cs**

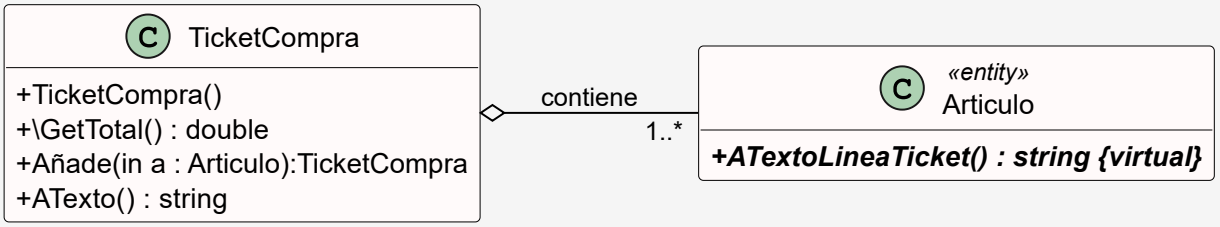
Utilidad del polimorfismo de datos (sustitución) y el enlace dinámico

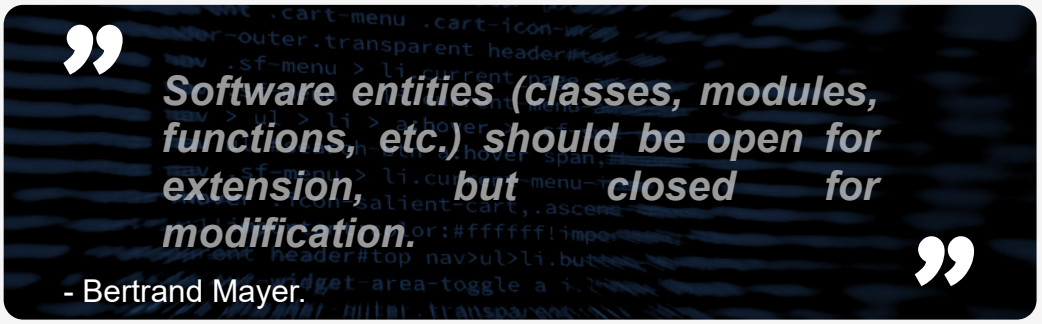
En ocasiones el software cambia y se añaden nuevas especificaciones, como pudieran ser nuevos tipos de artículos en la jerarquía. Con el polimorfismo de datos, podremos adaptarnos a futuros cambios (Nuevas formas de un objeto), **sin realizar cambios** traumáticos y costosos en nuestros objetos ni en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

Ejemplo:

Supongamos que queremos modelar una clase `TicketCompra` que me permita añadir artículos al mismo. Además, vamos a añadir un método para mostrar el ticket y una propiedad calculada que me devuelva el total de la compra. Además, vamos a añanir un método virtual `virtual stting ATextoLineaTicket()` en la clase `Articulo` que nos permita mostrar una línea del ticket con el artículo y la invalidaremos en todas las subclases.

Un posible **diseño simplificado** en UML para expresar esto podría ser ...





Lo primero sería la implementación de `ATextoLineaTicket()` en toda la jerarquía de artículos, para que cada uno de ellos muestre su información de forma adecuada.

```
public class Articulo
{
    // ... código omitido para abreviar.
    public virtual string ATextoLineaTicket() => $"""
                        {Nombre,-20} {Precio,26:F2}€
                        """;
}


public class ArticuloRebajado : Articulo
{
    // ... código omitido para abreviar.
    public override string ATextoLineaTicket() => $"""
                        {Nombre,-20} {PrecioBase,8:F2}€ con {PorcentajeRebaja:D2}% {Precio,8:F2}€
                        """;
}


public class ArticuloReacondicionado : Articulo
{
    // ... código omitido para abreviar.
    // Como artículo pero añadiendo un R
    public override string ATextoLineaTicket() => $"""
                        {base.ATextoLineaTicket()} R ({Descripcion})
                        """;
}


public class  ArticuloEnExposicion : ArticuloRebajado
{
    // ... código omitido para abreviar.
    // Como artículo rfebajado pero añadiendo un E
    public override string ATextoLineaTicket() => $"""
                        {base.ATextoLineaTicket()} R ({Descripcion})
                        """;
}
```

Una propuesta de implementación de la clase `TicketCompra` podría ser:

```
public class TicketCompra
{
    // Puesto que no definimos constructor y dejamos el de por defecto,
    // inicializamos la lista de artículos directamente.
    private List<Articulo> Articulos { get; } = [];

    public double Total
    {
        get
        {
            double total = 0d;
            foreach (Articulo a in Articulos)
                total += a.Precio;
            return total;
        }
    }
    // Definimos un interfaz fluido como los que utiliza la clase StringBuilder.
    public TicketCompra Añade(Articulo a)
    {
        Articulos.Add(a);
        return this;
    }
    public string ATexto()
    {
        StringBuilder ticket = new();
        foreach (Articulo a in Articulos)
            ticket.AppendLine(a.ATextoLineaTicket());
        ticket.AppendLine(new string('-', 70))
              .AppendLine($"Total: {Total,41:F2}€");
        return ticket.ToString();
    }
}
```

Si implementamos el siguiente programa principal de test...

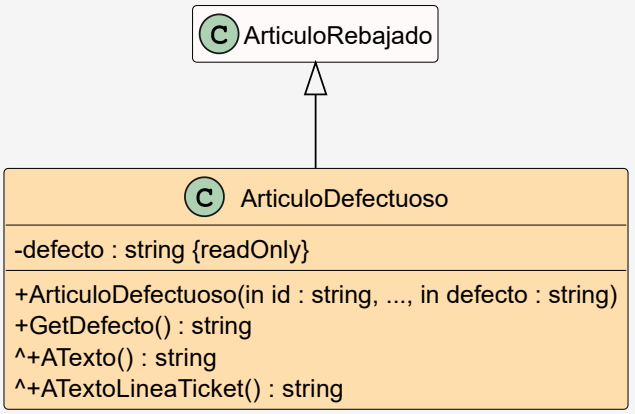
```
public static void Main()
{
    TicketCompra ticket = new();
    ticket.Añade(new Articulo(
                    id: "A001",
                    nombre: "Camiseta",
                    precio: 19.99))
    .Añade(new ArticuloRebajado(
                    id: "A002",
                    nombre: "Pantalón",
                    precio: 39.99,
                    porcentajeRebaja: 20))
    .Añade(new ArticuloReacondicionado(
                    id: "A003",
                    nombre: "Zapatillas",
                    precio: 59.99,
                    fechaReacondicionamiento: new (2023, 1, 15),
                    empresa: "Reacondicionados S.A.",
                    descripcion: "Con caja original"))
    .Añade(new ArticuloEnExposicion(
                    id: "A004",
                    nombre: "Chaqueta",
                    precio: 89.99,
                    inicioExposicion: new (2025, 2, 1)))
    Console.WriteLine(ticket.ATexto());
}
```

Mostrará por consola el siguiente ticket de compra:

```
Camiseta                                19,99?
Pantalón                39,99? con 20%  31,99?
Zapatillas                              59,99? R (Con caja original)
Chaqueta                89,99? con 75%  22,50? E (desde 01/02/2025)
-----------------------------------------------------------------
Total:                                  134,47?
```

Si nos fijamos aunque el ticket solo maneje **artículos genéricos**, ha sabido calcular correctamente a través del enlace dinámico como hay que mostrar la información de cada uno de ellos a través del método `ATextoLineaTicket()`. Además, de csaber calcular el `Total` sumando la propiedad `Precio` de cada uno de los artículos.

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 como la propuesta del diagrama adjunto.

Donde una posible implementación podría ser ...

```
public class ArticuloDefectuoso : ArticuloRebajado
{
    public string Defecto { get; }
    public ArticuloDefectuoso(
                        string id,
                        string nombre,
                        in double precio,
                        in ushort porcentajeRebaja,
                        string defecto) : base(id, nombre, precio, porcentajeRebaja)
    {
        Defecto = defecto;
    }

    public override string ATexto() => $"""
                        {base.ATexto()}
                        Defecto: {Defecto}
                        """;
    public override string ATextoLineaTicket() => $"""
                        {base.ATextoLineaTicket()} D ({Defecto})
                        """;
}
```

> **Importante**
>
> Si ahora añadimos un nuevo artículo de este tipo a nuestro ticket. **No tendremos que modificar nuestra clase** `TicketCompra` y esto será gracias al **polimorfismo de datos**.

```
public static void Main()
{
    TicketCompra ticket = new();
    // ... código omitido para abreviar.
    .Añade(new ArticuloDefectuoso(
                    id: "A005",
                    nombre: "Gorra",
                    precio: 15.99,
                    porcentajeRebaja: 10,
                    defecto: "Pequeño rasguño"));
    Console.WriteLine(ticket.ATexto());
}
```

Mostrará por consola el ticket de compra modificado:

```
Camiseta                                19,99?
Pantalón                39,99? con 20%  31,99?
Zapatillas                              59,99? R (Con caja original)
Chaqueta                89,99? con 75%  22,50? E (desde 01/02/2025)
Gorra                   15,99? con 10%   14,39? D (Pequeño rasguño)
---------------------------------------------------------------------
Total:                                  148,86?
```

Puedes descargar el código del ejemplo anterior del siguiente enlace:

herencia_ticket_articulos.cs

El caso especial de la clase object en CSharp

La Clase `Object` definida en `System`, es una clase especial de la cual heredan de forma **implícita** todos los objetos, tanto valor como refrencia, 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. Además, 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:

object.ToString()

El método `public virtual string ToString()`, 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 nuestras clases de ejemplo.

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** mostrará por consola el **nombre completo de la clase**, que en el caso caso del siguiente ejemplo sería ***EjemploHerencia.Articulo***:

```
namespace EjemploHerencia;

public class Articulo { ... }

public class Program
{
    public static void Main()
    {
        Articulo a = new (
                    id: "A001",
                    nombre: "Camiseta",
                    precio: 19.99);
        Console.WriteLine(a);
    }
}
```

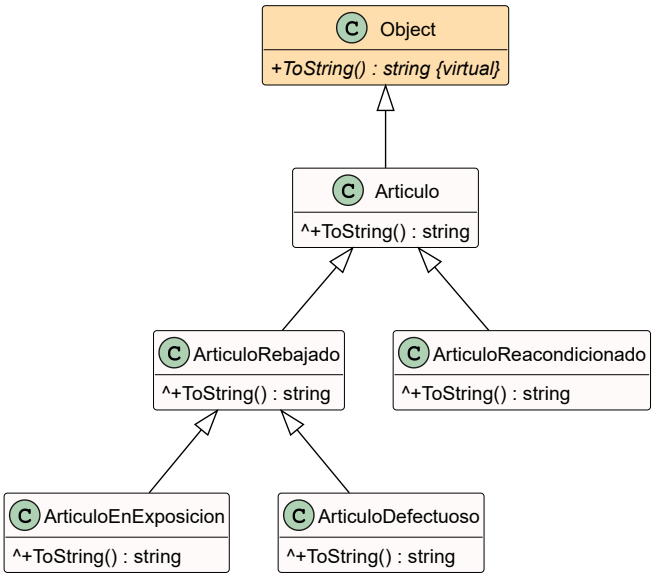
Mostrará por consola:

```
EjemploHerencia.Articulo
```

Si ahora **sustituimos el nombre del método** `ATexto()` por el método `ToString()` toda la jerarquía de artículos podrá mostrar su información de forma adecuada y podremos evitar tener que llamar a `ATexto()` cada vez que queramos mostrar un artículo. Pues `Console.WriteLine(a)` llamará automáticamente a `ToString()`.

Ten en cuenta que al ser un método de `object` no debemos declararlo como `virtual` en la clase `Articulo`, sino aplicar el modificador `override` para invalidarlo, ya que `ToString()` ya está definido en la clase `object`.

```
public class Articulo
{
    // ... código omitido para abreviar.
    public override string ToString() => $"""
            Id: {Id}
            Nombre: {Nombre}
            Precio: {Precio:F2}€
            """;
}
```



Ahora el Main mostrará por consola:

```
Id: A001
Nombre: Camiseta
Precio: 19,99?
```

object.Equals() y object.GetHashCode()

Los métodos `virtual bool Equals(object obj)` y `virtual int GetHashCode()` **deben implementarse siempre juntos** y se usan para comparar el objeto sobre el que se aplica con cualquier otro que se le pase como parámetro en **profundidad** y para obtener un valor de hash único del objeto, respectivamente.

Ambos metodos **se imvalidan de forma automática** en aquellas clases que **tienen en su definición** la palabra reservada `record`, esto es, se comportan como `Value Objects`.

Veamos un ejemplo en el cual los invalidaremos en la clase `Articulo`.

Una opción es comparar propiedad por propiedad no calculada. Pero en este caso, como la propiedad `Id` es única para cada artículo, podemos comparar directamente su valor. Además, como `Id` es de tipo `string`, podemos usar el Equals de `string` para comparar su valor puesto que ya está implementado en la clase `string`.

La función `GetHashCode()` nos devolverá **un valor de hash único** para el objeto. En este caso, usaremos la función `HashCode.Combine()` para combinar los valores de las propiedades que consideremos relevantes para la comparación, como `Id`, `Precio` y `Nombre`. También podríamos usar el método `GetHashCode()` de `string` para obtener un valor de hash único para la propiedad `Id` por ejemplo `Id.GetHashCode()`.

```
class Articulo
{
    // ... código omitido para abreviar.
    public override bool Equals(object? obj) => obj is Articulo a && a.Id.Equals(Id);

    public override int GetHashCode() => HashCode.Combine(Id, Precio, Nombre);
}
```

En las subclases, como `ArticuloRebajado`, `ArticuloReacondicionado` o `ArticuloEnExposicion`, podemos invalidar estos métodos para que también se comparen las propiedades específicas de cada subclase, pero en este caso no es necesario, ya que al heredar de `Articulo` se ejecutaría el método `Equals()` de la superclase y se compararían por `Id`.

```
public static void Main()
{
    ArticuloRebajado ar1 = new(
                        id: "A002",
                        nombre: "Pantalón",
                        precio: 39.99,
                        porcentajeRebaja: 20);
    ArticuloDefectuoso ad2 = new(
                        id: "A002",
                        nombre: "Pantalón",
                        precio: 39.99,
                        porcentajeRebaja: 20,
                        defecto: "Cremallera rota");
    Console.WriteLine(ar1.Equals(ad2));
}
```

En el ejemplo anterior, al comparar `ar1` y `ad2`, ambos tienen el mismo `Id` y por tanto el resultado de la comparación será `true` aunque son de tipos diferentes.

object.GetType()

El métodos `virtual Type GetType()` es un método que nos permite obtener el **tipo del objeto en tiempo de ejecución**. Este método es muy útil cuando queremos saber el tipo real de un objeto, especialmente cuando trabajamos con herencia y polimorfismo. Por ejemplo ...

```
Articulo a = new ArticuloRebajado("A002", "Pantalón", 39.99, 20);
Console.WriteLine(a.GetType().Name); // Mostrará "ArticuloRebajado"
```

🎓 **Caso de estudio modelo de examen:**

Vamos a aplicar un poco todos los conceptos que hemos visto hasta ahora, a través de un caso de estudio '*simplificado*'.

Vamos a suponer que el propietario de una **campa de aparcamiento vehículos** a largo plazo y que decidimos 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 vehiculos que entran y salen.

El sistema es capaz de identificar de un **vehículo** al pasar, la siguiente información común que podemos modelar a través de la clase `Vehiculo` con las siguientes **propiedades publicas de solo lectura**:

- `Matricula` que será un **identificador único** con formato '`DDDD LLL`' donde `D` será un dígito de 0 a 9 y `L` una letra mayúscula excluidas las vocales. La modelaremos a través de un `Value Object` que la guaradará como una cadena de texto.
- `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 de un **tipo enumerado**:

  ```
  public enum Color
  {
      Blanco, Morado, Cian, Azul, Rojo, Verde, Negro, Naranja, Gris
  }
  ```

  Los tipos enumerados se te proporcionarán junto al programa principal y deben declararse fuera del ámbito de esta clase para evitar un conflicto de nombres con las propiedades correspondientes.
- `Marca` que será un conjunto de valores de logos que la IA es capaz de identificar en las imágenes como un **tipo enumerado** de entre los siguientes:

  ```
  public enum Marca
  {
      DESCONOCIDA, BMW, SEAT, AUDI, RENAULT, MAN, DAF, CITROEN, TOYOTA, SUZUKI, YAMAHA, MERCEDES, PEGA
  }
  ```

- `Ocupantes` que la IA cree que hay en el interior del vehículo.
- `Tipo` **propiedad calculada e invalidable**, que nos devolverá **como texto** el **tipo de vehículo** que ha identificado la IA **dentro de una determinada categoría**. Si la IA no ha podido identificar la categoría del vehículo, devolverá el valor *"SinIdentificar"*.

Además, esta clase inavalidará los métodos `Equals(object obj)`, `int GetHashCode()` y `string ToString()` de la clase object De tal manera que muesa la iformación del vehículo en una sola línea de texto.

```
public override string ToString() => $"""
    {GetType().Name} {Marca} {Tipo} {Matricula} color {Color} y {Ocupantes} ocupantes.
    """;
```

Fíjate que hemos usado el método `GetType().Name` para mostrar el **nombre de la clase real del objeto**. Además, `Tipo` tendrá el valor *"SinIdentificar"*, si no es invalidado en las subclases.

La IA, de momento, sabe **clasificar muchos vehículos en estás tres categorías**: `Coche`, `Moto` y `Camión`. Las cuales vamos a **modelizar a través de herencia**. Cada categoría, como hemos comentado, tendrá a su vez **diferentes tipos** a través de un conjunto de valores enumerados.

- **Coche:** Podrá tomar uno de los valores del siguiente conjunto de valores enumrados:

  ```
  public enum TipoCoche
  {
      SinIdentificar, Berlina, Coupe, Sedan, Cabrio, TodoTerreno, MonoVolumen, Crossover
  }
  ```

- **Moto:** Podrá tomar uno de los valores del siguiente conjunto de valores enumerados:

  ```
  public enum TipoMoto
  {
      SinIdentificar, Scooter, Motocross, Naked, Trail, Supermotard
  }
  ```

- **Camion:** Podrá tomar uno de los valores del siguiente conjunto de valores enumerados:

  ```
  public enum TipoCamion
  {
      SinIdentificar, Articulado, Frigorífico, Cisterna, Trailer
  }
  ```

Todas las subclases de `Vehiculo` invalidarán la propiedad `Tipo` para devolver el tipo de vehículo correspondiente **como texto**. De tal manera que **guardarán el valor enumerado de tipo como campo privado**. Además, **los camiones**, guaradrán dos propiedades adicionales identificables por la IA que son el número de **ejes** y **carga máxima (MMA)** en kilos. Por tanto, la subclase `Camion` invalidará el método `ToString()` para mostrar esta información adicional intercalaándola entre el nombre de la categoría y la marca del camión.

```
Plaza n: Camion 2 ejes con MMA de 6000 Kg DAF Frigorífico 8798 JWR color Blanco y 1 ocupa
```

¿Se te ocurre como hacerlo reutilizando el ToString() de `Vehículo` y no repitiendo el código?

Por último, nuestra IA, pasará la generalización de `Vehiculo`, 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). Estás plazas se numerarán de la **1** a la **5**.

La clase `CampaVehiculos` tendrá pues los siguientes miembros:

Privados:

- `List<Vehiculo?> Plazas` : Propiedad de solo lectura que contiene la lista de vehículos que se encuentran aparcados en la campa. Si en una plaza no hay vehículo, se guardará un `null`.
- `int? PlazaVacia` : Propiedad calculada que me retornará el **primer índice vacío** en un array de vehículos o **null** si no lo encuentra ninguno.
- `int PlazasOcupadas` : Propiedad calculada que me retornará el número de plazas ocupadas en la campa.
- `int? Busca(Vehiculo v)` : Método que me retornará el **índice** en la lista de vehículos que ocupa el vehículo `v` o **null** si no lo encuentra.

> **Idea**
>
> Recuerda que el tipo `int?` puede valer null, por lo que si tenemos `int? dato`, podemos usar las propiedades `HasValue` y `Value` para comprobar si tiene un valor y obtenerlo, respectivamente. Por ejemplo...
> `int valor = (dato.HasValue) ? dato.Value : 0;`

Públicos:

- Constructor que recibe un entero con la capacidad de la campa (número de plazas) y que inicializará la propiedad `Plazas` a dicha dimensión con `null` en todas sus plazas pues aún no hay vehículos aparcados.
- `(bool puedeEntrar, int plaza, string? problema) PuedeEntrar(Vehiculo v)` : Método que me retornará un **tupla** con un **booleano** que indica si el vehículo puede entrar en la campa, el número de plaza donde se podría aparcar y un mensaje de aviso en caso de que no pueda entrar. Retornos posibles...
  - Si la campa está llena, retornará `false`, **0** y el mensaje '*Aparcamiento lleno*'.
  - Si la matrícula del vehículo ya se encuentra registrada, retornará `false`, **0** y el mensaje '*Ya se encuentra en el aparcamiento el vehículo DDDD LL*'.

    - Si la campa no está llena y la matrícula no se encuentra registrada, retornará `true`, el número de plaza donde se podría aparcar (1 a Numero Plazas) y **null** como mensaje.
- `void Entra(Vehiculo v)`: Método que recibe un vehículo y lo aparca en la plaza indicada por el método `PuedeEntrar(Vehiculo v)`. Si no se puede aparcar, **lanzará una aserción** con el problema indicado por dicho método.
- `(bool puedeSalir, int plaza, string? problema) PuedeSalir(Vehiculo v)`: Método que me retornará un **tupla** con un **booleano** que indica si el vehículo puede salir de la campa, el número de plaza donde se encuentra aparcado y un mensaje de aviso en caso de que no pueda salir. Retornos posibles:
    - Si la campa está vacía, retornará `false`, **0** y el mensaje '*`Aparcamiento vacío`*'.
    - Si la matrícula del vehículo no se encuentra registrada, retornará `false`, **0** y el mensaje '*`No se registró la entrada del vehículo DDDD LL`*'.
    - Si la matrícula del vehículo se encuentra registrada, retornará `true`, el número de plaza donde se encuentra aparcado (1 a Numero Plazas) y **null** como mensaje.
- `void Sale(Vehiculo v)`: Método que recibe un vehículo y lo saca de la plaza indicada por el método `PuedeSalir(Vehiculo v)`. Si no se puede sacar, **lanzará una aserción** con el problema indicado por dicho método.
- `string ToString()` imvalidado para que nuestre las plazas y los datos que se recibieron de la IA de cada vehículo en el siguiente formato:

```
Plaza n: <datos del vehículo devueltos por ToString()>
Plaza n: Vacía
```

Purdes decargar, **el código con el programa principal** que simulará la ejecución de la IA crando instancias de vehículo para probar tus clases y métodos: **campa_vehiculos.cs**

La ejecución del programa principal, con la siguiente entrada de vehículos, debería producir la **siguiente salida por consola**:

```
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             -> Aparcado en la plaza 4
Entrando Vehiculo SinIdentificar 0000 DGP -> Aparcado en la plaza 5
Entrando Moto Naked 1111 GRF              -> Aparcamiento lleno
Entrando Coche Coupe 1020 DRG             -> Aparcamiento lleno
Saliendo Camion Frigorífico 8798 JWR      -> No se registró la entrada del vehículo 8798
Saliendo Camion Frigorífico 8798 JWR      -> No se registró la entrada del vehículo 8798
Saliendo Moto Naked 1111 GRF              -> No se registró la entrada del vehículo 1111
Entrando Moto Naked 1111 GRF              -> Aparcamiento lleno
Entrando Camion SinIdentificar 8798 JWR   -> Aparcamiento lleno
Veículos en el aparcamiento...

Plaza 1: Coche SEAT Coupe 1020 DRG color Azul y 3 ocupantes.
Plaza 2: Camion 2 ejes con MMA de 6000 Kg DAF Frigorífico 8798 JWR color Blanco y 1 ocupa
Plaza 3: Coche BMW TodoTerreno 7643 LRF color Rojo y 4 ocupantes.
Plaza 4: Coche SEAT Coupe 1020 DRG color Azul y 3 ocupantes.
Plaza 5: Vehiculo DESCONOCIDA SinIdentificar 0000 DGP color Negro y 2 ocupantes.
```

Nota

Intenta realizar primero por tu cuenta la implementación de la especificación anterior. Posteriormente comparala con la **propueta de solución** que encontrarás en el siguiente enlace: **campa_vehiculos_solucion.cs**