PROGRAMACIÓN ORIENTADA A OBJETOS (POO) INTRODUCCIÓN

INTRODUCCIÓN

- Paradigma de programación que pretende mejorar aspectos de la programación imperativa tradicional tales como:
  - Abstracción con la que representamos el problema.
  - Portabilidad del código y por tanto su reusabilidad.
  - Modularidad del código y por tanto legibilidad.
- Atacaremos los problemas dividiéndolos en unidades lógicas denominadas objetos, que colaborarán entre ellos para resolver el problema.

POO
DEFINICIONES

DEFINICIONES - TAD

¿Qué Es Tipo Asbtracto De Datos O TAD?

- Especificación abstracta, completa y no ambigua de una estructura de datos junto con el conjunto de operaciones que se pueden hacer sobre ese tipo de datos.
- Puede tener una o más implementaciones.

DEFINICIONES - CLASE

¿Qué Es Una Clase?

- Es la implementación total o parcial de un TAD.
- Define objetos que van a tener la misma estructura y comportamiento.
- Añade los conceptos de paso de mensajes, Herencia y Polimorfismo que no se contemplan en los TAD.
- Existen autores que las definen con 2 naturalezas:
  - Como Tipo: Implementa un TAD con sus atributos y operaciones.
  - Como módulo: Organización y encapsulación de software.

DEFINICIONES - CLASE

¿Por Qué Está Formada Una Clase Como Implementación De Un TAD?

- Un Nombre: Que escribe a la clase.
- Atributos: Son datos necesarios para describir los objetos (instancias) creados a partir de la clase.
  - La combinación de sus valores determina el estado de un objeto.

Cuenta
+ Saldo : real + Titular : cadena
+ Reintegro() : real + Ingreso(cantidad:real) : void

- Roles: relaciones que una clase establece con otras clases.
- Operaciones, Métodos, Servicios:
  - Debería se el único modo de acceder a los atributos.
  - Describe la operaciones posibles sobré un objeto de esa clase.

DEFINICIONES – OBJETO

¿Qué Es Un Objeto?

- De manera informal podemos decir que es una instancia en memoria de una clase creada en tiempo de ejecución.

¿Por Qué Está Formado?

- Un Estado: que vendrá dado por el valor de sus atributos y su rol durante la ejecución.
- Un Comportamiento: que será el modo en que las operaciones cambian a su estado.
- Una Identidad: que me permitirá distinguirlo de otros.
  - Dos objetos son iguales si tienen el mismo estado.
  - Ojo !! No es lo mismo identidad que igualdad.

c1:Cuenta	c2:Cuenta	c3:Cuenta
Saldo = 30000 Titular = Juanjo	Saldo = 30000 Titular = Juanjo	Saldo = 60000 Titular = Maria

c1≡c1
c1=c2
c1≠c3

DEFINICIONES – ATRIBUTO

¿Qué Es Un Atributo O Propiedad?

- Describirá los objetos de una clase y sus valores indicarán el estado de un objeto.

¿Qué Tipos Hay Atendiendo A La Forma De Acceder A Ellos?

De Instancia

- Serán diferentes en cada objeto.
- Necesitaré de un objeto instanciado en memoria para acceder a ellos.

De clase

- Tendrán el mismo valor en todos los objetos de la clase, por tanto almacenan características comunes a todos ellos.
- No necesito un objeto instanciado para acceder a ellos.
- Se diferencian de una variable global en que están encapsuladas.

DEFINICIONES – ATRIBUTO

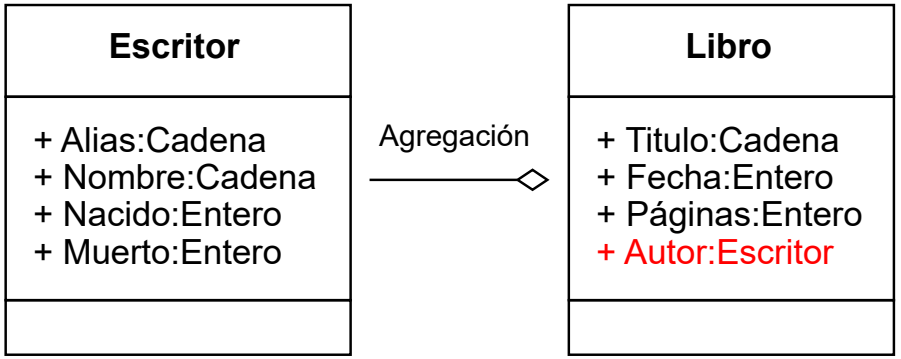
¿Qué Tipos Hay Atendiendo A Su Tipo De Datos?

- Básicos: Tipos simples como entero, real, cadena, etc…
- Compuestos: Otras clases. Estableciéndose relaciones Todo-Parte entre ellas.

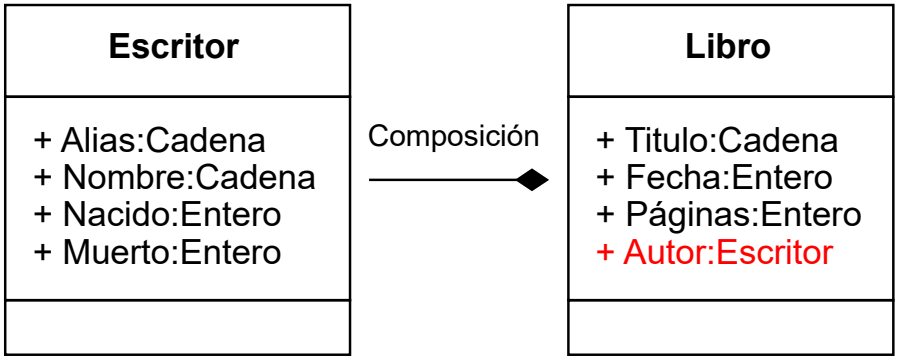
¿Qué Tipos De Relaciones Todo-Parte Hay?

- Como subobjetos: Almacenamos el Valor.
- Como referencias: Almacenamos una referencia al objeto.

REFERENCIA o AGREGACION



SUBOBJETO o COMPOSICION

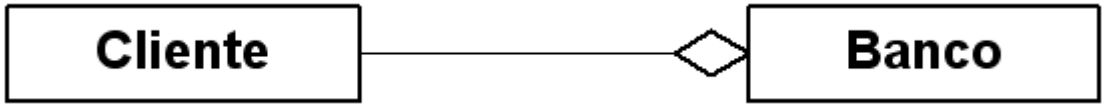


DEFINICIONES – ATRIBUTO

¿Cuándo Utilizar Agregación O Composición?

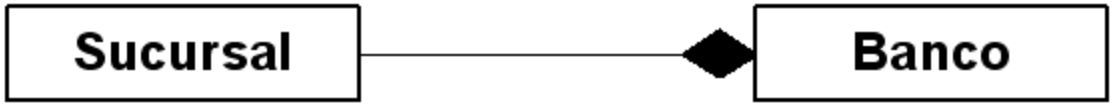
- La agregación indica independencia de los objetos, esto es si desaparece el contenedor no desaparece el agregado.

  Ej: Si desaparece el banco no desaparece el cliente.



- La composición indica dependencia de los objetos, esto es si desaparece el contenedor, también desaparece el agregado.

  Ej: Si desaparece el banco desaparecen sus sucursales.



DEFINICIONES – ATRIBUTO

¿Cómo Se Representaría Físicamente En Memoria Con Objetos Instanciados De La Anterior Definición De Clases?

COMPOSICIÓN

☹ Gastamos memoria.
☹ No podemos compartir.

AGREGACIÓN

☹ **Aliasing** con memoria gestionada por el usuario.
☺ Compartición (Integridad referencial).
☺ Mejor manejo de objetos complejos.
☺ Los objetos se crean cuando se necesitan.
☺ Necesario para el polimorfismo.

DEFINICIONES – OPERACIÓN, MÉTODO, SERVICIO ...

¿Que Es Una Operación O Método?

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

¿Qué Tipos De Métodos Hay? (Ya Lo Sabemos, Los Hemos Usado).

Métodos de instancia o también (de objeto).

- Necesito tener un objeto instanciado en memoria para acceder a ellos.
- Pueden acceder tanto a atributos de instancia como de clase.

Métodos de clase o también estáticos.

- No necesito tener un objeto instanciado en memoria para acceder a ellos.
- En principio solo pueden acceder a los atributos de clase (static).

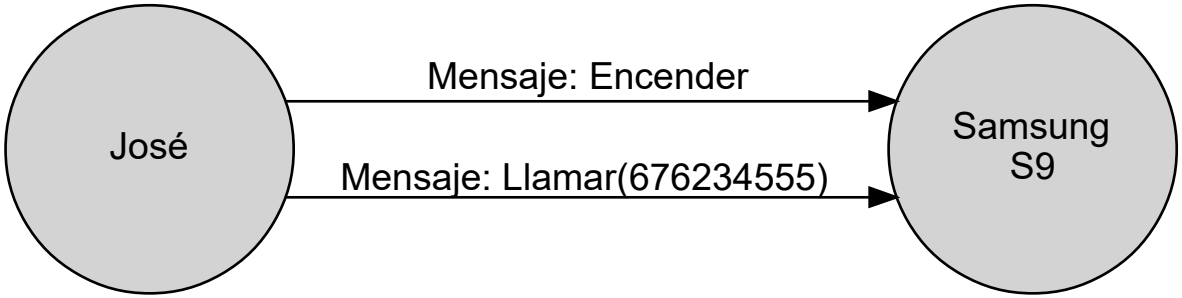
Métodos de acceso y actualización

- También se les conoce como Accesores - Mutadores, Propiedades en C# o Setters - Getters (Java).

DEFINICIONES – PASO DE MENSAJE A UN OBJETO

¿Qué Es El Paso De Mensajes?

- La invocación de un método sobre un objeto se le denomina.



- Esto significa que lo que hacemos realmente es enviar un mensaje a un determinado objeto receptor, indicándole la signatura método que debe ejecutar.
- Recordemos que al conjunto formado por el identificador de un método y el número y tipo de sus parámetros formales se le denomina SIGNATURA.

DEFINICIONES – ENCAPSULACIÓN

¿Qué Es La Encapsulación?

- En POO, se denomina encapsulación al ocultamiento del estado, es decir, de los atributos, de un objeto de manera que sólo se puede cambiar mediante las operaciones definidas para ese objeto o sus accesores - mutadores.
- De esta forma el usuario de la clase solo interacciona con los objetos abstrayéndose de como están implementados.
- Se evita que el usuario pueda cambiar su estado de maneras imprevistas e incontroladas.

DEFINICIONES – CONSTRUCTORES Y DESTRUCTORES

¿Qué Es Un Constructor?

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

¿Qué Es Un Destructor?

- Un único método especial encargado de eliminar una instancia en memoria de un objeto.
- En algunos lenguajes OO no hace falta definirlos, pues de la labor de eliminación de instancias se encarga un RECOLECTOR DE BASURA cuando un objeto ya no es referenciado por nadie.

IMPLEMENTACIÓN CONCEPTOS POO EN C#

IMPLEMENTACIÓN CONCEPTOS POO EN C#

Definiendo El Nombre Y Los Atributos De Una Clase

Supongamos la siguiente definición de TAD como ejemplo…

PuntoConsola
- fila : ushort - columna : ushort
+ Muestra(CosoleColor: color) : void + Desplaza(numPosiciones:ushort, angulo:real) : void

En primer lugar definiremos la clase, la cual tiene 2 atributos Fila y Columna marcados como privados, (No pueden ser accedidos desde fuera de la clase), para ello antepondremos la cláusula private → SIEMPRE!!!

```
class <NombreDeLaClase>
{
   <atributos>
   <constructor/es>
   <métodos>
   <propiedades>
}
```

```
class PuntoConsola
{
   // <atributos>
   private ushort fila;
   private ushort columna;
}
```

IMPLEMENTACIÓN CONCEPTOS POO EN C#

Definiendo El Constructor/es De Una Clase

- A continuación de los atributos crearemos un método constructor para la clase.
- En C# el método constructor tiene el mismo nombre que la clase y no lleva tipo de retorno (Es implícito).

```
class PuntoConsola
{
 // <atributos>
    private ushort fila;
    private ushort columna;

 // <constructor>
 public PuntoConsola(ushort fila, ushort columna)
 {
     this.fila = fila;
     this.columna = columna;
 }
 // <métodos>
 // <propiedades>
}
```

¿Qué Es This?

- Es una referencia implícita a la instancia en memoria del objeto que en ese momento estamos creando o está accediendo a un método de la clase.

- Nos ayuda a diferenciar entre los identificadores de los atributos y los parámetros de entrada del constructor.

¿Cómo Definir Un Constructor Para Copiar Objetos?

```
public PuntoConsola(PuntoConsola punto)
{
    fila = punto.fila;
    columna = punto.columna;
}
```

- Es un constructor, no obligatorio, que copiará el estado de otra instancia de un objeto de la clase PuntoConsola.
- Como vemos no hemos usado this, porque no hay posibilidad de confusión.

- Más adelante veremos que para realizar copias usaremos un método especial llamado Clone.

IMPLEMENTACIÓN CONCEPTOS POO EN C#

Definiendo El Destructor Una Clase

- Cuando un objeto deja de ser referenciado, al cabo de tiempo es eliminado de la memoria por el Recolector de Basura (Garbage Collector). Llamando a su destructor por defecto que nosotros podremos redefinir.

```
class PuntoConsola
{
    // <atributos>
    // <constructores>

    // <destructor>
    ~PuntoConsola()
    {
        fila = ushort.MaxValue;
        columna = ushort.MaxValue;
    }
}
```

IMPLEMENTACIÓN CONCEPTOS POO EN C#

Definiendo Métodos O Operaciones Sobre Una Clase

- Desde cualquier método de instáncia, podremos acceder a sus atributos. Ya sea a través de this o “directamente” si no hay confusión.
- Para nuestro ejemplo tendremos...

```
class PuntoConsola
{
    // <atributos>
    // <constructores / Destructor>
    // <métodos>
    public void Muestra(ConsoleColor color)
    {
        Console.SetCursorPosition(columna, 24 - fila);
        Console.ForegroundColor = color;
        Console.Write("*");
    }
    public void Desplaza(ushort posiciones, double anguloGrados)
    {
        double anguloRadianes = anguloGrados * Math.PI / 180d;
        fila += (ushort)( posiciones * Math.Sin(anguloRadianes));
        columna += (ushort)( posiciones * Math.Cos(anguloRadianes));
    }
}
```

IMPLEMENTACIÓN CONCEPTOS POO EN C#

Definiendo Accesores Y Mutadores. ("Propiedades")

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

- Para el accesor, crearemos un método con el prefijo get seguido del nombre del atributo.
- Para el mutador, crearemos un método con el prefijo set seguido del nombre del atributo.
- En el cuadro de al lado vemos cómo se definirían para el atributo fila de nuestra clase de ejemplo PuntoConsola.

```
class PuntoConsola {

    // <atributos>
    // <constructores / Destructor>
    // <métodos>

    // <accesores / Mutadores>
    // <propiedades>
    public ushort GetFila() {
        return fila;
    }

    public void SetFila(ushort fila) {
        Debug.Assert(fila <= 24);
        this.fila = fila;
    }
}
```

IMPLEMENTACIÓN CONCEPTOS POO EN C#

Resumen Directrices Generales Iniciales De Implementación

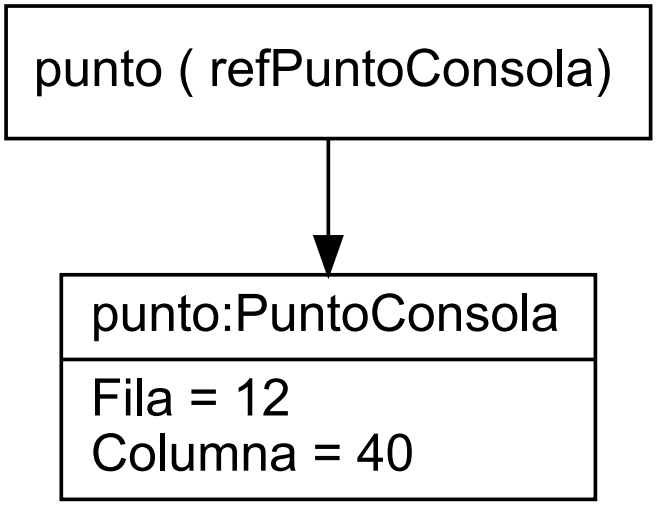
- Definir datos miembro o atributos privados (Encapsulación).
- Definir constructores, y destructor. Este último, solo si es necesarios.
- Definir accesores/mutadores o propiedades solo cuando sea necesario.
- Definir operaciones sobre el objeto sobre el objeto como métodos públicos.
- Recordar mantener siempre la integridad del estado de nuestro objeto.

Instanciando Nuestro Primer Objeto En Memoria

- Puesto que son objetos y son tipos referencia, deberé utilizar siempre el operador new para crearlos y uno de los constructores definidos.

```
PuntoConsola punto = new PuntoConsola(12, 40);
```

- En el identificador punto no tendremos el objeto en si, sino una referencia al mismo.
- Pero para nosotros cuando accedamos a punto es como si estuviéramos accediendo al propio objeto.



IMPLEMENTACIÓN CONCEPTOS POO EN C#

Interactuando Con Nuestro Objeto - I

- A partir de ahora ya podemos cambiar el estado del objeto a través de sus operaciones/métodos usando el mecanismo de paso de mensajes anteriormente descrito.

```
PuntoConsola punto = new PuntoConsola(12, 40);
punto.Muestra(ConsoleColor.DarkBlue);
punto.Desplaza(4, 45d);
punto.Muestra(ConsoleColor.Blue);
```

- Pero para hacer una copia deberemos construir o instanciar uno nuevo...

```
PuntoConsola puntoCopia = new PuntoConsola(punto);
// En el futuro con Clone
```

y nunca haremos

```
PuntoConsola puntoCopia = punto; ☠☠
```

puesto que estaríamos referenciando al mismo objeto en memoria.

IMPLEMENTACIÓN CONCEPTOS POO EN C#

Interactuando Con Nuestro Objeto - II

Ahora podremos modificar ambos objetos de forma independiente…

```
punto.Desplaza(10, 180d);
puntoCopia.Desplaza(15, 0d);
punto.Muestra(ConsoleColor.DarkCyan);
puntoCopia.Muestra(ConsoleColor.DarkRed);
punto.Desplaza(10, 120d);
puntoCopia.Desplaza(10, 160d);
punto.Muestra(ConsoleColor.Cyan);
puntoCopia.Muestra(ConsoleColor.Red);
punto.setFila(20);
puntoCopia.Muestra(ConsoleColor.DarkMagenta);
```

Hacer Hasta El Ejercicio 4

POO
HERENCIA

HERENCIA EN POO

- Una de las características principales de la POO.
- Representará el tipo de relación "Es un/a".
- La herencia nos servirá para reutilizar código y no repetirnos.

Definición

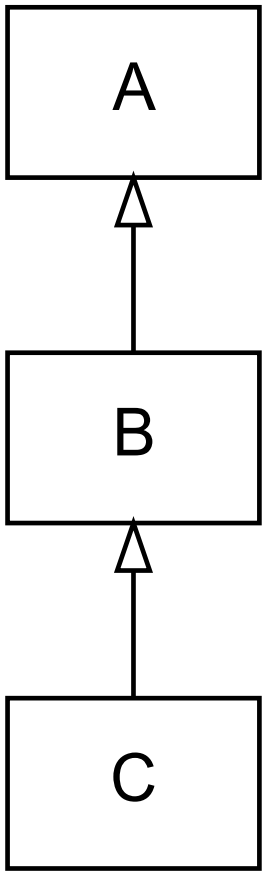
- Tipo de relación entre clases, en la cual una clase denominada subclase (o clase hija), comparte la estructura y/o comportamiento definidos en una o más clases, llamadas superclases (o clase padre o clase base).

- Podemos decir que 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.

HERENCIA EN POO

Nomenclatura

- B hereda de A
- A es la superclase y B la subclase
- C hereda de B y A
- B y C son subclases de A
- B es un descendiente directo de A
- C es un descendiente indirecto de A



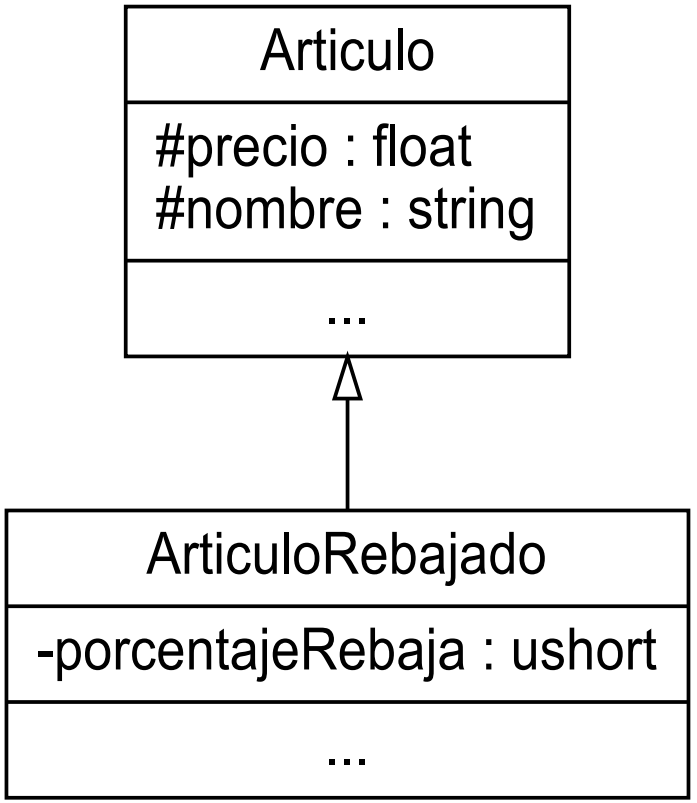
HERENCIA EN POO

Herencia Simple

- Cuando la subclase hereda de una sola superclase.

Ejemplo

- Tenemos la superclase Articulo con un precio y un nombre
- Una subclase de Articulo denominada ArticuloRebajado que además añade al articulo el atributo rebaja.
- Como vemos en UML la relación de herencia se representa con una flecha de punta hueca desde la subclase a la superclase.

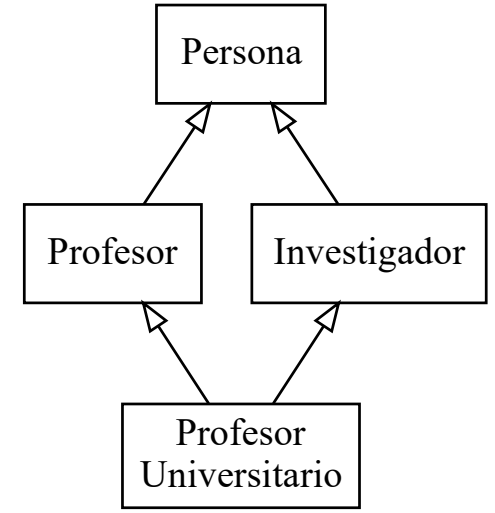


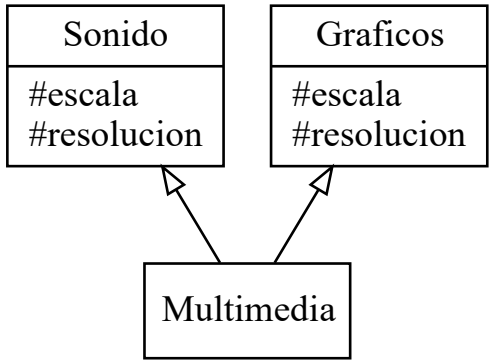
Herencia Múltiple

- Cuando una subclase hereda características de varias superclases.
- Tiene más desventajas que ventajas. Por eso C# y Java NO la permiten.

Desventajas

- Menor velocidad de ejecución.
- Herencia repetida (Transitividad). En el ejemplo Profesor Universitario hereda 2 veces los atributos de Persona.
- Diseños más complejos y más difíciles de aprender y utilizar por el programador.
  (Se puede rediseñar con herencia simple).
- Colisiones de Nombres. En el ejemplo la subclase Multimedia hereda atributos de las clases base Sonido y Gráficos con el mismo nombre… Cuando hagamos referencia a Escala, ¿Cómo podemos saber a cual estamos haciendo referencia?





HERENCIA EN C#

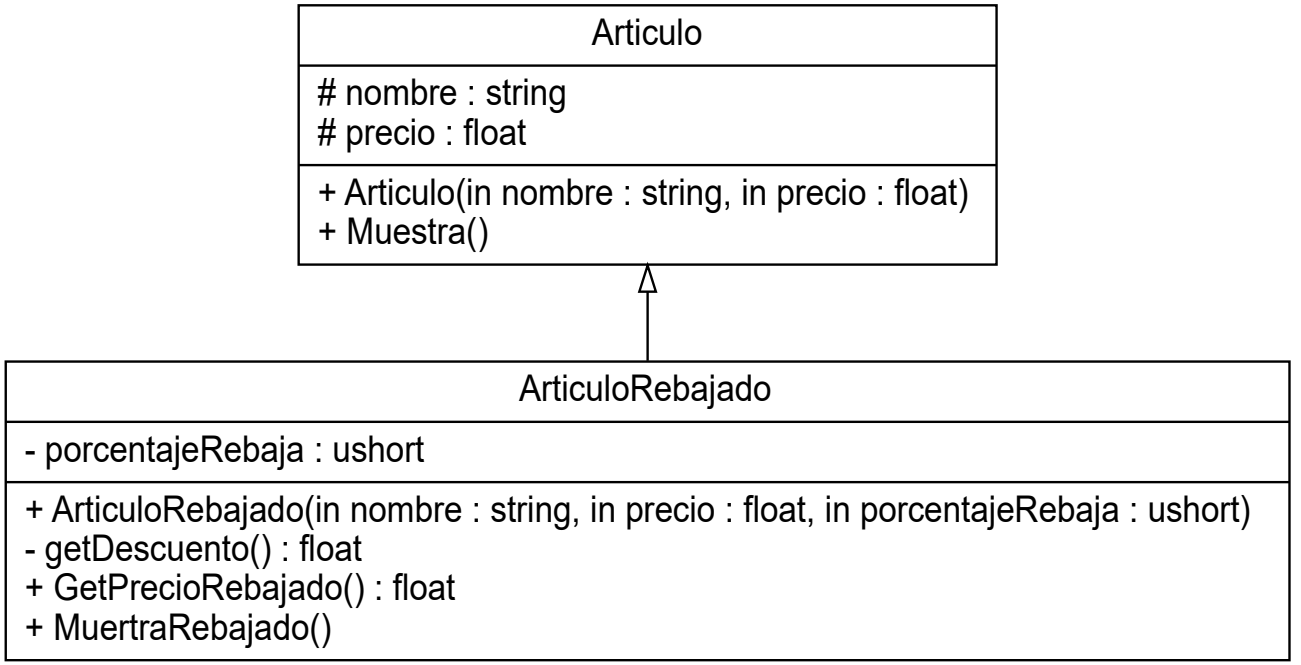
Sintaxis

- Separaremos el nombre de la subclase y la superclase por el carácter ':'

```
class <nombreSubClase> : <nombreSuperclase>
{
    // <definición de la subclase>
}
```

Ejemplo

- Partamos de la siguiente relación de herencia, expresada en el diagrama UML de la derecha.



HERENCIA EN C#

Definiremos Primero La Superclase Articulo

```
class Articulo {
    protected float precio;
    protected string nombre;

    public Articulo(string nombre, float precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    public void Muestra() {
        Console.WriteLine($"Nombre: {nombre}\nPrecio: {precio}€");
    }
}
```

- 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 atributo no puede ser accedido desde fuera de la clase como en el caso de private, pero si desde las subclases de la misma.

HERENCIA EN C#

Definiremos Ahora La Subclase ArticuloRebajado

```
class ArticuloRebajado : Articulo {
    public ushort porcentajeRebaja;

    public ArticuloRebajado(
                string nombre,
                float precio,
                ushort porcentajeRebaja) : base(nombre, precio) {
        this.porcentajeRebaja = porcentajeRebaja;
    }
    private float getDescuento() {
        return (precio * porcentajeRebaja) / 100f;
    }
    public float GetPrecioRebajado() {
        return precio - getDescuento();
    }
    public void MuestraRebajado() {
        Console.WriteLine($"Nombre: {nombre}\n" +
                          $"Rebaja: {porcentajeRebaja}%\n" +
                          $"Antes: {precio}€\nAhora:
                                   {GetPrecioRebajado()}€");
     }
}
```

HERENCIA EN C#

Palabra Reservada **Base**

- Si nos fijamos su constructor 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 automáticamente seria llamado al llamar al de la subclase.

- Al igual que this era una referencia implícita a 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. Me servirá en los casos en los que en la subclase y en la superclase tengamos un método con el mismo nombre.

HERENCIA EN C#

Ejemplo De Instanciación De Subclases

- Supongamos el siguiente código...

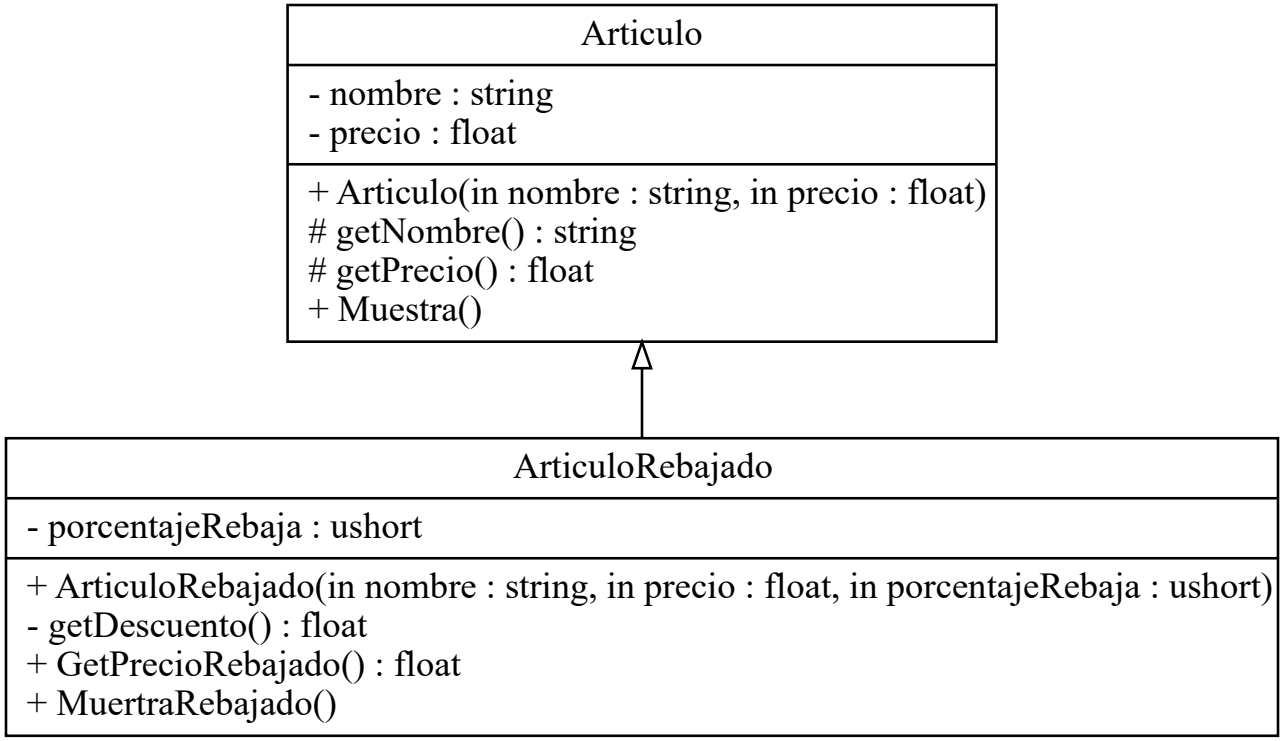
```
static void Main()
{
    Articulo articulo = new Articulo("Camisa", 30f);
    ArticuloRebajado articuloRebajado =
                        new ArticuloRebajado("Polo", 88f, 50);

    articulo.Muestra();

    articuloRebajado.Muestra();
    articuloRebajado.MuestraRebajado();
}
```

- Como vemos, el objeto articuloRebajado hereda el método Muestra() de la clase artículo que mostrará solo los datos que tiene de Articulo.

HERENCIA EN C#

Caso Práctico

- ¿Serías capaz de refactorizar (rehacer) el ejemplo anterior para que cumpla el siguiente diagrama?
- ¿Qué diferencias aprecias?, ¿Qué aporta el nuevo diseño?



HERENCIA EN POO

¿Qué Pasaría Si En ArticuloRebajado Cambiamos La Signatura Del Método Void MuestraRebajado() Por Void Muestra()?

- Tendremos métodos con signaturas idénticas en la superclase y la subclase.

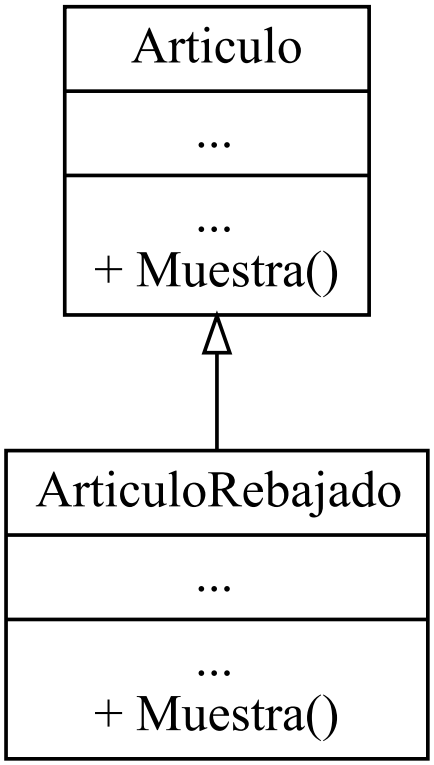
- Podremos hacer 2 cosas:
  - Reemplazo: Se sustituye completamente la implementación del método heredado manteniendo la signatura.
    Según el autor también se le conoce como Ocultación (Hiding) o Shadowing.
  - Refinamiento: Se añade nueva funcionalidad al comportamiento heredado.
    Según el autor también se le conoce como Sobreescritura.

HERENCIA EN C#

Reemplazo O Ocultación En C#

- Lo que buscamos es definir una nueva funcionalidad para una operación heredada.
- Antepondremos la palabra reservada new en la clase hija a la operación o método con la misma signatura que queremos ocultar en la clase base o superclase.

```
class ArticuloRebajado : Articulo
{
    ...
    public new void Muestra()
    {
        Console.WriteLine(
        $"Nombre: {getNombre()}\n" +
        $"Rebaja: {porcentajeRebaja}%\n" +
        $"Antes: {getPrecio()}€\n”+
        $"Ahora: {GetPrecioRebajado()}€");
    }
    ...
}
```



HERENCIA EN C#

Refinamiento O Sobreescritura En C# - I

- Haremos lo que hacía la clase padre más nueva funcionalidad.

- En estos casos diremos que los métodos redefinibles en la superclase son métodos virtuales.
- Para ello utilizaremos la palabra reservada virtual precediendo a la declaración del método virtual y la palabra override precediendo la declaración de un método que redefine a uno virtual en la superclase.

Paso1: Marcando el método cómo sobreescribible por sobclases

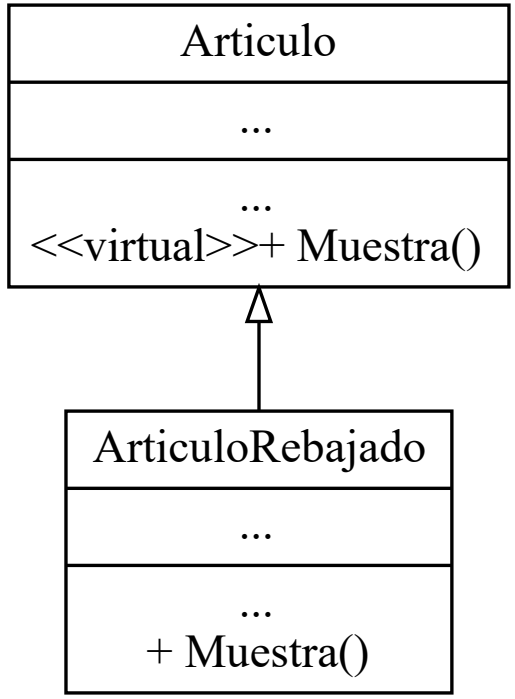
```
Class Articulo
{
     ...
     public virtual void Muestra()
     {
            Console.WriteLine($"Nombre: {nombre}");
            Console.WriteLine($"Precio: {precio}€");
     }
     ...
}
```

HERENCIA EN C#

Refinamiento O Sobreescritura En C# - II

Paso2: Marcando el método cómo sobreescrito en la subclase

```
class ArticuloRebajado : Articulo
{
    ...
    public override void Muestra()
    {
        // Llamamos a la funcionalidad
        // de Mostrar en la superclase
        base.Muestra();

        // Añadimos la nueva funcinalidad
        // de mostrar en la subclase.
        Console.WriteLine(
            $"Rebaja:
                    {porcentajeRebaja}%");
        Console.WriteLine(
            $"Precio Rebajado:
                    {GetPrecioRebajado()}€");
    }
    ...
}
```



HERENCIA EN C#

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

- Podemos decir que es, cuando un identificador que hemos declarado del tipo la superclase, referencia a un objeto de la subclase.
- OJO !! Sólo se podrá acceder a través de dicho identificador a lo común en ambos objetos, esto es, lo que se define en la superclse.

HERENCIA EN C#

Polimorfismo De Datos Y Principio De Sustitución De Liskov En C# - I

- También se le conoce como UPCASTING
- Creamos un objeto de la subclase y lo asignamos a la superclase.

```
ArticuloRebajado ar =
                  new ArticuloRebajado("Polo", 88f, 50);

Articulo a = ar;
```

- También podemos hacerlo directamente en la declaración

```
Articulo a = new ArticuloRebajado("Polo", 88f, 50);
```

HERENCIA EN C#

Polimorfismo De Datos Y Principio De Sustitución De Liskov En C# - II

¿Podemos Hacer La Operación Contraria ?

- Sí, si realmente la referencia que tenemos a un artículo es un artículo rebajado.
- A esta operación se le denomina DOWNCASTING.
- Deberemos forzar la conversión con un cast explícito.

```
Articulo a = new ArticuloRebajado("Polo", 88f, 50);
...
ArticuloRebajado ar = (ArticuloRebajado)a;
```

- Sin embargo el siguiente código no sería válido.

```
Articulo a = new Articulo("Polo", 88f);
...
ArticuloRebajado ar = (ArticuloRebajado)a;
```

HERENCIA EN C#

Operadores De Utilidad Para El Downcasting - I

- Operador is:

  Nos sirve para preguntarle a un objeto si es de un determinado tipo.

```
Articulo a = new ArticuloRebajado("Polo", 88f, 50);
if (a is ArticuloRebajado) {
    ArticuloRebajado ar = (ArticuloRebajado)a;
    Console.WriteLine(ar);
}
```

- Operador as: (Es un syntactic sugar de is)

  Realiza directamente el downcasting y si no puede asigna null

```
Articulo a = new ArticuloRebajado("Polo", 88f, 50);
ArticuloRebajado ar = a as ArticuloRebajado;
Console.WriteLine(ar);

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

HERENCIA EN C#

Operadores De Utilidad Para El Downcasting – II

- Operador de coalescencia nulo ??: (Es un syntactic sugar)

  Sirve para simplificar expresiones del tipo

```
ArticuloRebajado ar = …;
…
Articulo a = ar != null ? ar : new ArticuloRebajado(…);
```

  y en su lugar poner...

```
Articulo a = ar ?? new ArticuloRebajado(…);
```

- Se podría usar en el downcasting de la siguiente forma…

```
Articulo a = new ArticuloReacondicionado(
                                "Samsung s10", 500,
                                DateTime.Now, "FOXCOM");
…
ArticuloRebajado ar = a as ArticuloRebajado
                      ??
                      new ArticuloRebajado(
                                a.GetNombre(), a.GetPrecio(), 0);
```

HERENCIA EN C#

Operadores De Utilidad Para El Downcasting – III

- Cláusula case de us switch para objetos (con when opcional):

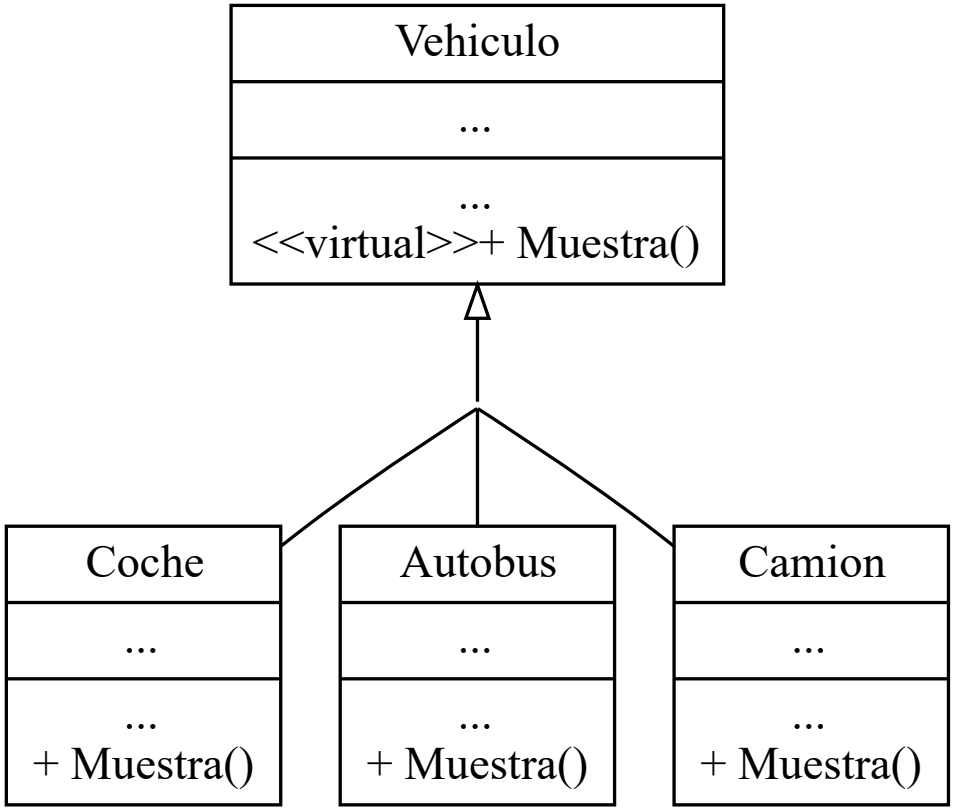
```
Articulo a = new ArticuloReacondicionado(
                        "Samsung s10", 500, DateTime.Now, "FOXCOM");
switch (a)
{
    case null:
        break;
    case ArticuloRebajado ar:
        break;
    case ArticuloReacondicionado are when are.GetEmpresa() == "FOXCOM":
        break;
    case ArticuloReacondicionado are:
        break;
}
```

- Entra en case si se puede hacer el downcasting al nuevo identificador cuyo ámbito de existencia será hasta el siguiente case.
- Si añadimos una especificación con when tiene que estar antes de la generalización.

HERENCIA EN POO

Definición De **Ligadura Dinámica**

- Supongamos el siguiente esquema de herencia:
- Donde tenemos la superclase Vehiculo con un método Muestra() Sobrescrito.



Si hacemos …

Vehiculo v;

y hacemos

v.Muestra();

¿A qué Muestra() estará llamando ?

Dependerá de si **v** está sustituido enlazará o ligará dinámicamente con el Muestra() apropiado.

HERENCIA EN C#

Ligadura Dinámica En C# - I

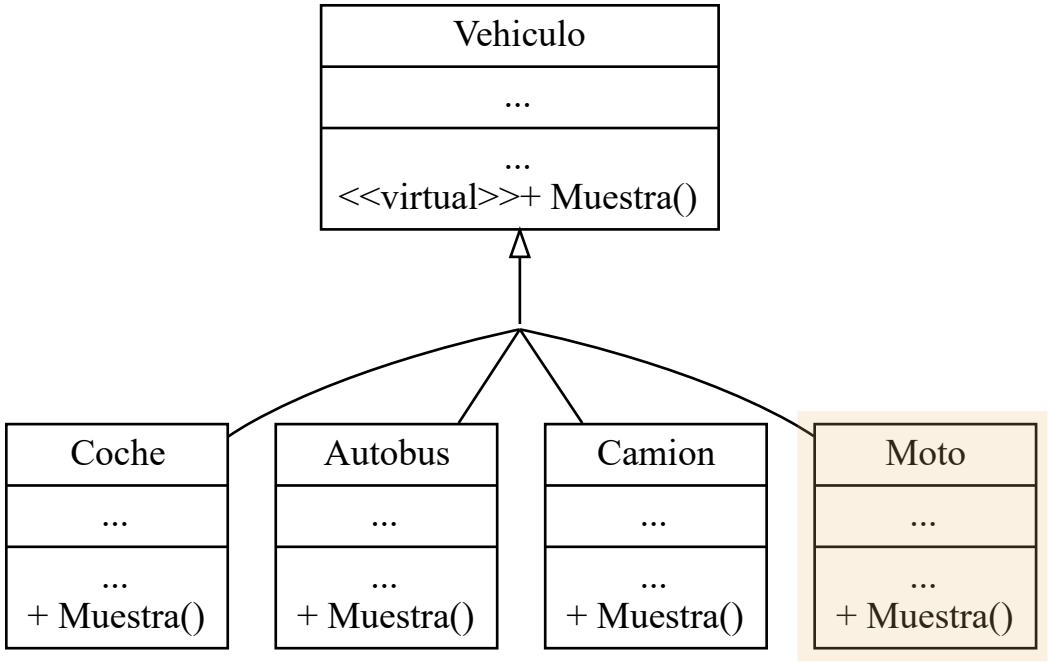
- Supongamos que en una implementación del esquema anterior tenemos…

```
class Parking {
    public void Entra(Vehiculo v) {
        Console.WriteLine("Estas añadiendo un: ");

        // Se enlazará dinámicamente
        // con el Mostrar() de la subclase subyacente.
        v.Mostrar();
    }
}
class Programa {
    static void Main()
    {
        Parking p = new Parking();

        // En cada llamada a Entra() l
        // a subclase se sustituye por la superclse.
        p.Entra(new Coche());
        p.Entra(new Autobus());
        p.Entra(new Camion());
    }
}
```

HERENCIA EN C#

¿Para Que Sirve Este Polimorfismo De Datos?

- En ocasiones el software cambia y se añaden nuevas especificaciones, como pudieran ser nuevos tipos de vehículos.
- 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.



Supongamos que añadimos las Motos como nuevo tipo de Vehiculo.

Podremos hacer...

p.Entra(new Moto());

Sin tener que cambiar nada en la clase Parking.

HERENCIA EN C#

El Caso Especial De La Clase Object En C# - I

- La Clase Object definida en System, es una clase especial de la cual heredan de forma implícita todos los objetos 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.
- 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 en el Framework 2.0
- Define una serie de métodos virtuales que podremos redefinir en cualquiera de las clases que nosotros creemos.

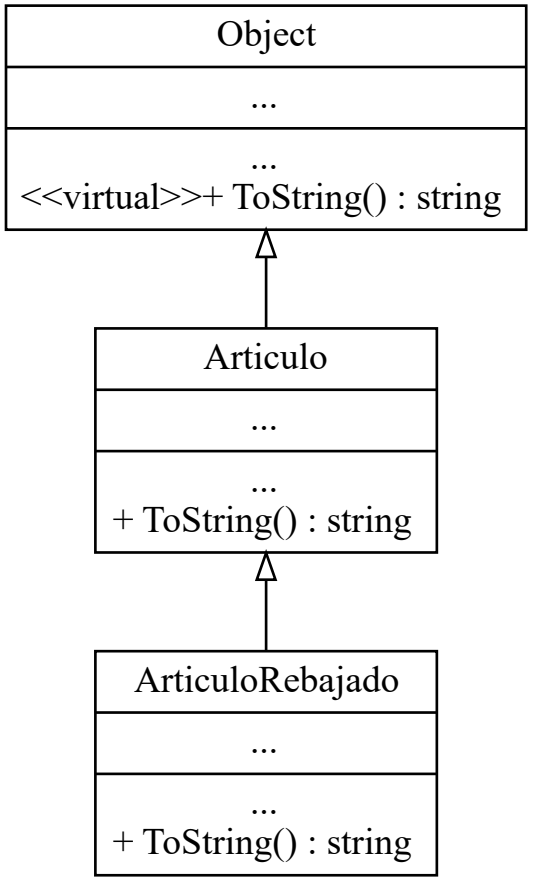
  Un ejemplo útil puede ser:

```
public virtual string ToString();
```

HERENCIA EN C#

El Caso Especial De La Clase Object En C# - II

Si volvemos a nuestra herencia de artículo realmente tendremos…



- Que Articulo hereda de Object y por tanto las clases Articulo y ArticuloRebajado heredan el método ToString().
- Este al ser virtual, se puede sobreercribir, quedando un código similar al siguiente.

```
class Articulo
{
    ...
    public override string ToString()
    {
        string texto =
            $"Nombre: {nombre}\n" +
            $"Precio: {precio}€";
        return texto;
    }
}
```

HERENCIA EN C#

El Caso Especial De La Clase Object En C# - III

```
class ArticuloRebajado : Articulo {
    …
    public override string ToString()
    {
        string texto =
            base.ToString() + "\n" +
            $"Rebaja: {porcentajeRebaja}%\n" +
            $"Precio Rebajado: {GetPrecioRebajado()}€";
        return texto;
    }
}

class Programa {
    static void Main()
    {
        Articulo a1 = new Articulo("Camisa", 30f);
        Articulo a2 = new ArticuloRebajado("Falda", 77f, 20);
        ArticuloRebajado ar = new ArticuloRebajado("Polo", 88f, 50);
        Console.WriteLine(a1);
        Console.WriteLine(a2);
        Console.WriteLine(ar);
    }
}
```

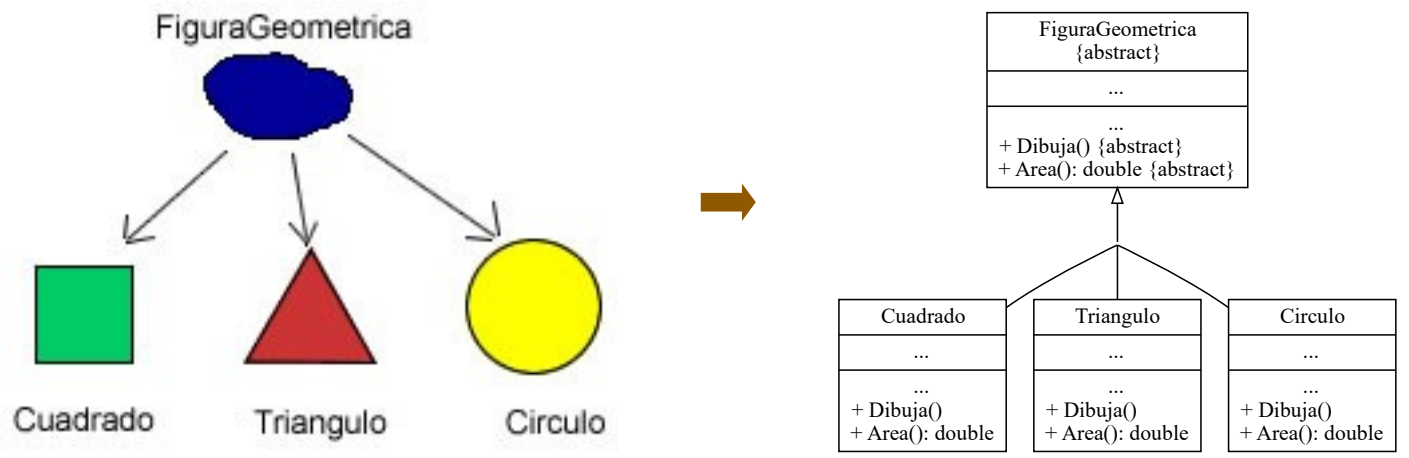
POO
ABSTRACCIÓN

CLASES ABSTRACTAS EN POO

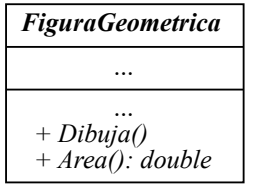
Concepto

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

Ejemplo



- Aunque en el diagrama de clases hemos marcado la clase abstracta y los métodos abstractos con la restricción {abstract} también es normal representar los mismos en *cursiva*.



CLASES ABSTRACTAS EN C#

Sintaxis Para Definirlas I

- Utilizaremos la palabra reservada abstract y al menos uno de sus métodos debe ser abstracto, esto es, dejaremos su implementación en manos de sus subclases.
- Para marcar un método como abstracto también utilizaremos la palabra reservada abstract.

Para el ejemplo de las figuras anterior el código resultante será:

```
abstract class FiguraGeometrica
{
    abstract public void Dibuja();
    abstract public double Area();
}

class Cuadrado : FiguraGeometrica
{
    override public void Dibuja() { ;}
    override public double Area() { return 0; }
}
```

CLASES ABSTRACTAS EN C#

Sintaxis Para Definirlas II

```
class Circulo : FiguraGeometrica
{
    override public void Dibuja() { ;}
    override public double Area() { return 0; }
}

class Triangulo : FiguraGeometrica
{
    override public void Dibuja() { ;}
    override public double Area() { return 0; }
}
```

- A las clases abstractas con todos sus métodos abstractos como en el caso de FiguraGeometrica se les denomina Clases Abstractas Puras.

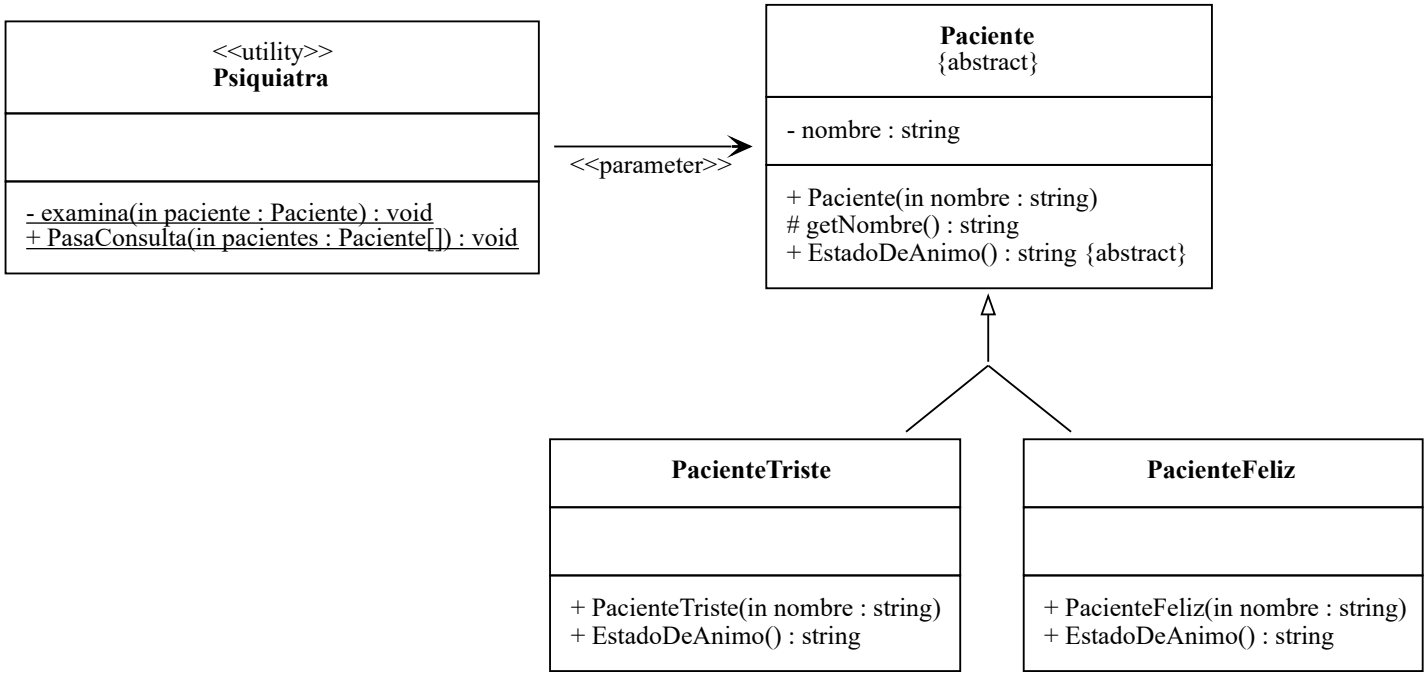
CLASES ABSTRACTAS EN C#

- Si buscamos polimorfismo de datos, lo mejor es buscar la abstracción.

Esto significa tratar de llevar la implementación hacia las clases hijas, vaciando de funcionalidad a las superclases.

- Veamos un ejemplo más de como predecir el futuro con el polimorfismo de datos y la abstracción.
- Vamos a implementar un programa que simule la consulta de un psiquiatra.

¿Qué Pasará Si Creo Una Nueva Clase De Paciente?



POO
EXCEPCIONES

EXCEPCIONES EN POO

Conceptos Básicos

- Es la forma en que los lenguajes orientados a objetos realizan el control de errores.

Frente A La Programación Estructurada Tradicional Nos Ofrecen:

- Claridad ya que evitamos la lógica adicional del caso de error.
- Evitar que un método devuelva un error como parámetro a través de la pila de llamadas como sucedía en la programación estructurada.
- Tratamiento asegurado de errores.

EXCEPCIONES EN C#

- Todas derivan de la clase System.Exception
- Existen algunas ya predefinidas.
  - https://msdn.microsoft.com/es-es/library/z4c5tckx(v=vs.110).aspx
- Podemos definir excepciones propias mediante el mecanismo de herencia.

¿Cómo Se Genera Una Excepción?

- Utilizaremos la instrucción **throw new <TipoExcepción>;**

```
public Articulo(Articulo articulo)
{
    if (articulo == null)
        throw new ArgumentNullException("articulo");

    articulo.nombre = nombre;
    articulo.precio = precio;
}
```

Pero… ¿Cómo Capturar Y Controlar Excepciones En C#?

BLOQUE TRY - CATCH

```
try
{
     <Bloque de código o módulo del que quiero controlar una excepción>
}
catch (<TipoDeLaExcepción> <parametro formal si lo voy a usar>)
{
     <Tratamiento de la excepción>
}
```

En el siguiente ejemplo capturaremos solo las excepciones de formato.

```
static void Main()
{
     try
     {
          Console.Write("Introduce un número real: ");
          double n = double.Parse(Console.ReadLine());
     }
     catch(FormatException)
     {
          Console.Write("Lo siento el numero introducido no es Real.");
     }
}
```

BLOQUE TRY - CATCH

¿Qué Pasará Si Se Produce Una Excepción Que No Es Del Tipo Que Capturamos?

```
static void Main()
{
    try
    {
        Console.WriteLine("Calculo de división... ");
        Console.Write("Introduce el numerador: ");
        double numerador = double.Parse(Console.ReadLine());
        Console.Write("Introduce el divisor: ");
        double divisor = double.Parse(Console.ReadLine());

        if (divisor == 0) throw new DivideByZeroException();

        Console.WriteLine($"La división es {numerador / divisor}");
    }
    catch (FormatException)
    {
        Console.Write("Lo siento el numero introducido no es Real.");
    }
}
```

El CLR captura la excepción por nosotros y finalizará la ejecución.

BLOQUE TRY - CATCH

Podremos Capturar Más De Una Excepción En El Mismo Bloque Try

- Lo haremos añadiendo bloques catch consecutivos.

```
try
{
 …

    if (divisor == 0) throw new DivideByZeroException();

    …
}
catch (FormatException)
{
    Console.Write("Lo siento, el numero introducido no es Real.");
}
catch (DivideByZeroException)
{
    Console.Write("Lo siento, no puedo dividir por cero.");
}
```

- Pero CUIDADO!! si el tipo del primer bloque catch es una superclase del segundo. El segundo bloque catch nunca se ejecutará.

BLOQUE TRY - CATCH

Caso Del Mal Uso Del Bloque Try-catch

- El siguiente código producirá un error de compilación.

- Una vez entra en un catch asociado a un try… ya no entra en los otros.

```
try {
     …
     if (divisor == 0) throw new DivideByZeroException();
     …
}
// No debemos capturar esta superclase.
// Ya lo hace el sistema por nosotros.
catch (Exception) {
     Console.Write("Algo malo ha pasado.");
}
// Estos bloques catch nunca se ejecutarán.
catch (FormatException) {
    Console.Write("Lo siento el numero introducido no es Real.");
}
catch (DivideByZeroException) {
    Console.Write("No puedo dividir por cero.");
}
```

BLOQUE TRY - CATCH - FINALLY

Liberación De Recursos

- El bloque finally es opcional y se añade tras todos los bloques catch.
- Se usará para liberar recursos usados dentro del bloque try.

- El bloque **finally** se ejecutará tanto si ha ido bien el bloque try, como si ha entrado por alguno de los bloques catch.

Fijémonos En El Siguiente Ejemplo.

```
public static void LeeFichero(int posicion)
{
    string ruta = @"c:\test.txt";
    StreamReader streamReader = new StreamReader(ruta);
    char[] buffer = new char[10];

    try {
        streamReader.ReadBlock(buffer, posicion, buffer.Length);
    }
    catch (FileNotFoundException) {
        Console.WriteLine($"El fichero {ruta} no existe.");
    }
```

BLOQUE TRY - CATCH – FINALLY (CONTINUACIÓN)

```
        catch (IOException e) {
            Console.WriteLine($"Error leyendo de {ruta}. {e.Message}");
        }
        finally {
            // En el caso de que el fichero no exista streamReader
            // será null y no deberemos borrarlo.
            // Si el fichero no existe, pero la posición
            // donde leemos es inválida
            // deberemos cerrarlo o de lo contrario se
            // quedará abierto por nuestro programa.

            if (streamReader != null)
            {
                streamReader.Close();
            }
        }
}
```

- Se puede tener un bloque **try** seguido de un **finally** (sin bloque catch), y realizar la captura de la excepción en otro ámbito más externo.

- El finally se ejecutará siempre y antes que el catch del ámbito superior.

ENCADENAR LANZAMIENTOS

- Puedo crear un bloque catch para capturar una excepción en un ámbito y añadir un mensaje específico para ese ámbito y posteriormente relanzarla para ser capturada en otro ámbito.
- Este proceso se puede repetir de forma sucesiva.

```
try {
    // ámbito 1
}
catch (ArgumentException e) {
    throw new ArgumentException("mensaje específico en ámbito 1", e);
}
```

- La mayoría de constructores de excepciones de las BCL, admiten una sobre carga con el parámetro Exception innerException. (a NULL por defecto).
- Este me permitirá recorrer todos los objetos excepción en el orden que se han ido relanzando y así acceder a mensajes específicos en cada ámbito.

https://docs.microsoft.com/es-es/dotnet/api/system.exception.innerexception?view=netframework-4.7.1

EJEMPLO AVANZADO DE ANIDACIÓN DE GESTIÓN EXCEPCIONES

```
try {    // bloque try 1
     try {   // bloque try 2
          …
          try {   // bloque try 3
               …
          }
          catch(E1) {
               throw new E1("Mensaje específico bloque 3", E1);
               // Capturo la excepción y la relanzo para que  sea
          }   // gestionada del bloque 3 por alguno de los otros bloques.
          …
     }
     catch(E1) {
          throw new E1("Mensaje específico bloque 2", E1);
     }
     catch(E2) { // No puede ser E1 ni una subclase de E1
          …
     }
     finally { // Se ejecuta siempre aunque relance E1
          …
     }
}
catch(E1) { // Aquí puedo acceder a la última E1 generada y a través
     …      // de InnerException ir recorriendo los mensajes específicos
}           // añadidos en cada bloque.
```

CREANDO NUESTRAS PROPIAS EXCEPCIONES

Deberemos De Heredar De System.Exception Como El Resto.

```
class ExcepcionDepartamento : Exception
{
    public ExcepcionDepartamento(string message) : base (message)
    {
        ;
    }
     …
}
```

- En el código de ejemplo hemos creado una excepción **ExcepcionDepartamento** que usaré para saber cuando no he controlado algo en los departamentos de mi compañía.
- Supongamos un método para imprimir nóminas de un departamento, donde no hemos contemplado uno de reciente creación.

CREANDO NUESTRAS PROPIAS EXCEPCIONES

```
class Ejemplo {
    enum Departamentos { Contable, Desarrollo, Marketing };
    static void ImprimeNominas(Departamentos departamento) {
        switch (departamento) {
            case Departamentos.Contable:
                Console.WriteLine(" Imprimiendo nóminas contabilidad.");
                break;
            case Departamentos.Desarrollo:
                Console.WriteLine("Imprimiendo nóminas contabilidad.");
                break;
            default: throw new ExcepcionDepartamento(
                            $"No se pueden imprimir nóminas de "+
                            $"este departamento de {departamento}.");
        }
    }
    …
    static void Main() {
        try {
            ImprimeNominas(Departamentos.Marketing);
        }
        catch (ExcepcionDepartamento e) {
            Console.WriteLine(e.Message);
        }
    }
}
```

JERARQUÍA DE EXCEPCIONES EN .NET

Ejemplo De Jerarquía De Las Excepciones Ya Definidas Más Usadas

- Fíjate que algunas no son recomendables capturarlas, lanzarlas o derivarlas.
- Recomendaciones de uso de excepciones estándar: https://msdn.microsoft.com/es-es/library/ms229007(v=vs.110).aspx
- Prácticas de uso recomendadas. https://msdn.microsoft.com/es-es/library/ms229030(v=vs.110).aspx