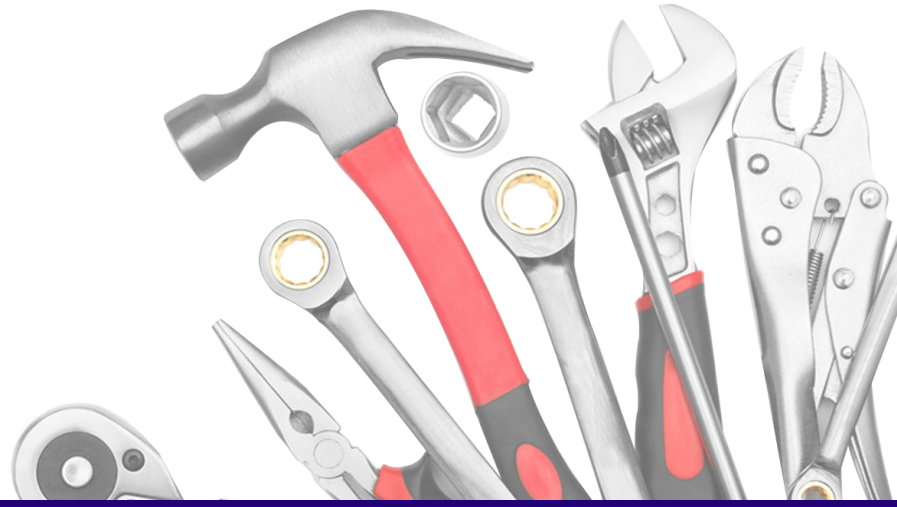


CFGS APLICACIONES MULTIPLATAFORMA - PROGRAMACIÓN

UNIDAD 7

USO AVANZADO DE CLASES

DIEGO VALERO ARÉVALO
BASADO EN APUNTES Y EJERCICIOS PROPORCIONADOS POR
Mª CARMEN DÍAZ GONZÁLEZ - IES VIRGEN DE LA PALOMA

U7 - **USO AVANZADO DE CLASES**
ÍNDICE

7.1 - COMPOSICIÓN

La **composición** es el **agrupamiento de uno o varios objetos y valores dentro de una clase**. La composición crea una relación de tipo **'tiene'** o **'está compuesto por'**.

EJEMPLO

Rectangulo.java

```java
public class Rectangulo{
    Punto p1;
    Punto p2;
}
```

PersonaMain.java

```java
public class Punto{
   int x;
   int y;
}
```

En este ejemplo, podríamos formar un objeto **`Rectángulo`** pasando por parámetros dos objetos de tipo **`Punto`** o cuatro **`int`**.

EJEMPLO

Cuenta.java

```java
public class Cuenta{
    Persona titular;
    Persona autorizado;
    double saldo;
    Movimiento movimientos[];
}
```

Persona.java

```java
public class Persona{
   String nombre;
   String dni;
}
```

Movimiento.java

```java
public class Movimiento{
   int tipo;
   Date fecha;
   double cantidad;
   String concepto;
}
```

Como vemos, una **`Cuenta`** tiene asociados **`Personas`** y **`Movimientos`**.

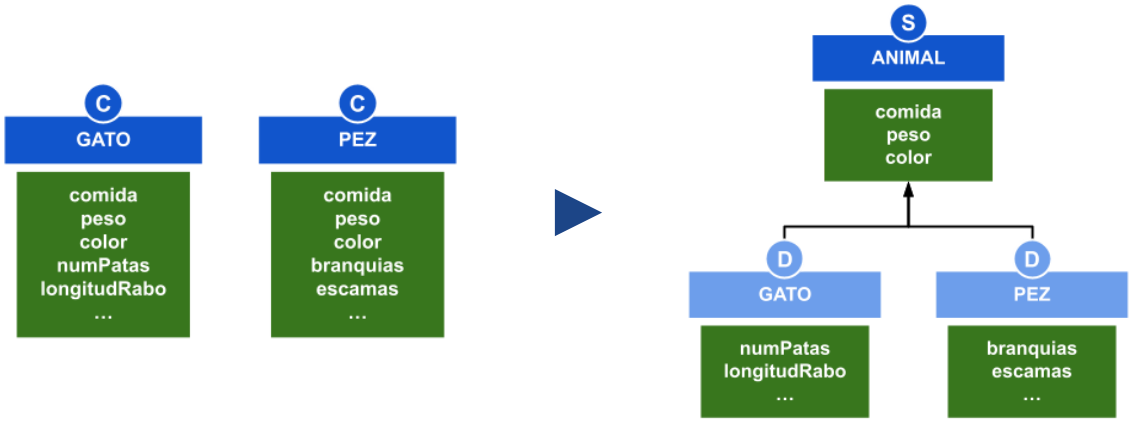
La composición de clases permite **hacer que un conjunto de clases colaboren entre sí**: Cada clase **se especializa en una tarea concreta** y esto permite dividir un problema complejo en varios sub-problemas pequeños. También facilita la **modularidad** y **reutilización del código**.

Dentro de la composición de clases veremos:

Herencia	Polimorfismo	Abstracción	Interfaces

7.2 - HERENCIA

La **herencia** es una de las capacidades más importantes y distintivas de la POO. Consiste en **derivar o extender una clase nueva a partir de otra ya existente** de forma que la clase nueva **hereda todos los atributos y métodos de la clase ya existente**. Veamos un ejemplo:



Si tenemos dos clases que tienen **atributos y/o métodos comunes**…

…lo mejor es **crear una nueva clase con ellos** y hacer que las **originales extiendan de esta** para **heredar** estos atributos.

Al hacer esto, decimos que una clase **hereda o extiende** de otra, en orden de que la clase **hija** o **derivada** (**D**) coge los atributos de la **superclase**, **base** o **padre** (**S**). Esto lo hemos visto ya por ejemplo al crear excepciones personalizadas cuando extendemos de **`Exception`**.

>> 7.2.1- EXTENDER DE OTRA CLASE

Para crear una **clase hija que refiera a una clase padre**, añadimos **`extends`** y el **nombre de la padre** en su cabecera.

```
public class NombreClase extends NombreSuperclase {
    ...
}
```

Una clase **sólo puede heredar de una única superclase**. Para heredar de varias, deberíamos **anidar las clases**.

>> 7.2.2- ACCESO A MIEMBROS HEREDADOS

Si recuerdas los tipos de accesos a las clases de la **unidad 3 - *Programación Orientada a Objetos* (apartado 3.1.3)**, tenemos **tres palabras para especificar y restringir** el uso de cada atributo y método:

`public`	`private`	`protected`
Se puede acceder **desde cualquier clase**.	Sólo se puede acceder **desde la misma clase**.	Se puede acceder **desde la propia clase**, **desde las subclases que heredan de ella** y **desde aquellas que estén en el mismo paquete**.

Nuestra forma de trabajar es declarar todos los atributos como **`private`**, pero esto restringe el acceso desde las clases que heredan de la misma, y no deberíamos hacerlos **`public`** porque no queremos que cualquiera pueda acceder. Para encontrar ese punto medio de acceso tenemos la palabra **`protected`**.

>> 7.2.3- ACCEDER A LOS CONSTRUCTORES DE LA SUPERCLASE

Si la **clase padre** tiene un **constructor**, **debemos usarlo** para manejar la información común y además añadirle los atributos de la clase hija. Para ello, lo llamaremos en el **constructor** de la **clase hija** con el llamador **`super()`**.

El llamador **`super()`** se utiliza para **acceder a las características del padre**. Podemos llamar tanto a **atributos** como a **métodos** de este.

Podemos hacerlo de dos maneras:

EJEMPLO

Electrodomestico.java

```java
public class Electrodomestico{

    protected String nombre;
    protected int precio;

    public Electrodomestico(String nombre, int precio){
        this.nombre = nombre;
        this.precio = precio;
    }
}
```

LavadoraA.java

```java
public class LavadoraA extends Electrodomestico{

    protected int carga;

    public LavadoraA(String nombre, int precio, int carga){
        super.nombre = nombre;
        super.precio = precio;
        this.carga = carga;
    }
}
```

LavadoraB.java

```java
public class LavadoraB extends Electrodomestico{

    protected int carga;

    public LavadoraB(String nombre, int precio, int carga){
        super(nombre, precio);
        this.carga = carga;
    }
}
```

Podemos llamar a los atributos del constructor de la clase padre nombrando **cada atributo** (**`lavadoraA`**) o **accediendo a través del mismo constructor** (**`lavadoraB`**) como vimos en la **unidad 3 - *Programación Orientada a Objetos (apartado 3.1.4)*** (la manera preferida y recomendada).

Si una clase padre **tiene un constructor** y **no lo llamamos con `super()` en la clase hija**, el compilador **llamará al que tenga por defecto**, y si no existe, dará **error**.

>> 7.2.4- HERENCIA Y SOBREESCRITURA: ACCEDER A LOS ATRIBUTOS Y MÉTODOS DE LA SUPERCLASE

Si queremos **usar un método de la clase padre** pero necesitamos adaptarlo a las **necesidades de la clase hija**, podemos hacerlo perfectamente **llamando al mismo método** y sobreescribirlo usando `@Override`.

Un método está **sobreescrito** o **reimplementado** cuando **se programa de nuevo en la clase hija**.

EJEMPLO

Excepciones.java

```
 1  public class Electrodomestico{
 2
 3     protected String nombre;
 4     protected int precio;
 5     public static final double IVA = 1.21;
 6
 7     public Electrodomestico(String nombre, int precio){
 8        this.nombre = nombre;
 9        this.precio = precio;
10     }
11
12     public double getPrecio(){
13        return precio;
14     }
15
16     public double calcularIva(){
17        return getPrecio()*IVA;
18     }
19  }
```

Lavadora.java

```
 1  public class Lavadora extends Electrodomestico{
 2
 3     protected int carga;
 4     public static final double SOBRECARGO = 15;
 5
 6     public Lavadora(String nombre, int precio, int carga){
 7        super(nombre, precio);
 8        this.carga = carga;
 9     }
10
11     @Override
▲12    public double calcularIva(){
13        return (getPrecio()+SOBRECARGO)*IVA;
14     }
15  }
```

Aquí por ejemplo hemos decidido que las lavadoras tendrán un sobrecargo de 15€ sobre el precio, del cual luego hay que calcular el IVA. Como ya hay un método para calcularlo en el padre (`Electrodomestico`), lo sobreescribimos en `Lavadora`. Observa que podemos acceder también a los métodos del padre, como `getPrecio()`.

Podrás ver que donde hay un **método sobreescrito** le acompañará en la columna de números de línea un **triángulo verde** (▲).

La sobreescritura de métodos heredados constituye la base del **polimorfismo** y la **selección dinámica de métodos**.

>> 7.2.5- CLASES Y MÉTODOS `FINAL`

Debemos tener muy en cuenta que:

> **No podemos heredar clases ni sobreescribir métodos** definidos como **`final`**.

7.3 - POLIMORFISMO

El **polimorfismo** consiste en que una clase pueda **adaptar** un método heredado a sus propias necesidades sin que haya que cambiar nada de la clase padre. Para ello usamos la **sobreescritura** (que ya hemos visto en el apartado **7.2.4** de esta **unidad**), que constituye la base de uno de los conceptos más potentes de Java: la **selección dinámica de métodos**:

SELECCIÓN DINÁMICA DE MÉTODOS

Es un mecanismo mediante el cual **la llamada a un método sobreescrito** se resuelve en **tiempo de ejecución** y **no durante la compilación**. Esto significa que **una variable de referencia a una superclase** se puede referir a **un objeto de una subclase**. Java se basa en esto para resolver llamadas a métodos sobreescritos en el **tiempo de ejecución**.

Lo que determina la **versión del método que será ejecutado** es el **tipo de objeto** al que se hace referencia, y no **el tipo de variable** de referencia.

De esta manera, combinando la herencia y la sobreescritura de métodos, una superclase puede **definir la forma general de los métodos que se usarán en todas sus subclases**.

>> 7.3.1- COMPROBAR SI UNA CLASE ES HIJA DE OTRA

Para saber si un determinado objeto es de un tipo podremos hacer uso del método **`instanceOf`**.

```
objeto instanceOf Clase
```

Este devuelve un booleano. Un objeto hijo será **instancia** tanto del **propio hijo** como del **padre**.

EJEMPLO

Excepciones.java

```
1  public class Padre{
2     public double llamada(){
3        System.out.println("Has llamado al padre");
4     }
5  }
```

Lavadora.java

```
1  public class Hija1 extends Padre{
2     @Override
▲3    public double llamada(){
4        System.out.println("Has llamado a la hija1");
5     }
6  }
```

Lavadora.java

```
1  public class Hija2 extends Padre{
```

```java
    @Override
    public double llamada(){
        System.out.println("Has llamado a la hija2");
    }
}
```

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Padre padre = new Padre();
        Hija1 hija1 = new Hija1();
        Hija2 hija2 = new Hija2();
        Padre padrePruebas;

        System.out.println("Asignación del padre:");
        padrePruebas = padre;
        padrePruebas.llamada();
        comprobarInstancia(padrePruebas);

        System.out.println("\nAsignación de la hija1:");
        padrePruebas = hija1;
        padrePruebas.llamada();
        comprobarInstancia(padrePruebas);

        System.out.println("\nAsignación de la hija2:");
        padrePruebas = hija2;
        padrePruebas.llamada();
        comprobarInstancia(padrePruebas);
    }

    public static void comprobarInstancia(Object padrePruebas) {
        if(padrePruebas instanceOf Hija2) {
            System.out.println("padrePruebas es instancia de Hija2");
        }
        else if(padrePruebas instanceOf Hija1) {
            System.out.println("padrePruebas es instancia de Hija1");
        }
        else if(padrePruebas instanceOf Padre) {
            System.out.println("padrePruebas es instancia de Padre");
        }
    }
}
```

Console

```
Asignación del padre:
Has llamado al padre
padre2 es instancia de Padre

Asignación de la hija1:
Has llamado a la hija1
padre2 es instancia de Hija1

Asignación de la hija2:
Has llamado a la hija2
padre2 es instancia de Hija2
```

En este ejemplo vemos cómo tenemos una clase padre y otras dos que extienden de la misma. En el **`main`** creamos varios objetos con cada una y otro más de tipo **`Padre`**, el cual usaremos para ir asignando los otros objetos e ir comprobando sus instancias, que puedes ir viendo en la salida por consola. Aquí podemos comprobar perfectamente cómo actúa la herencia y el polimorfismo.

7.4 - ABSTRACCIÓN

La **abstracción** se da cuando en una clase declaramos **métodos sin contenido**, es decir, que tenemos **sólo su cabecera**. De esta manera creamos **métodos abstractos**, que hacen que inmediatamente la clase en la que están deba declararse como **abstracta**, la cual **no permite crear objetos de ese tipo** pero sí heredar a partir de ella. Las clases que hereden deberán **implementar obligatoriamente los métodos abstractos** de la clase padre.

Una **clase abstracta** puede contener tanto **métodos abstractos** (por lo menos uno) como **no abstractos**. Para declarar ambos se usa el modificador `abstract`:

```
public abstract class nombreClase{...}
```

```
public abstract void/tipo nombreMetodo();
```

Las clases abstractas son útiles cuando necesitamos definir **una forma generalizada de clase** que será compartida por las subclases, dejando **parte del código en la clase abstracta** (métodos "**normales**") y **delegando otra parte en las subclases** (métodos **abstractos**).

No se pueden declarar como abstractos ni los **constructores** ni los **métodos** `static`.

La **abstracción** va directamente relacionada con la herencia y es **parecida a sobreescribir métodos**, pero la finalidad es tener **métodos comunes que el padre no necesita implementar** pero que son **útiles para las subclases**, donde cada una los deberá implementar a su manera.

EJEMPLO

Animal.java

```
1  public abstract class Animal{
2      public abstract void emiteSonido();
3  }
```

Gato.java

```
1  public class Gato extends Animal{
2      @Override
3      public void emiteSonido(){
4          System.out.println("Miau");
5      }
6  }
```

Perro.java

```
1  public class Perro extends Animal{
2      @Override
3      public void emiteSonido(){
4          System.out.println("Guau");
5      }
6  }
```

Main.java

```
1  public class Main {
2      public static void main(String[] args) {
3          Gato gato = new Gato();
4          Perro perro = new Perro();
5
6          gato.emiteSonido();
7          perro.emiteSonido();
8
9      }
10 }
```

Console

```
Miau
Guau
```

Aquí puedes ver que hay dos clases que heredan de **`Animal`** e implementan el método abstracto **`emiteSonido()`**. Como es un método que puede **ser útil** para cualquier subclase de la misma pero el padre no la necesita, se declara como **abstracto**. Cada clase la implementará como mejor le convenga.

Los **métodos abstractos implementados** se señalan en su línea con un **triángulo morado** (▲).

7.5 - INTERFACES

Imagina una clase donde todos los atributos y métodos sean abstractos y que pudiese servir como **plantilla para otras clases**. Pues esto es lo que se conoce como **interfaz**, que son aquellas clases que **definen la estructura** de la misma pero **no cómo se implementan**.

Mediante la construcción de un interfaz se pretende **especificar qué caracteriza a una colección de objetos** e, igualmente, **especificar el comportamiento** que deben reunir los objetos que quieran entrar dentro de esa categoría o colección.

Para facilitar la creación de interfaces, dejamos a un lado el especificador **`abstract`** y sustituimos **`class`** por **`interface`**.

```
public interface nombreInterfaz{
        ...
}
```

Es verdad que la implementación (herencia) de una interfaz **no evita el código duplicado ni favorece la reutilización del mismo**, porque directamente no traen código, pero las interfaces juegan un papel muy importante a la hora de aplicar el **polimorfismo** a una jerarquía de clases.

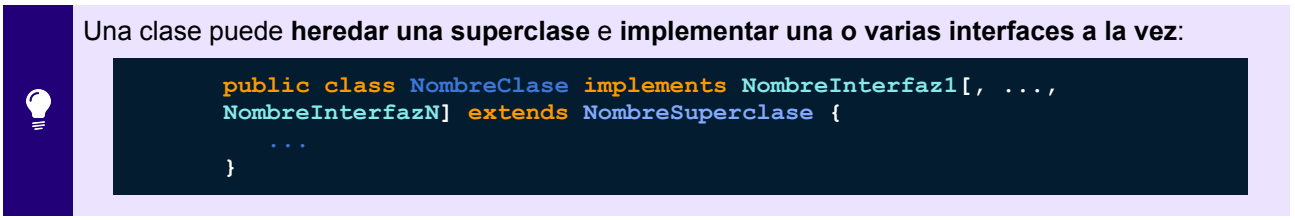
Las interfaces **sustituyen a las clases abstractas** porque así las subclases pueden **implementar más de una**, al contrario que la herencia donde se **restringe sólo a una superclase**. Además, esto favorece la separación de la **especificación de una clase (qué hace)** de la **implementación (cómo lo hace)**. Esto se ha comprobado que da lugar a programas más robustos y con menos errores.

Al crear interfaces debemos tener en cuenta que:

- Una interfaz **no puede instanciar objetos**, solo sirve para **implementar clases**.
- Una clase puede **implementar varias interfaces** (separadas por comas).
- Una clase que **implementa una interfaz** debe **implementar todos y cada uno de los métodos definidos en la interfaz**.
- Las clases que implementan una **interfaz que tiene definidas constantes** pueden usarlas en cualquier parte del código de la clase, simplemente indicando su nombre.

Para que una clase pueda utilizar una interfaz, al igual que en la herencia donde usábamos la palabra reservada **`extends`** para indicar el padre, aquí usamos la palabra reservada **`implements`** seguida **del o los nombre/s de la/s interfaz/ces**.

```
public class NombreClase implements NombreInterfaz1[, ..., NombreInterfazN] {
    ...
}
```



Una clase puede **heredar una superclase** e **implementar una o varias interfaces a la vez**:

```
public class NombreClase implements NombreInterfaz1[, ...,
NombreInterfazN] extends NombreSuperclase {
    ...
}
```

U7 - **BATERÍA DE EJERCICIOS**

Se debe realizar gestión de errores y excepciones siempre que sea necesario en todos los ejercicios.

Línea	Utilizando la clase Punto creada en la unidad 3, crear por composición la clase Linea (que estará compuesta de dos puntos, el de inicio y el de fin. Deberá poder crearse una línea a partir de dos puntos, o 4 coordenadas (sobrecarga de constructores). Además será necesario un método que indique la longitud de la línea y otro que muestre las coordenadas de la misma.
Rectángulo	Utilizando la clase Punto creada en la unidad 3, modificar la clase Rectangulo de la misma unidad, para que un Rectangulo esté compuesto por dos puntos. Si los métodos de calcular área y perímetro no están dentro de la clase Rectangulo, modificarla y moverlos.
Mascotas	Crear un programa para gestionar el inventario de una tienda de mascotas. De cada mascota se deberá guardar nombre y fechaNacimiento. Habrá 3 tipos de mascotas: perros, gatos y loros. La configuración de las clases será la siguiente: • De los perros se necesitan los atributos raza y pulgas. Y deberá tener un método que sea emiteSonido que indicará el sonido que hace el perro. Además será posible mostrar todas sus características • De los gatos se necesitan los atributos peloLargo y color. Y deberá tener un método que sea emiteSonido. Además será posible mostrar todas sus características • De los loros se necesitan los atributos origen y habla. Y deberá tener un método que sea emiteSonido, otro que sea volar, y otro saluda. Además será posible mostrar todas sus características. Crear una clase principal en la que se creen distintas instancias de las mascotas y se llamen a sus distintos métodos.
Biblioteca	Escribe un programa para una biblioteca que contenga libros y revistas. Las características comunes que se almacenan tanto para las revistas como para los libros son el código, el título, el año de publicación y el número de páginas. • Los libros tienen además un atributo autor. • Las revistas tienen además un número de revista. • Tanto las revistas como los libros deben tener (aparte de los constructores) un método que devuelve el valor de todos los atributos en una cadena de caracteres (para poder mostrar por pantalla, pero no se deberá hacer un print en la clase). También tienen un método que devuelve el año de publicación, otro el código y otro el número de páginas.
Electrodomésticos	Crearemos una supeclase llamada Electrodomestico con las siguientes características: • Sus atributos son precio base, color, consumo energético (letras entre A y F) y peso. Indica que se podrán heredar. • Por defecto, el color será blanco, el consumo energético será F, el precioBase es de 100 € y el peso de 5 kg. Usa constantes para ello. • Los colores disponibles son blanco, negro, rojo, azul y gris. No importa si el nombre está en mayúsculas o en minúsculas. • Los constructores que se implementarán serán ◦ Un constructor por defecto. ◦ Un constructor con el precio y peso. El resto por defecto. ◦ Un constructor con todos los atributos. • Los métodos que implementara serán: ◦ Métodos get de todos los atributos. ◦ comprobarConsumoEnergetico(char letra): comprueba que la letra es correcta, sino es correcta usará la letra por defecto. Se invocará al crear el objeto y no será visible.

◦ comprobarColor(String color): comprueba que el color es correcto, sino lo es usa el color por defecto. Se invocará al crear el objeto y no será visible.
◦ precioFinal(): según el consumo energético, aumentará su precio, y según su tamaño, también. Esta es la lista de precios:

Letra	Precio
A	100 €
B	80 €
C	60 €
D	50 €
E	30 €
F	10 €

Tamaño	Precio
Entre 0 y 19 kg	10 €
Entre 20 y 49 kg	50 €
Entre 50 y 79 kg	80 €
Mayor que 80 kg	100 €

Crearemos una subclase llamada **Lavadora** con las siguientes características:
• Su atributo es carga, además de los atributos heredados.
• Por defecto, la carga es de 5 kg. Usa una constante para ello.
• Los constructores que se implementarán serán:
◦ Un constructor por defecto.
◦ Un constructor con el precio y peso. El resto por defecto.
◦ Un constructor con la carga y el resto de atributos heredados. Recuerda que debes llamar al constructor de la clase padre.
• Los métodos que se implementara serán:
◦ Método get de carga.
◦ precioFinal():, si tiene una carga mayor de 30 kg, aumentará el precio 50 €, sino es así no se incrementara el precio. Llama al método padre y añade el código necesario. Recuerda que las condiciones que hemos visto en la clase Electrodomestico también deben afectar al precio.

Crearemos una subclase llamada **Television** con las siguientes características:
• Sus atributos son resolución (en pulgadas) y sintonizador TDT (booleano), además de los atributos heredados.
• Por defecto, la resolución será de 20 pulgadas y el sintonizador será false.
• Los constructores que se implementarán serán:
◦ Un constructor por defecto.
◦ Un constructor con el precio y peso. El resto por defecto.
◦ Un constructor con la resolución, sintonizador TDT y el resto de atributos heredados.
Recuerda que debes llamar al constructor de la clase padre.
• Los métodos que se implementara serán:
◦ Método get de resolución y sintonizador TDT.
◦ precioFinal(): si tiene una resolución mayor de 40 pulgadas, se incrementara el precio un 30% y si tiene un sintonizador TDT incorporado, aumentará 50 €. Recuerda que las condiciones que hemos visto en la clase Electrodomestico también deben afectar al precio.

Ahora crea una clase ejecutable que realice lo siguiente:
• Crea un array de Electrodomesticos de 10 posiciones.
• Asigna a cada posición un objeto de las clases anteriores con los valores que desees.
• Ahora, recorre este array y ejecuta el método precioFinal().
• Deberás mostrar el precio de cada clase, es decir, el precio de todas las televisiones por un lado, el de las lavadoras por otro y la suma de los Electrodomesticos (puedes crear objetos Electrodomestico, pero recuerda que Television y Lavadora también son electrodomésticos). Recuerda el uso operador instanceof.

	Por ejemplo, si tenemos un Electrodomestico con un precio final de 300, una lavadora de 200 y una televisión de 500, el resultado final sera de 1000 (300+200+500) para electrodomésticos, 200 para lavadora y 500 para televisión.
Figura	Crearemos clase abstracta Figura, que deberá tener dos métodos abstractos, calculaPerimetro() y calculaArea(). Crea las clases Triángulo, Rectángulo y Círculo de forma que hereden de la clase abstracta. Añade los atributos y métodos que consideres necesarios para definir cada una de las figuras. (Se puede utilizar composición utilizando la clase Punto que ya tenemos creada).
Mascota 2	Modificar el ejercicio 3, de forma que el método emiteSonido() sea un método abstracto de la clase Mascota.
Banco	Vamos a hacer una aplicación que simule el funcionamiento de un banco. Crea una clase CuentaBancaria con los atributos: iban y saldo. Implementa métodos para: ◦ Consultar los atributos. ◦ Ingresar dinero. ◦ Retirar dinero. ◦ Traspasar dinero de una cuenta a otra. También habrá un atributo común a todas las instancias llamado interesAnualBasico, que en principio puede ser constante. La clase tiene que ser abstracta y debe tener un método calcularIntereses() que se dejará sin implementar. También puede ser útil implementar un método para mostrar los datos de la cuenta. De esta clase heredarán dos subclases: CuentaCorriente y CuentaAhorro. La diferencia entre ambas será la manera de calcular los intereses: • A la primera se le incrementará el saldo teniendo en cuenta el interés anual básico. • La segunda tendrá una constante de clase llamada saldoMinimo. Si no se llega a este saldo el interés será la mitad del interés básico. Si se supera el saldo mínimo el interés aplicado será el doble del interés anual básico. Implementa una clase principal para probar el funcionamiento de las tres clases: Crea varias cuentas bancarias de distintos tipos, pueden estar en un array si lo deseas; prueba a realizar ingresos, retiradas y transferencias; calcula los intereses y muéstralos por pantalla; etc.
Empresa	Vamos a implementar dos clases que permitan gestionar datos de empresas y sus empleados. Los empleados tienen las siguientes características: • Nombre, DNI, sueldo bruto (mensual), edad, teléfono y dirección. • El nombre y DNI de un empleado no pueden variar. • Es obligatorio que todos los empleados tengan al menos definido su nombre, DNI y el sueldo bruto. Los demás datos no son obligatorios. • Será necesario un método para imprimir por pantalla la información de un empleado. • Será necesario un método para calcular el sueldo neto de un empleado. El sueldo neto se calcula descontando del sueldo bruto un porcentaje que depende del IRPF. El porcentaje del IRPF depende del sueldo bruto anual del empleado (sueldo bruto x 12 pagas).(*) **Sueldo bruto anual** — **IRPF** Inferior a 12.000 € — 20% De 12.000 a 25.000 € — 30% Más de 25.000 € — 40%

	Por ejemplo, un empleado con un sueldo bruto anual de 17.000 € tendrá un 30% de IRPF. Para calcular su sueldo neto mensual se descontará un 30% a su sueldo bruto mensual. Las empresas tienen las siguientes características: • Una empresa tiene nombre y CIF (datos que no pueden variar), además de teléfono, dirección y empleados. Cuando se crea una nueva empresa esta carece de empleados. • Serán necesarios métodos para: ◦ Añadir y eliminar empleados a la empresa. ◦ Mostrar por pantalla la información de todos los empleados. ◦ Mostrar por pantalla el DNI, sueldo bruto y neto de todos los empleados. ◦ Calcular la suma total de sueldos brutos de todos los empleados. ◦ Calcular la suma total de sueldos netos de todos los empleados. Implementa las clases Empleado y Empresa con los atributos oportunos, un constructor, los getters/setters oportunos y los métodos indicados. Puedes añadir más métodos si lo ves necesario. Estas clases no deben realizar ningún tipo de entrada por teclado. Implementa también una clase para realizar pruebas: Crear una o varias empresas, crear empleados, añadir y eliminar empleados a las empresas, listar todos los empleados, mostrar el total de sueldos brutos y netos, etc. (*) El IRPF realmente es más complejo pero se ha simplificado para no complicar demasiado este ejercicio.
Vehículos	Debes crear varias clases para un software de una empresa de transporte. Implementa la jerarquía de clases necesaria para cumplir los siguientes criterios: • Los vehículos de la empresa de transporte pueden ser terrestres, acuáticos y aéreos. • Los vehículos terrestres pueden ser coches y motos. Los vehículos acuáticos pueden ser barcos y submarinos. • Los vehículos aéreos pueden ser aviones y helicópteros. • Todos los vehículos tienen matrícula y modelo (datos que no pueden cambiar). La matrícula de los coches terrestres deben estar formadas por 4 números y 3 letras. La de los vehículos acuáticos por entre 3 y 10 letras. La de los vehículos aéreos por 4 letras y 6 números. • Los vehículos terrestres tienen un número de ruedas (dato que no puede cambiar). • Los vehículos acuáticos tienen eslora (dato que no puede cambiar). • Los vehículos aéreos tienen un número de asientos (dato que no puede cambiar). • Los coches pueden tener aire acondicionado o no tenerlo. • Las motos tienen un color. • Los barcos pueden tener motor o no tenerlo. • Los submarinos tienen una profundidad máxima. • Los aviones tienen un tiempo máximo de vuelo. • Los helicópteros tienen un número de hélices. No se permiten vehículos genéricos, es decir, no se deben poder instanciar objetos que sean vehículos sin más. Pero debe ser posible instanciar vehículos terrestres, acuáticos o aéreos genéricos (es decir, que no sean coches, motos, barcos, submarinos, aviones o helicópteros). El diseño debe obligar a que todas las clases de vehículos tengan un método imprimir() que imprima por pantalla la información del vehículo en una sola línea. (no utilizar el toString). Implementa todas las clases necesarias con: atributos, constructor con parámetros, getters/setters y el método imprimir. Utiliza abstracción y herencia de la forma más apropiada.

	Implementa también una clase para hacer algunas pruebas: Instancia varios vehículos de todo tipo (coches, motos, barcos, submarinos, aviones y helicópteros) así como vehículos genéricos (terrestres, acuáticos y aéreos). Crea un array y añade todos los vehículos. Recorre el array y llama al método imprimir de todos los vehículos.
Figura 2D	Implementa una interface llamada Figura2D que declare los métodos: • **double perimetro()**: Para devolver el perímetro de la figura. • **double area()**: Para devolver el área de la figura. • **void escalar(double escala)**: Para escalar la figura (aumentar o disminuir su tamaño). Solo hay que multiplicar los atributos de la figura por la escala (> 0). • **void imprimir()**: Para mostrar la información de la figura (atributos, perímetro y área) en una sola línea. Existen 4 tipos de figuras. • **Cuadrado**: Sus cuatro lados son iguales. • **Rectángulo**: Tiene ancho y alto. • **Triángulo**: Tiene ancho y alto. • **Círculo**: Tiene radio. Crea las 4 clases de figuras de modo que implementen la interfaz Figura2D. Define sus métodos. Crea una clase para realizar las siguientes pruebas: 1. Crea un array de figuras. 2. Añade figuras de varios tipos. 3. Muestra la información de todas las figuras. 4. Escala todas las figuras con escala = 2. 5. Muestra de nuevo la información de todas las figuras. 6. Escala todas las figuras con escala = 0.1. 7. Muestra de nuevo la información de todas las figuras.
Empresa agroalimentaria	Se plantea desarrollar un programa Java que permita la gestión de una empresa agroalimentaria que trabaja con tres tipos de productos: productos frescos, productos refrigerados y productos congelados. Todos los productos llevan esta información común: • nombre. • fecha de caducidad. • número de lote. A su vez, cada tipo de producto lleva alguna información específica: • Los productos frescos deben llevar: ◦ La fecha de envasado. ◦ El país de origen. • Los productos refrigerados deben llevar: ◦ El código del organismo de supervisión alimentaria. ◦ La fecha de envasado. ◦ La temperatura de mantenimiento recomendada. ◦ El país de origen. • Los productos congelados deben llevar: ◦ La fecha de envasado ◦ El país de origen ◦ Temperatura de mantenimiento recomendada. Implementar dichas clases. Después, crear un programa que gestione todos los productos de la empresa, que permita: • Añadir productos • Listar todos los productos • Listar los productos de cada tipo por separado • Modificar un producto existente • Borrar productos

• Nota: para añadir productos se deberá comprobar que no existe en la empresa (obligatorio utilizar el método equals)
• Nota 2: Controlar con excepciones todos los datos que sean necesarios, además de restringir los atributos que no se puedan ver modificados.
• Nota 3: Si se introduce algún dato de un producto mal, será necesario que se vuelvan a pedir los datos del mismo hasta que esté bien o que de la opción de no introducir más veces el producto.