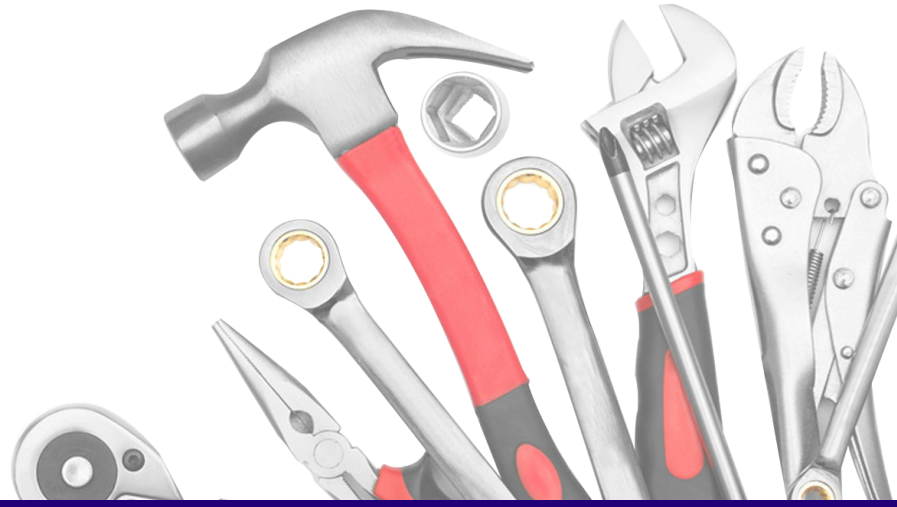


CFGS APLICACIONES MULTIPLATAFORMA - PROGRAMACIÓN

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 1

· Herencia, polimorfismo, abstracción, interfaces.

7.2 - HERENCIA 1

>> 7.2.1- EXTENDER DE OTRA CLASE 2

>> 7.2.2- ACCESO A MIEMBROS HEREDADOS 2

· public, private, protected.

>> 7.2.3- ACCEDER A LOS CONSTRUCTORES DE LA SUPERCLASE 3

>> 7.2.4- HERENCIA Y SOBREESCRITURA: ACCEDER A LOS ATRIBUTOS Y MÉTODOS DE LA SUPERCLASE 4

· public, private, protected.

>> 7.2.5- CLASES Y MÉTODOS FINAL 5

7.3 - POLIMORFISMO 5

· Selección dinámica de métodos.

>> 7.3.1- COMPROBAR SI UNA CLASE ES HIJA DE OTRA 5

7.4 - ABSTRACCIÓN 7

7.5 - INTERFACES 8

U7 - BATERÍA DE EJERCICIOS 10

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`**.

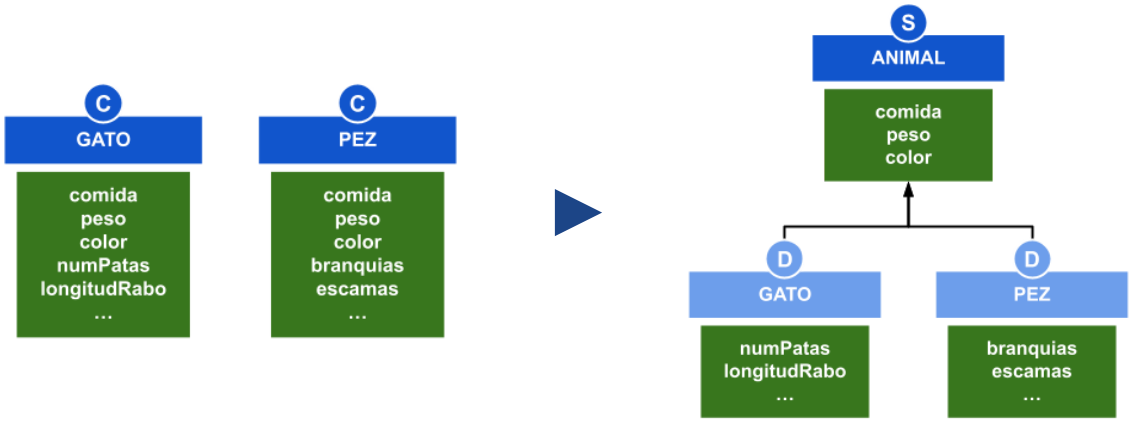
La composición de clases permite **hacer que un conjunto de clases colaboren entre sí**: Cada clase **se especializa en una tarea concreta** y esto permite dividir un problema complejo en varios sub-problemas pequeños. También facilita la **modularidad** y **reutilización del código**.

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.

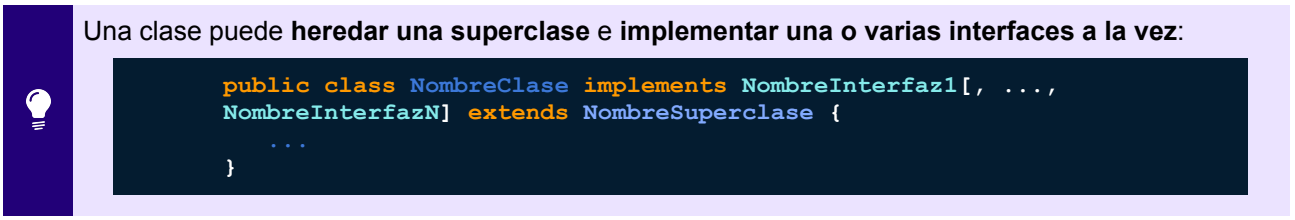
Las interfaces **sustituyen a las clases abstractas** porque así las subclases pueden **implementar más de una**, al contrario que la herencia donde se **restringe sólo a una superclase**. Además, esto favorece la separación de la **especificación de una clase (qué hace)** de la **implementación (cómo 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**

1	Implementa un programa que pida al usuario un valor entero A utilizando un nextInt() y luego muestre por pantalla el mensaje “Valor introducido: …”. Se deberá tratar la excepción InputMismatchException que lanza nextInt() cuando no se introduce un entero válido. En tal caso se mostrará el mensaje “Valor introducido incorrecto”.
2	Implementa un programa que pida dos valores int A y B utilizando un nextInt() (de Scanner), calcule A/B y muestre el resultado por pantalla. Se deberán tratar de forma independiente las dos posibles excepciones, InputMismatchException y ArithmeticException, mostrando en cada caso un mensaje de error diferente en cada caso.
3	Implementa un programa que cree un array tipo double de tamaño 5 y lo rellene con 5 valores que solicite al usuario. Tendrás que manejar la/las posibles excepciones y seguir pidiendo valores hasta rellenar completamente el vector.
4	Implementa un programa que cree un vector de enteros de tamaño N (número aleatorio entre 1 y 100) con valores aleatorios entre 1 y 10. Luego se le preguntará al usuario qué posición del vector quiere mostrar por pantalla, repitiéndose una y otra vez hasta que se introduzca un valor negativo. Maneja todas las posibles excepciones.
5	Implementa un programa con tres funciones: ◦ void imprimePositivo(int p): Imprime el valor p. Lanza una ‘Exception’ si p < 0 ◦ void imprimeNegativo(int n): Imprime el valor n. Lanza una ‘Exception’ si p >= 0 ◦ La función main para realizar pruebas. Puedes llamar a ambas funciones varias veces con distintos valores, hacer un bucle para pedir valores por teclado y pasarlos a las funciones, etc. Maneja las posibles excepciones.
6	Implementa una clase Gato con los atributos nombre y edad, un constructor con parámetros, los getters y setters, además de un método imprimir() para mostrar los datos de un gato. El nombre de un gato debe tener al menos 3 caracteres y la edad no puede ser negativa. Por ello, tanto en el constructor como en los setters, deberás comprobar que los valores sean válidos y lanzar una ‘Exception’ si no lo son. Luego, haz una clase principal con main para hacer pruebas: instancia varios objetos Gato y utiliza sus setters, probando distintos valores (algunos válidos y otros incorrectos). Maneja las excepciones