Composición de clases, herencia e interfaces.

Índice:

1. Composición

La composición de clases consiste en crear una clase nueva agrupando objetos de clases que ya existen.

Por ejemplo, a partir de la clase Vehiculo y de la clase Cliente puede crearse una nueva clase VehiculoAlquilado. Esta nueva clase relaciona cada vehículo con el cliente que lo alquila, es decir, por cada alquiler que se realice se crea un objeto de esta clase.

1.1. Ejemplo

Composición: clase VehiculoAlquilado.

```
public class Vehiculo {

    private String matricula;
    private String marca;
    private String modelo;
    private double tarifa;
    private boolean disponible;
```

```
public class Cliente {
    private int id;
    private String nombre;
```

```
public class VehiculoAlquilado {

    private Cliente cliente;
    private Vehiculo vehiculo;
    private LocalDate fechaAlquiler;
    private int numeroDias;
```

Utilización:

```
Cliente cliente = new Cliente(1,"Paco");
Vehiculo vehiculo = new Vehiculo("1200-MMC", "SEAT",
        "Ibiza", 56.2, true);

VehiculoAlquilado ve = new VehiculoAlquilado(cliente,
        vehiculo, LocalDate.now(), 4);
```

2. Herencia.

Dentro de la programación orientada a objetos se dice que una clase “extiende a otra” cuando hereda sus atributos y métodos. A la clase original se le llama **clase base** o **superclase** y a la nueva clase se llama **clase derivada o subclase.**

2.1. Extensión de clases

Para trabajar con la herencia en Java se utiliza el término “extender” o “extend” en inglés. Una subclase hereda de una superclase y además añade nuevos métodos y atributos que le dan un comportamiento diferente a la subclase.

La herencia permite la “reutilización de código”.

Para indicar que una clase hereda o extiende otra clase:

public nombre_subclase extends nombre_superclase {

}

Estudiaremos un ejemplo para entender el mecanismo de la herencia.

La idea es la de diseñar una aplicación para gestionar una empresa de alquiler de vehículos de los tipos **turismo**, **deportivo** y **furgoneta**. La clase **Vehiculo** define los atributos y métodos de todos los vehículos de la empresa (matricula, marca, modelo, tarifa, disponible), sin embargo al tener vehículos de distintos tipos podemos pensar en definir subclases para ellos. Así serán subclases turismo, deportivo y furgoneta. Todas las subclases son vehículos pero cada una tiene características propias.

- para un turismo interesa conocer el número de puertas y el tipo de cambio de marchas
- para un deportivo su cilindrada
- para una furgoneta su capacidad en kilos y su volumen en metros cúbicos.

La declaración de las clases sería (se omiten getters/setters).

```
public class Vehiculo {

    private String matricula;
    private String marca;
    private String modelo;
    private double tarifa;
    boolean disponible;

    public Vehiculo(String matricula, String marca,
            String modelo, double tarifa, boolean disponible) {
        super();
        this.matricula = matricula;
        this.marca = marca;
        this.modelo = modelo;
        this.tarifa = tarifa;
        this.disponible = disponible;
```

```java
public class Turismo extends Vehiculo {

    private int numeroPuertas;
    private String tipoCambio;

    public Turismo(String matricula, String marca,
            String modelo, double tarifa, boolean disponible, int numeroPuertas,
            String tipoCambio) {
        super(matricula, marca, modelo, tarifa, disponible);
        this.numeroPuertas = numeroPuertas;
        this.tipoCambio = tipoCambio;
    }
```

```java
public class Furgoneta extends Vehiculo {

    private int capacidad;
    private int volumen;

    public Furgoneta(String matricula, String marca,
            String modelo, double tarifa, boolean disponible, int capacidad,
            int volumen) {
        super(matricula, marca, modelo, tarifa, disponible);
        this.capacidad = capacidad;
        this.volumen = volumen;
    }
```

```java
public class Deportivo extends Vehiculo {

    private int cilindrada;

    public Deportivo(String matricula, String marca,
            String modelo, double tarifa, boolean disponible, int cilindrada) {
        super(matricula, marca, modelo, tarifa, disponible);
        this.cilindrada = cilindrada;
    }
```

Puede observarse que en el constructor de las subclases se realiza una llamada al constructor de la superclase con el código: **super(matricula, marca, modelo, color, tarifa, disponible);,** reutilizando el código generado en la superclase. Esto mismo puede hacerse en el método toString.

2.2 El operador instanceof

El operador instanceof se utiliza para determinar si un objeto es una instancia de una clase. La forma de utilizarlo sería: **if nombreObjeto instanceof nombreClase**. Si devuelve verdadero el objeto pertenece a la clase indicada. **Ojo, si preguntamos si un objeto de una clase derivada es una instancia de la clase base, el operador devolverá siempre verdadero.**

2.3. Polimorfismo

El polimorfismo (varias formas) se produce cuando mediante la herencia, una subclase recibe un método de la superclase y ese mismo método es sobreescrito en esta subclase. Se tienen por tanto varias implementaciones del mismo método, y se activarán unas u otras según sea el tipo (clase) del objeto que hace la llamada.

En el ejemplo anterior la superclase Vehiculo tiene un método toString() que es sobreescrito en las subclases. En cada subclase este método se sobreescribe llamando al toString() de la superclase "super.toString()" y añadiendo los atributos propios de la clase derivada.

```
Vehiculo v = new Vehiculo("1200-MMC", "SEAT", "Ibiza", 56.2, true);
Furgoneta f = new Furgoneta("2021-HXT", "Mercedes", "VITO", 67.2, true, 2000, 21);

System.out.println(v.toString());
System.out.println(f.toString());

//polimorfismo. Como f es una furgoneta, al llamar a toString
//se pone en marcha el de furgoneta
Vehiculo vv = f;
System.out.println(vv.toString());
```

Reglas para sobreescribir un método:

- La lista de argumentos del método debe ser exactamente la misma.
- El tipo de retorno debe de ser el mismo o un subtipo del tipo de retorno declarado originalmente.
- El nivel de acceso **no debe de ser más restrictivo**.
- El nivel de acceso **puede ser menos restrictivo**.
- Los métodos de instancia pueden ser sobreescritos solamente si han sido heredados por la subclase.
- Los métodos sobreescritos pueden arrojar cualquier excepción no verificada (de tiempo de ejecución) por el compilador.
- Los métodos sobrescritos **NO** pueden arrojar excepciones verificadas por el compilador.
- No se puede sobreescribir un método marcado como **final**.
- No se puede sobreescribir un método marcado como **estático** (**static**).
- Si un método no puede ser heredado, no puede ser sobreescrito.

2.4. Clases abstractas.

- **Una clase abstracta es una clase que puede tener algún método abstracto**.
- **Un método abstracto es aquel que no tiene código.**
- Una clase que extiende a una clase abstracta debe implementar los métodos abstractos (escribir el código) o bien volverlos a declarar como abstractos, con lo que ella misma se convierte también en clase abstracta.

La clase abstracta se declara simplemente con el modificador **abstract** en su declaración: **public abstract class NombreClase {...}**.

Los métodos abstractos se declaran también con el mismo modificador, pero sin código: **public abstract tipo_devuelto metodo();**.

La clase derivada debe declarar e implementar los métodos abstractos de la clase base. En caso contrario, el compilador genera un error indicando que no se han implementado dichos métodos abstractos y que, o bien, se implementan, o bien se declara la clase abstracta.

Pueden declararse referencias a clases abstractas pero no se pueden instanciar. Es decir, si la clase FiguraGeometrica es abstracta:

```
FiguraGeometrica f; //está permitido
FiguraGeometrica f= new FiguraGeometrica(); //provoca un error.
```

3. Compatibilidad de tipos

Una superclase es compatible con los tipos que derivan de ella, pero no al contrario. Dicho de otra forma, un objeto de la clase Turismo puede almacenar sólo turismos, sin embargo, un objeto de la clase Vehiculo puede almacenar un turismo, una furgoneta o un deportivo.

3.1. Conversión ascendente de tipos.

Si se asigna un objeto de la clase derivada a la clase base, o lo que es lo mismo, si un objeto de la clase Furgoneta se asigna a un objeto de la clase Vehiculo, Java realiza una conversión ascendente de tipos (upcasting). Convierte la Furgoneta en Vehículo. **Esta conversión siempre puede hacerse.**

3.2. Conversión descendente de tipos.

Si una instancia de la clase Vehiculo almacena una referencia a un objeto de una de sus subclases, por ejemplo una Furgoneta, es posible hacer una conversión descendente de tipos o "downcasting", o lo que es lo mismo convertir el Vehiculo en un Furgoneta.

```
Vehiculo v = new Vehiculo("1200-MMC", "SEAT", "Ibiza", 56.2, true);
Furgoneta f = new Furgoneta("2021-HXT", "Mercedes", "VITO",
        67.2, true, 2000, 21);

//conversión ascendente de tipos (upcasting)
//guardar una furgoneta en un vehiculo
v=f;

//conversión descendente de tipos (downcasting)
//guardamos v (furgoneta) en ff. Es necesario hacer casting
Furgoneta ff = (Furgoneta) v;
```

3.3. Jerarquía de herencia.

Cualquier clase Java puede ser considerada clase base para extender sus atributos y métodos. La clase derivada que se obtenga, puede a su vez, ser extendida de nuevo.

En Java todas las clases están relacionadas en una única jerarquía de herencia puesto que toda clase hereda explícitamente de otra o bien implícitamente de Object.

4. Interfaces

Podemos definir una interface como una clase abstracta pura, es decir, sus métodos no tienen implementación. Por tanto, la palabra reservada abstract no es necesaria, ni en la interfaz ni en sus métodos.

Una interface (o interfaz en español) sólo puede contener métodos sin implementar (sin código) y declaraciones de constantes que luego puedan ser utilizadas por otras clases

El objetivo es definir una plantilla (interfaz) para la construcción de clases.

La sintaxis de declaración de una interfaz es la siguiente:

```
public interface IdentificadorInterfaz {
// Cuerpo de la interfaz . . .
}
```

Una interfaz declarada como public debe ser definida en un archivo con el mismo nombre de la interfaz y con extensión .java. Las cabeceras de los métodos declarados en el cuerpo de la interfaz se separan entre sí por caracteres de punto y coma y todos son declarados implícitamente como public. Por su parte, todas las constantes incluidas en una interfaz se declaran implícitamente como public, static y final y es necesario inicializarlas en la misma sentencia de declaración.

Se dice que una clase "**implementa**" una interfaz, o lo que es lo mismo, que la clase desarrolla los métodos declarados en la interfaz. Esto se indica de la siguiente forma:

```
public class NombreClase implements NombreInterfaz {...}
```

Una clase puede implementar una o varias interfaces: en ese caso, la clase debe proporcionar la declaración y definición de todos los métodos de cada una de las interfaces o bien declararse como clase abstract.

Una interfaz hereda de otra interfaz, no la implementa.

Una interfaz no puede ser instanciada, es decir, no podemos hacer new de una interfaz.

Las interfaces son útiles para:

- implementar algo parecido a la herencia múltiple en java (con clases no es posible).
- modo de lograr el polimorfismo. Al definir interfaces permitimos la existencia de variables polimórficas y la invocación polimórfica de métodos.
- permiten declarar constantes que van a estar disponibles para todas las clases que queramos (implementando esa interfaz). Nos ahorra código evitando tener que escribir las mismas declaraciones de constantes en diferentes clases.

5. El modificador final.

La palabra reservada **“final”** puede utilizarse en varios contextos.

- Clases final.
  - No se pueden extender o dicho de otra forma, no puede heredarse de ellas.
  - Pueden instanciarse y utilizarse con normalidad.
- Métodos final.
  - En un entorno de herencia, si una clase que puede ser extendida tiene métodos **final**, estos no podrán ser sobrescritos (@Override) en las clases derivadas.
- Variables primitivas static final.
  - Representan valores constantes.
- Objetos o arrays final.
  - No puede modificarse la referencia. El IDE presenta un error.

```
8          final Persona persona = new Persona(1, "Sara");
9          persona = new Persona(2, "Elena");
```

- Argumento de función final.
  - De igual forma que antes, no podrá modificarse la referencia. El IDE presenta un error.

```
14     public static void cambioNombre(final Persona p) {
15         p = new Persona(3, "Gema");
16     }
```