Utilización avanzada de clases (II): interfaces

Contenido

Interfaces

La idea general de la POO y uno de sus principios es la **abstracción**, en el sentido de que trata de representar objetos del mundo real con modelos abstractos. Diseñar modelos consiste en centrarse en las características esenciales de los objetos e ignorar el resto. Por ejemplo, un lápiz es un objeto que usamos para dibujar. Otras propiedades, como el material del que está hecho o la longitud pueden ser importantes a veces, pero no definen la idea de lápiz.

Supongamos que tenemos que crear un programa de edición gráfica. Una de las funcionalidades básicas del programa será dibujar. Antes de dibujar el programa pide al usuario que elija una herramienta para dibujar. Puede ser un bolígrafo, un lápiz, un pincel, un rotulador, un spray, etc. Cada una de estas herramientas tiene propiedades específicas, pero hay un rasgo esencial a todas ellas: que **sirven para dibujar** (funcionalidad).

Vamos a crear una clase abstracta para que todas las herramientas de dibujo tengan un grosor y un color

```
public abstract class Herramienta {
    int grosor;
    String color;

    public Herramienta(int grosor, String color) {
        this.grosor = grosor;
        this.color = color;
    }
}
```

Ahora consideremos la **clase Lápiz**, que será una abstracción de un lápiz. Un lápiz tiene que tener un grosor y un color, pero además una serie de métodos para dibujar, como pueden ser dibujar una recta, dibujar una curva o dibujo libre:

```
public class Lapiz extends Herramienta{


    public Lapiz(int grosor, String color) {
        super(grosor, color);
    }

    public void dibujarRecta() {
        System.out.println("dibujando una recta con el lápiz");
    }

    public void dibujarCurva() {
        System.out.println("dibujando una curva con el lápiz");
    }

    public void dibujarLibre() {
        System.out.println("dibujo libre con el lápiz");
    }
}
```

Definamos clases para las demás herramientas, por ejemplo un pincel (se haría exactamente igual que el anterior ejemplo, pero sustituyendo la palabra lápiz por pincel.

Ambas clases tienen los mismos métodos (dibujarRecta, dibujarCurva, dibujarLibre), pero cada una lo implementará de forma diferente. Se supone que si una clase es una herramienta de dibujo debería implementar estos métodos.

Java permite declarar esto con las interfaces. En este caso la interfaz quedaría así:

```
public interface Dibujable {

    public void dibujarRecta();

    public void dibujarCurva();

    public void dibujarLibre();
}
```

Esta interfaz declara, sin implementar, los métodos que dibujan.

Ahora las clases que pueden dibujar llevarán añadido implementarán esta interfaz con la palabra clave `implements` Dibujable en la declaración de la clase. Si una clase implementa una interfaz, tiene que implementar todos sus métodos abstractos (aclararemos esto más adelante):

```
public class Lapiz extends Herramienta implements Dibujable{

    public Lapiz(int grosor, String color) {
        super(grosor, color);
    }

    public void dibujarRecta() {
        System.out.println("dibujando una recta con el lápiz");
    }

    public void dibujarCurva() {
        System.out.println("dibujando una curva con el lápiz");
    }

    public void dibujarLibre() {
        System.out.println("dibujo libre con el lápiz");
    }
}
```

(Lo mismo haremos con la clase Pincel, y con cualquier herramienta que queramos crear como clase).

En muchos casos es más importante saber qué puede hacer un objeto que cómo lo hace. Esta es una de las razones por las que se utilizan las interfaces.

Cómo declarar interfaces

Una interfaz puede considerarse como un tipo especial de clase que no puede ser instanciada.

Para declarar una interfaz se escribe la palabra clave `interface` (en vez de `class`) antes del nombre de la interfaz:

```
interface Interfaz { }
```

Una interfaz puede contener:

- Constantes (`public static final`); **Las variables declaradas en una interfaz no son variables de instancia**. En cambio, son implícitamente *public*, *final*, y *static*, y deben inicializarse. Por lo tanto, son esencialmente constantes.
- Métodos abstractos sin implementación (aquí la palabra clave `abstract` no es necesaria). Se indica solamente la signatura.
- Métodos por defecto con implementación (llevan la palabra clave `default`).
- Métodos privados con implementación (llevan la palabra clave `private`).
- Métodos estáticos con implementación (llevan la palabra clave `static`).

Una interfaz no puede contener atributos (sólo constantes) ni constructores. Vamos a declarar una interfaz que contenga todos los posibles miembros:

```
interface Interfaz {

   int CONSTANTE = 0; /* esto es una constante, igual que si
pusiera public static final int CONSTANTE = 0 */

   void MétodoDeInstancia1(); // método abstracto sin implementación

   void MétodoDeInstancia2(); // método abstracto sin implementación

   static void métodoEstático() { // método estático implementado
        System.out.println("Interfaz: Método estatico");
    }

   default void métodoPorDefecto() { //método por defecto implementado
        System.out.println("Interfaz: Método por defecto.");
    }

    private void métodoPrivado() { //método privado implementado
        System.out.println("Interfaz: Método privado");
    }
}
```

Los métodos estáticos, por defecto y privados deben tener implementación en la interfaz.

Cómo implementar interfaces

Una clase puede implementar una interfaz utilizando la palabra clave implements. Cualquier clase que implemente una interfaz tiene que implementar todos los métodos **abstractos** de la interfaz.

Vamos a implementar la anterior interfaz.

```
class Clase implements Interfaz {

    @Override
    public void MétodoDeInstancia1() {
        System.out.println("Clase: método de instancia 1");
    }

    @Override
    public void MétodoDeInstancia2() {
        System.out.println("Clase: método de instancia 2");
    }
}
```

Ahora podemos crear una instancia de la clase y llamar a sus métodos.

```
Class instancia = new Clase();
instancia.MétodoDeInstancia1(); //imprime "Clase: método de instancia 1"
instancia.MétodoDeInstancia2(); // imprime "Clase: método de instancia 2"
instancia.métodoPorDefecto(); // imprime "Interfaz: método por defecto.
```

La variable instancia es del tipo Clase, aunque también podría haberse utilizado Interfaz para denotar el tipo:

```
Interfaz instancia = new Clase();
```

Cómo implementar y extender interfaces múltiples

Una de las características más importantes de las interfaces es que permite simular la herencia múltiple: una clase puede implementar varias interfaces.

```
interface A { }
interface B { }
interface C { }

class D implements A, B, C { }
```

Una interfaz puede extender una o más interfaces utilizando la palabra clave extends:

```
interface A { }
interface B { }
interface C { }

interface E extends A, B, C { }
```

Una clase puede también extender otra clase e implementar múltiples interfaces:

```
class A { }

interface B { }
interface C { }

class D extends A implements B, C { }
```

La herencia múltiple con interfaces se utiliza a menudo en la librería estándar de clases de Java. La clase String, por ejemplo, implementa tres interfaces a la vez:

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence{
// ...
}
```

Los métodos de las interfaces

1) Métodos abstractos

Como hemos dicho anteriormente, son los métodos principales en las interfaces. No llevan implementación, sino que se indica solamente su signatura, y tendrán que ser desarrollados en las clases que implementen la interfaz que los incluye.

Si la declaración de un método en una interfaz no lleva modificadores, se entiende que el método es public abstract.

2) Métodos por defecto

Los métodos por defecto, a diferencia de los métodos abstractos, tienen implementación:

```
interface Interfaz {
    default void acción() {
        System.out.println("Acción");
    }
}
```

Para denotar un método por defecto, se utiliza la palabra clave `default`.

Para invocar un método por defecto hace falta un objeto de una clase que implemente la interfaz. No pueden ser invocados directamente con el nombre de la interfaz (por ejemplo: Interfaz.acción).

```
class Clase implements Interfaz {
}
...

Clase c = new Clase();
c.acción(); // Acción
```

Los métodos por defecto pueden sobreescribirse en una clase, como cualquier otro método normal:

```
class Clase implements Interfaz {
    public void acción() {
        System.out.println("Método acción sobreescrito");
    }
}
...
Clase c = new Clase();
c.acción(); // Método acción sobreescrito
```

¿Por qué son necesarios los métodos por defecto? La razón principal es la de soportar compatibilidad hacia atrás. Consideremos un ejemplo.

Supongamos que programamos un juego que tiene diferentes personajes que se desplazan en un mapa. Eso lo representamos con la interfaz Mover:

```
interface Mover {
    void pasoAdelante();
    void giraIzquierda();
    void giraDerecha();
}
```

De modo que tenemos la interfaz y muchas clases que la implementan. Por ejemplo, el personaje Batman:

```
class Batman implements Mover {
    public void pasoAdelante() {...}
    public void giraIzquierda() {...}
    public void giraDerecha() {...}
}
```

Ahora de repente decides que los personajes deberían poder darse la vuelta. Eso implica que necesitas añadir el método darseVuelta a la interfaz Mover. Puedes implementar el método para todas las clases que implementan la interfaz, pero otra forma es declarar un método default en la interfaz, y así te evitas el tener que implementarlo clase por clase. A veces los métodos por defecto ayudan a evitar la duplicación de código. De hecho, en nuestro caso los métodos darseVuelta pueden hacer lo mismo para todas las clases:

```
interface Mover {
    void pasoAdelante();
    void giraIzquierda();
    void giraDerecha();

    default void darseVuelta() {
        giraIzquierda();
        giraIzquierda();
    }
}
```

Si después quieres personalizar un método por defecto para el personaje Batman, basta con sobreescribirlo:

```
class Batman implements Mover {
    public void pasoAdelante() {...}
    public void giraIzquierda() {...}
    public void giraDerecha() {...}

    public void darseVuelta() {
        giraDerecha();
        giraDerecha ();
    }
}
```

Otro ejemplo lo vemos en las interfaces que forman parte de la librería estándar de Java. Supongamos que los responsables de Java deciden ampliar una interfaz ampliamente utilizada con un método nuevo en la próxima versión. Esto significa que si vas a actualizar la versión de Java y tienes clases que implementan esa interfaz en tu código, tendrías que implementar el nuevo método para que tu código compile.

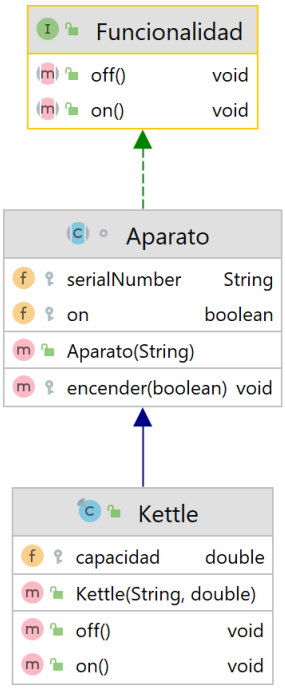
3) Métodos privados

A veces los métodos por defecto tienen mucho código. Para descomponerlos, Java permite declarar métodos privados dentro de la interfaz:

```
interface Interfaz {
    default void acción() {
        String respuesta = subAcción();
        System.out.println(respuesta);
    }

    private String subAcción() {
        return "Acción";
    }
}
```

4) Métodos estáticos



Las interfaces pueden contener métodos estáticos implementados.

```
interface Coche {
    static float convertirAMillasHora(float kmh) {
        return 0.62 * kmh;
    }
}
```

Para hacer uso de un método estático basta con invocarlo directamente desde la interfaz

```
Coche.convertirAMillasHora(4.5);
```

El objetivo principal de incluir métodos estáticos en una interfaz es el de definir una funcionalidad común a todas las clases que la implementen, ayudando a evitar duplicidades en el código.

Interfaces vacías (marker interfaces)

En algunas situaciones una interfaz puede no tener ningún miembro. Por ejemplo, la interfaz Serializable es una interfaz vacía:

```
public interface Serializable{
}
```

Otros ejemplos de interfaz vacía son Cloneable, Remote, etc. Se utilizan para dar información importante a la máquina virtual de Java (JVM).

Utilizar clases abstractas e interfaces conjuntamente

A veces se utilizan interfaces y clases abstractas a la vez para hacer que la jerarquía de clases sea más flexible. En este caso, una clase abstracta contiene miembros comunes e implementa una o múltiples interfaces, y las clases concretas extienden la clase abstracta y posiblemente otras interfaces.

Veamos el siguiente ejemplo:

```
interface Funcionalidad {
    void on();
    void off();
}

abstract class Aparato implements Funcionalidad {

    protected String serialNumber;
    protected boolean on;

    public Aparato(String serialNumber) {
        this.serialNumber = serialNumber;
    }

    protected void encender(boolean on) {
        this.on = on;
```

```
    }
}

class Kettle extends Aparato {

    protected double capacidad;

    public Kettle(String serialNumber, double capacidad) {
        super(serialNumber);
        this.capacidad = capacidad;
    }

    @Override
    public void on() {
        // implementa la lógica para activar la kettle
        encender(true);
    }

    @Override
    public void off() {
        // implementa la lógica para apagar la kettle
        encender(false);
    }
}
```

En la librería estándar de clases de Java hay jerarquías de clases. Por ejemplo, la jerarquía collections, que combina clases abstractas e interfaces para hacerla más mantenible y flexible al utilizarla en el código.

Preguntas test:

1. ¿Qué palabra reservada se emplea siempre en la declaracion de una interfaz?
    a. interface
    b. extends
    c. implements

2. ¿Qué palabra reservada se emplea siempre en la declaración de una clase que implementa una interfaz?
    a. implements
    b. extends
    c. interface

3. ¿Cuántas interfaces puede implementar una clase en Java?
    a. Tantas interfaces como metodos abstractos quiera implementar
    b. Una clase sólo puede implementar una interfaz
    c. Puede implementar un número indefinido de interfaces

4. Los atributos incluidos en una interfaz se declaran implicitamente como...
    a. private, static y abstract
    b. public, static y abstract
    c. Ninguna de las anteriores

5. Una interfaz sirve para:
    a. Almacenar datos numéricos

b. Definir una serie de funcionalidades que se implementarán el las clases.
c. Implementar los métodos abstractos de una clase abstracta.

6. Un método declarado, pero no implementado en una interfaz se llama:
   a. Método estático.
   b. Método abstracto.
   c. Método de instancia.

7. Las interfaces se parecen a las clases abstractas en que:
   a. No tienen atributos
   b. Todos sus métodos son abstractos
   c. No son instanciables
   d. Carecen de métodos privados

8. Las clases anónimas:
   a. Sirven para crear atributos sin nombres.
   b. Sirven para implementar métodos privados.
   c. Se emplean para heredar de varias clases.
   d. Sirven para implementar interfaces localmente, creando un objeto único, cuya clase no lleva ningún nombre.

Preguntas cortas:

9. Partiendo del siguiente código, declara la clase D de forma que implemente las interfaces A, B y C

```
interface A { }
interface B { }
interface C { }
```

10. Partiendo del siguiente código, escribe una clase Hispanohablante que implemente la clase Hablante.

```
interface Hablante {
    void decirHola();
    void decirAdiós();
}
```

Ejercicios

11. Construir una interfaz Relaciones (y posteriormente una clase que la implemente) que incluya los siguientes métodos:

```
// Devuelve verdadero si a es mayor que b
boolean esMayor(Object b) ;
// Devuelve verdadero si a es menor que b
boolean esMenor(Object b) ;
// Devuelve verdadero si a es igual que b
boolean esIgual(Object b) ;
```

12. Escribe un programa para una biblioteca que contenga libros y revistas.
    a. Publicación: Las características comunes que se almacenan tanto para las revistas como para los libros son el código, el título y el año de publicación. Debe contener un constructor parametrizado con todos los atributos. Es una clase pensada sólo para que libros y revistas hereden de ella, así no se puede instanciar.
    b. Los libros tienen además un atributo prestado. Los libros, cuando se crean, no están prestados.
    c. Las revistas tienen un número. En el momento de crear las revistas se pasa el número por parámetro.
    d. Tanto las revistas como los libros deben tener (aparte de los constructores) un método toString() que devuelve el valor de todos los atributos en una cadena de caracteres. También tienen un método que devuelve el año de publicación y otro para el código.
    e. Para prevenir posibles cambios en el programa se tiene que implementar una interfaz Prestable con los métodos prestar(), devolver() y prestado(). La clase Libro implementa esta interfaz.
    f. En la clase Main, tenemos
        i. Un método estático cuentaPrestados(): recibe por parámetro un ArrayList de Libros y devuelve cuántos de ellos están prestados.
        ii. Un método estático publicacionesAnterioresA(): recibe por parámetro un ArrayList de Publicaciones y un año, y devuelve cuántas publicaciones tienen fecha anterior al año recibido por parámetro.
        iii. En el método main,
            1. Crear un ArrayList de libros con 3 libros. Prestar uno de los libros. Mostrar por pantalla los datos del ArrayList. Mostrar cuántos de ellos están prestados.
            2. Crear 2 revistas.
            3. Crear un ArrayList de publicaciones con las cinco publicaciones anteriores. Mostrar por pantalla los datos almacenados en el ArrayList. Mostrar cuántas publicaciones hay anteriores a 2020.

Referencias

https://hyperskill.org/knowledge-map/251?track=8
https://www.paraninfo.es/catalogo/9788428342865/programacion--edicion-2021-
https://es.scribd.com/document/154572650/Programacion-en-Java2-Serie-Schaum-Mc-Graw-Hill