Tema 5. Interfaces

Definición

Una interfaz es una colección de métodos abstractos (contiene solo la signatura o cabecera de dichos métodos, sin el cuerpo) y propiedades (campos o atributos).

La principal ventaja o el principal uso que tienen las interfaces es el de organizar el código de manera más clara y eficiente, evitando también duplicidades, de la misma forma que lo hace la herencia o las clases abstractas. La característica distintiva de las interfaces, es que permiten simular herencia múltiple, como se verá más adelante.

Tienen algunas ventajas adicionales, como es la de obligar a que todas las clases que las implementen definan los mismos métodos (pero con la posibilidad de una implementación particular en cada clase), ofreciendo así uniformidad en el código.

También sirve para establecer relaciones entre clases en principio independientes. De esta forma, es posible "agrupar" clases que en principio no lo estaban, y extraer de ellas sus características (campos y/o métodos comunes), simplificando el código y complejidad de las originales. Aplicable por ejemplo a la hora de refactorizar código.

Por último, al definir interfaces permitimos la existencia de variables polimórficas y la invocación polimórfica de métodos, igual que con las clases abstractas o la herencia.

Implementación

La interfaz *de una clase* declara lo que puede y no puede hacer dicha clase, es decir, su especificación, la manera en que puede ser usada, pero sin mostrar la implementación interna de dicho comportamiento. Muestra el *qué*, pero no el *cómo*. Por otro lado, la implementación es el *cómo* la clase ejecuta dicho comportamiento de forma interna. Es decir, define el código completo de dicha clase.

En el API de Java, por ejemplo, disponemos de la interfaz *List* que es implementada por las clases *ArrayList* y *LinkedList*. Misma interfaz, diferentes implementaciones.

Se puede pensar en que la interfaz es la parte *pública* de una clase, mientras que la implementación es la parte *privada* de dicha clase. Sirve para poner en práctica lo que se denomina ocultamiento o encapsulación de la información.

Una buena práctica a la hora de programar, especialmente en proyectos más o menos grandes, es desarrollar primero las interfaces que se van a necesitar, definiendo el comportamiento externo de las clases, para más tarde hacer su implementación interna. Esto aumenta la robustez y mantenibilidad del código.

Interfaz / Implementación

Externa / Interna

Público / Privado

Declaración / Definición

El *QUÉ* / El *CÓMO*

Herencia

Java no permite herencia múltiple como tal en sus clases. Es decir, una clase puede heredar única y exclusivamente de otra más, pero de una sola, y no de varias.

Sin embargo, es posible simular esta herencia múltiple utilizando precisamente interfaces, las cuales son muy similares a las clases abstractas. Algunas características de las interfaces son:

- Una interfaz utiliza la **palabra clave *interface*** en su cabecera, mientras que una clase abstracta utiliza *abstract class*.

- Permiten definir constantes disponibles para cualquier clase que implemente dicha interfaz. Los atributos o **campos** definidos en una interfaz son ***public static final*** por defecto, sin necesidad de especificarlo así. Es decir, accesibles desde cualquier parte del código (*public*), estáticos, y por tanto utilizables sin instanciar la clase (*static*) y constantes, por lo que no se modifican a lo largo de la ejecución del programa (*final*).

- En cuanto a los **métodos**, son ***public abstract***, sin necesidad de definirlos como tal, y no pueden ser de otra forma*. Mientras que en una clase abstracta sus métodos pueden ser abstractos, o bien tener definido el cuerpo de la implementación, en una interfaz no se implementa ninguno de los métodos que declara. *Una excepción, cuando se habla de interfaces, serían los métodos *default de Java 8* (explicados en el siguiente punto).

- Una interfaz **no tiene constructor** de ningún tipo, no se pueden crear objetos de tipo interfaz (no se puede instanciar de la forma tradicional con el operador *new()*).

Así, por ejemplo, suponiendo que *MiInterfaz* y *MiClase* son una interfaz y una clase respectivamente, y que la segunda implementa la primera, nunca se podría hacer:

```
MiInterfaz if = new MiInterfaz();
```

Después del *new*, siempre tiene que ir una clase que implemente dicha interfaz:

```
MiInterfaz if = new MiClase();
```

Para entender mejor esto, se puede pensar una analogía en la que la interfaz actúa como un adjetivo, por ejemplo el adjetivo *Azul*. Tiene sentido crear objetos de tipo *CocheAzul*, o *PelotaAzul* (clases que implementan la interfaz *Azul*). Sin embargo, no tendría sentido crear un objeto de tipo *Azul* a secas, sin saber qué es azul.

- Una clase puede heredar de una interface, igual que puede hacerlo de una clase abstracta. Las diferencias en este punto son 2.

1/ **Una clase puede implementar una o varias interfaces, pero sólo puede extender como máximo una clase** (abstracta o no). Implementar varios interfaces es lo más parecido que tiene Java a la herencia múltiple.. Es decir, una misma clase puede: no heredar, heredar de 1 sola clase, heredar de 1 sola interfaz, heredar de 1 clase y de 1 interfaz, o bien heredar de 1 clase y a su vez de 2 o más interfaces.

2/ **Las clases heredan de diferente manera**, según lo hagan de una clase o de una interfaz. Por ejemplo, aquí tenemos una clase que extiende a otra:

```
public class Perro extends Animal { ... }
```

Mientras que si implementa una interfaz, la signatura de la clase sería:

```
public class Cuadrado implements Figura { ... }
```

Como se puede ver, **la diferencia es utilizar *extends* (clase) o *implements* (interfaz)**. Si una clase hereda de otra clase, y además, de una o varias interfaces, la signatura es:

```
public class Navaja extends Cuchillo implements Herramienta, Arma { ... }
```

- Si una clase implementa una interfaz, se dan 2 escenarios. O bien implementa todos los métodos de dicha interfaz, o bien se trata de una clase abstracta que solo implementa parte o ninguno de los métodos de dicha interfaz, y delega su implementación a las clases inferiores.

NOTA: una interfaz no puede implementar (*implements*) otra interfaz, puesto que las interfaces, por definición, no contienen implementación. Pero una interfaz si que puede extender (*extends*) a otra interfaz, delegando la implementación hacia clases inferiores.

¿Cómo saber si una clase es candidata a ser definida como una interfaz?

a) Si necesitamos algún método con cuerpo ya sabemos que no va a ser una interfaz porque todos los métodos de una interfaz han de ser abstractos. Encontramos una excepción a esto con los métodos *default*.

b) Si necesitamos que una clase "herede" de más de una superclase (herencia múltiple), esas superclases son candidatas a ser interfaces.

c) A veces es igual de aceptable definir una clase como interfaz que como clase abstracta, pero en estos casos se prefiere optar por una interfaz, porque es más flexible y extensible: nos permite que una clase implemente varias interfaces. En cambio, una clase no puede heredar de varias clases abstractas.

Métodos *default*

Los métodos *default* permiten proporcionar la implementación por defecto para un método en una interfaz, de tal forma que si por ejemplo tenemos un conjunto de clases tipo: interfaz + clases que la implementan, se podría añadir un nuevo método a la interfaz, sin tener que modificar todas las clases hijas. Es decir, permiten añadir, a posteriori, funcionalidades a una interfaz, sin "romper" y tener que reescribir las clases hijas que la implementan. Por tanto, se aumenta la mantenibilidad de código.

También se les conoce como *Defender Methods* o *Virtual Extension Methods*.

Cuando una clase implementa una interfaz que contiene un método *default*, se puede dar uno de los siguientes casos:

- Si no se menciona el método *default* en dicha clase, se utiliza la implementanción por defecto del interfaz.
- El método *default* se puede volver a hacer abstracto redefiniendo el método como *abstract* en la clase que implemente la interfaz, delegando así su implementación final a otra clase aún inferior en la jerarquía, que herede de la clase de la que hablamos.

- Se puede redefinir el método de forma similar a la que se sobrescriben los métodos en cualquier otro ámbito de la herencia de clases.
- Cuando una clase implementa dos interfaces que tienen definido un método default con la misma signatura (declaración) es necesario redefinir el método en la clase hija e indicar a que interfaz se llama de los dos, o bien redefinir el método con un comportamiento nuevo. De lo contrario, el compilador indicará un error en tiempo de compilación. Esto se conoce como el *diamon problem* cuando hablamos de herencia múltiple. Un ejemplo:

```
public interface Persona {
   default void saludar() {
      System.out.println("Hola, soy una persona."); }
}
public interface Hombre {
   default void saludar() {
      System.out.println("Hola, soy un hombre."); }
}
public class Pedro implements Persona, Hombre {
   // Pedro implementa dos interfaces con un método del mismo nombre.
   // Debe redefinir dicho método con una implementación propia,
   // o indicando a qué interfaz se llama en este caso
   public void saludar() {
      Hombre.super.saludar(); }
}
```

- Cabe señalar que una interfaz no puede implementar un método *default* que redefina (sobreescriba) un método de la clase *Object*, como por ejemplo los métodos *toString(), equals()* o *hashCode()*. Esto daría un error del compilador.

Por tanto, al introducir esta clase de métodos, se puede ver que la diferencia entre interfaces y clases abstractas se acorta aún más. Las separan 2 propiedades:

- Herencia simple / Herencia múltiple. Una clase solo puede extender una clase abstracta, pero implementar múltiples interfaces. Los métodos *default* en las interfaces permiten una herencia múltiple **de métodos**, pero **no de atributos**. Es decir, cada clase que implemente la interfaz, hereda la implementación de sus métodos *default*, pero, puesto que los campos de la interfaz son estáticos, en realidad ninguna de las clases hijas heredan dichos campos, sino que son objetos "de clase", inherentes a la propia interfaz, y utilizables invocando directamente a la interfaz.

- Campos mutables / Campos constantes. Una clase abstracta puede contener tanto campos variables como constantes, mientras que en una interfaz, todos los campos que se incluyan serán siempre y por defecto *public* ***final*** *static* (constantes). Por tanto, no se podrán sobreescribir o cambiar su valor durante la ejecución.

Métodos *static*

Primero, vamos a poner un ejemplo para discernir un poco mejor los elementos *static* de aquellos que no lo son, en Java. Imaginemos una clase *Ejemplo* con los campos:

```
public class Ejemplo {

	private int x;

	private static int z;

}
```

Cada objeto de tipo *Ejemplo* tendrá su propia *x*, mientras que solo habrá una *z* común y compartida para todos los objetos de tipo *Ejemplo*. De tal forma, que en cada objeto *Ejemplo* la *x* podrá tener un valor diferente, y ser asignado de forma individual, mientras que la *z* tendrá un valor común a través de todos los objetos, e incluso aunque no exista ninguno (antes de crearlos, o después de que se destruyan).

Los métodos *static* son aquellos métodos que pueden ser invocados directamente sobre una clase, sin necesidad de que exista una instancia de la misma. Por eso a veces también se les llama "métodos de clase".

En contraposición, tenemos los métodos de instancia, los cuales se invocan siempre sobre un objeto (una instancia) de una clase. Por ello se dice que los primeros se invocan de forma estática, mientras que los segundos se invocan de forma dinámica, sobre un objeto creado en tiempo de ejecución.

Los métodos de clase tienen 2 limitaciones principales:

- No pueden acceder a campos de instancia (no estáticos). Lógico, dado que dichos campos se asocian a objetos.
- No puede invocar métodos de instancia (no estáticos). Puesto que, como antes, estos métodos de instancia se asocian a objetos.

Básicamente ambas limitaciones quieren decir que desde contextos estáticos (métodos o clases estáticas) no se puede hacer referencia a contextos no estáticos. Pero si puede ser al contrario, desde un contexto no estático acceder a uno estático.

Son muy útiles como métodos "utilitarios", que proporcionen alguna funcionalidades genéricas, como por ejemplo funciones matemáticas, o transformaciones sobre listas.

Con la introducción de los métodos *default* en Java 8, ahora es posible también crear métodos de clase (métodos *static*) directamente en las interfaces, algo que han incorporado ya algunas de las contenidas en la API de Java, como la interfaz *Comparator* con su método reverse*Order()*, el cuál es un método *static*.

Hablando de interfaces, los métodos *default* comentados en el apartado anterior son muy similares a los métodos *static*, con la diferencia de que estos últimos no pueden ser redefinidos (sobreescritos) en clases hijas que implementen la interfaz, mientras que los métodos *default* si que pueden ser redefinidos, como se vió antes.

Además, igual que en los métodos *default*, un método *static* de una interfaz de Java no puede redefinir un método de la clase *Object*, se obtendría error de compilación.

Comparadores: *Comparable, Comparator*

Comparable

La interfaz *Comparable* permite que la clase que la implemente pueda realizar la comparación de objetos (ser comparada), habilitando así la ordenación de elementos.

Para ello hace falta que la clase que queramos implemente dicha interfaz:

```
public class FechaComparable implements Comparable<FechaComparable> { ... }
```

Al implementar la interfaz, esa clase está obligada también a implementar todos sus métodos, que en este caso solo tiene uno, el método *compareTo(Object o)*. Este método es el que albergará toda la lógica de comparación que queramos expresar entre los elementos de la clase que sean comparables.

Por ejemplo, pongamos la clase *Integer*, la cual implementa la interfaz *Comparable*:

```
public int compareTo(Integer numero) {

      return Integer.parseInt(this) – Integer.parseInt(numero); }
```

Como se ve, en el caso de *Integer* la comparación es tan sencilla como comparar los valores int del objeto que representan. **Si el primero (*this*) es menor que el segundo, devolverá un entero negativo, si es mayor, un entero positivo, y si son iguales, devolverá 0**. **Es el comportamiento esperado según la interfaz *Comparable***.

¿Por qué es bueno usar la implementación de interfaces en Java?

Como por ejemplo hemos hecho con la interfaz *Comparable*. Hay varios motivos, pero el principal es que mediante la implementación de interfaces todos los programadores usan la misma signatura y semántica para comparar elementos (o realizar otras operaciones). Si por ejemplo hubiera que usar la clase que realiza comparaciones desarrollada por otro programador, dado que todos usarían la misma implementación (*Comparable*), se sabría que la comparación de objetos debe de hacerse invocando determinados métodos de determinada forma, ya conocida. En definitva, mejora el desarrollo y la comprensibilidad del código.

Comparator

La interfaz *Comparator* permite que las clases se definan como comparadores, cada uno de los cuales compara un tipo de objetos determinado, en base a unos criterios de ordenación propios. Hace falta que la clase elegida implemente dicha interfaz:

```
public class ComparadorFechas implements Comparator<Fecha> { ... }
```

Al implementar la interfaz, esa clase está obligada también a implementar todos sus métodos abstractos, que en este caso solo es uno, el método *compare(Object o1, Object o2)*. Este método es el que albergará toda la lógica de comparación que queramos expresar entre los elementos de la clase que sean comparables.

Criterio	***Comparable***	***Comparator***
Lógica de ordenación	La lógica de ordenación está contenida en los propios elementos que son ordenados. Esto es lo que se llama *orden natural* de los elementos.	La lógica de ordenación está contenida en una clase separada (el comparador). Por tanto, se pueden crear diferentes modos de ordenación según diferentes atributos de los objetos a ordenar.
Implementación	La clase de los elementos a ordenar debe implementar la interfaz (y sus métodos).	La clase de los elementos a ordenar no tiene porqué implementar la interfaz, sino que puede hacerlo otra clase aparte.
Forma de ordenación	*int compareTo(Object obj);* Compara *this* con *obj*. Y devuelve un *int* (entero): *# +: this > obj* *# 0: this = obj* *# -: this < obj*	*int compare(Object o1, Object o2);* Compara el *o1* con *o2*. Y devuelve un *int* (entero): *# +: o1 > o2* *# 0: o1 = o2* *# -: o1 < o2*
Forma de llamada	*Collections.sort(List);* Los objetos de la lista deben ser *comparables* y el método los ordenará según su orden natural (orden interno).	*Collections.sort(List, Comparator);* - o bien - *lista.sort(Comparator);* Los objetos de la lista se ordenarán en base a los criterios del *comparador* proporcionado.
Paquete (importar)	*java.lang.Comparable*	*java.util.Comparator*