Interfaces en POO

ÍNDICE

1. Introducción

Una interfaz es una plantilla para la construcción de clases. Permite definir qué métodos debe contener una clase (o conjunto de clases) pero sin implementarlos. Lo anterior quiere decir que, normalmente, una interfaz se compone de un conjunto de declaraciones de cabeceras de métodos (sin implementar) que especifican qué debe hacer una clase, es decir, qué comportamiento debe tener. Una clase puede implementar una o varias interfaces y ello la llevará a tener que dar lógica a todos los métodos indicados en la interfaz o interfaces que implemente. Es como un contrato en el que se compromete a ofrecer el comportamiento pactado. **Nos permite definir qué hará una clase pero no cómo lo hará**.

Además, una interfaz puede emplearse también para declarar constantes que luego puedan ser utilizadas por otras clases.

Para tratar de ir entendiendo más las interfaces, podemos basarnos en un ejemplo típico: el de las figuras geométricas. Podemos definir una interfaz llamada `FiguraGeometrica` que contenga un método para calcular el área de la figura. Si tenemos una clase `Circulo` y una clase `Rectangulo` que implementen dicha interfaz, ambas clases deberán tener un método de calcular el área pero cada una con su propia lógica (con su propia fórmula del cálculo del área).

2. Declaración de la interfaz

La declaración de una interfaz es similar a una clase, aunque emplea la palabra reservada **`interface`** en lugar de **`class`** y no incluye ni la declaración de atributos de instancia ni la implementación del cuerpo de los métodos (únicamente las cabeceras). La sintaxis de declaración de una interfaz es la siguiente:

```
public interface FiguraGeometrica {
	// 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* (tal y como ocurre con las clases). 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`** y **`abstract`** (se pueden omitir).

Por ejemplo, la interfaz `FiguraGeometrica` declara la cabecera de un único método para calcular el área:

```
public interface FiguraGeometrica {
	double getArea();
}
```

Los métodos, obviamente, pueden recibir parámetros para cumplir su función si así es necesario.

3. Constantes de interfaz

En una interfaz, también podemos definir constantes que serán comunes a todas las clases que implementen dicha interfaz.

Todas las constantes incluidas en una interfaz se declaran implícitamente como **`public`**, **`static`** y **`final`** (también se pueden omitir) y es necesario inicializarlas en la misma sentencia de declaración.

Supongamos que, por cuestiones de lógica de negocio, únicamente vamos a trabajar con figuras geométricas que tengan un área superior 13'4 cm². Podemos implementar una interfaz que declare e inicialice dicha interfaz. Nuestra interfaz quedaría así:

```
public interface FiguraGeometrica {
	double AREA_MINIMA = 13.4;

	double getArea();
}
```

4. Implementación de la interfaz en clases

Para declarar una clase que implemente una interfaz es necesario utilizar la palabra reservada **`implements`** en la cabecera de declaración de la clase. Las cabeceras de los métodos (identificador y número y tipo de parámetros) deben aparecer en la clase tal y como aparecen en la interfaz implementada. Por ejemplo, la clase `Rectangulo` implementa la interfaz `FiguraGeometrica` y, por lo tanto, debe declarar el método `getArea()` de la interfaz.

```
public class Rectangulo implements FiguraGeometrica {
	@Override
	public double getArea() {
		// Sin implementar...
		return 0;
	}
}
```

Podemos ver que los métodos vienen decorados con la anotación `Override` que indica que vamos a sobrescribir (dar una nueva funcionalidad) a dicho método. Es opcional pero nosotros lo vamos a poner siempre.

Nuestra clase `Rectangulo` podría quedar de la siguiente forma:

```
public class Rectangulo implements FiguraGeometrica {
	private double base;
	private double lado;

	public Rectangulo(double base, double lado) {
		this.base = base;
		this.lado = lado;
	}

	public double getBase() {
		return base;
	}

	private void setBase(double base) {
		this.base = base;
	}

	public double getLado() {
		return lado;
	}

	private void setLado(double lado) {
		this.lado = lado;
	}

	@Override
	public double getArea() {
		return getBase() * getLado();
	}
}
```

Podemos, crear la clase `Circulo` que, a su vez, también implementa la interfaz `FiguraGeometrica`:

```
public class Circulo implements FiguraGeometrica {
	private double radio;

	public Circulo(double radio) {
		this.radio = radio;
	}

	public double getRadio() {
		return radio;
	}

	private void setRadio(double radio) {
		this.radio = radio;
	}

	@Override
	public double getArea() {
		return Math.PI * Math.pow(radio, 2);
	}
}
```

5. Implementación de múltiples interfaces

Una clase puede implementar más de una interfaz. Para ello, debemos separar el nombre de las interfaces por comas, tal y como se puede ver en el siguiente ejemplo.

Para verlo más claro, vamos a utilizar un ejemplo académico en el que tenemos una interfaz llamada `Primera` y otra interfaz llamada `Segunda`.

```
public interface Primera {
	void metodo1();
	void metodo2(int numero);
}
```

```java
public interface Segunda {
	boolean metodo3();
	double metodo4();
}
```

En este momento podemos definir una clase llamada `MultiInterfaz` que implemente ambas interfaces:

```java
public class MultiInterfaz implements Primera, Segunda {

	@Override
	public void metodo1() {
		// ...
	}

	@Override
	public void metodo2(int numero) {
		// ...
	}

	@Override
	public boolean metodo3() {
		// ...
	}

	@Override
	public double metodo4() {
		// ...
	}
}
```

6. Extensión de interfaces

Una interfaz puede incorporar (heredar) el comportamiento de otra interfaz mediante el uso de la palabra clave **`extends`**. Cuando una clase implementa una interfaz que hereda otra interfaz, debe proporcionar implementaciones para todos los métodos requeridos por la cadena de herencia de la interfaz. Supongamos que tenemos la siguiente interfaz llamada `Tercera` que extiende la interfaz `Primera` que ya hemos visto.

```
public interface Tercera extends Primera {
	int metodo5(int num);
}
```

Si creamos una clase llamada `MultiInterfaz2` que implemente la interfaz `Tercera`, la estructura de dicha clase, como mínimo, deberá ser así:

```
public class MultiInterfaz2 implements Tercera {

	@Override
	public void metodo1() {
		// ...
	}

	@Override
	public void metodo2(int numero) {
		// ...
	}

	@Override
	public int metodo5(int num) {
		// ...
	}
}
```

Los métodos 1 y 2 deben ser implementados como contrato con la interfaz `Primera`. El método 5 debe ser implementado debido al contrato con la interfaz `Tercera`.

7. Utilización de una interfaz como tipo de dato

Uno de los principales usos de las interfaces es utilizarlas como tipos de datos. De tal forma que un método no reciba una clase como parámetro sino una interfaz que es indicativa del comportamiento de una clase o conjunto de clases.

Supongamos que tenemos un programa que nos permita crear círculos y rectángulos y mostrar su área. Haciendo uso de las clases y la interfaz creada en este tema lo podríamos hacer de la siguiente forma:

```
private static void mostrarArea(Rectangulo rectangulo)
{
    System.out.printf("El área de la figura es %.2f cm2%n", rectangulo.getArea());
}

private static void mostrarArea(Circulo circulo)
{
    System.out.printf("El área de la figura pasada por parámetro es %.2f cm2%n", circulo.getArea());
}

public static void main(String[] args)
{
    Rectangulo rectangulo = new Rectangulo(8, 5);
    Circulo circulo = new Circulo(7);

    mostrarArea(rectangulo);
    mostrarArea(circulo);
}
```

La salida por pantalla correspondiente sería la siguiente:

```
El área de la figura es 40,00 cm2
El área de la figura pasada por parámetro es 153,94 cm2
```

Podemos ver que necesitamos un método por cada figura geométrica, es decir, por cada clase que implemente la interfaz `FiguraGeometrica`. Si tuviésemos 100 tipos de figuras geométricas, ¿necesitaríamos 100 métodos `mostrarArea()` cada uno para un tipo de figura geométrica? Si nos fijamos, lo único que hacemos es mostrar por pantalla el área de la figura correspondiente (llamando al método `getArea()`). Si ambos tipos de figuras tienen dicho método ya que implementan la propia interfaz… ¿no habrá alguna forma de generalizar y decirle al método `mostrarArea()` que le pasamos una figura geométrica? La respuesta es sí y nuestro código podría quedar con un único método de la siguiente forma.

```
private static void mostrarArea(FiguraGeometrica figura) {
	System.out.printf("El área de la figura geométrica es %.2f cm2%n", figura.getArea());
}

public static void main(String[] args) {
	Rectangulo rectangulo = new Rectangulo(8, 5);
	Circulo circulo = new Circulo(7);

	mostrarArea(rectangulo);
	mostrarArea(circulo);
}
```

En el anterior ejemplo, no tenemos un método por cada figura geométrica sino que reutilizamos el mismo para cualquier figura que implemente la interfaz `FiguraGeometrica` ya que lo único que hacemos es llamar al método `getArea()` que debe ser codificado por cualquier clase que implemente la interfaz `FiguraGeometrica`.

Podemos ver que se puede emplear el identificador de una interfaz en cualquier lugar donde se pueda utilizar el identificador de un tipo de dato (o de una clase). El objetivo es garantizar la sustitución por cualquier instancia de una clase que la implemente.

8. Interfaces desde JDK 8

Una vez entendido el concepto de interfaz es importante hacer una puntualización. JDK 8 agregó una función a interface que hizo un cambio significativo en sus capacidades. Antes de JDK 8, una interfaz no podía definir ninguna implementación de ningún tipo. Por lo tanto, antes de JDK 8, una interfaz podría definir únicamente el qué, pero no el cómo, como hemos aprendido.

JDK 8 cambió lo anterior y permitió agregar una implementación predeterminada a un método de la interfaz. Además, ahora se admiten los métodos de interfaz estática y, a partir de JDK 9, una interfaz también puede incluir métodos privados. Por lo tanto, ahora es posible que la interfaz especifique algún comportamiento.

Sin embargo, tales métodos constituyen lo que son, en esencia, características de **uso especial**, y la intención original detrás de la interfaz aún permanece. Por lo tanto, como regla general, con frecuencia se crearán y utilizarán interfaces en las que no se utilizarán estas nuevas funciones.

9. Interfaz Cloneable

En ocasiones, puede que necesitemos clonar un objeto, es decir, hacer una copia del objeto con exactamente los mismos valores en cada uno de los atributos. Sin embargo, el objeto original y el clonado serán distintos, esto es, tendrán diferente dirección de memoria. Lo anterior implica que si cambiamos el valor de un atributo en uno de los objetos (original o clonado) el otro no se verá afectado.

Para clonar un objeto en Java, podemos hacer que la clase de dicho objeto implemente la interfaz `Cloneable`. Esta *`interface`* sólo dispone de un método que es el método `clone()` que se encarga de implementar el sistema de clonación.

Por ejemplo, si tenemos una clase `Coche` y queremos tener la posibilidad de clonar coches, debemos implementar la interfaz `Cloneable` y darle contenido al método `clone()`.

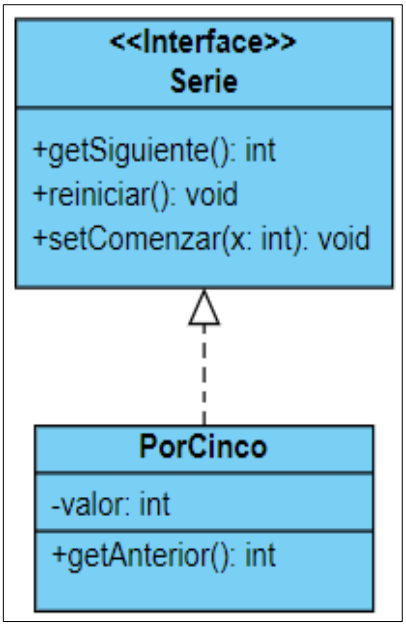
```
public class Coche implements Cloneable {

	private String matricula;
	private LocalDate fechaMatriculacion;

	public Coche(String matricula, LocalDate fechaMatriculacion) {
		super();
		this.matricula = matricula;
		this.fechaMatriculacion = fechaMatriculacion;
	}

	public String getMatricula() {
		return matricula;
	}

	public void setMatricula(String matricula) {
		this.matricula = matricula;
	}

	public LocalDate getFechaMatriculacion() {
		return fechaMatriculacion;
	}

	public void setFechaMatriculacion(LocalDate fechaMatriculacion) {
		this.fechaMatriculacion = fechaMatriculacion;
	}

	@Override
	public Coche clone() {
		//return super.clone();

		Coche clonado = new Coche(getMatricula(), getFechaMatriculacion());
		return clonado;
	}
}
```

Para clonar el objeto, debemos hacer uso del método `clone()` del objeto de tipo `Coche`:

```
public static void main(String[] args) {
	Coche original = new Coche("1234-ABC", LocalDate.of(2022, 03, 04));
	Coche copia = original.clone();
}
```

10. UML

En UML, las interfaces se indican mediante la palabra *Interface* (junto con su nombre) de la siguiente forma. Posteriormente, se indican los métodos que debe implementar cualquier clase que implemente, a su vez, la propia `interface`.



Para indicar que una clase implementa dicha interface, se unen mediante una flecha de punta cerrada y línea discontinua desde la clase implementadora hasta la interface.