POO: Interfaces en Java

Sitio: Instituto de Tecnología ORT
Curso: Taller de Programación 1 - Plan 2 años 1°C 2020
Libro: POO: Interfaces en Java

Imprimido por: Gentile Fabián
Día: domingo, 19 de abril de 2020, 18:59

Tabla de contenidos

1. No a la Herencia Múltiple

Como vimos en los textos previos, Java no acepta la Herencia Múltiple entre clases. Es decir, las clases de Java no pueden heredar directamente de más de una clase (sí indirectamente, a través del árbol de jerarquías).

La razón principal de esta decisión fue que quienes crearon Java, con vasta experiencia en POO usando C++, sabían que la Herencia Múltiple parece aportar más problemas que los que resuelve, y como quisieron hacer de Java un lenguaje simple, en lo posible sin ambigüedades, decidieron dejar a la Herencia Múltiple afuera.

No obstante, eran conscientes de que se necesitaba un mecanismo a través del cual las clases pudieran adoptar como propias características de distintas entidades, aún cuando éstas no estuviesen dentro de la misma línea jerárquica. La opción elegida fue permitir que una clase pudiese adoptar constantes y *firmas* de métodos de variados orígenes, sin importar si su origen pertenece a la misma línea jerárquica (es decir sin herencia de implementación), siempre y cuando la clase implemente esas *firmas*.

Esos orígenes variados son las Interfaces.

2. Qué es una interfaz

En Java, una interfaz es una especie de plantilla con una o más *firmas* (sólo la declaración del método, sin la implementación) y, si fuese necesario, atributos constantes.

En las interfaces se especifica el qué pero no el cómo, permitiendo así extender el *contrato funcional* de una clase más allá de su propio árbol de herencia. Serán las clases que *implementen* estas interfaces las que definan el comportamiento de los métodos declarados.

La sintaxis de su declaración es muy similar a la de una clase.

```
public interface Incrementable {
    void incrementar();
}
```

También, al igual que las clases entre sí, una interfaz *soporta herencia*, pero a diferencia de las clases, las interfaces sí pueden heredar de más de una interfaz. Y como son tipos de datos válidos, también se pueden declarar variables con su tipo (variables tipo interfaz) y por lo tanto *podrán recibir instancias de objetos que cumplan con la interfaz*.

3. Un ejemplo

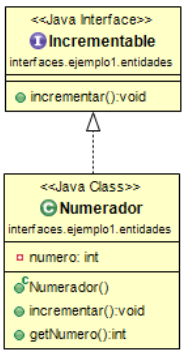
Empecemos por el principio. Lo primero que se me ocurre cuando digo *incrementable* es algo numérico, así que crearemos una clase Numerador que *implemente* la interfaz *Incrementable*. Esta clase sólo tendrá como atributo el número a incrementar (en 1), y su código será el siguiente:

```
package interfaces.ejemplo1.entidades;

public class Numerador implements Incrementable {
    private int numero;

    public Numerador() {
        this.numero = 0;
    }
    public int getNumero() {
        return this.numero;
    }
    @Override
    public void incrementar() {
        this.numero++;
    }
}
```

Y aquí presentamos el diseño UML de la clase Numerador indicando que implementa *Incrementable*.



Al igual que sucede con la herencia desde clases abstractas, Numerador está obligada a implementar el método *incrementar()*, o en su defecto deberá ser declarada abstracta.

Este pequeño programa prueba la nueva clase:

```
package interfaces;

import interfaces.ejemplo1.entidades.Numerador;

public class Ejemplo1 {

    public static void main(String[] args) {
        Numerador num = new Numerador();
        System.out.println("El valor actual es " + num.getNumero());
        System.out.println("incremento...");
        num.incrementar();
        System.out.println("El valor actual es " + num.getNumero());
    }
}
```

Y éste es el resultado de su ejecución:

```
El valor actual es 0
incremento...
El valor actual es 1
```

Hasta aquí otra vez nada especial. En definitiva, tuvimos que implementar todos los métodos en Numerador. Más allá de que al declarar que esta clase implementa *Incrementable*, la estamos obligando a cumplir con el "contrato" que indica que debe tener una implementación del método *incrementar()*, no parece que haya nada que no pueda hacerse con los mecanismos que ya conocemos (por ejemplo, hacer que Numerador herede de una clase abstracta).

Pero supongamos que ahora queremos que los objetos de esta clase sepan *mostrarse*. Todo indica que alcanza con agregar prolijamente un "syso" del valor en Numerador dentro de un método que se llame, por ejemplo, *mostrar()*.

OK, perfecto.

Pero se me ocurre que hay muchas cosas que pueden ser *mostrables*. Una palabra, una foto, un gráfico… ¡tantas cosas! Y me gustaría no tener que preocuparme por aprender el nombre de cada método que signifique "mostrar el objeto en cuestión". El verbo *mostrar* calza perfecto, pero no es lo mismo mostrar un número que mostrar un cartel o una imagen.

Pareciera que, otra vez, una interfaz es el camino:

Declaremos la interfaz *Mostrable*:

```
package interfaces.ejemplo1.entidades;

public interface Mostrable {
    void mostrar();
}
```

3.1. Combinando interfaces

Hasta aquí otra vez nada especial. En definitiva, tuvimos que implementar todos los métodos en Numerador. Más allá de que al declarar que esta clase implementa Incrementable la estamos obligando a cumplir con el "contrato" que indica que debe tener una implementación del método *incrementar()*, no parece que haya nada que no pueda hacerse con los mecanismos que ya conocemos (por ejemplo, hacer que Numerador herede de una clase abstracta).

Pero supongamos que ahora queremos que los objetos de esta clase sepan *mostrarse*. Todo indica que alcanza con agregar prolijamente un "syso" del valor en Numerador dentro de un método que se llame, por ejemplo, *mostrar()*.

OK, perfecto.

Pero se me ocurre que hay muchas cosas que pueden ser *mostrables*. Una palabra, una foto, un gráfico… ¡tantas cosas! Y me gustaría no tener que preocuparme por aprender el nombre de cada método que signifique "mostrar el objeto en cuestión". El verbo *mostrar* calza perfecto, pero no es lo mismo mostrar un número que mostrar un cartel o una imagen.

Pareciera que, otra vez, una interfaz es el camino:

Declaremos la interfaz *Mostrable*:

```
package interfaces.ejemplo1.entidades;

public interface Mostrable {
    void mostrar();
}
```

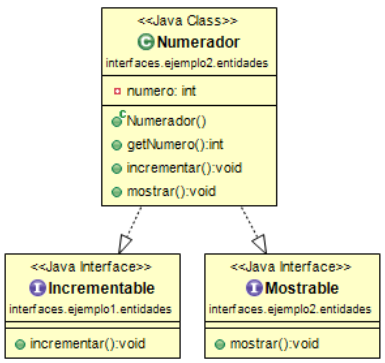
E implementemos esta interfaz en una nueva versión de Numerador:

```
package interfaces.ejemplo2.entidades;

import interfaces.ejemplo1.entidades.Incrementable;

public class Numerador implements Incrementable, Mostrable {
    private int numero;
    public Numerador() {
        this.numero = 0;
    }
    public int getNumero() {
        return this.numero;
    }
    @Override
    public void incrementar() {
        this.numero++;
    }
    @Override
    public void mostrar() {
        System.out.println(this.numero);
    }
}
```

Ahora nuestro numerador quedó implementado con el siguiente diseño:



3.2. Implementación sin fronteras

Para extender el ejemplo, además de declarar a Numerador como *mostrable* e implementar el método correspondiente, agregaremos dos clases más: Frase y BarraDeAvance. La primera contendrá un string y será *mostrable* pero no *incrementable*, y la última implementará ambas interfaces, pero en vez de incrementar el valor de un entero agregará caracteres (por ejemplo un asterisco) a un string inicialmente vacío, uno por cada incremento.

Este es el código de BarraDeAvance:

```
package interfaces.ejemplo3.entidades;

import interfaces.ejemplo1.entidades.Incrementable;
import interfaces.ejemplo2.entidades.Mostrable;

public class BarraDeAvance implements Incrementable, Mostrable {
    private char caracter;
    private String valor;

    public BarraDeAvance(char caracter) {
        this.caracter = caracter;
        this.valor = "";
    }
    @Override
    public void incrementar() {
        this.valor += String.valueOf(caracter);
    }
    @Override
    public void mostrar() {
        System.out.println("[" + valor + "]");
    }
}
```

Asi es la clase Frase:

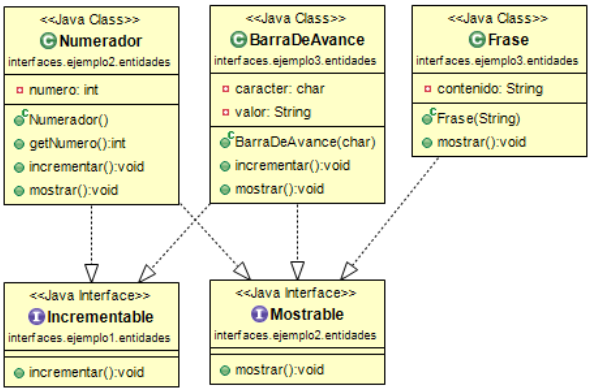
```
package interfaces.ejemplo3.entidades;

import interfaces.ejemplo2.entidades.Mostrable;

public class Frase implements Mostrable {
    private String contenido;

    public Frase(String contenido) {
        this.contenido = contenido;
    }
    @Override
    public void mostrar() {
        System.out.println(this.contenido);
    }
}
```

Y este es el UML de las tres clases con sus respectivas interfaces:



En el diagrama UML queda bien clara la relación entre las clases y sus interfaces. Salvo en el caso de Frase (que sólo implementa una de las dos interfaces) las otras dos implementan ambas, cosa que no podrían hacer a través de un árbol jerárquico. Además, BarraDeAvance y Numerador poco tienen que ver entre sí, salvo que comparten las interfaces *Mostrable* e *Incrementable*. Las tres clases declaradas no forman parte del mismo "linaje", y sin embargo pueden compartir y cumplir con las mismas interfaces sin impedimento.

El programa que figura a continuación crea instancias de estas tres clases y muestra algunos detalles de ellas más su contenido. Luego muestra el resultado de incrementar el valor de los objetos *incrementables* cinco veces:

```
package interfaces;

import interfaces.ejemplo1.entidades.Incrementable;
import interfaces.ejemplo2.entidades.Mostrable;
import interfaces.ejemplo2.entidades.Numerador;
import interfaces.ejemplo3.entidades.BarraDeAvance;
import interfaces.ejemplo3.entidades.Frase;

public class Ejemplo3 {

    public static void main(String[] args) {
        Numerador num = new Numerador();
        Frase frase = new Frase("Hola! Estamos probando interfaces en Java.");
        BarraDeAvance barra = new BarraDeAvance('*');
        mostrarDetallesDelObjeto(num);
        mostrarDetallesDelObjeto(frase);
        mostrarDetallesDelObjeto(barra);
        for (int i=0; i < 5; i++) {
            num.incrementar();
            num.mostrar();
            barra.incrementar();
            barra.mostrar();
        }
    }

    private static void mostrarDetallesDelObjeto(Object objeto) {
        System.out.println("Clase del objeto: " + objeto.getClass().getName());
        System.out.println("Nombre simple de la clase: " + objeto.getClass().getSimpleName());
        boolean esMostrable = (objeto instanceof Mostrable);
        System.out.println("Es Mostrable? " + esMostrable);
        if (esMostrable) {
            System.out.println("La siguiente línea ejecuta el método mostrar()");
            ((Mostrable) objeto).mostrar();
        }
        System.out.println("Es Incrementable? " + (objeto instanceof Incrementable));
        System.out.println("----------------------------------------");
    }
}
```

Y esto es lo que vemos en la consola:

```
Clase del objeto: interfaces.ejemplo2.entidades.Numerador
Nombre simple de la clase: Numerador
Es Mostrable? true
La siguiente línea ejecuta el método mostrar()
0
Es Incrementable? true
----------------------------------------
Clase del objeto: interfaces.ejemplo3.entidades.Frase
Nombre simple de la clase: Frase
Es Mostrable? true
La siguiente línea ejecuta el método mostrar()
Hola! Estamos probando interfaces en Java.
Es Incrementable? false
----------------------------------------
Clase del objeto: interfaces.ejemplo3.entidades.BarraDeAvance
Nombre simple de la clase: BarraDeAvance
Es Mostrable? true
La siguiente línea ejecuta el método mostrar()
[]
Es Incrementable? true
----------------------------------------
1
[*]
2
[**]
3
[***]
4
[****]
5
[*****]
```

4. Uso concreto de interfaces

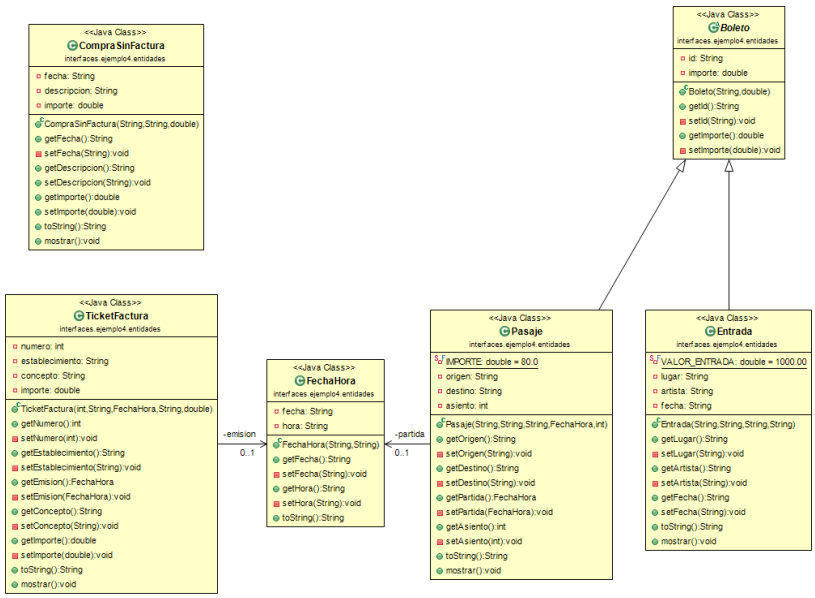
Veamos un caso más aproximado al uso real.

Supongamos que queremos llevar cuenta de los gastos que tenemos en una salida. Hay gastos de transporte, compras, otros gastos (la entrada de un cine, una función de teatro o un recital), gastos en comida, alguna cosa que nos gustó, etc. Lo ideal sería que tener de cada cosa un comprobante.

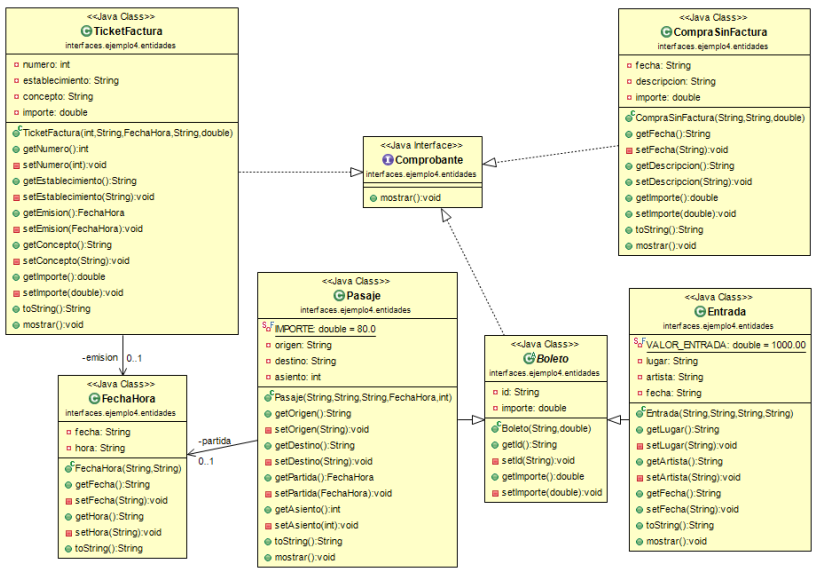
Imaginemos una salida a un recital fuera de la ciudad (por ejemplo en La Plata): tengo la entrada al recital, dos boletos de viaje (ida y vuelta), una comida rápida antes de entrar al recital y una cena tardía. Cada uno de estos eventos fue una compra y tengo el comprobante (bueno… de la comida rápida no porque fue en un puesto ambulante, pero una nota a mano me alcanza).

Luego de revisar cada tipo de gasto, veo que tanto la Entrada como los Pasajes son Boletos, mientras que los otros comprobantes son una Factura y una nota que hace de recordatorio del gasto callejero (una CompraSinFactura).

Este es UML con las clases descubiertas:



Ahora que el evento ya pasó, solamente me quedan el recuerdo… y los *comprobantes*. Y para poder aunarlos como tales, creamos la interfaz *Comprobante*, por lo que las clases quedan relacionadas así:



Ahora, las cuatro clases que tenemos son también *comprobantes*. Pero sin embargo, hacia la interfaz van sólo tres flechas que indican

implementación. La clase Boleto (compartida por Pasaje y Entrada) declara implementar *Comprobante* porque cada boleto que haya será un comprobante. No obstante, no implementa el método *mostrar()*, pues un boleto "ideal" no puede saber qué mostrar. Son sus subclases quienes implementan el método declarado en la interfaz, y por lo tanto Boleto es una clase *abstracta*.

Dado que ahora pueden ser identificados como comprobantes, podemos guardarlos en una colección y mostrarlos:

```
package interfaces;

import java.util.ArrayList;
import interfaces.ejemplo4.entidades.CompraSinFactura;
import interfaces.ejemplo4.entidades.Comprobante;
import interfaces.ejemplo4.entidades.Entrada;
import interfaces.ejemplo4.entidades.FechaHora;
import interfaces.ejemplo4.entidades.Pasaje;
import interfaces.ejemplo4.entidades.TicketFactura;

public class Ejemplo4 {
    public static void main(String[] args) {
        ArrayList<Comprobante> comprobantes = new ArrayList<>();
        comprobantes.add(new Entrada("AAZ1240002136", "Estadio Unico La Plata", "Superbanda", "22/09/17"));
        comprobantes.add(new Pasaje("X2134544", "CABA", "La Plata", new FechaHora("22/09/17", "18:30"), 27));
        comprobantes.add(new CompraSinFactura("22/09/17", "Sandwich y Bebida", 250.0));
        comprobantes.add(new Pasaje("X2141784", "La Plata", "CABA", new FechaHora("23/09/17", "01:00"), 18));
        comprobantes.add(new TicketFactura(123345, "Parrilla Los Nenes", new FechaHora("23/09/17", "02:42"), "Cena",
260.0));
        System.out.println("\nComprobantes de gastos:");
        for (Comprobante comprobante : comprobantes) {
            comprobante.mostrar();
        }
    }
}
```

y esta es la salida por consola:

```
Comprobantes de gastos:
*****************************************
Superbanda - Estadio Unico La Plata 22/09/17
Entrada #AAZ1240002136, $1000.00
*****************************************
-------------------------------------------------
- T r a n s p o r t e s   A U T O P I S T E R O S -
-                                               -
-       CABA    -       La Plata
-       Fecha y Hora: [22/09/17 18:30]
-       Asiento: 27      ID Pasaje #X2134544
-       Importe: $80.00
-                                               -
-------------------------------------------------
CompraSinFactura [fecha=22/09/17, descripcion=Sandwich y Bebida, importe=250.0]
-------------------------------------------------
- T r a n s p o r t e s   A U T O P I S T E R O S -
-                                               -
-       La Plata        -       CABA
-       Fecha y Hora: [23/09/17 01:00]
-       Asiento: 18      ID Pasaje #X2141784
-       Importe: $80.00
-                                               -
-------------------------------------------------
*****************
Factura 123345
Parrilla Los Nenes
[23/09/17 02:42]
Cena.......$260.00
*****************
```

5. Conclusión

La declaración de métodos abstractos permite que cada clase tenga su propia implementación de una misma "acción ideal". Un interfaz lleva este concepto al extremo. Una interfaz podrá verse como un "molde", una "declaración de principios". Dado que no permite implementaciones (al menos hasta la versión 8 de Java) solamente deja declarar nombres de métodos y adicionalmente atributos de tipos básicos y declarados como *static* y *final* (constantes). Así, todos sus métodos deberán ser implementados en otro lugar.

La principal diferencia entre una clase abstracta y una interfaz es que mientras la primera obliga a respetar el esquema definido por el árbol jerárquico de clases, la segunda proporciona un mecanismo de encapsulamiento de los métodos sin forzar el uso de herencia. Esta ventaja hace que una interfaz puede ser implementada por cualquier número de clases, permitiendo a cada clase compartir la misma interfaz de programación sin importar qué implementación hagan otras clases de la misma interfaz.

Tanto la "abstracción extrema" como la posibilidad de simular Herencia Múltiple hacen de las interfaces una herramienta importante dentro del mundo de la POO, posibilitando la construcción de definiciones sólidas más allá de las implementaciones actuales y futuras.