Abstracción I

¡Comencemos!

¿Qué aprenderás?

- Comprender una interface para separar código.
- Implementar polimorfismo mediante interfaces para el desarrollo de herencias entre clases.

Introducción

En este capítulo continuaremos con el polimorfismo en relación con la aplicación en interfaces, importantes a la hora de llevar a cabo el concepto de abstracción. Además, veremos la primera parte de lo que es abstracción en POO, ya que es un concepto que se utiliza bastante a la hora de hacer aplicaciones más dinámicas.

¡Vamos con todo!

Las interfaces

Como vimos anteriormente, se puede utilizar el polimorfismo entre superclases y subclases, pero no es la única forma de implementar el polimorfismo, ya que también existe algo llamado interfaces.

Las interfaces se declaran utilizando la palabra reservada “interface” en lugar de “class” y proveen una lista de prototipos de métodos, lo que significa que solo se declara tipo de retorno, nombre y parámetros de entrada. Otras clases pueden implementar (heredar) a las interfaces, para lo que deben agregar la palabra “implements” después del nombre de la clase. Al haber implementado una interface, la clase podrá contener sus propias versiones de los métodos de la interface.

```
   //Creación de interface
   public interface nombreInterface{
     void imprimirHola();
   }
   //Implementación de interface:
   public class nombreClase implements nombreInterface[,
nombreOtraInterface]{  @Override
     public void imprimirHola(){
        System.out.println("hola");
    }
   }
```

Además de estos prototipos de método en las interfaces se pueden crear constantes que, como su nombre lo indica, sirven para almacenar valores (como las variables) que no cambiarán.

Las interfaces permiten conocer la lista de métodos que tendrán las clases que las implementen sin conocer el comportamiento específico de cada una, ya que cada implementación puede ser diferente. Los métodos que se declaran en la interface deben existir en todas las clases que la implementen, por ende, ayudan a establecer la forma de las clases, lo que define protocolos de comunicación fácilmente, además de hacer la aplicación de fácil entendimiento para quienes tengan que modificarla.

Por ejemplo, al utilizar un control remoto de televisor, presionamos los botones sabiendo lo que hacen pero no nos interesa saber cómo lo hacen, solo queremos que haga lo que le pedimos que haga.

Imagen 1. Ejemplo de abstracción.
Fuente: Desafío Latam.

Las interfaces no ayudan mucho con la reutilización de código, pero lo mantienen ordenado.

Ejercicio Guiado: El juego

Veamos un ejemplo en código del uso de las interfaces y cómo podríamos utilizarlas aplicando polimorfismo con ellas. Crearemos una porción de la estructura de un juego donde solo se puede avanzar hacia adelante y saltar (algo así como escapar de un enemigo que rompe todo a su paso).

Paso 1: Crear un proyecto en Eclipse y crear la siguiente interface dentro de un package llamado `Interfaces`:

```
public interface Personaje {
 void mover(int x);
}
```

Paso 2: Crear un package `Personajes` y una implementación de `Personaje` llamada `Protagonista`.

```
public class Protagonista implements Personaje{
}
```

Paso 3: Importar la interface `Personaje` dentro de la clase `Protagonista`, y Eclipse arrojará un mensaje diciendo que debemos implementar los métodos de dicha interface, dándonos la opción de hacerlo automáticamente, tal como se muestra:

```
package Personajes;

import Interfaces.Personaje;

public class Protagonista implements Personaje {
}
```

Esto nos pide que agreguemos implementaciones de los métodos que no están en `Protagonista` y que existen en la interface, es decir, nos obliga a que `Protagonista` tenga la forma de su interface `Personaje`.

Paso 4: Dar clic a la primera opción para que Eclipse genere las implementaciones por nosotros; la clase quedaría así:

```
public class Protagonista implements Personaje{
  @Override
  public void mover(int x) {
    // TODO Auto-generated method stub
  }
}
```

La implementación del método no es más que una sobre-escritura del mismo.

Paso 5: Agregar una variable llamada `xActual` para indicar la posición del personaje y que `mover()` modifique esa variable.

```
public class Protagonista implements Personaje{

   private int xActual;

   @Override
   public void mover(int x){
   xActual = xActual + x;
  }
}
```

Paso 6: Crear otra clase llamada `Enemigo` en el package `Personajes` que implementa la interface `Personaje`. Esto con la idea de crear comportamientos específicos para `Enemigo` y `Protagonista`, es decir, el `Enemigo` avanzará desde el punto A al punto B de una forma progresiva y el `Protagonista` se moverá instantáneamente de un punto a otro.

```
public class Enemigo implements Personaje{
 private int xActual;

 @Override
 public void mover(int x){
 while(xActual < x){
 xActual++;
 }
 }
}
```

Paso 7: Crear una Interface de `Jugador` que le permita a las clases implementar un comportamiento de jugador.

```
package Interfaces;
   public interface Jugador {
   void saltar();
   void ejecutarAccion(String accion);
}
```

Paso 8: Implementar la interface `Jugador` en la clase `Protagonista`.

```
package Personajes;

import Interfaces.Jugador;
import Interfaces.Personaje;

public class Protagonista implements Personaje , Jugador {
...
}
```

Paso 9: Agregar los métodos que no hemos implementado gracias a la opción de la imagen y la clase queda así:

```
 private int xActual;

 @Override
 public void mover(int x){
   xActual = xActual + x;
  }
 @Override
 public void saltar() {
  // TODO Auto-generated method stub
 }
 @Override
 public void ejecutarAccion(String accion) {
  // TODO Auto-generated method stub
 }
```

Paso 10: Crear un comportamiento específico para el protagonista. Primero, modificar la implementación de `saltar()`, agregar una variable representando el eje "y" del plano (`yActual`), que actualmente tiene solo una dimensión y, posteriormente, haremos que el personaje suba y baje paulatinamente.

```
private int yActual = 1;
@Override
public void saltar() {
 //Aumentamos hasta 5
 while(yActual < 5){
   yActual++;
 }
 //Cuando sea 5, disminuimos a 1 nuevamente
 while(yActual > 1){
    yActual--;
  }
}
```

Ahora que tenemos la acción de saltar y mover, vamos a crear un método que permita llamar a estos comportamientos.

Paso 11: Crear la implementación del método `ejecutarAccion` (String accion).

```
@Override
public void ejecutarAccion(String accion) {
 if(accion.equals("saltar") && yActual == 1){
   saltar();
 } else if(accion.equals("avanzar")){
   mover(1);
 }
}
```

Y listo, hemos utilizado dos interfaces en el `Protagonista`, diciendo que tiene la forma de un `Personaje` controlado por el jugador gracias a la forma de la interface Jugador (polimorfismo).

Cierre

Hemos visto en este capítulo que en el mundo de la programación existe un concepto abstracto llamado “Polimorfismo”, el cual está directamente relacionado con la herencia entre clases. Es fundamental para cualquier desarrollador/a resolver problemas del día a día mediante este concepto, ya que cubre diversas necesidades en la creación de aplicaciones. Si a todo lo anterior le agregamos la incorporación de interfaces, podemos crear aplicaciones más robustas y compactas.