Práctica 8. Clases abstractas e interfaces

Clases abstractas

En los últimos capítulos, basamos nuestros ejemplos en torno a una clase Animal que luego heredamos en diferentes subclases según el tipo de animal que queríamos implementar. Usando las diferentes clases que definimos, podríamos crear objetos “Perro”, objetos “Gato” e incluso objetos “Animales”. Pero piénselo: ¿tiene sentido crear un objeto Animal? Cada animal que existe pertenece a una especie distinta y tiene sus propias características. No puede haber una instancia de Animal porque el concepto de "animal" es demasiado general.

Lo mismo podría decirse de una clase de vehículo. Hay diferentes tipos de vehículos, como automóviles, bicicletas, aviones, trenes, etc., pero no hay ningún objeto concreto que podamos llamar "vehículo" y eso es todo. Para crear una clase de vehículo, necesitamos saber cómo se mueve, qué transporta, su tamaño y otros atributos que nos ayudan a clasificarlo en su propia clase.

Java proporciona la palabra clave "abstracto" para denotar una clase de la que no se puede crear una instancia, en otras palabras, intentar crear un objeto de una clase abstracta producirá un error de compilación. Esto, a su vez, ahorra memoria en su aplicación Java, haciéndola más liviana en cuanto a memoria.

```
Public abstract class animal {
}
```

Quizás se esté preguntando, ¿cuál es el punto de tener una clase de la que no podemos crear objetos? Bueno, las clases abstractas son una forma conveniente de indicar que el comportamiento de una clase solo debe definirse en sus subclases. En el ejemplo del animal, no tendría mucho sentido definir los métodos comer (), dormir () o moverse (), porque cada animal come, duerme y se mueve a su manera. Cada subclase debe implementar esos métodos de acuerdo con sus necesidades. Podríamos definir algún código "base" en la superclase sin hacerlo abstracto, pero como nunca se accedería directamente a ese código, no tiene mucho sentido. Hay una cosa que aún debe colocarse dentro de una clase abstracta, y son los atributos que podría usar cualquier tipo de animal, como la edad, el nombre o el género. Su inicialización aún debería estar dentro del constructor de la clase, y esto se llamará desde el constructor de la subclase como siempre.

Cuando una clase abstracta tiene métodos que cambian cuando se heredan a una subclase, podemos definir esos métodos como abstractos para dejar la implementación a las subclases. Eche un vistazo a los métodos abstractos de nuestra clase Animal anterior:

```
public abstract void eat();
public abstract void sleep();
```

Cada clase que hereda de Animal **debe** proporcionar una implementación para estos métodos, la única excepción sería si la clase que hereda de Animal también es una clase abstracta. Una subclase que no proporciona implementaciones para todos los métodos abstractos en su superclase también debe declararse abstracta.

Ejercicio 1:

Intente crear una instancia de un objeto Animal después de declararlo abstracto. ¿Se compila? * Haga que los métodos de comer y dormir de la clase Animal sean abstractos como se muestra. Ahora intente * ejecutarlo y observe si hay algún error.

Dado que la clase "Animal" ahora es una clase abstracta, no podrá crear una instancia de un objeto "Animal", pero aún puede usar muchas funciones útiles que ya hemos aprendido en este curso, como usar un "Animal" variable para contener el objeto de sus subclases o instanciar una matriz "Animal" con ellos.

Ejercicio 2:

Haga una matriz “Animal” y ejecútela con muchos objetos diferentes de sus subclases y use la instrucción <<for>> para revisar sus elementos y hacer que cada uno de ellos * llame al método “eat ()”. ¿Tiene el comportamiento que esperabas? Usemos esta misma matriz, pero esta vez intentemos una instrucción << para cada >>, ¿funciona de la misma manera?

Interfaces

Las interfaces nos permiten incluir funcionalidades similares en diferentes clases. En otras palabras, dentro de una interfaz, podemos indicar uno o más métodos que serán definidos en la clase que **implementa** la interfaz. Esto es útil para compartir código entre clases que no necesariamente heredan de la misma superclase.

Las interfaces no son clases, no se pueden instanciar ni necesitan un constructor para el caso. Son, de forma simplificada, un modelo de funcionalidad que se compromete a seguir cuando lo implementa en una clase. No puede implementar una interfaz y solo definir la mitad de sus métodos, ni puede implementarla sin definir ningún método. **Debe**, por cualquier medio, definir la funcionalidad para cada uno de los métodos listados en la interfaz dentro de la clase que lo implementa, o de lo contrario el programa no se compilará.

Veamos un ejemplo, tenemos una clase de automóvil con algunos atributos y métodos simples:

```
public class Car{
    String model;
    int mileage;
    double speed;

    public Car (String model) {
        this.model = model;
    }

    public void start(){
        System.out.println(model+" is running");
    }

    public void accelerate(){
        System.out.println("Accelerating");
        speed++;
    }
}
```

Es importante que un vehículo comercial cumpla con los estándares de seguridad más básicos, como tener cinturones de seguridad. Podríamos agregar atributos y métodos para implementar la funcionalidad de los cinturones de seguridad, pero ¿qué pasa si necesitamos crear clases para otros tipos de vehículos que también necesitan cinturones de seguridad, pero no heredan de Car, como un avión? Como ya sabe, no podemos heredar de varias clases al mismo tiempo. En casos como este, podemos usar una herramienta llamada interfaz. Las interfaces son similares a las clases en el sentido de que deben definirse en su propio archivo (el nombre del archivo es el nombre de la interfaz).

```
public interface Safety{
    public boolean checkSeatBelts();
    public void lockSeatBelts();
}
```

Tenga en cuenta que solo se incluye la firma de los métodos. Esto funciona de manera similar a los métodos abstractos de clases abstractas, **deben** definirse en cualquier clase que implemente la interfaz. Para usar realmente los métodos en la interfaz, usamos la palabra clave **implements** como hemos mencionado:

```
public class Car implements Safety{
}
```

Una clase puede implementar cualquier número de interfaces siempre que se definan los métodos que incluyen. Aquí está nuestra implementación de los métodos contenidos en la interfaz de seguridad:

```
public class Car implements Safety{
    String model;
    int mileage;
    double speed;

    public Car (String model) {
        this.model = model;
    }

    public void start(){
        if(checkSeatBelts())
            System.out.println(model+" is running");
        else
            System.out.println("There was a problem starting" + model);
    }

    public void accelerate(){
        System.out.println("Accelerating");
        speed++;
    }

    public boolean checkSeatBelts(){
        System.out.println("Seatbelts are working");
        return true;
    }

    public void lockSeatBelts(){
        System.out.println("Seatbelts are locked");
    }
}
```

Antes de que podamos arrancar el coche, hay una llamada al método checkSeatBelts (), pero como nuestro coche es perfectamente seguro, siempre devuelve verdadero en este ejemplo. Ésta es una forma de implementar los métodos incluidos en la interfaz de seguridad. Para crear una clase de avión, o cualquier vehículo que necesite cinturones de seguridad, se puede utilizar el mismo proceso.

Ejercicio 3:
Cree una clase Car que implemente la interfaz definida anteriormente y ejecute un código para verificar * el resultado de sus métodos.

Aparte de los métodos, otra cosa que puede incluir dentro de una interfaz son las constantes. Las constantes son atributos que se definen como "estáticos finales", generalmente con el modificador de acceso "público" para su uso general. En este caso, "estático" significa que, independientemente de la implementación de la interfaz, el valor de este atributo es siempre el mismo para todos, y los cambios realizados por una implementación se reflejarán también en todas las demás implementaciones. "Final" es de donde realmente proviene la parte constante, indica que el atributo que sigue solo se inicializará una vez y nunca más, dándole un valor permanente para el resto de la instancia del programa. Las constantes deberían ser útiles para todas las clases que implementan la interfaz.

```
public static final double pi = 3.141592653;
public static final double e = 2.7182818284;
```

Si encuentra que otra clase nunca usa una constante de la interfaz, debería considerar mover la constante a otra diferente. Hablando de que…

Interfaz de herencia
Las interfaces pueden extender una interfaz diferente, pero no tienen que definir el comportamiento de los métodos heredados, esto simplemente le dice a la clase que implementa la interfaz secundaria que también necesita implementar todos los métodos de su predecesor. También comparte las constantes definidas en las otras interfaces cuando se implementa.

```
public interface SafetyLights extends Safety {
    public void checkLights();
    public void toggleLights();
}
```

Esto generalmente se hace cuando dos interfaces tienen comportamientos que coexisten entre sí y tienen sentido en el contexto, como una interfaz de grabadora y una interfaz de reproductor de audio, que podrían heredarse en una interfaz de reproductor de música.

Interfaz como variables
De la misma manera que usa variables de superclase como contenedores para los objetos de sus subclases, puede usar una variable de interfaz para contener cualquier objeto de una clase que implemente esta misma interfaz, con las mismas restricciones que vienen con el primer caso. (Solo puede usar los métodos de la interfaz, no puede llamar a las variables / constantes definidas fuera de la definición de la interfaz).

```
public static void main (String[] args){
    Safety c = new Car("Ferrari");

    c.checkSeatBelts();
    c.start();
}
```

Ejercicio 4:
Usando su clase de automóvil y la interfaz de seguridad, ejecute el código anterior y verifique el resultado. Si * hay errores, ¿cómo los solucionaría?

Las matrices de interfaz también son posibles y siguen las mismas reglas que las matrices de superclases, puede identificar el tipo de objeto que contiene con instanceof y luego convertirlo para recuperar su funcionalidad deseada.

```
public static void main (String[] args){
    Safety [] testCars =  new Car [3];

    testCars[0] =  new Car("Ferrari");
    testCars[1] =  new Car("Audi");
    testCars[2] =  new Car("Nissan");

    for(Safety c: testCars){
        if(c instanceof Car)
            ((Car)c).start();

    }
}
```

Práctica:

Una compañía paga a sus empleados por semana los empleados son de cuatro tipos:

- Empleados asalariados que reciben un salario semanal fijo, sin importar el número de horas trabajadas.
- Empleados por horas, que reciben un sueldo por hora y pago por tiempo extra, por todas las horas trabajadas que excedan 40 horas.
- Empleados por comisión, que recién un porcentaje de sus ventas.
- Empleados asalariados por comisión, que reciben un salario base más un porcentaje de sus ventas.

Para este periodo de pago, la compañía ha decidido recompensar a los empleados asalariados por comisión, agregando un 10% a sus salarios base. La compañía desea implementar una aplicación en Java que realice sus cálculos de nómina en forma polimórfica.

Actividades:

- Hacer un reporte con portada de hoja completa que incluya el código fuente, el link de su código en GitHub y una captura de pantalla de la práctica funcionando.
- En caso de contener menús será necesario agregar en el reporte una captura de pantalla por cada opción seleccionada del menú.
- Subir a blackboard código fuente (archivos java) y reporte como archivos separados, no se aceptarán archivos en carpetas comprimidas.