Práctica 6. Polimorfismo

Polimorfismo

Cuando hablamos de herencia, aprendimos que una clase puede hacer uso de los métodos y atributos de otra clase extendiéndola. Esta es una característica muy útil de la programación orientada a objetos, que permite a los desarrolladores reutilizar código de diferentes clases. Para comprender mejor cómo se utilizan las diferentes clases de padres e hijos en el mismo programa, echemos un vistazo a nuestra clase de animales:

```
public class Animal{
    String name;
    int age;

    public Animal (String name, int age){
        this.name = name;
        this.age = age;
    }

    public void eat (){
        System.out.println(name + "is eating");
    }

    public void sleep () {
        System.out.println(name + "is sleeping");
    }
}
```

Ahora, para este ejemplo, supongamos que queremos heredar la clase Animal para representar a nuestros dos animales favoritos: perros y gatos. Debemos prestar mucha atención a los diferentes atributos y métodos contenidos en Animal y considerar si alguno de ellos debería implementarse de manera diferente cuando se extienda a otra clase, en este caso, que representa a un perro y un gato. Teniendo en cuenta el método del sueño, podríamos decir que ambos animales duermen de la misma manera, por lo que no hay mucha necesidad de modificar su comportamiento. Pero, ¿tanto los perros como los gatos comen exactamente la misma comida? Realmente no, así que podemos aprovechar esta oportunidad para demostrar cómo anular un método en una subclase.

Esto significa que crearemos un método con la misma firma exacta que otro en una clase principal, pero modificando el código interno. Cada vez que se llama al método utilizando el objeto de la clase secundaria, se ejecutará el código modificado en lugar del de la clase principal.

```
public class Dog {

    public Dog(String name, String breed, int age){
        super(name,age);
    }

    @Override

    public void eat (){
        System.out.println(name + "is eating Pedigree");
    }
}
```

Ahora, si llamáramos al método "eat ()" con un objeto Animal, Dog y Cat, veríamos que cada método eat ahora imprime una línea diferente en la consola. La anotación "**@Override**" es un tipo especial de etiqueta que le indica a Java que el siguiente método está siendo reemplazado de una clase principal. Nuestro

código se compilará bien sin él, pero por convención, se recomienda incluir esta etiqueta cuando queramos anular un método para documentar mejor el código.

Ejercicio 1:
Realiza las clases de Animales, Perros y Gatos. Implemente un programa que cree un objeto para * cada uno de ellos y llame a sus métodos de dormir y comer. ¿Qué líneas corresponden a los métodos * definidos en la clase principal y cuáles a las clases secundarias?

Tenga en cuenta que los constructores no se pueden anular de la misma manera que con los métodos, ya que los constructores no son realmente métodos. Cuando creamos una subclase sin especificar un constructor, Java siempre ejecutará el constructor en la superclase. Por otro lado, si definimos un constructor en la subclase, el constructor de la superclase se ejecutará primero (en la llamada a "super ()"), y luego las instrucciones que hayamos definido. Por lo que está bastante claro que en realidad no estamos anulando nada.

Variables polimórficas
Ya debería estar familiarizado con la forma en que instanciamos un objeto:

```
Animal someAnimal = new Animal();
```

Declaramos una variable de la clase "Animal" para contener nuestro objeto "Animal" instanciado con la palabra clave "new" y el constructor de la clase. A estas alturas, ya sabes que cualquier clase que amplíe “Animal” consiste en una definición más específica del concepto, pero incluso si tienen características diferentes, seguirían siendo animales, sin importar cuán diferentes sean entre sí. Entonces, si un "Dog" sigue siendo un "Animal" por extensión, esto es algo que podríamos hacer sin ningún problema:

```
Animal someAnimal =  new Dog();
```

Aquí, nuestro objeto “Dog" se interpreta como un animal más y se almacena en una variable "Animal". Esto hace que sea imposible acceder a los métodos o atributos exclusivos de nuestra clase “Dog” y eso se debe a que nos referimos a él como un "Animal" mediante el uso de una variable de clase "Animal", y la clase "Animal" no sabe lo que hay dentro de la clase "Dog", por lo que no podemos acceder a ellos con esta variable.
Entonces, si no podemos acceder a los atributos o métodos de nuestras subclases cuando almacenamos una instancia de él en una variable de su superclase, ¿cuál es el propósito? Bueno, si recuerdas el "método anulado" de antes, cuando llamamos a este método anulado con un objeto de subclase, hace lo que está definido dentro de la subclase en lugar de la definición original en la superclase. Entonces, si tenemos un objeto “Dog” dentro de una variable de clase "Animal", ¿a qué método llamaría la variable si usáramos un método anulado?

Ejercicio 2:
Cree una variable de clase "Animal" e inicialícela con un objeto "Dog", use todos los métodos * de la clase Animal" y "Dog" y vea lo que sucede cuando llama a:
- Un método escrito exclusivamente en el " Clase Animal
- La clase "Dog" anula un método.
- Y un método exclusivo de la clase "Dog"

Como probablemente hayas visto, cuando tenemos un objeto de subclase dentro de una de sus variables de superclase, si llamamos a un método que existe dentro de la superclase y está anulado en la subclase, el método que la variable termina llamando es el que está dentro de la subclase. definición.
Otra cosa a tener en cuenta es que, si bien podemos almacenar un objeto de subclase dentro de una variable de superclase, no se puede decir lo mismo del caso opuesto. A continuación, se muestran algunos ejemplos de variables de superclase y subclase que contienen diferentes tipos de objetos relacionados.

```
Animal anAnimal =  new Animal();
Dog aDog =  new Dog();
Animal anotherAnimal = new Dog();
Dog doggie =  new Animal();      //Error
Cat another =  newDog ();        //Error
```

Aquí puede ver que algunos de los casos anteriores le darán un error dependiendo de la relación entre el objeto y la variable que lo contiene. En pocas palabras, podemos tener un "Dog" dentro de una variable "Animal" porque un Perro es un Animal, pero cuando intentamos crear una instancia de un objeto "Animal" dentro de una variable "Dog", el compilador nos detendrá en seco. y, en palabras simples, dirá: *"¡Espera, un perro puede ser un animal, pero no hay nada que diga que un animal es un perro!"*. Para continuar con esta última nota, un animal no siempre se refiere a un perro, podría ser un gato o un elefante por lo que sabemos y, efectivamente, ninguno de ellos se considera perro, ¿verdad? Con el mismo pensamiento en mente, podemos estar absolutamente seguros de que un "Dog" no es un "Gato", y aunque ambos son considerados como un "Animal", tienen características propias que los hacen muy diferentes a cada uno. otro, haciéndolo una coincidencia incompatible.

Polimorfismo en argumentos y parámetros

Ahora que conocemos las variables polimórficas, es hora de que comencemos a ver cómo se puede aplicar de manera útil en nuestros programas.
Para esta parte, vamos a necesitar utilizar un atributo booleano simple dentro de la clase animal, que se llamará <<vaccinated>>.

```
public class Animal{
    String name;
    int age;
    boolean vaccinated;
```

Es bueno recordar:
¡Encapsula tus atributos! Una buena práctica en POO es hacer que sus clases sean lo más independientes posible del programa principal, para hacerlo, use el modificador de acceso privado en sus atributos y agregue un getter y un setter para cada uno de ellos.
Continuaremos con la lección asumiendo que, aunque no los señalemos en los ejemplos, tenemos un setter y un getter para cada atributo en cada clase.

Vamos a crear una clase "Veterinarian", que tendrá atributos muy simples para empezar:

```
public class Veterinarian {
    String name;
    double salary;

    public Veterinarian (String name, double salary){
        this.name = name;
        this.salary = salary;
    }

    public void vaccinate (Animal animal){
        animal.setVaccinated (true);
        System.out.println(animal.getName ()+ "was vaccinated");
    }
}
```

Aquí, podemos ver que nuestro "Veterinario" tiene un método que recibe un objeto "Animal" y luego procede a usar un setter para hacer que el atributo <<vaccinated >> sea verdadero. Luego, imprime el nombre del animal que vacunaron y declara este hecho en la consola.
Cosas simples, ¿verdad? Bueno, ¿y si te dijera que necesitamos enviar a través del argumento del método "vaccinate" un objeto "Dog"? dirías que:

a) No sería posible hacerlo, ya que no hay ningún método sobrecargado en la clase que recibe un objeto "Dog".
b) Funcionaría bien.
c) No hay ningún atributo "vaccinated" presente en la clase "Dog", por lo que no lo afectaría.

La respuesta correcta es ¡B !, ¡funcionaría bien! Puede que ya hayas averiguado por qué, o no, pero vamos a explicarlo para que quede completamente claro.
Ya has oído hablar de la sobrecarga de métodos, que explica cómo necesitas otro método, similar al primero y con diferentes parámetros para enviar diferentes argumentos a través de la llamada a el mismo método. Bueno, ya establecimos que un "Dog" es un "Animal" y, como tal, puede estar contenido dentro de una variable "Animal" a lo largo de la última sección. Esto todavía se aplica aquí, porque la forma en que funciona es: enviamos un objeto "Dog" a través de los argumentos del método, pero el método lo contiene dentro de la variable "Animal" llamada <<animal>>, y luego, como heredamos todos los atributos y dentro de la clase "Animal" con la clase "Dog", podemos usar el método setVaccinated () definido en la clase "Animal" sin problemas.

Matriz polimórfica
Supongamos que queremos crear un grupo de objetos que comparten una clase principal común pero que no son necesariamente la misma clase. Podemos crear una matriz de los objetos de su clase principal y podríamos acceder a los atributos y métodos que comparten con su clase principal. Teniendo en cuenta nuestra clase “Animal”, podemos crear una serie de animales que pueden contener cualquier animal, incluidos perros y gatos. ¡Asegúrese de que los objetos de la matriz estén instanciados correctamente antes de intentar acceder a ellos!

```
Animal [] animalArray = new Animal [3];
animalArray[0] = new Animal("An animal", 1);
animalArray[1] = new Dog("Chispita", 4);
animalArray[2] = new Cat("Miau", 2);
```

Aquí podemos ver que una matriz de objetos de clase padre puede contener cualquier objeto de clase hijo. Podemos usar un bucle for para iterar a lo largo del contenido de la matriz para acceder a sus setters/getters o métodos, por ejemplo. Esta parece una buena oportunidad para introducir una variante del bucle for, conocida como "**for-each**". Los llamamos para una declaración, pero en lugar de definir variables de contador, simplemente definimos un objeto de marcador de posición de la misma clase que la matriz, insertamos un símbolo de dos puntos (“**:**”) y el nombre de la matriz. Java determinará automáticamente que queremos iterar la matriz de principio a fin, asignando los objetos contenidos en cada índice al objeto marcador de posición, hasta que llegue al final de la matriz. ¿No es inteligente?

```
for(Animal a: animalArray){
    a.eat();
}
```

Ejercicio 3:
Pruebe el bucle for-each con la matriz de animales.

Esto está perfectamente bien si solo necesitamos acceder a atributos y métodos ya definidos en la superclase, pero ¿qué sucede si necesitamos utilizar la funcionalidad definida en una subclase? Acceder a un método definido en las clases “Dog” o “Cat” nos daría un error si intentáramos acceder a ellos desde el arreglo definido anteriormente ya que solo conoce los atributos y métodos definidos en “Animal”. Para estos casos, Java tiene el operador ***instanceof*** que, cuando se usa junto con una instrucción if, nos permite verificar si un objeto fue instanciado desde una clase específica. Agreguemos un método *bark()* a nuestra clase “Dog”, solo con fines de demostración:

```
public void bark(){
    System.out.println(name + ":Woof");
}
```

La siguiente parte es muy importante: intentaremos acceder a un método que no está definido en la clase principal. Si intentáramos llamar al método "bark()" desde el objeto Animal, obtendríamos un error de compilación. Como estamos seguros de que solo los perros deben ladrar, podemos lanzar el objeto Animal a un objeto Dog y luego llamar al método de ladrido.

```
for(Animal a: animalArray){
    a.eat();

    if (a instanceof Dog){
        ((Dog) a ).bark();
    }
}
```

La transmisión de una variable le dice a Java que sabe lo que está haciendo y que debe ignorar cualquier posible advertencia o error al acceder a métodos o atributos no definidos. ¡Úsalo con precaución!

Ejercicio 4:
Pruebe la salida del código anterior.

Desafío
● Implementar una clase "Zoo" que contenga una ArrayList de diferentes objetos Animal y métodos para agregar, eliminar y mostrar animales.
● Agregue más funcionalidad a la clase de veterinario.
● Crear diferentes subclases de animales con sus propios métodos.

Práctica:

1. Utilizar los conceptos obtenidos en esta práctica para crear un programa que utilice una clase “Zoo” que contenga un ArrayList de diferentes objetos animales y métodos para agregar, eliminar y mostrar animales.
2. Agregue mas funcionalidades a la clase “Veterinarian”
3. Crear diferentes subclases de animales con sus propios métodos.

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.