Herencia en java

Autor: Luis José Sánchez González

(adaptación Antonio H.)

La herencia es una de las características más importantes de la POO. Si definimos una serie de atributos y métodos para una clase, al crear una subclase, todos estos atributos y métodos siguen siendo válidos.

En el apartado anterior se define la clase `Animal`. Uno de los métodos de esta clase es `duerme`. A continuación podemos crear las clases `Gato` y `Perro` como subclases de `Animal`. De forma automática, se puede utilizar el método `duerme` con las instancias de las clases `Gato` y `Perro` ¿no es fantástico?

La clase `Ave` es subclase de `Animal` y la clase `Pinguino`, a su vez, sería subclase de `Ave` y por tanto hereda todos sus atributos y métodos.

Clase abstracta (`abstract`)

Una clase abstracta es aquella que no va a tener instancias de forma directa, aunque sí habrá instancias de las subclases (siempre que esas subclases no sean también abstractas). Por ejemplo, si se define la clase `Animal` como abstracta, no se podrán crear objetos de la clase `Animal`, es decir, no se podrá hacer `Animal mascota = new Animal()`, pero sí se podrán crear instancias de la clase `Gato`, `Ave` o `Pinguino` que son subclases de `Animal`.

Para crear en Java una subclase de otra clase existente se utiliza la palabra reservada `extends`. A continuación se muestra el código de las clases `Gato`, `Ave` y `Pinguino`, así como el programa que prueba estas clases creando instancias y aplicándoles métodos. Recuerda que la definición de la clase `Animal` se muestra en el apartado anterior.

```
/**
 * Gato.java
 * Definición de la clase Gato
 * @author Luis José Sánchez
 */

public class Gato extends Animal {

  private String raza;


  public Gato (Sexo s, String r) {
    super(s); //en la primera línea se llama al constructor del padre
    raza = r;
  }

  public Gato (Sexo s) {
    super(s);
    raza = "siamés";
  }

  public Gato (String r) {
    super(Sexo.HEMBRA);
    raza = r;
  }

  public Gato () {
    super(Sexo.HEMBRA);

    raza = "siamés";
  }

  public String toString() {
     return super.toString() //se llama al método del padre aprovechándolo
            + "Raza: " + this.raza //y se le añade información después
            + "\n*************************\n";
  }
```

```
  /**
   * Hace que el gato maulle.
   */
  public void maulla() {
    System.out.println("Miauuuu");
  }

  /**
   * Hace que el gato ronronee
   */
  public void ronronea() {
    System.out.println("mrrrrrr");
  }

  /**
   * Hace que el gato coma.
   * A los gatos les gusta el pescado, si le damos otra comida
   * la rechazará.
   *
   * @param comida la comida que se le ofrece al gato
   */
  public void come(String comida) {
    if (comida.equals("pescado")) {
      System.out.println("Hmmmm, gracias");
    } else {
      System.out.println("Lo siento, yo solo como pescado");
    }
  }

  /**
   * Pone a pelear dos gatos.
   * Solo se van a pelear dos machos entre sí.
   *
   * @param contrincante es el gato contra el que pelear
   */
  public void peleaCon(Gato contrincante) {

    if (this.getSexo() == Sexo.HEMBRA) {
      System.out.println("no me gusta pelear");
    } else {
      if (contrincante.getSexo() == Sexo.HEMBRA) {
        System.out.println("no peleo contra gatitas");
      } else {
        System.out.println("ven aquí que te vas a enterar");
      }
    }
  }
}
```

Observa que se definen nada menos que cuatro constructores en la clase `Gato`. Desde el programa principal se dilucida cuál de ellos se utiliza en función del número y tipo de parámetros que se pasa al método. Por ejemplo, si desde el programa principal se crea un gato de esta forma

```
Gato gati = new Gato();
```

entonces se llamaría al constructor definido como

```
public Gato () {
  super(Sexo.HEMBRA);
  raza = "siamés";
}
```

Por tanto `gati` sería una gata de raza siamés. Si, por el contrario, creamos `gati` de esta otra manera desde el programa principal

```
Gato gati = new Gato(Sexo.MACHO, "siberiano");
```

se llamaría al siguiente constructor

```
public Gato (Sexo s, String r) {
  super(s);
  raza = r;
}
```

y `gati` sería en este caso un gato macho de raza siberiano.

La palabra `super()` hace una llamada al método equivalente de la superclase. Fíjate que se utiliza tanto en el constructor como en el método `toString()`. Por ejemplo, al llamar a `super` dentro del método `toString()` se está llamando al `toString()` que hay definido en la clase `Animal`, justo un nivel por encima de `Gato` en la jerarquía de clases.

¿Qué ocurre si olvidamos poner super(...) como primera línea del constructor de la subclase? Hay dos posibilidades:

- si la superclase tiene un constructor sin parámetros **super()** , el compilador lo incluirá en segundo plano de forma automática y no saltará un error. De cualquier manera se considera contrario al buen estilo de programación, ya que no queda claro si se trata de un olvido. Por ello incluiremos siempre la palabra clave super y sus parámetros entre paréntesis.

- La otra posibilidad es que no haya un constructor sin parámetros, en cuyo caso saltará un error.

Para los constructores, a modo de resumen, la inicialización de un objeto de una subclase comprende dos pasos. La invocación al constructor de la superclase (primera línea del constructor: super…) y el resto de instrucciones propias del constructor de la subclase.

A continuación tenemos la definición de la clase `Ave` que es una subclase de `Animal`.

```java
/**
 * Ave.java
 * Definición de la clase Ave
 * @author Luis José Sánchez
 */

public class Ave extends Animal {

  public Ave(Sexo s) {
    super(s);
  }

  public Ave() {
    super();
  }

  /**
   * Hace que el ave se limpie.
   */
  public void aseate() {
    System.out.println("Me estoy limpiando las plumas");
  }

  /**
   * Hace que el ave levante el vuelo.
   */
  public void vuela() {
    System.out.println("Estoy volando");
  }
}
```

9.5.1 Sobrecarga de métodos

Un método se puede **redefinir** (volver a definir con el mismo nombre) en una subclase. Por ejemplo, el método `vuela` que está definido en la clase `Ave` se vuelve a definir en la clase `Pinguino`. En estos casos, indicaremos nuestra intención de sobreescribir un método mediante la etiqueta `@Override`.

Si no escribimos esta etiqueta, la sobreescritura del método se realizará de todas formas ya que `@Override` indica simplemente una intención. Ahora imagina que quieres sobreescribir el método `come` de `Animal` declarando un `come` específico para los gatos en la clase `Gato`. Si escribes `@Override` y luego te equivocas en el nombre del método y escribes `comer`, entonces el compilador diría algo como: "¡Cuidado! algo no está bien, me has dicho que ibas a sobreescribir un método de la superclase y sin embargo `comer` no está definido".

A continuación tienes la definición de la clase `Pinguino`.

```java
/**
 * Pinguino.java
 * Definición de la clase Pinguino
 * @author Luis José Sánchez
 */

public class Pinguino extends Ave {

  public Pinguino() {
    super();
  }

  public Pinguino(Sexo s) {
    super(s);
  }

  /**
    * El pingüino se siente triste porque no puede volar.
    */
  @Override
  public void vuela() {
    System.out.println("No puedo volar");
  }
}
```

Con el siguiente programa se prueba la clase `Animal` y todas las subclases que derivan de ella. Observa cada línea y comprueba qué hace el programa.

```java
/**
 * PruebaAnimal.java
 * Programa que prueba la clase Animal y sus subclases
 * @author Luis José Sánchez
 */

public class PruebaAnimal {
  public static void main(String[] args) {

    Gato garfield = new Gato(Sexo.MACHO, "romano");
    Gato tom = new Gato(Sexo.MACHO);
    Gato lisa = new Gato(Sexo.HEMBRA);
    Gato silvestre = new Gato();

    System.out.println(garfield);
    System.out.println(tom);
    System.out.println(lisa);
    System.out.println(silvestre);
```

```
    Ave miLoro = new Ave();
    miLoro.aseate();
    miLoro.vuela();

    Pinguino pingu = new Pinguino(Sexo.HEMBRA);
    pingu.aseate();
    pingu.vuela();
  }
}
```

En el ejemplo anterior, los objetos `miLoro` y `pingu` actúan de manera **polimórfica** porque a ambos se les aplican los métodos `aseate` y `vuela`.

Polimorfismo

En Programación Orientada a Objetos, se llama **polimorfismo** a la capacidad que tienen los objetos de distinto tipo (de distintas clases) de responder al mismo método.