Herencia en Java

Las jerarquías de las clases

Sabemos que resolver un problema utilizando objetos implica identificar las entidades que participan en el problema. Asimismo, debemos modelar cómo interactúan esas entidades entre sí. Entonces, definimos clases, con sus atributos y operaciones y arribamos a la definición de responsabilidades de esas clases.

Como vimos, cada clase deberá tener sus responsabilidades bien definidas. Hecho el análisis, podríamos llegar a detectar que algunas clases (y lo que hacen o deben hacer) son demasiado parecidas entre sí. Asimismo, veremos que algunas clases que se parezcan tanto en atributos como en operaciones pueden tener algunas diferencias bien marcadas. Es importante, entonces, dar a esta multiplicidad de clases algún tipo de **orden**. Según estas similitudes y diferencias, podemos agrupar todos aquellos atributos y operaciones en común en un lugar y dejar aisladas las diferencias.

Definiremos así un **orden jerárquico** entre las clases: por un lado, las que son más genéricas, más abstractas, más conceptuales; por otro lado, las que van especificando los atributos y los comportamientos. Hablamos, entonces, de un orden jerárquico superior, más genérico, y de un orden jerárquico inferior, más específico. No se trata de agrupar según mejor o peor, es solo un ordenamiento en cuanto a la especificidad de las características y operaciones de las clases.

Relación ES-UN

Para este tema, tomemos a los perros. ¿Quién no ama a los perros? Todos los perros tienen un nombre y una edad, y todos ladran y juegan. Es así como podemos decir que un perro ES-UN perro. Porque **tiene edad y nombre, y ladra y juega**. Aquí es importante prestar atención a los verbos. Con el tiempo iremos viendo que es más importante las operaciones de la clase que sus atributos. Es más importante lo que un objeto hace que aquellos que almacena. Porque recordemos: si tiene un atributo, entonces debería ser privado y tener una operación para acceder o modificarlo.

Si analizamos un Yorkshire Terrier (o “yorkie”, como se lo llama comúnmente), veremos que juega y ladra; si analizamos un Siberian Husky, también veremos cómo juega y ladra, aunque

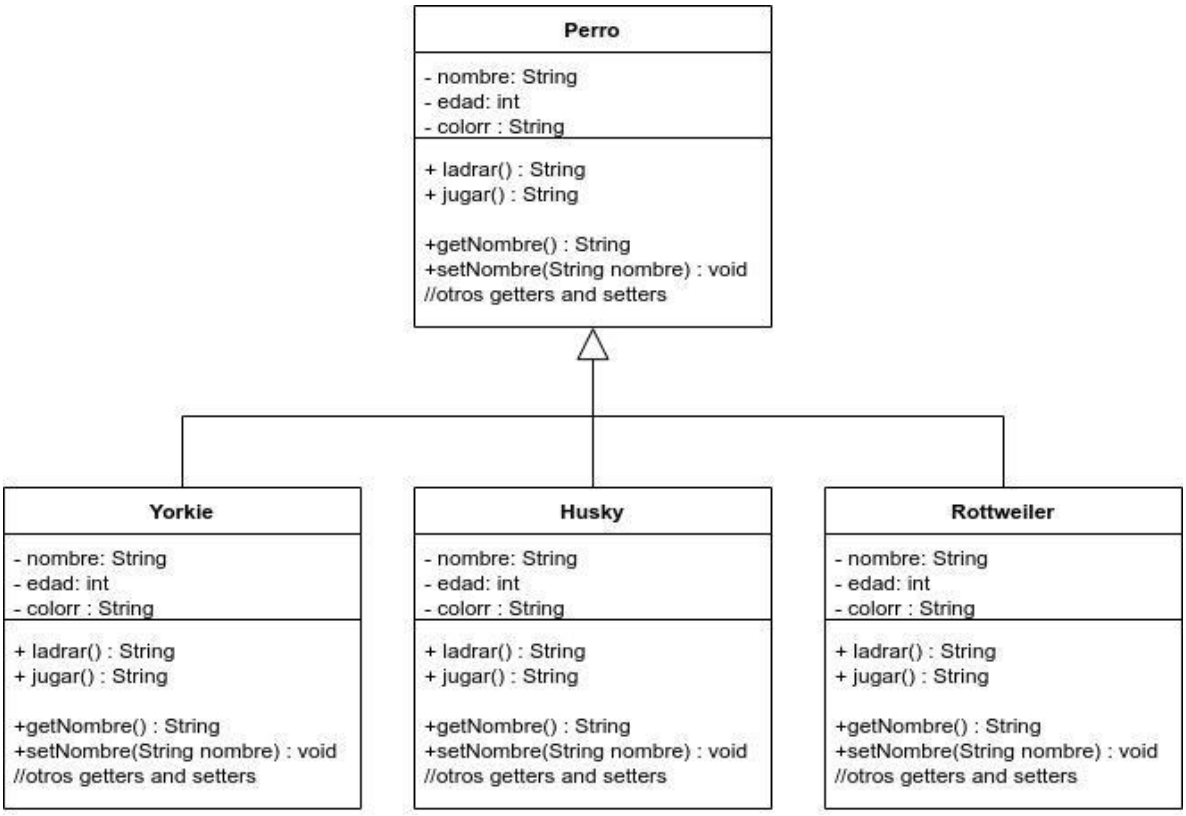
lo haga de manera muy distinta al yorkie. Entonces, tanto el yorkie como el husky ladran y juegan, y ambos tienen nombre y edad. Por lo tanto, podría asumirse que si tienen y hacen todo lo que hace un perro, el husky ES-UN Perro y el yorkie también ES-UN Perro.

El conjunto de las cosas que la clase hace (una de las cosas que la clase hace es tener atributos, definiendo *getters*/*setters*), es decir, el conjunto de operaciones públicas se conoce con el nombre de "Interfaz Pública" de la clase.

Por todo esto, podemos decir que un Yorkie es un Perro. Un Yorkie hace, al menos, todo lo que hace un perro. Por lo tanto, es un Perro.

Si analizamos otro negocio, y siguiendo el mismo tren de pensamiento, podríamos decir que Profesor ES UN Empleado, y más aún: un Empleado ES UNA Persona, por lo tanto, un Profesor ES UNA Persona.

Si analizamos el caso de nuestros amados pichichos, podríamos llegar a una jerarquía como la siguiente:



Algo de notación UML habíamos visto anteriormente. Acá solo se introduce el concepto nuevo de "extensión" o "herencia", que es una flecha, sin relleno; típicamente, cuando hay más de una, tienen a unirse en un nodo único, aunque no es requisito.

En nuestro universo, introdujimos una raza más, el Rottweiler, que suele tener una reputación de ser un perro un poco agresivo respecto de los otros dos, pero también EN-UN perro. Hace también lo que todos los otros perros: ladrar y jugar además de tener nombre, edad, color, etc., atributos a los que accedemos por *getters* y *setters*.

Definimos, entonces, a la **herencia** como es un "ordenamiento entre clases que define una **relación ES-UN**". Decimos que el Yorkie, el Terrier, el Husky o el Rottweiler ES UN perro, porque tiene y hace todo lo que hace un perro.

¿Para qué nos sirve la herencia?

Esta es una pregunta interesante, ya que afirmamos que la herencia es uno de los pilares de la orientación a objetos. Si analizamos el esquema anterior, tanto Yorkie como Husky, así como también Rottweiler hacen LO MISMO que hace el perro. Si hacen lo mismo que el perro, ¿para qué escribir el código de lo que hacen una y otra vez en cada raza? ¿No sería más conveniente escribirlo una sola vez en la clase Perro y que Rottweiler, Yorkie, Siberian Husky, Retriever, Poodle, Pitbull, etc., "obtengan" este comportamiento desde perro?

De hacer esto, decimos que Yorkie, Husky, etc., **"heredan"** el comportamiento de un perro, es decir, la clase Yorkie hereda de la clase Perro todas sus operaciones.

Supongamos la clase Perro:

```
public class Perro {

    private int edad;
    private String nombre;

    public String ladrar() {
        return "WOF!";
    }
    public String jugar() {
        return "JUGAR!";
    }
    //getters y setters
}
```

Si quisiéramos crear una clase Yorkie y mostrar que Yorkie ES UN perro, entonces tenemos que hacer:

```
public class Yorkie extends Perro {

}
```

Con la palabra **"extends"** indico, en Java, que un Yorkie es un perro y, por tanto, **hereda** todo su comportamiento, es decir, Yorkie heredará todos los atributos y los métodos de Perro. **¡Pero atención! Heredará todos los atributos y los métodos no privados**, esto es, todo lo que esté definido como *private* en Perro es privado a esa clase y no se pasa a ninguna otra, ni siquiera su clase hija Yorkie.

Si establecimos que un Yorkie es un Perro y lo plasmamos en código, entonces a partir de este punto podemos hacer:

```
Perro p = new Perro();
System.out.println(d.ladrar()); // < GUAU!

Yorkie y = new Yorkie();
System.out.println(y.ladrar());// < GUAU!
```

Si observamos el código de más arriba, en la clase Yorkie no se ha definido un método "ladrar". Pero sí se está heredando desde la clase Dog. Para que este código compile y luego funcione, lo que el compilador hace es lo siguiente:

¿La operación ladrar, sin parámetros, está definida en Yorkie?

a. Sí, entonces se compila correctamente y se puede ejecutar luego.

b. No, intento ir a buscarla a la clase de la que extiende Yorkie, es decir, en Dog.

c. ¿Está la operación (método) ladrar, que no recibe parámetros en Perro?

Sí, entonces la sintaxis es correcta.

No, intento buscarla en la clase de la que extiende Dog.

¿Está en esa clase?... Y repetir así sucesivamente hasta encontrar el método buscado. Si no se encuentra en ninguna parte de la cadena (de la jerarquía), entonces el código no puede compilarse correctamente.

Al momento de la ejecución, ocurre algo similar. El *runtime* analiza ladrar () sin parámetros. ¿Está definida en Yorkie? No, pero sí lo está en la clase padre perro. Entonces, voy a ejecutar el código que corresponde a la operación ladrar(), que se hereda de Perro.

Es importante que este proceso se repite en cada escalón de la jerarquía y se va "escalando" hasta encontrar la operación que **coincida con la firma** del método llamado. Tengamos en

cuenta aquí, para más adelante, la cantidad, el tipo y el orden de los parámetros de la llamada.

En este punto, tenemos la primera consecuencia importante de la herencia. Si vemos la clase Yorkie, **no repetimos nada del código** para hacer que un Yorkie ladre, juegue, etc. La herencia es un mecanismo que apunta fundamentalmente a la **reutilización de código**.

Operaciones. Interfaz pública

Todos sabemos que los Yorkies ladran, también lo hacen los Huskies, los Rottweilers, los Dóberman, etc. Con lo hecho hasta aquí ahorramos código: para cualquiera de estas clases, no deberíamos escribir nuevamente el ladrar() para ladrar, el jugar() para jugar, etc. ¿Conocen a los Labradores? ¡Esa raza tan de color amarillo, tan amorosa, que se lleva tan bien con los niños! Si tuviéramos en nuestra jerarquía a un Labrador, también heredaría de Perro. Porque un Labrador ES-UN perro.

Lo que sí podríamos hacer es **agregar operaciones**. Entonces, como ya heredamos el comportamiento para ladrar y jugar, podemos agregarle al Labrador, por ejemplo, una operación para cuidar a los niños, que denominaríamos **cuidarNinos()**. O al Rottweiler podríamos agregarle métodos para cuidar la casa cuando no estemos, llamado **cuidarCasa()**. Entonces, el Labrador y el Rottweiler son perros, porque hacen al menos todo lo que hacen los perros. Y lo plasmamos en código, usando “extends Perro”. Pero hacen algo más.

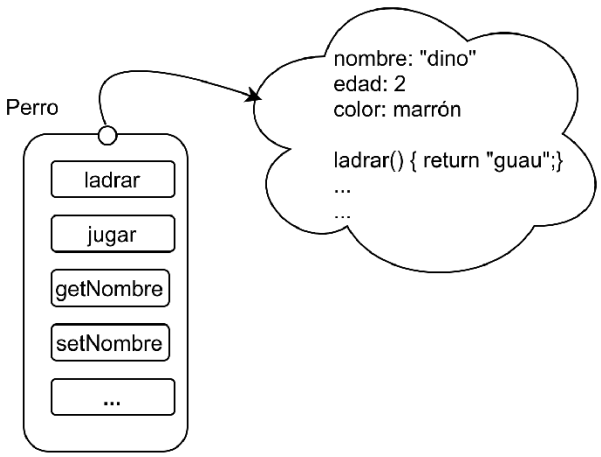
Se dice que Labrador ES-UN perro porque “cumple con la interfaz pública” de perro. Es importante esta distinción, porque la relación ES UN no se da por respetar exactamente lo que hace otra clase, sino que, además, puede agregar comportamiento. Por todo esto es que podemos hacer:

```
Perro p = new Perro();
p.ladrar();

Perro x = new Rottweiler();// Rottweiler ES UN Perro
x.ladrar();
```

¿Cómo es posible? Rottweiler cumple con los que todos los perros hacen, inclusive ladrar(). Lo que ocurre es que x es una variable que almacena un puntero, una referencia a un objeto que es un perro, porque Rottweiler extiende de Perro. ¿Recuerdan la analogía de las referencias y el control remoto?

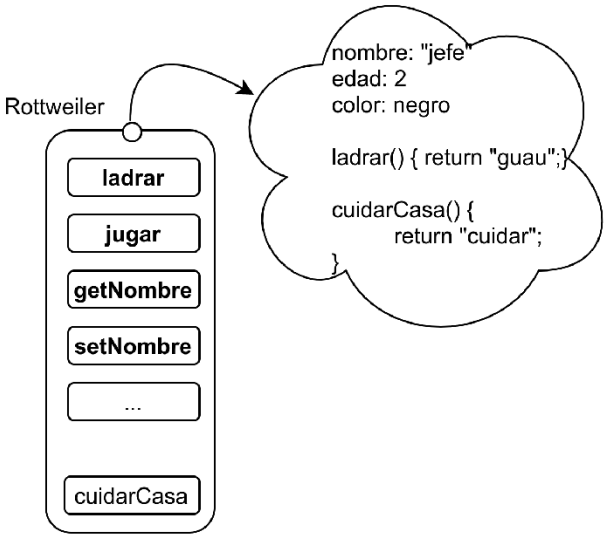
Esto ya lo conocemos:

```
Perro p =new Perro();

System.out.print(p.ladrar());
```



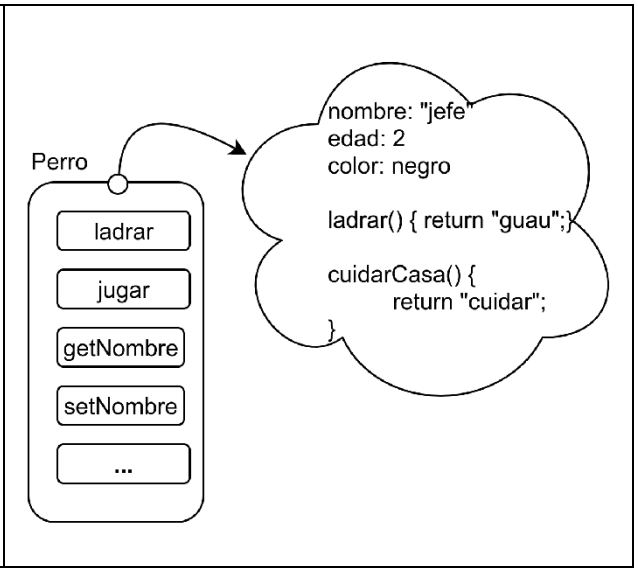
Y con la misma lógica podemos hacer:

```
Rottweiler r = new Rottweiler();

r.ladrar();

r.cuidarCasa();
```



¿Pero observan lo que está en negrita? ¿No corresponden a las mismas operaciones del Perro? Entonces, ¿por qué no podríamos utilizar un control remoto de Perros para controlar un Rottweiler? Es un perro después de todo.

```
Rottweiler r = new Rottweiler();

r.ladrar();

r.cuidarCasa();
```



En el peor de los casos, con nuestro control remoto de Perro "simple", no le podremos decir que cuide la casa. Pero ya trataremos ese tema más adelante.

Uso de la keyword super

Al momento de construir objetos, usamos los constructores. Ya vimos el uso de la palabra this() con paréntesis, con o sin argumentos, para referenciar a un constructor de la propia clase. Cuando estamos en un contexto de herencia y queremos hacer referencia a un constructor de la clase padre, se utiliza super() con paréntesis, con o sin argumentos, tal cual hacíamos con this:

```
public class Perro {
   private Sting nombre;

   public Perro(String nombre) {
      this.nombre = nombre;
   }
   public String ladrar() {
      return super.ladrar();
   }
   //getters & setters
}
```

```
public class Rottweiler extends Perro {
  private int agresividad;

  public Rottweiler(String nombre, int agresividad) {
```

```
    super(nombre);
    this.agresividad = agresividad;
  }
}
```

Aquí es importante notar tres cosas:

- El atributo “nombre” está definido como privado. Entonces, si quisiera usar el atributo en el constructor no lo tendría disponible. Recordemos que se hereda todos los atributos y los métodos “no-privados”. Entonces, esto no sería correcto, no compilaría correctamente:

```
public Rottweiler(String nombre, int agresividad) {
    this.nombre = nombre;//no compila
    this.agresividad = agresividad;
  }
```

- Por más de que se trate de un contexto de herencia, las restricciones de los constructores siguen aplicándose. Al definir en perro un constructor con parámetros, se ha perdido el constructor por *default*. Un Rottweiler, antes de ser Rottweiler, necesita su esencia de perro, porque un Rottweiler es un Perro. Lo que el compilador hace por nosotros en estos casos, es hacer una llamada al constructor por *default* de la clase padre, es decir, hace por nosotros, la llamada a super(). Pero en Perro, lo perdimos. Entonces, tampoco compilaría correctamente. En este caso, tenemos dos alternativas: a) Explicitar el constructor vacío en perro; b) Usar el super(nombre) que sí es un constructor que heredamos de perro.

  Siguiendo con las restricciones de los constructores, al igual que el this() y por las mismas razones, la llamada super() con o sin parámetros debe ir siempre en la primer línea y no podríamos llamarlo más de una vez.

- Aún si tuviéramos acceso al atributo nombre (veremos más adelante cómo) e hiciéramos:

```
  public Rottweiler(String nombre, int agresividad) {
    this.nombre = nombre;
    this.agresividad = agresividad;
}
```

Estaríamos repitiendo código. Por lo tanto, el uso de `super(nombre)` nos permite reutilizar código, reusamos el constructor heredado al cual le pasamos un String con el nombre.

El ejemplo con el código correcto sería:

```
public class Perro {
   private Sting nombre;

   public Perro() {

   }

   public Perro(String nombre) {
      this.nombre = nombre;
   }
}
```

```
public class Rottweiler extends Perro {
   private int agresividad;

   public Rottweiler (int agresividad) {
      //aquí la llamada implícita a super() no tiene problemas
      this.agresividad = agresividad;
   }

   // quiero un constructor que le pase el nombre
   public Rottweiler (String nombre) {
      //this.nombre = nombre;< este código ya lo escribí en Perro!
      super(nombre);//RE-utilizo lo que ya escribí
   }
   //sobrecarga de constructores de Rottweiler
   public Rottweiler (String nombre, int nivelAgresividad) {
      super(nombre);//RE-utilizo lo que ya escribí
      this.agresividad = nivelAgresividad;
   }
}
```

Te compartimos disparadores para que reflexiones sobre lo visto en la lectura:

- ¿Qué piensas de las limitaciones de la herencia en Java?
- ¿Crees que debería aplicar la herencia múltiple como en otros lenguajes o crees que la decisión de mantenerlo simple es acertada?