Clases abstractas

En este punto, estamos familiarizados con la herencia. Gracias a ella podemos agrupar atributos y comportamientos en común a ciertas clases y ubicarlos en una clase “padre”. Ahora, en vez de duplicar el código en cada clase, este estará en un solo lugar. De esta forma, cualquier modificación en el código en la clase padre hará que todas las subclases vean reflejada la modificación.

Además, sabemos que con la herencia podemos hacer que las subclases hereden todo el comportamiento **no privado** de su superclase y también que el comportamiento heredado puede "redefinirse", de ser necesario, con la sobreescritura de métodos.

Static y Dynamic Binding

Las ataduras o “bindings” requieren de un análisis complejo. Por ahora, intentaremos bajarlo a términos sencillos para poder avanzar y lograr entender qué ocurre cuando utilizamos herencia y hacemos código del estilo de:

```
Perro p = new Rottweiler();
```

En Java, como en muchos otros lenguajes, hay dos formas de “atar” un tipo de dato de la variable con el objeto creado que va a almacenar. El Static Binding ocurre en tiempo de compilación, es estático en el sentido que se analiza la atadura sin ejecutar el programa. Recordemos que el compilador dice si la sintaxis de nuestro código es correcta o no, pero nunca ejecuta el código.

Por el contrario, el Dynamic Binding se realiza durante la ejecución del programa, lo que llamamos “tiempo de ejecución”. Esto puede observarse cuando falla el programa al momento de un casteo incorrecto, lo que nos obliga a utilizar el operador instanceof. El instanceof se utiliza para analizar instancias y eso es algo que ocurre necesariamente durante la ejecución.

Entonces, ¿qué implica ejecutar la siguiente instrucción?

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

© Universidad de Palermo. Prohibida la reproducción total o parcial de imágenes y textos.

1	3	2

En *1 se declara una referencia: es una variable. Es importante aclarar que aquí todavía no se ha creado ningún objeto. Solo reservamos un pequeño lugar para **almacenar una referencia** a la que podremos acceder por su nombre: “p”. Esta variable “p” tiene un tipo de dato, es decir, puede almacenar objetos de ese tipo, en este caso, Perro. Luego, eventualmente, “p” apuntará un objeto Perro, es decir, un objeto de tipo Perro.

En *2 Llamamos al constructor y, ahora sí, se crea un objeto: se le indica a la VM que destine el espacio suficiente en la memoria para guardar un objeto Perro, los valores de sus atributos y sus métodos.

En *3 Asignación: se "ata" (*bind*) la referencia al objeto. Aquí hacemos que la referencia "apunte" a un objeto Perro.

En este punto, tanto la referencia como el objeto referenciado son del mismo tipo: Perro. Sin embargo, como vimos, es posible que la referencia y el objeto referenciado sean de distinto tipo. Ahora bien, según lo que sabemos, cualquier objeto que respete la premisa ES UN respecto de la referencia puede ser apuntado por esta: Rottweiler ES UN Perro, por eso Perro p = new Rottweiler() es correcto. En resumen y de forma sencilla, podemos decir que todo objeto, lo que extienda del tipo de la referencia, puede ser apuntado por esta (es decir, todo lo que extienda de lo que está en *1 puede aparecer en *2).

Lo interesante ocurre en tiempo de ejecución. Típicamente, los perros tendrán un método ladrar. Según lo que vimos más arriba, esto es válido en tiempo de compilación.

```
Perro d = new Rottweiler();
d.ladrar();
```

¿Pero, en tiempo de ejecución, qué método ladrar se ejecutará? En tiempo de ejecución, es decir, cuando el programa corre, la JVM llamará al método correspondiente del objeto referenciado. Por tanto:

```
Perro d = new Rottweiler();
d.ladrar();//ladra como un rottweiler

d = new Husky();
d.ladrar();//ladra como un husky

d = new Yorkie();
d.ladrar();//ladra como un yorkie
```

© Universidad de Palermo. Prohibida la reproducción total o parcial de imágenes y textos.

Aquí tenemos un Perro. Rottweiler ES UN Perro. Así que "d" almacena un objeto Perro. Lo mismo ocurre con el Husky y el Yorkie, es decir, el Perro d se puede comportar como un Rottweiler o se puede comportar como un Husky o un Yorkie.

Clases abstractas

Siguiendo con el ejemplo de nuestros peludos amigos, sabemos que los perros tienen diferentes razas. Tenemos Rottweilers, Dóbermans, Huskies, Poodles, y un sinfín de etcéteras. Supongamos que estamos simulando una refinada tienda de mascotas, en la que tenemos que mostrar las características de cada raza de perro, incluido mostrar a los posibles compradores cómo ladra cada uno, entre otras cosas. Hay comportamientos que tienen en común, por ejemplo, "comer". Todos los perros comerán de igual forma: se acerca al plato y degluten los granitos de alimento. Sin embargo, como vimos, cada uno tiene su forma específica de ladrar. Es decir, en cada raza, hay comportamiento que podemos heredar de Perro, pero hay otro que siempre debemos redefinir, como por ejemplo "ladrar". Podemos decir que "ladrar" es algo que **todos los perros hacen, pero cada uno lo hace a su manera.**

¿Cómo podemos representar este escenario de la realidad en código? ¿Cómo podríamos decir que el Perro tiene un comportamiento "ladrar", pero solo las hijas de perro deben decir *cómo ladrar*?

Para ello, introducimos el concepto de **clase abstracta.** Con las clases abstractas podemos definir comportamiento, es decir, qué debe hacer esa clase. Pero no definiremos cómo lo debe hacer. En una clase abstracta Perro, aparecería el comportamiento ladrar, pero sin código que especifique cómo llevar a cabo ese comportamiento. Ese comportamiento se denomina **comportamiento abstracto** y cada método que define este comportamiento se lo llama **método abstracto**.

Definimos a las clases abstractas y el comportamiento en abstracto con la palabra clave "abstract". Como el comportamiento es abstracto (solo decimos qué hacer), los métodos abstractos no tienen código asociado. Se dice que los métodos no tienen "cuerpo". Veamos un ejemplo:

```
public abstract class Perro {

    public abstract String ladrar(); // no hay { }
```

© Universidad de Palermo. Prohibida la reproducción total o parcial de imágenes y textos.

```
}
```

La clase abstracta tiene métodos abstractos, es decir, tiene métodos que dicen qué hacer pero no cómo hacerlo. Sería razonable pensar que una clase abstracta es “incompleta” de alguna forma, que no puede ejecutarse correctamente. Esto es lo que da una de las características de las clases abstractas: **no se puede instanciar**. Por el contrario, las hijas de esa clase serán las que puedan instanciarse.

Llamaremos **clase concreta** (o simplemente “clase”) a toda clase que no es abstracta. **Las clases abstractas dictan “qué hay que hacer”, las clases hijas concretas dirán “cómo hay que hacerlo”**. De esta manera, podemos usar como referencia la clase abstracta, "atándole" una instancia de una clase concreta a esta.

Uso de clases abstractas

Veamos un ejemplo sobre cómo usar una clase abstracta:

```
public Rottweiler extends Perro {


}
```

No parece revestir ninguna complejidad adicional, ¿verdad?

Sin embargo, en este caso, la clase Rottweiler arrojará un **error de compilación**, porque no indica cómo llevar a cabo el comportamiento que establece la clase Perro. Es decir, si Perro dice qué se debe hacer, la subclase Rottweiler debe “explicar” cómo hacerlo. Se denomina a esta operación “implementar” el método. En este caso, debemos implementar el método ladrar().

Si Rottweiler quiere cumplir con la premisa ES UN contra Perro, entonces debe respetar la definición de lo que los Perros hacen: debe implementar un método que se llame ladrar, que devuelva un String y que no reciba parámetros. En pocas palabras, **debe sobrescribir todos los métodos abstractos definidos en Perro.** Entonces:

```
public Rottweiler extends Perro {


    public String ladrar() {
        return “WOOF”;
    }
```

© Universidad de Palermo. Prohibida la reproducción total o parcial de imágenes y textos.

```
}
```

Lo mismo con los demás:

```
public Husky extends Perro {


    public String ladrar() {
        return "GUAU GUAUUOOUU";
    }


}
```

```
public Yorkie extends Perro {


    public String ladrar() {
        return "WIF WIF";
    }


}
```

Cuando implementamos los métodos, estos dejan de ser abstractos; por eso, en Rottweiler y los demás ya no usamos la palabra clave "*abstract*". Las reglas para para la implementación de métodos son las de la sobreescritura: respetar tipo, cantidad y orden de los parámetros. Si no lo hacemos, no estamos cumpliendo con lo que define a un Perro. De ser así, obtendremos un error de compilación.

Al momento de usar una clase abstracta, nuevamente no nos encontraremos con novedades:

```
public Test extends {
  public static void main (String [] args) {
    Perro p = new Yorkie();
    System.out.println(p.ladrar()); // WIF WIF
  }
```

Una clase abstracta es una clase como cualquier otra y, por tanto, puede tener atributos y métodos concretos. Aun así, se debe tener en cuenta que solo los abstractos serán los que definan lo que es un Perro. Antes, un Rottweiler extendía de Perro y heredaba todo el comportamiento no privado de un Perro, lo que lo hacía un Perro: Rottweiler ES UN Perro porque hace todo lo que hacen los Perros. En este caso, es exactamente igual. Un Rottweiler

© Universidad de Palermo. Prohibida la reproducción total o parcial de imágenes y textos.

heredará todo el comportamiento no privado de Perro aunque, adicionalmente, para que Rottweiler sea Perro debe indicar cómo llevar a cabo el comportamiento abstracto definido en Perro.

```
public abstract class Perro {
    private String nombre;


    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getNombre() {
        return this.nombre;
    }


    public String ladrar();


}
```

```
public Rottweiler extends Perro {
    public String ladrar() {
        return "WOOF";
    }
}
```

```
public Test extends {
  public static void main (String [] args) {
    Perro p = new Rottweiler();
    p.setNombre("Drake"); // heredado de Perro
    System.out.println(p.ladrar()); // WOOF implementado en
Rottweiler
  }
```

¿Por qué tendríamos métodos concretos en una clase que no se puede instanciar? Porque estos métodos son susceptibles de ser reutilizados. Asimismo, que una clase abstracta no se pueda instanciar no significa que no pueda tener constructores; el objetivo es el mismo: definir constructores para reutilizar código en las subclases concretas.

```
public abstract class Perro {
    private String nombre;

    public Perro() { // **1 cuidado con las subclases!
    }
```

© Universidad de Palermo. Prohibida la reproducción total o parcial de imágenes y textos.

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


    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getNombre() {
        return this.nombre;
    }


    public String ladrar();


}
```

```
public Rottweiler extends Perro {

    int nivelAgresividad;

    public Rottweiler(int nivelAgresividad) {
     this.nivelAgresividad = nivelAgresividad; // **1
    }

    public Rottweiler(String nombre, int agresividad) {
      super(nombre); // reutilizamos el constructor de Perro
      this.nivelAgresividad = agresividad;
    }


    public String ladrar() {
        return "WOOF";
    }


}
```

```
public Test extends {
  public static void main (String [] args) {
    Perro p = new Rottweiler("Drake", 1);
    System.out.println(p.ladrar());// WOOF -implementado en
Rottweiler
  }
```

© Universidad de Palermo. Prohibida la reproducción total o parcial de imágenes y textos.

Algo más acerca de las clases abstractas y los métodos abstractos

Como vimos, **las clases abstractas definen comportamiento abstracto mediante métodos abstractos**. Es importante recalcar que, si la clase tiene al menos un método abstracto, debe ser declarada como abstracta. De lo contrario, obtendremos un error de compilación.

Sin embargo, si la clase es definida como abstracta, entonces no es necesario definir un método abstracto. En algunas ocasiones, resulta útil tener una clase abstracta con el solo hecho de contener métodos para ser reutilizados, pero no nos interesará tener instancias de esa clase, sino solo de las subclases. Por ejemplo, en una tienda, podemos tener la clase abstracta Electrodoméstico sin ningún método abstracto, pero con atributos y métodos para la marca, el precio, el cálculo de pagos diferidos, descuentos, etc. Sin embargo, en la tienda se venderán Lavarropas, Heladeras, Batidoras, etc.

Vimos que una clase abstracta puede definir algunos métodos abstractos y otros concretos. Que la clase sea abstracta no impide que las subclases sobrescriban los métodos concretos. A diferencia de los métodos abstractos que estamos obligados a implementar, podemos sobrescribir los concretos a voluntad. De esta forma, podemos tomar los métodos concretos de una clase abstracta como un “comportamiento por defecto".

Es importante aclarar que es la primera subclase concreta en la jerarquía deberá implementar todos los métodos abstractos. De lo contrario, si una subclase es también abstracta puede diferir la implementación de algunos abstractos e implementar otros, dejando a la siguiente primera subclase concreta la implementación de todos los abstractos que resten. Por ejemplo:

```
public abstract class Canino {

  public abstract String ladrar():
  public abstract void jugar();

}
```

```
//perro es subclase, pero es abstracta tambien
public abstract class extends Perro {
  public void jugar() { //implemento jugar
    System.out.println(“perro jugando con pelota”);
  }
  //no implemento ladrar pero la clase compila correctamente
  //porque es abstracta, difiero el ladrar()
}
```

© Universidad de Palermo. Prohibida la reproducción total o parcial de imágenes y textos.

```
public Rottweiler extends Perro { //Rottweiler es concreta
  //no puedo tener métodos sin cuerpo, y heredo ladrar. Entonces
  public String ladrar() {
   return "WOOF";
  }
}
```

Te invitamos a utilizar clases abstractas en tu código existente. ¿Crees que puedes convertir una de las clases que tienen hijas en una clase abstracta? Define algo de comportamiento en abstracto y experimenta con el código: ¿qué efecto tiene en las hijas una clase abstracta?

© Universidad de Palermo. Prohibida la reproducción total o parcial de imágenes y textos.