Programación orientada a objetos (parte 2)

Clases abstractas e interfaces

Nuestra última discusión se quedó en una clase Mesa y dos subclases suyas: MesaCircular y MesaRectangular. ¿Se acuerdan del problema que había? En realidad no tenía sentido instanciar un objeto de clase Mesa porque no puede tener significación alguna.

Estas clases son comunes cuando programamos orientado a objetos, por lo que Java nos permite modelarlo con dos maneras diferentes. El primero son las **clases abstractas**, de las cuales *no* podemos instanciar ningún objeto. Una clase se declara abstracta poniend `abstract` en su declaración. Otra cosa que caracteriza a las clases abstractas es que pueden tener **métodos abstractos**, métodos especificados sin implementarse, entendiéndose que cuando una clase extienda a la clase abstracta, debe especificar la implementación.

En el ejemplo, anterior, podríamos convertir la clase Mesa en abastracta como sigue:

```
public abstract class Mesa{

      //Atributos
      private String material;
      private int añosDeUso;

      //Métodos
      public Mesa(){}

      public Mesa(String material, int años){
       material = material;
       añosDeUso = años;
      }

      public void setAñosDeUso(int años){
       añosDeUso = años;
      }

      public String getAñosDeUso(){
       return añosDeUso;
      }

      public abstract double calcularArea();
}
```

¿Ven qué es nuevo en este código? la clase y el método `calcularArea` se especificaron explícitamente como `abstract`. Noten que a diferencia del construc-

tor trivial, un método abstracto *no* tiene llaves. Las llaves defininen la implementación del método, y no es lo mismo poner llaves vacías (la implementación es nada) a no ponerlas (la implemenetación se especificará después.) Podemos completar entonces la clase MesaCircular exctamente como lo hicimos en el documento anterior.

Antes de Java 1.8, las clases sólamente podían extender a *una* otra clase. Por ejemplo, la clase MesaRectangular podía extender a Mesa o a Rectángulo, pero sólamente a una de ellas. Si se quiere usar **herencia múltiple** (es decir, extender a más de una clase), utilizamos **Interfaces**. Pueden pensar en una interface como una clase abstracta que no permitía *ninguna* implementación. Básicamente es un contrato, un caparazón vacío que especifica todo lo que debería poder hacer una clase que **implemente** (como se pueden imaginar, aquí no usamos `extends` sino `implement`) la interfase, pero todo es diferente entre ellos.

No quiero entrar mucho en detalle ahora para no confundirlos, pero desde Java 1.8 las Interfaces ya permiten implementación de los llamados **métodos default**. ¿Cuándo usar interfaces y cuándo clases abastractas? En general, yo prefiero usar interfaces con métodos default, pero las clases abstractas son más flexibles con la visibildad (ya casi llegamos a eso).

Estructura de un documento de Java

Por fin llegamos a entender la letanía de `public static void main(String[] args)`. Primero hablaremos de la **visibilidad** de una clase y sus miembros. Cuando declaramos a una clase como `public`, todas las clases pueden verla; mientras que si la nombramos `private`, sólo pueden hacerlo las clases dentro de su paquete.

Para los métodos y atributos funciona más o menos igual: los declarados `public` le son accesibles a todas las clases y los `private` sólo lo son al objeto mismo. Por ejemplo, todos los atributos los hemos declarados siempre como privados, y es porque sólo un objeto debería ser capaz de cambiarlos y leerlos (recuerden el principio de encapsulamiento). Hay una tercera capa de visibilada, `protected`, que da acceso a todas las clases del paquete.

La otra palabra que no saben interpretar es `static`. Para no entrar mucho en detalle, y comor regla rápida de decisión, los métodos y atributos estáticos son los que *deben asociarse a toda la clase y no a un objeto particular*. Por ejemplo, un método estático debe tener sentido incluso cuando no haya ningún objeto de la clase instanciada, y un atributo estático debe ser compartido entre todos los objetos instanciados de esa clase. Por esto cuando usamos métodos además del `main` en una clase los hemos declarado todos estáticos.

Polimorfismo

El último concepto de programación orientada a objetos que quiero revisar es el de **polimorfismo**. Del griego 'muchas formas', en POO polimorfismo se refiere a la idea de que una clase puede compartir comportmiento con su superclase pero también añadirle algo propio.

Por ejemplo, supongan que queremos trabajar con esta clase

```
public class Bicicleta{
 public static void metodoDeClase(){
  System.out.println("Método estático en la clase Bicicleta");
 }
 public void metodoDeObjeto(){
  System.out.println("Método en la clase Bicicleta");
 }
}
```

Y que la siguiente clase la extiende

```
public class BicicletaDeMontaña extends Bicicleta{
 public static void metodoDeClase(){
  System.out.println("Método estático en la clase BicicletaDeMontaña");
 }
 public void metodoDeObjeto(){
  System.out.println("Método en la clase BicicletaDeMontaña");
 }
}
```

Ahora imaginen que corremos el siguiente código e intenten analizar qué sucede.

```
public class EjecutableBicicleta{
 public static void main(String[] args){
  BicicletaDeMontaña biciMontaña = new BicicletaDeMontaña();
  Bicicleta bici = biciMontaña;

  BicicletaDeMontaña.metodoDeClase();
  bici.metodoDeObjeto();
 }
}
```

La consola imprimiría esto:

```
Método estático en la clase Bicicleta
Método en la clase BicicletaDeMontaña
```

¿Por qué pasa esto? Cuando una clase extiende a otra, hay tres posibles comportamientos con los métodos:

1. **Sobreescribir** el método es reescribir un método con el mismo nombre, tipo y argumentos en la subclase que determine un nuevo comportamiento para los objetos de la subclase. Por ejemplo, cuando reescribimos `metodoDeObjeto()`en la subclase `BicicletaDeMontaña`. Cuando sobreescribimos se acostumbra poner la anotación `@Override` arriba de la declaración del método.

2. Los métodos estáticos los podemos **esconder**. Cuando escondemos un método, el método de la superclase sigue existiendo, y cuál se usa se elige en ejecución según cómo se instanció el objeto. Por ejemplo, en el caso de la bicicleta, eligió el de la superclase porque en la segunda línea lo declaramos de tipo Bicicleta.

3. Finalmente, **sobrecargar** (no sé si es la palabra correcta, en inglés usamos *overload*) un método es extender la lista de argumentos, en cuyo caso la llamada depende sólamente de qué argumentos se le envíen. Usando esto pueden añadir comportamiento en una subclase que quizá no tenía sentido en la superclase. Por ejemplo, `calcularArea()` de una mesa redonda require un sólo parámetro (el radio) mientras que `calcularArea()` de una mesa rectangular requiere dos parámetros (base y altura). En ese caso, el siguiente código es perfectamente válido.

Lo que sucede en las primeras dos opciones se debe al polimorfismo de la programación orientada a objetos. Noten que pueden declarar tanto `BicicletaDeMontaña bici = new BicicletaDeMontaña()` como `Bicicleta bici = new BicicletaDeMontaña()`. El polimorfismo tiene muchas utilidades, pero ahora basta con que sepan que permite esconder y sobreescribir métodos y que además permite tener arreglos polimórficos. Por ejemplo, el siguiente código es perfectamente válido:

```
public static void main(String[] args){
 Bicicleta[] muchasBicletas = new Bicicleta[2];
 muchasBicicletas[0] = new Bicicleta();
 muchasBicicletas[1] = new BicicletaDeMontaña();
}
```

Noten que en el arreglo tenemos bicicletas y bicicletas de montaña, pero esto sí se puede porque *ambas son bicicletas* (a diferencia de, por ejemplo, querer guardar `double`s e `int`ss juntos.)

Tipos genéricos

En este momento no importa mucho que sepan hacerlo (si quieren lo podemos ver después), pero quiero una pregunta natural en este momento es si pueden escribirse métodos que reciban argumentos generales. Por ejemplo, un método que sume debería poder recibir enteros y doubles sin problema, pues tiene sentido sumar ambos. La respuesta es sí, pueden escribirse métodos **genéricos**

parametrizados sobre tipos de variable.