Programación orientada a objetos (parte 1)

Antes de empezar, aprendan que en la jerga de programación orientada a objetos, a las funciones se les llama **métodos**.

La semana pasada vimos cómo guardar variables básicas como números enteros y cadenas de texto, pero usualmente las cosas que queremos programar son bastante más complicadas que eso. Por ejemplo, imaginen que queremos empezar a programar el robot. De entrada, el chassis tiene al menos dos motores y una estructura potencialmente complicada en el tren motriz, un brazo, un intake… ¡y cada una de esas partes tiene otras! Estas cosas complicadas de programar no sólo se dan en el contexto de robótica, y por eso a finales de los sesenta apareció Simula, el primer lenguaje de *programación orientada a objetos (POO)*. (Pueden ver más del desarrollo histórico aquí o por supuesto, en Wikipedia).

Clases y objetos

La idea de la POO es precisamente representar *objetos* del mundo 'real'. Los objetos tienen algunas propiedades que los caracterizan (por ejemplo, los robots tienen número de equipo) y algunas cosas que pueden hacer (por ejemplo, un robot puede subir un brazo o moverse). A las primeras les llamamos los **atributos** del objeto, y a las segundas les llamamos sus **métodos**.

Ahora bien, la programación orientada a objetos no tendría ninguna ventaja real si para cada robot hubiera que iniciar desde cero. Pueden tener diferente equipo, mecanismos y diseño, pero son fundamentalmente una misma *cosa*. A esta *cosa* que ambos robots representan le llamamos su **clase**. (Nota cultural: Platón fue el primero en pensar en programación orientada a objetos, por ejemplo, en Parménides).

Pueden pensar en una clase como la idea abstracta a la que el objeto pertence. Tanto su robot de FRC como los de FTC son robots, por ejemplo, aunque tengan cosas diferentes. Decimos que cada uno de ellos es una **instancia** (yo prefiero el término **realización**, pero nadie lo usa) de la clase.

Tal vez con un ejemplo les sea más claro: tanto Gödel como Cantor son matemáticos (estamos pensando en la clase Matemático) pero ambos tienen diferente nombre, año de nacimiento y especialidad. Si pensamos que sólo esto los distingue, podríamos programar una sola vez lo que hacen, como demostrar teoremas y tomar café y luego hacerle entender a la computadora que tenemos dos objetos de la clase Matemático con diferentes atributos.

¡Ustedes ya han trabajado con objetos! ¿Se acuerdan de la palabra `new` cuando inicializan un arreglo? En Java, `new` es una palabra clave que crea un objeto de cierta clase. Para ver cómo funciona, abajo les doy el código de una clase `Mesa` con algunos métodos y atributos.

```
public class MesaRectangular{

   //Atributos
   private double largo;
   private double ancho;
   private String material;
   private int añosDeUso;

   //Métodos
   public Mesa(){}

   public Mesa(double l, double a, String material, int años){
    largo = l;
    ancho = a;
    material = material;
    añosDeUso = años;
   }

   public calcularArea(){
    return largo*ancho;
   }

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

   public int getAñosDeUso(){
    return añosDeUso;
   }

}
```

Vamos analizando el código poco a poco. Lo primero es la declaración de la clase con nombre "MesaRectangular". Les recuerdo que los nombres de clase *siempre* empiezan con mayúscula (a diferencia de los nombres de variable, que con las convenciones de Java se escriben con `minusculaCamelCase` y los de constantes, que se escriben en puras `MAYUSCULAS`). En una sección posterior les explico qué significa public y private.

La clase tiene tres atributos: `largo`, `anchoy` `material`. Noten que a los atributos no les damos ningún valor, sólo los declaramos.

Los dos primeros métodos se llaman **constructores**. Los constructores son los que le permiten a la computadora instanciar nuevos objetos de la clase. El primer constructor se llama el **constructor vacío** o **trivial**. Por convención, cada clase debe tener un constructor trivial. Noten además que el constructor es el *único* método que empieza con mayúscula, porque debe llamarse igual que la

clase. El segundo, si quieren verlo así, es un constructor "útil" que nos permite instanciar un objeto de la clase MesaRectangular con determinado largo, ancho y material. Ojo, un constructor no tiene que tener todos los atributos. Si les gusta la combinatoria, es fácil (inténtenlo) hacer la cuenta de que una clase con n atributos puede tener hasta 2^n constructores distintos (la prueba está en el artículo deconjunto potencia) en Wikipedia.

La palabra mágica `new` es la que nos permite instanciar un objeto de la clase. Por ejemplo, en el código de abajo **importamos** la clase Mesa para poder usarla y creamos dos objetos de tipo Mesa.

```
import MesaRectangular;

public class EjecutableMesa{

 public static void main(String[] args){
  MesaRectangular mesaCocina = new MesaRectangular(3, 4, "cristal", 1);
  MesaRectangular mesaComedor = new MesaRectangular(5, 8, "madera", 7);
 }

}
```

Los objetos son diferentes completamente. Tienen diferentes tamaños, material y años de uso, pero al final los dos son mesas rectangulares.

Por último, noten que ni `getAñosDeUso` ni `setAñosDeUso` necesitan recibir parámetros. Esto es porque el método está programado *dentro de la misma clase* y por lo tanto tiene acceso a sus atributos. Por analogía, si quisieran calcular su edad en una calculadora tienen que escribir qué año es y en qué año nacieron (la calculadora no tiene acceso a esa información) pero si quieren hacerlo mentalmente, no necesitan pedirle su edad a nadie, nadie se las tiene que "escribir" porque tu año de nacimiento es información tuya, y por lo tanto tienes acceso a ella.

Encapsulación y herencia

Lo que acabamos de explicar es un concepto fundamental en la programación orientada a objetos. El término **encapsulamiento** engloba precisamente la idea de que sólo un objeto sea capaz de cambiar sus propios atributos con métodos del objeto mismo. Por ejemplo, ningún externo debería ser capaz de cambiar el atributo saldo de un objeto que represente tu cuenta de banco. La mesa tiene encapsulados sus atributos y vemos que incluye un método `setAñosDeUso` para cambiar el atributo `añosDeUso`. Análogamente, el atributo `material` tiene un método `getAñosDeUso` para compartirle al mundo exterior la edad de la mesa, que de otra forma sería privada y por lo tanto inaccesible (ya casi llegamos a explicar público y privado).

Otro concepto fundamental en POO es el de **herencia**. Como ejemplo para entender mejor, retomemos el caso de la mesa. Nuestra clase modela una mesa rectangular, pero también hay mesas redondas. Si quisiéramos que cada mesa sea capaz de calcular su área y de cambiar sus años de uso, podemos escribir una clase `Mesa` y pensar que `MesaCircular` y `MesaRectangular` son **subclases** o casos particulares de la clase `Mesa`. Noten que hay dos tipos de herencia:

1. Los métodos `setAñosDeUso` y `getAñosDeUso` son igual en todas las mesas.
2. ¡Pero el método `calcularArea` no puede ser igual en todas las subclases! El área del círculo y del rectángulo se calculan diferente.

Para resolver el primer problema, implementamos la clase Mesa abajo

```
public 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;
   }

}
```

Y ahora las subclases (perdonen la aproximación burda de π)

```
public class MesaCircular extends Mesa{

   //Atributos
   double radio;

   //Métodos
   public Mesa(){
    super();
   }
```

```
    public Mesa(double r, String material, int años){
     super(material, años);
     radio = r;
    }

    public calcularArea(){
     return 3.14*radio*radio;
    }

}
public class MesaRectangular extends Mesa{

    //Atributos
    double base;
    double altura;

    //Métodos
    public Mesa(){
     super();
    }

    public Mesa(double b, double h, String material, int años){
     super(material, años);
     base = b;
     altura = h;
    }

    public calcularArea(){
     return base*altura;
    }

}
```

Analicemos estas subclases. Primero noten que después del nombre de la clase, declaramos `extends Mesa`. El concepto de **extender** es precisamente del que hablábamos antes; estamos diciendo que MesaCircular es *un caso particular de Mesa* y por lo tanto hereda todas las cosas que Mesa tenía. Con más detalle, Cualquier objeto que declaremos de tipo MesaRectangular o de tipo MesaCircular tendrán todos los atributos y los métodos de la clase Mesa. Noten que el constructor vacío ahora no es tan vacío (por eso prefiero constructor trivial) y llama a un método `super`. `super` es el constructor de la **superclase**, o sea, la clase a la que extendemos, y de ahí es que se heredan los métodos y atributos de ella.

Para corroborar que efectivamente solucionamos el primer problema, prueba el

siguiente código:

```
import MesaCircular;
import MesaRectangular;

public EjecutablePrueba{
 public status void main(String[] args){
   MesaRectangular mesaComedor = new MesaRectangular(5, 8, "madera", 7);
   MesaCircular mesaSala = new MesaCircular(1, "marmol", 3);

   System.out.println(mesaComedor.getAñosDeUso());
   mesaSala.setAñosDeUso(5);
   System.out.println(mesaSala.getAñosDeUso());
 }
}
```

Sin embargo, hay algo raro con nuestra solución: ¡es posible instanciar un objeto de clase Mesa sin que sea circular ni rectangular! ¿Ven cómo no tiene sentido? La idea de una "Mesa sin forma" no existe, necesitamos saber de qué forma es la mesa para que tenga sentido su existencia. Para estas situaciones, Java introduce la noción de **clase abstracta** e **interfaz**, temas del siguiente documento.