El mecanismo de sobrecarga

Sabemos que durante el proceso de abstracción observamos la realidad y obtenemos un modelo en código.

Al momento de modelar es importante tener en cuenta el contexto, ya que estamos haciendo un modelo particular de la realidad para resolver un problema específico. Al momento de modelar, pensamos en las clases, los objetos, sus atributos y su comportamiento. Si partimos de la realidad, es importante que mantengamos los vínculos semánticos con ella. Es decir, si estoy modelando un auto, sé que un auto tendrá una marca, un modelo y tendrá cierto comportamiento. Seguramente tendrá comportamientos como acelerar() y desacelerar() por ejemplo. En ese sentido, nunca tendrá un método "reducirValorDeAtributoVelocidad()". Efectivamente, en el fondo, eso es lo que ocurre, al frenar se reducirá el valor del atributo velocidad actual, pero cuando nos subimos a un auto, hay un pedal para acelerar y otro para frenar, el auto acelera y desacelera.

Firma de métodos

Para entender el concepto de sobrecarga es importante definir a qué llamamos la "firma" de un método. Para ello, debemos analizar la anatomía completa de un método. Veamos:

```
public  static  int  multiply   (int a, int b)   {   }
```

1	2	3	4	5	6

1. **El modificador de visibilidad**: indica si el método es accesible a otros objetos. En este caso, el método es público, así que cualquier otro objeto de cualquier otra clase podrá ver y usarlo.
2. **Indicador de clase:** este es un modificador opcional y lo usamos para indicar cuando el método corresponde a una operación de la clase o de los objetos. Analizaremos esto en detalle más adelante.
3. **Tipo de dato de retorno:** indica de qué tipo de dato es el valor que retorna este método. Si el método no retorna ningún valor, se utiliza la palabra clave "void".

4. **Nombre del método**: es el nombre de la operación.
5. **Parámetros:** los parámetros van entre paréntesis y separados por coma. Cada uno tendrá un tipo de dato y un nombre.
6. **Cuerpo del método:** entre las llaves se pone el código que debe ejecutarse cuando se llama a este método.

Analizada la anatomía, definiremos lo que se conoce como la firma de un método. **La firma de un método es la forma de identificarlo, y corresponde a su nombre y sus parámetros únicamente**. Haremos hincapié en los parámetros, ya que es de suma importancia para la firma la cantidad de parámetros, el tipo de cada uno y el orden en que aparecen; pero no el nombre de cada uno. Por ejemplo estas firmas son distintas:

```
void someOperation(int someInt, int otherInt, boolean b)
void someOperation(int someInt, boolean b, int otherInt)
```

Sin embargo estas firmas se las considera iguales:

```
void someOperation(int someInt, int otherInt, boolean b)
void someOperation(int otherInt, int someInt, boolean b)
```

Las variantes en el comportamiento

Supongamos un programa sencillo de dibujo de figuras. El programa tendrá operaciones para dibujar un círculo, un cuadrado, una flecha, etc. Podemos agregarle algún color de relleno y un texto. Analicemos la operación de crear un círculo:

```
public class DrawingEngine {
  public void drawCircle() {
  //draw on screen
  }
}
```

Hasta acá todo bien, pero no parece ser muy útil. El método nos dibuja un círculo chiquito en la esquina superior izquierda de la pantalla y nada más. ¿Qué pasaría si le quisiéramos dibujar un círculo en el centro de la pantalla? Podríamos definir un método tipo:

```
public class DrawingEngine {
  public void drawCircleCentered() {
    //draw on center of screen
  }
}
```

Parece una alternativa viable. Ahora bien, ¿qué pasa si quisiéramos dibujar un círculo en una posición arbitraria de la pantalla? Entonces podríamos hacer:

```
public class DrawingEngine {
    public void drawCirclePosition(int x, int y) {
      //draw on point x,y
    }
}
```

Más aún, si queremos color:

```
public class DrawingEngine {

 public void drawCirclePositionWithColor(int x, int y, String color) {
    //draw on point x,y with supplied color
 }
}
```

La pregunta que corresponde hacerse ahora sería: ¿podríamos hacer esto para cada opción posible que quiere el usuario? Probablemente no sea práctico, además, la función del

programa es “dibujar círculos”. Si vamos al famoso editor de imágenes de Windows, el botón que apretamos es “dibujar circulo”, ¿verdad? Semánticamente, no hacemos otra cosa que dibujar círculos, solo que con diferentes características y la pantalla, el lienzo donde dibujamos, reacciona de acuerdo a nuestras diferentes entradas. Entonces, ¿por qué no definir directamente un método para dibujar círculos únicamente? Veamos:

```
public class DrawingEngine {
  public void drawCircle (int x, int y) {
    //
  }
}
```

Si observamos bien, se cambió el nombre del método para indicar que se dibuja un círculo. Esa es la acción del usuario. A veces, el usuario dibuja un círculo, pero lo quiere de color. Nuevamente, y desde el punto de vista semántico, el usuario “dibuja un círculo”. Entonces deberíamos tener:

```
public class DrawingEngine {
  public void drawCircle (int x, int y) {
    //
  }
  public void drawCircle (int x, int y, String color) {
    //
  }
}
```

Faltaría agregar un tamaño. Como dijimos, el usuario realiza la acción de dibujar círculos, así que vamos a representar en nuestro modelado esa acción, haremos una operación adicional a la que tenemos para incluir el tamaño:

```
public class DrawingEngine {
  public void drawCircle (int x, int y) {
    //
  }
  public void drawCircle (int x, int y, String color) {
    //
  }
  public void drawCircle (int x, int y, String color, int radius) {
    //
  }
```

Entonces, ¿qué tenemos hasta aquí? Tenemos que nuestro editor permite al usuario hacer una cosa: dibujar círculos. La operación dibujar círculo se comportará de forma diferente de acuerdo a los parámetros que suministremos. Nuevamente, diferenciamos cada variante de la operación de dibujar círculo por su tipo, cantidad y orden de los parámetros, y no su nombre. Es decir, obtuvimos variantes respetando el nombre pero alterando los parámetros de la firma del método.

Adicionalmente, ¿han notado como algunos editores gráficos, al momento de seleccionar la forma, directamente la dibujan con algunos valores preestablecidos? Nuestro drawCircle seguiría siendo algo aburrido, pero sería válido:

```
public drawCircle() {
  this.drawCircle(5,5, “white”,5); //x=5, y=5, radius=5
}
```

Y más aún, **reutilizamos el código de otro método** que ya habíamos definido. Para hacer nuestro “circulo por defecto“ no tuvimos que crear código adicional. Es así como en el “main” de nuestro editor de texto tendríamos:

```
public class DrawingProgramUI {
  public static void main(String [] args) {
    DrawingEngine de = new DrawingEngine();
    de.drawCircle();
    de.drawCircle(3,5);
    de.drawCircle(3,5, “red”);
    de.drawCircle(3,5, “red”, 10);
  }
}
```

Funcionamiento

Si analizamos el *main* anterior, podríamos preguntarnos cómo es que Java sabe a qué método llamar en cada caso si los nombres de los métodos son iguales. Aquí, al igual que con los constructores anteriores, por el tipo, orden y cantidad de parámetros.

A este mecanismo se lo conoce como **sobrecarga u *overloading***. Puede ser de métodos o, como ya vimos, de constructores. Esto se debe a que los constructores son métodos, solo que están destinados a crear objetos. Los principales objetivos de la sobrecarga son tener variantes y responder de diferentes maneras a los estímulos y, adicionalmente, reusar código. Por ejemplo, veamos:

```
public class DrawingEngine {
  public void drawCircle (int x, int y) {
    //code to draw pixels on screen for a circle
  }
  public void drawCircle (int x, int y, String color) {
    // reuse drawing
    this.drawCircle(x, y);
```

```
  //then code to add color
 }
 public void drawCircle (int x, int y, String color, int radius) {
  //reuse colored drawing
  this.drawCircle(x, y, color)
  //then code to resize
 }

 //default circle:
 public void drawCircle() {
  //reuse
  this.drawCircle(5,5,"White",5);
 }
```

Si pensamos que el hecho de dibujar los píxeles, darles color y demás es una operación compleja, resulta de gran beneficio poder reutilizarla, no solo porque no tenemos que escribir una y otra vez el mismo código, sino que una vez que lo escribimos y lo probamos, lo podemos volver a utilizar una y otra vez de forma segura.

Conclusiones

Recordemos que los objetos se comunican mandándose mensajes y que cada mensaje es un estímulo, que lleva una carga paga para que el destino los interprete y trabaje con ella. Esa carga son los parámetros o argumentos. Hemos logrado que un objeto reaccione de manera diferente según los estímulos suministrado.

En la práctica, hemos logrado que un objeto tenga igual comportamiento (dibujar un círculo), pero que se lleve a cabo de forma diferente de acuerdo con los parámetros que le pasamos. En otros lenguajes a esto se lo llama **"polimorfismo de funciones"**; en el caso de Java, simplemente se lo conoce como **sobrecarga de métodos**. El polimorfismo es un

concepto más avanzado que veremos luego. Y solo para aquellos que se estén preguntando, en Java no tenemos sobrecarga de operadores, como hay en otros lenguajes.

La POO intenta modelar la realidad lo más fielmente posible.

- ¿Creés que la sobrecarga tiene un correlato con lo que vemos en la realidad?
- ¿Creés que este concepto es aplicable a las cosas nos rodean o te resulta algo que en teoría resulta útil solo en el código?

Te invito a reflexionar en base a los disparadores presentados.