Referencias. Pase de argumentos en Java

Referencias

Sabemos que en Java tenemos tipos de datos para valores escalares o “tipos de datos primitivos” y, por otro lado, tenemos los “tipos de dato referencia”. Es por eso que tenemos tanto al tipo de dato “int” como al “Integer”, al “float” como al “Float”.

Los tipos de datos de referencia son tipos de datos que constituyen objetos. En Java, como en otros lenguajes, se utilizan “punteros” o “referencias” a los objetos. Sin embargo, a diferencia de otros lenguajes, como C o C++, los punteros en Java son implícitos, simplemente no tenemos que encargarnos de ellos. De todas formas, están ahí y tienen sus consecuencias al momento de crear nuestros programas.

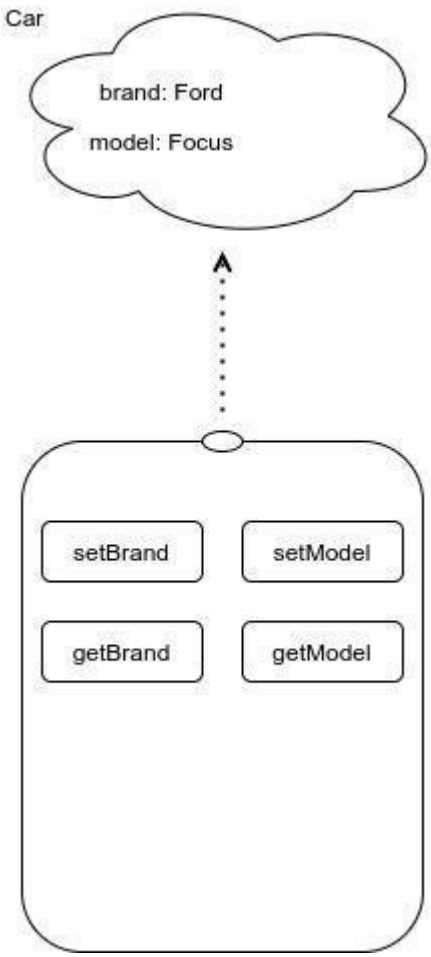
Por ejemplo, tomemos el caso de una clase Car, que produce objetos que son autos. Entonces, cuando definimos:

```
public class Car {

   private String brand;

   private String model;

  // getters & setters

}
```

Y hacemos:

```
public void static main(String[] args) {

  Car c = new Car();

  c.setBrand(“Ford”);

  c.setModel(“Focus”);

}
```

En la variable “c” no se está almacenando un objeto auto, sino una referencia al objeto auto. De hecho, cuando llamamos a setBrand estamos diciendo: “ejecutar el método setBrand al objeto apuntado por la referencia guardada en c”. Es como cuando tomamos el control remoto y apuntamos al televisor y apretamos un botón para cambiar de canal. Es decir, en la variable “c” guardamos un control remoto con el cual controlar al objeto auto referenciado.



Ahora bien, qué pasa cuando complejizamos un poco las cosas y pensamos en cómo se trasladan los objetos al momento de ejecutar los métodos. Algo ya tenemos aquí, el setBrand y el setModel reciben un objeto de tipo String, pero para hacerlo más claro agreguemos lo siguiente:

```
public class Car {

   private String brand;
```

```
   private String model;

   private Engine engine;

  public void setEngine(Engine e) {

    this.engine = e;

  }

  public Engine getEngine() {

    return this.engine;

  }

  // other getters & setters

}

public class Engine {

  private started;

  public void start() {

    //brooom

    this.started = true;

  }

  public boolean isRunning() {

    return this.started;

  }

}
```

En ese caso, ¿qué creen que ocurriría aquí?

```
public static void main(String[] args) {

  Car c = new Car();

  c.setBrand("Ford");

  c.setModel("Focus");

  Engine x = new Engine();

  c.setEngine(x);

  x.start();

  System.out.println("CAR engine is running:"

                  + c.getEngine().isRunning()); // …is running: true

  System.out.println("engine is running:"

                  + c.getEngine().isRunning());// …is running: true

}
```

Si x apuntaba al objeto motor y le pasamos x a setEngine, entonces le estamos pasando la referencia al objeto motor. En este caso externo, entonces, un cambio al auto (le damos marcha al motor "por fuera" del auto) tiene una consecuencia en el auto, por más que no sea este quien haya dado marcha al motor.

Parámetros

En los lenguajes, las palabras "parámetro" y "argumentos" suelen usarse como sinónimos y, aunque no está mal, es importante saber distinguir unos de otros

Los parámetros, llamados "parámetros formales", son aquellos definidos al momento de escribir el código. Es decir, aquí:

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

 return a*b;
```

```
}
```

Los parámetros formales serían “a” de tipo entero y “b” de tipo entero.

Por otro lado, los argumentos, llamados “parámetros reales”, son aquellos valores o expresiones que se pasan a la función en el momento de la ejecución. Es decir, si en un momento hiciéramos:

```
int result = multiply(3, 5);
```

Los argumentos serían el entero 3 y el entero 5.

A veces, algunos lenguajes hacen una diferenciación entre lo que son los “parámetros reales” y los “argumentos”. Esto se da en el contexto del uso de operadores como, por ejemplo, en el uso del operador if ternario:

```
retry ? callServer() : showError()
```

Aquí, si el valor de retry es true, se llama a callServer(); de lo contrario, se llama a showError(). En este caso, retry se lo denomina “argumento”.

Formalmente, al resultado de la ejecución de una operación suele llamarse “parámetro de resultado” o “parámetro de retorno”.

Sabiendo, entonces, la distinción y los pormenores de cada término, a partir de aquí usaremos argumento y parámetro como sinónimos.

Pasaje de parámetros

El tema de pasaje de parámetros es central en todos los lenguajes de programación. Cada lenguaje implementa su técnica particular y cada una tiene sus pros y sus contras. Hay muchas técnicas, entre otras:

- Pasaje por valor
- Pasaje por referencia
- Pasaje por resultado
- Pasaje por nombre

Las más comúnmente usadas son las dos primeras.

Pasaje por valor

En el pasaje por valor, al método o función se le suministran copias exactas de los valores. De este modo, cualquier modificación que realicemos sobre los objetos suministrados no tendrán ningún efecto por sobre los objetos fuera de esa función. En Java, ocurre esto con los valores primitivos. Por ejemplo, si hiciéramos:

```
public quirkyMultiply(int a, int b) {

  a = a+1;

  b = b+1;

  return a*b;

}
```

Y en el main:

```
int a = 3;

int b = 2;

int result = quirkyMultiply(a,b);

System.out.println(result); // << 12

System.out.println(a);//3 aunque hayamos sumado 1 dentro de
quirkyMultiply

System.out.println(b);//2 aunque hayamos sumado 1 dentro de
quirkyMultiply
```

Pasaje por referencia

En el método de pasaje por referencia, lo que se pasa al método es una referencia al valor. En este sentido, cualquier cosa que hagamos con ese valor se reflejará fuera de la función.

Tal cual vimos en el ejemplo del auto más arriba. Dicho esto, **Java no pasa por referencia. Java pasa estrictamente por valor, siempre**. ¿Cómo se explica, entonces, el comportamiento que vimos antes? Java, como algunos otros lenguajes, pasa el valor de la referencia. Esto implica que no todo lo que hagamos a los valores suministrados afectará al exterior. Si modificamos un poco el código anterior del auto podremos verificarlo:

```
public class Car {

   private String brand;

   private String model;

   private Engine engine;

   public void setEngine(Engine e) {

   // estoy modificando al "e" suministrado, el atributo started será
false

    e = new Engine();

    this.engine = e;

  }

  public Engine getEngine() {

    return this.engine;

  }

  // other getters & setters

}
```

```
public class Engine {

   boolean started = false  ;
```

```
   public void start() {
     //brooom
     this.started = true;
   }
  public boolean isRunning() {
    return this.started;
  }
}
```

Entonces si hiciéramos un main:

```
public class Test {
  public static void main(String[] args) {
    Car c = new Car();
    Engine x = new Engine();
    c.setEngine(x);
    x.start();
    System.out.println("CAR engine is running:"
              + c.getEngine().isRunning()); // …is running: false

    System.out.println("engine is running:"
              + x.isRunning());// …is running: true
   }
```

```
}
```

En conclusión, aquí no todo lo que hagamos dentro del método con el parámetro suministrado afectará el exterior. Dijimos que **en Java todo es pasaje por valor**. Ahora bien, teniendo claro esto, podemos decir que, a los efectos prácticos, en el 99,9% de los casos, cuando lidiamos como objetos, podemos trabajar como si Java pasara por referencia.

¿Crees que Java debería cambiar su manera de pasar argumentos? ¿Consideras que Java podría beneficiarse de dar a elegir al desarrollador la forma de pasar argumentos como en otros lenguajes?