Igualdad e identidad de objetos

En este punto sabemos que las clases tienen atributos y operaciones. Hablando de objetos, cada atributo puede tomar valores y los métodos pueden operar sobre esos valores. Ya determinamos que mientras que puedo tener múltiples objetos iguales, las instancias son únicas. Entonces, si tenemos dos objetos que son iguales, eso no quiere decir que sean idénticos. Dos objetos son iguales cuando los valores de sus atributos tienen el mismo valor, pero no necesariamente cuando su identidad es la misma. Si un objeto tiene la misma identidad que otro, entonces es el mismo objeto.

Ya estamos al tanto de los conceptos de igualdad e identidad de los objetos. Se dice que dos objetos son idénticos cuando son exactamente el mismo objeto, es decir, se trata de la misma instancia. Yendo a la realidad, podemos pensar que, si bien yo puedo tener dos objetos exactamente idénticos, entonces deberían ser el mismo objeto, deberían ocupar el mismo espacio físico, y eso no es posible.

Veamos un ejemplo: supongamos la clase auto, llamada Car, con atributos marca, modelo y color; y algunas operaciones, acelerar y frenar. Como ya sabemos, marca y color son privados y tendremos operaciones públicas para acceder a sus valores. Supongamos que podemos hacer lo siguiente:

```
Car c = new Car("Ford", "Focus", "Red");

Car d = new Car("Ford", "Focus", "Red");
```

A simple vista podemos deducir:

- Existen dos instancias de Auto.
- Existen dos objetos, dos autos.
- Ambos objetos tienen iguales valores en sus atributos. Son dos Ford Focus rojos.

Cabe entonces preguntarnos: ¿son a y b dos objetos iguales? Sí, pero ¿son a y b dos objetos idénticos? Definitivamente, no. ¿Qué hace que los objetos apuntados por c y por d sean iguales? Que tienen iguales valores en sus atributos. ¿Qué hace que NO sean dos idénticos? Que son dos instancias únicas porque usamos "new" para crear una nueva instancia de la clase Auto en los dos casos.

Comparar identidad

Para comparar valores de variables de tipo escalar o “primitivas” usamos el operador de igualdad “==”. Si bien eso es correcto, cuando se utiliza el operador “==” entre variables de tipo referencia, el comportamiento es similar, aunque no exactamente igual. Dado que los valores escalares no son referencias (es decir, no son punteros a objetos), lo que se compara con el “==” es su valor absoluto, por ejemplo:

```
int x = 3;

int y = 5;

if(x == y) {

System.out.println(“ambos valores son iguales”)

}
```

En el caso de los autos, c y d, no almacenan un valor absoluto, sino que almacenan un objeto; más estrictamente, una referencia a un objeto, un puntero a una dirección de memoria donde se almacena ese objeto.

Entonces, si hiciéramos:

```
if(c == d) {

System.out.println(“ambos autos son iguales”)

}
```

Así como está, esta condición nunca se podría dar. Porque el auto apuntado por c es una instancia diferente del auto apuntado por d. Sin embargo, si hiciéramos:

```
Car x = c;
```

Básicamente, lo que estamos haciendo es decir que la variable “x” almacena lo mismo que la variable “c”, es decir que si “c” apuntaba a un objeto auto, entonces “x” apuntará exactamente al mismo auto que el apuntado por “c”.

Entonces, si hiciéramos la comparación…

```
if(a == x) {

System.out.println(“ambos autos son iguales”)

}
```

… esta sería verdadera.

Puede sonar bastante obvio, pero es un muy común intentar compara objetos con el “==”. Lamentablemente, si no comprendemos la diferencia entre comparar valores escalares y comparar valores de referencia ni los conceptos de igualdad e identidad obtendremos resultados inesperados.

Cuando se usa el operador de identidad, en realidad se están comparando las posiciones de memoria donde se alojan los objetos. Más arriba mencionamos que “==” era similar para objetos, aunque no era lo mismo, y que “==” comparaba valores absolutos. En el caso de las referencias, lo que estamos comparando son los punteros. Decir “if(a == x) {…}” es preguntar “¿Apunta x al mismo objeto que x?”. Es decir, “El objeto apuntado por a, ¿es el mismo objeto apuntado que está siendo apuntado por x?”. A nivel de hardware, lo que estamos comparando es, básicamente, posiciones de memoria, porque no puede haber dos objetos diferentes en el mismo lugar físico. Cada vez que hacemos “new” y el objeto resultante lo guardo en una variable, estoy reservando un lugar en la memoria para guardar los valores de ese objeto y correr sus operaciones.

Comparar igualdad

Ya sabemos cómo comparar si dos objetos son idénticos. Entonces, ¿cómo podemos saber si dos objetos son iguales? En Java, para comparar si dos objetos son iguales, se utiliza el método *public boolean equals(Object anotherObject)*.

El **método equals()** se hereda (como muchos otros métodos) desde la clase Object, la clase padre (ancestro en común) de todas las clases en Java. Es por eso que recibe como

parámetro otro objeto. Aunque es posible, rara vez se utiliza comparar un objeto de un tipo con otro de otro tipo. Podemos, pero es muy difícil comparar peras con manzanas.

El método equals() es un método que, por defecto compara identidad. Por lo tanto, deberemos reemplazar ese comportamiento por el que nosotros necesitemos. En este sentido, y como ya mencionamos, nuestras clases y sus operaciones dependen del problema a resolver. Es decir, adecuaremos el comportamiento del equals() para que se ajuste a nuestro negocio. ¿Qué significa que un auto sea igual a otro? Según el negocio, puede ser que un auto es igual a otro si su precio es igual; o según su chapa de licencia, es igual; o qué tal su antigüedad; o podemos decir que un auto es igual a otro si todos los valores de sus atributos son iguales. Veamos un ejemplo:

```
public class Box { /

private int x;

private int y;

private int z;



//getters & setters



public boolean equals(Object another) {

   Box anotherBox = (Box)another;

   return (this.x == anotherBox.getX() // x,y,z are primitive!

            && this.y == anotherBox.getY()

            && this.z == anotherBox.getZ() )

}
```

Entonces, si tenemos:

```
Box b = new Box(12, 30, 45);

Box c = new Box(12, 30, 45);

b == c; // false ¡

b.equals(c); // true! x,y,z measurements ARE the SAME
```

Por suerte existen clases con equals con criterios razonables para nosotros. Es el caso del equals de String, que compara carácter a carácter uno con otro y retorna si son iguales o no.

Las clases que no tengan el equals() implementado requieren de un trabajo adicional manual para implementarlo. Por suerte, los IDEs siempre traen una herramienta para generación del método equals() y permiten, mediante un asistente, modificar el criterio (qué atributos queremos comparar, qué condiciones adicionales, etc., que necesitamos para resolver el problema) mediante el cual el equals() dará verdadero o falso al momento de comparar dos objetos entre sí.

Equals y Hashcode

Lamentablemente, no todo es tan fácil. Así como tenemos el método equals(), también se hereda desde Object *public int hashCode()*.

Este método tiene como objetivo proveer un número unívoco que identifique un objeto, de forma tal que nos dé una doble comprobación junto con el equals para verificar que dos objetos son iguales o distintos. **Según la documentación de Java, si implementamos el método equals, debemos necesariamente implementar el método hashcode**, especialmente cuando usemos nuestros objetos en alguna estructura de datos (colección, mapa, etc.). El método hashcode se utiliza como "complemento obligatorio" del equals().

Según la documentación, el método equals define las siguientes reglas:

1. **Equals debe ser reflexivo**: un objeto debe ser igual asimismo: a.equals(a) debe dar true.
2. **Debe ser simétrico**: a.equals(b) debe dar el mismo resultado que b.equals(a).
3. **Debe ser transitivo**: si a.equals(b) da true y b.equals(c) da true, entonces a.equals(c) debe dar true.

4. **Debe ser consistente**: el resultado de equals() debe cambiar solo si cambia algo dentro de la lógica de la implementación. Es decir, para un objeto a y uno b, cada uno con ciertos valores en sus atributos, el resultado de correr a.equals(b) debe dar siempre el mismo resultado.

El método hashCode() debe emitir un número que identifique unívocamente un objeto. Por definición, el método hashcode, según la documentación, define las siguientes reglas:

1. **Debe ser consistente con sí mismo**: cada vez que llame al método hashCode() sobre un mismo objeto, con iguales valores en sus atributos, el valor retornado por el método hashCode() debe ser el mismo.
2. **Debe ser consistente con equals()**: si dos objetos son iguales según el método equals(), entonces ambos objetos deben producir el mismo número al ejecutar el método hashcode().
3. **Dos objetos distintos según equals() no están obligados a producir números distintos de hashcode**: sin embargo, no asegurarse de esto puede tener un impacto negativo en la performance de algunas estructuras de datos.

Veamos un ejemplo: a la clase Caja anterior, haremos un equals y un hashcode:

```
public class Box { /

    private int x;

    private int y;

    private int z;



    //getters & setters



    public boolean equals(Object another) {

        //típicamente, se compara identidad primero: si son

        //idénticos, entonces serán iguales.

        if(this == another) return true;
```

```
        Box anotherBox = (Box)another;

        return (this.x == anotherBox.getX()

            && this.y == anotherBox.getY()

            && this.z == x.getZ())

    }


    public boolean hashCode() {

      int result = x;

      result = 31 * result + y;

      result = 31 * result + z;

      return result;

    }

}
```

Normalmente, se utilizan implementaciones de hashCode utilizando números primos y haciendo algunos cálculos de forma tal que se minimicen las posibilidades de obtener un número repetido para diferentes objetos.

Afortunadamente, los IDEs, así como tienen herramientas para generar los equals() de cada clase, también las tienen para generar los métodos hashCode eficientemente. Implementaciones eficientes de equals y hashCode para nuestra clase Box serían (generadas por el IDE):

```
public boolean equals(Object o) {

    //si es idéntico, seguro es igual

    if (this == o) return true;
```

```
    //si no son de la misma clase, no puedo comparar peras con manzanas
    //el método getClass da la clase de la cual este objeto proviene
    if (o == null || this.getClass() != o.getClass()) return false;

    //comparar valores de los atributos:
    Box anotherBox = (Caja) o;

    if (x != anotherBox.getX()) return false;
    if (y != anotherBox.getY()) return false;
    return z == anotherBox.getZ();
}

@Override
public int hashCode() {
    int result = x;
    result = 31 * result + y;
    result = 31 * result + z;
    return result;
}
```

Afortunadamente, hay varias clases que ya tienen implementado, además del equals(), el hashCode(). String no es la excepción.

Los problemas de no implementar ambos métodos

Mencionamos que implementar el método hashCode es obligatorio para lograr resultados de comparación totalmente confiables. Podemos demostrar por qué. Supongamos que tenemos la siguiente clase:

```
class Team {



    String city;

    String sport;



    //getters & setters



    @Override

    public final boolean equals(Object o) {

        // implementación simple

        Team t = (Team)o;

        // String ya tiene implementado equals

        return this.city.equals(t.getCity() &&

                this.sport.equals(t.getSport())

    }

}
```

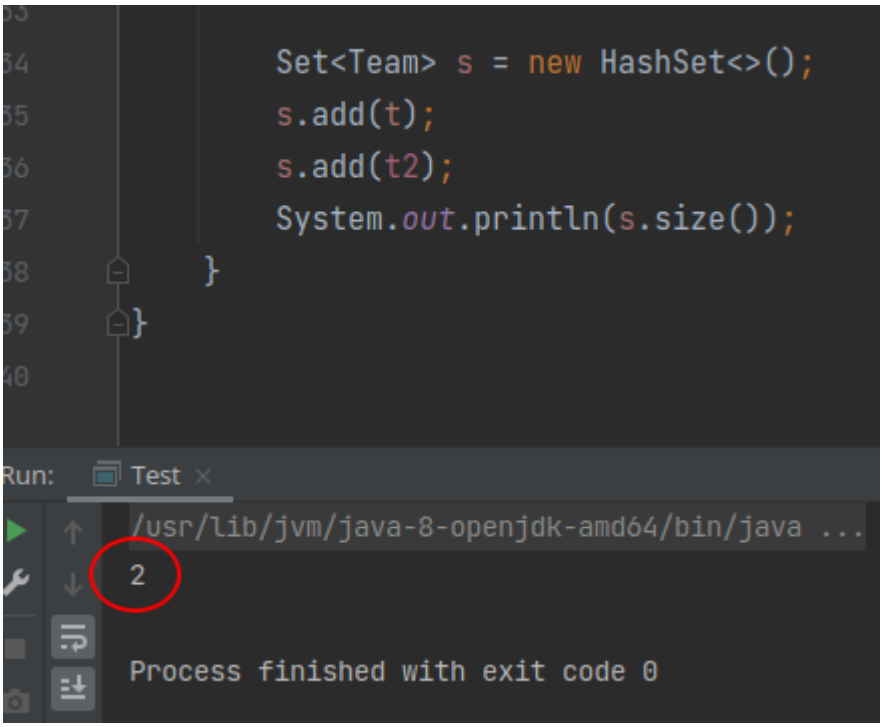
Si no sobrescribimos hashCode aquí, se usa la implementación por defecto, que retorna un número que depende del identificador interno de instancia en la JVM. Entonces, tener dos equipos que jueguen al fútbol en la ciudad de París violaría la segunda regla del hashCode.

Esto se debe a que el equals() daría verdadero, pero el hashCode() daría dos números distintos.

Así, al poner esto en una estructura de datos que depende de la igualdad de objetos, como el Set, tendríamos:

```
Team t = new Team();

t.setCity("Paris");

t.setSport("Football/soccer");



Team t2 = new Team();

t2.setCity("Paris");

t2.setSport("Football/soccer");



Set<Team> s = new HashSet<>();

s.add(t);

s.add(t2);

System.out.println(s.size());//”2”
```

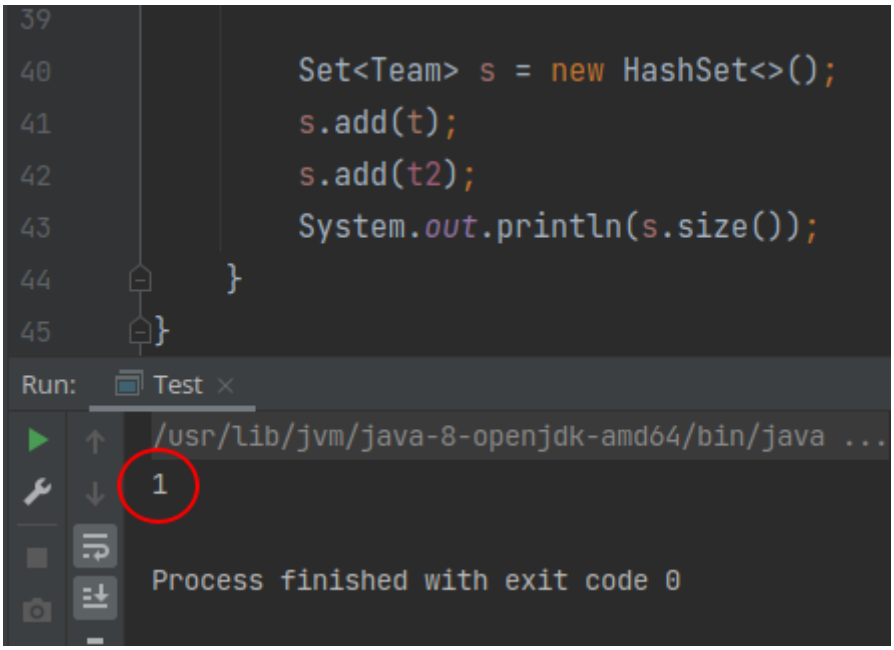
Al correr el código obtenemos:



Sabemos que el HahSet no admite dos objetos iguales repetidos. Aquí, el size() da “2” y eso no está bien. Sin embargo, luego de agregar el hashCode():

```
@Override

public int hashCode() {

    int result = city != null ? city.hashCode() : 0;

    result = 31 * result + (sport != null ? sport.hashCode() : 0);

    return result;

}
```

Y correr el programa de nuevo obtenemos:



Que es lo que esperamos de una estructura de datos como el Set.

¿Qué piensas del método que tiene Java de comparar objetos? ¿Crees que es correcto? ¿Crees que es muy complejo?