By default, the Java super class **java.lang.Object**provides two important methods for comparing objects: **equals()** and **hashcode()**.

**Method Definition and Default Implementation**

* **equals(Object obj):** a method provided by ***java.lang.Object*** that indicates whether some other object passed as an argument is ***"equal to"*** the current instance. The default implementation provided by the JDK is based on memory location — two objects are equal if and only if they are stored in the same memory address.
* ***hashcode():***a method provided by **java.lang.Object**that returns an integer representation of the object memory address. By default, this method returns a random integer that is unique for each instance. This integer might change between several executions of the application and won't stay the same.

## The Contract Between equals() and hashcode()

The default implementation is not enough to satisfy business needs, especially if we're talking about a huge application that considers two objects as equal when some business fact happens. In some business scenarios, developers provide their own implementation in order to force their own equality mechanism regardless the memory addresses.

As per the Java documentation, developers should override both methods in order to achieve a fully working equality mechanism — it's not enough to just implement the **equals()** method.

**If two objects are equal according to the equals(Object) method, then calling the hashcode() method on each of the two objects must produce the same integer result.**

In the following sections, we provide several examples that show the importance of overriding both methods and the drawbacks of overriding **equals()** without **hashcode()**.

## Practical Example

We define a class called **Student** as the following:

package com.programmer.gate.beans;

public class Student {

private int id;

private String name;

public Student(int id, String name) {

this.name = name;

this.id = id;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

For testing purposes, we define a main class **HashcodeEquals**that checks whether two instances of **Student** (who have the exact same attributes) are considered as equal.

public class HashcodeEquals {

public static void main(String[] args) {

Student alex1 = new Student(1, "Alex");

Student alex2 = new Student(1, "Alex");

System.out.println("alex1 hashcode = " + alex1.hashCode());

System.out.println("alex2 hashcode = " + alex2.hashCode());

System.out.println("Checking equality between alex1 and alex2 = " + alex1.equals(alex2));

}

}

Output:

alex1 hashcode = 1852704110

alex2 hashcode = 2032578917

Checking equality between alex1 and alex2 = false

Although the two instances have exactly the same attribute values, they are stored in different memory locations. Hence, they are not considered equal as per the default implementation of **equals()**. The same applies for **hashcode()** — a random unique code is generated for each instance.

## Overriding equals()

For business purposes, we consider that two students are equal if they have the same ID, so we override the **equals()** method and provide our own implementation as the following:

@Override

public boolean equals(Object obj) {

if (obj == null) return false;

if (!(obj instanceof Student))

return false;

if (obj == this)

return true;

return this.getId() == ((Student) obj).getId();

}

In the above implementation, we are saying that two students are equal if and only if they are stored in the same memory address **OR**they have the same ID. Now if we try to run **HashcodeEquals**,we will get the following output:

alex1 hashcode = 2032578917

alex2 hashcode = 1531485190

Checking equality between alex1 and alex2 = true

As you noticed, overriding **equals()**with our custom business forces Java to consider the ID attribute when comparing two **Student** objects.

### equals() With ArrayList

A very popular usage of **equals()**is defining an array list of **Student** and searching for a particular student inside it. So we modified our testing class in order the achieve this.

public class HashcodeEquals {

public static void main(String[] args) {

Student alex = new Student(1, "Alex");

List < Student > studentsLst = new ArrayList < Student > ();

studentsLst.add(alex);

System.out.println("Arraylist size = " + studentsLst.size());

System.out.println("Arraylist contains Alex = " + studentsLst.contains(new Student(1, "Alex")));

}

}

After running the above test, we get the following output:

Arraylist size = 1

Arraylist contains Alex = true

## Overriding hashcode()

Okay, so we override **equals()**and we get the expected behavior — even though the hash code of the two objects are different. So, what's the purpose of overriding **hashcode()**?

### equals() With HashSet

Let's consider a new test scenario. We want to store all the students in a **HashSet**, so we update **HashcodeEquals**as the following:

public class HashcodeEquals {

public static void main(String[] args) {

Student alex1 = new Student(1, "Alex");

Student alex2 = new Student(1, "Alex");

HashSet < Student > students = new HashSet < Student > ();

students.add(alex1);

students.add(alex2);

System.out.println("HashSet size = " + students.size());

System.out.println("HashSet contains Alex = " + students.contains(new Student(1, "Alex")));

}

}

If we run the above test, we get the following output:

HashSet size = 2

HashSet contains Alex = false

WAIT!! We already override **equals()**and verify that alex1 and alex2 are equal, and we all know that **HashSet** stores unique objects, so why did it consider them as different objects ?

**HashSet** stores its elements in memory buckets. Each bucket is linked to a particular hash code. When calling students.add(alex1), Java stores alex1 inside a bucket and links it to the value of alex1.hashcode(). Now any time an element with the same hash code is inserted into the set, it will just replace alex1. However, since alex2 has a different hash code, it will be stored in a separate bucket and will be considered a totally different object.

Now when **HashSet** searches for an element inside it, it first generates the element's hash code and looks for a bucket which corresponds to this hash code.

Here comes the importance of overriding **hashcode()**, so let's override it in **Student** and set it to be equal to the ID so that students who have the same ID are stored in the same bucket:

@Override

public int hashCode() {

return id;

}

Now if we try to run the same test, we get the following output:

HashSet size = 1

HashSet contains Alex = true

See the magic of **hashcode()**! The two elements are now considered as equal and stored in the same memory bucket, so any time you call contains() and pass a student object holding the same hash code, the set will be able to find the element.

The same is applied for **HashMap, HashTable**, or any data structure that uses a hashing mechanism for storing elements.

## Conclusion

In order to achieve a fully working custom equality mechanism, it is mandatory to override **hashcode()** each time you override **equals().** Follow the tips below and you'll never have leaks in your custom equality mechanism:

* If two objects are equal, they MUST have the same hash code.
* If two objects have the same hash code, it doesn't mean that they are equal.
* Overriding **equals()**alone will make your business fail with hashing data structures like: **HashSet, HashMap, HashTable** ... etc.
* Overriding **hashcode()**alone doesn't force Java to ignore memory addresses when comparing two objects.