Java hashCode() and equals() Methods

Last Updated: January 30, 2022 By: Lokesh Gupta Java Basics Java Equals, Java Hashcode

Learn about Java `hashCode()` **and** `equals()` **methods**, their **default implementation, and how to correctly override them**. Also, we will learn to implement these methods using 3rd party classes `HashCodeBuilder` and `EqualsBuilder`.

> **`hashCode()`** and **`equals()`** methods have been defined in `Object` class which is parent class for all java classes. For this reason, all java objects inherit a default implementation of these methods.

Table of Contents:

1. Uses of hashCode() and equals() Methods
2. Override the default behavior
3. EqualsBuilder and HashCodeBuilder
4. Generate hashCode() and equals() using Eclipse
5. Important things to remember
6. Special Attention When Using in ORM

1. Uses of hashCode() and equals() Methods

1. `equals(Object otherObject)` – verifies the equality of two objects. Its default implementation simply checks the object references of two objects to verify their equality.
   By default, two objects are equal if and only if they are refer to the same memory location. Most Java classes override this method to provide their own comparison logic.
2. `hashcode()` – returns a unique integer value for the object in runtime.
   By default, integer value is derived from memory address of the object in heap (but it's not mandatory).
   The object's hash code is used for determining the index location, when this object needs to be stored in some HashTable like data structure.

1.1. Contract between hashCode() and equals()

Overriding the the `hashCode()` is generally necessary whenever `equals()` is overridden to maintain the general contract for the `hashCode()` method, which states that **equal objects must have equal hash codes**.

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode`**() must consistently return the same integer**, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent between the two executions of the same application or program.
- **If two objects are equal** according to the `equals()` method, then calling the `hashCode()` on each of the **two objects must produce the same integer** result.
- It is ***not*** **required that if two objects are unequal** according to the `equals()`, then calling the `hashCode()` on each of the **both objects must produce distinct integer** results.
  However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

2. Overriding the Default Behavior

Everything works fine until we do not override any of both methods in our classes. But, sometimes, the application needs to change the default behavior of some objects.

Let us understand **why we need to override equals and hashcode** methods.

2.1. The default behavior of Employee class

Let's take an example where your application has `Employee` object. Let us create a minimal possible structure of `Employee` class:

```
public class Employee
{
    private Integer id;
    private String firstname;
    private String lastName;
    private String department;
    //Setters and Getters
}
```

Above `Employee` class has some fundamental attributes and their accessor methods. Now consider a simple situation where you need to **compare two Employee objects**. Both employee objects have the same `id`.

```
public class EqualsTest {
    public static void main(String[] args) {
        Employee e1 = new Employee();
        Employee e2 = new Employee();

        e1.setId(100);
        e2.setId(100);

        System.out.println(e1.equals(e2));  //false
    }
}
```

No prize for guessing. The above method will print "*false*."

But is it correct after knowing that both objects represent the same employee? In a real-time application, this should return `true`*.*

2.2. Should we override only equals() method?

To achieve correct application behavior, we need to override `equals()` method as below:

```
public boolean equals(Object o) {
    if(o == null)
    {
        return false;
    }
    if (o == this)
    {
        return true;
    }
    if (getClass() != o.getClass())
    {
        return false;
    }

    Employee e = (Employee) o;
    return (this.getId() == e.getId());
}
```

Add this method to the `Employee` class, and `EqualsTest` will start returning "`true`".

So are we done? Not yet. Let’s test the above-modified `Employee` class again in a different way.

```
import java.util.HashSet;
import java.util.Set;

public class EqualsTest
{
    public static void main(String[] args)
    {
        Employee e1 = new Employee();
        Employee e2 = new Employee();

        e1.setId(100);
        e2.setId(100);

        //Prints 'true'
        System.out.println(e1.equals(e2));

        Set<Employee> employees = new HashSet<Employee>();
        employees.add(e1);
        employees.add(e2);

        System.out.println(employees);  //Prints two objects
    }
}
```

The above example prints two objects in the second print statement.

If both employee objects have been equal, in a `Set` which stores unique objects, there must be only one instance inside `HashSet` because both objects refer to the same employee. What is it we are missing??

2.3. Overriding hashCode() is necessary

We are missing the second important method `hashCode()`. As java docs say, if we override `equals()` then we ***must*** override `hashCode()`. So let's add another method in our `Employee` class.

```
@Override
public int hashCode()
{
    final int PRIME = 31;
    int result = 1;
    result = PRIME * result + getId();
    return result;
}
```