equals() and hashCode() methods in Java

Java.lang.object has two very important methods defined: public boolean equals(Object obj) and public int hashCode().

equals() method

In java equals() method is used to compare equality of two Objects. The equality can be compared in two ways:

- **Shallow comparison:** The default implementation of equals method is defined in Java.lang.Object class which simply checks if two Object references (say x and y) refer to the same Object. i.e. It checks if x == y. Since Object class has no data members that define its state, it is also known as shallow comparison.
- **Deep Comparison:** Suppose a class provides its own implementation of equals() method in order to compare the Objects of that class w.r.t state of the Objects. That means data members (i.e. fields) of Objects are to be compared with one another. Such Comparison based on data members is known as deep comparison.

Syntax :

```
public boolean equals  (Object obj)

// This method checks if some other Object
// passed to it as an argument is equal to
// the Object on which it is invoked.
```

Some principles of equals() method of Object class : If some other object is equal to a given object, then it follows these rules:

- **Reflexive :** for any reference value a, a.equals(a) should return true.
- **Symmetric :** for any reference values a and b, if a.equals(b) should return true then b.equals(a) must return true.
- **Transitive :** for any reference values a, b, and c, if a.equals(b) returns true and b.equals(c) returns true, then a.equals(c) should return true.
- **Consistent :** for any reference values a and b, multiple invocations of a.equals(b) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.

Note: For any non-null reference value a, a.equals(null) should return false.

```
// Java program to illustrate
// how hashCode() and equals() methods work
import java.io.*;

class Geek
{

    public String name;
    public int id;

    Geek(String name, int id)
    {

        this.name = name;
        this.id = id;
    }

    @Override
    public boolean equals(Object obj)
    {

    // checking if both the object references are
    // referring to the same object.
    if(this == obj)
            return true;

        // it checks if the argument is of the
        // type Geek by comparing the classes
        // of the passed argument and this object.
        // if(!(obj instanceof Geek)) return false; ---> avoid.
        if(obj == null || obj.getClass()!= this.getClass())
            return false;

        // type casting of the argument.
        Geek geek = (Geek) obj;

        // comparing the state of argument with
        // the state of 'this' Object.
        return (geek.name == this.name && geek.id == this.id);
    }

    @Override
    public int hashCode()
    {

        // We are returning the Geek_id
        // as a hashcode value.
        // we can also return some
        // other calculated value or may
        // be memory address of the
        // Object on which it is invoked.
        // it depends on how you implement
        // hashCode() method.
        return this.id;
    }

}

//Driver code
class GFG
{

    public static void main (String[] args)
    {

        // creating the Objects of Geek class.
        Geek g1 = new Geek("aa", 1);
        Geek g2 = new Geek("aa", 1);

        // comparing above created Objects.
        if(g1.hashCode() == g2.hashCode())
        {

            if(g1.equals(g2))
                System.out.println("Both Objects are equal. ");
            else
                System.out.println("Both Objects are not equal. ");

        }
        else
        System.out.println("Both Objects are not equal. ");
    }
}
```

Output:

```
Both Objects are equal.
```

In above example see the line:

```
// if(!(obj instanceof Geek)) return false;--> avoid.-->(a)
```

We've used this line instead of above line:

```
if(obj == null || obj.getClass()!= this.getClass()) return false; --->(y)
```

Here, First we are comparing the hashCode on both Objects (i.e. g1 and g2) and if same hashcode is generated by both the Objects that does not mean that they are equal as hashcode can be same for different Objects also, if they have the same id (in this case). So if get the generated hashcode values are equal for both the Objects, after that we compare the both these Objects w.r.t their state for that we override equals(Object) method within the class. And if both Objects have the same state according to the equals(Object) method then they are equal otherwise not. And it would be better w.r.t. performance if different Objects generates different hashcode value.

Reason : Reference **obj** can also refer to the Object of subclass of Geek. Line (b) ensures that it will return false if passed argument is an Object of subclass of class Geek. But the **instanceof** operator condition does not return false if it found the passed argument is a subclass of the class Geek. Read InstanceOf operator.

hashCode() method

It returns the hashcode value as an Integer. Hashcode value is mostly used in hashing based collections like HashMap, HashSet, HashTable….etc. This method must be overridden in every class which overrides equals() method.
Syntax :

```
public int hashCode()

// This method returns the hash code value
// for the object on which this method is invoked.
```

The general contract of hashCode is:

- During the execution of the application, if hashCode() is invoked more than once on the same Object then it must consistently return the same Integer value, provided no information used in **equals(Object)** comparison on the Object is modified. It is not necessary that this Integer value to be remained same from one execution of the application to another execution of the same application.
- If two Objects are equal, according to the the **equals(Object)** method, then hashCode() method must produce the same Integer on each of the two Objects.
- If two Objects are unequal, according to the the **equals(Object)** method, It is not necessary the Integer value produced by hashCode() method on each of the tw Objects will be distinct. It can be same but producing the distinct Integer on ea

of the two Objects is better for improving the performance of hashing based Collections like HashMap, HashTable…etc.

Note: Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce distinct hash codes.

Related link : Overriding equal in Java
Reference: JavaRanch

This article is contributed by **Nitsdheerendra**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Recommended Posts:

Java.util.BitSet class methods in Java with Examples | Set 2
Shadowing of static functions in Java
How does default virtual behavior differ in C++ and Java ?
How are Java objects stored in memory?
How are parameters passed in Java?
Are static local variables allowed in Java?
final variables in Java
Default constructor in Java
Assigning values to static final variables in Java
Comparison of Exception Handling in C++ and Java
Does Java support goto?
Arrays in Java
Access specifier of methods in interfaces
Inheritance and constructors in Java
More restrictive access to a derived class method in Java

Improved By : akabhishek881