Use of java hashCode and equals.

Object class provides two methods hashcode() and equals() to represent the identity of an object. It is a common convention that if one method is overridden then other should also be implemented.

Equals is always called after the hashCode method in a java hashed collection while adding and removing elements.

Before explaining why, lets see what is the contract between these two methods hold. As per the Java API documentation:

* Whenever hashcode is invoked on the same object more than once during an execution of a Java application, the hashcode() method must consistently return the same integer, provided no information used in equals() comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
* 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.
* It is NOT required that if two objects are unequal according to the equals(Java.lang.Object) method, then calling the hashCode() method on each of the two 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 hashtables.

Now, consider an example where the key used to store in the Hashmap is an Integer. Consider that <Integer class doesn't implement hashcode() method. The code would look like:

map.put(new Integer(5),"Value1");

String value = (String) map.get(new Integer(5));

System.out.println(value);

//Output : Value is null

Null value will be displayed since the hashcode() method returns a different hash value for the Integer object created at line 2 and JVM tries to search for the object at different location.

Now if the Integer class has hashcode() method like:

public int hashCode() {

return value;

}

Everytime a new Integer object is created with same integer value passed; the Integer object will return the same hash value. Once the same hash value is returned, JVM will go to the same hashmap bucket every time and if in case there are more than one objects present for the same hash value it will use equals() method to identify the correct object.

Another step of caution that needs to be taken is that while implementing the hashcode() method the fields that are present in the hashcode() should not be the one which could change the state of object.

Consider the example:

public class FourWheeler implements Vehicle {

private String name;

private int purchaseValue;

private int noOfTyres;

public FourWheeler(){}

public FourWheeler(String name, int purchaseValue) {

this.name = name;

this.purchaseValue = purchaseValue;

}

public void setPurchaseValue(int purchaseValue) {

this.purchaseValue = purchaseValue;

}

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + ((name == null) ? 0 : name.hashCode());

result = prime \* result + purchaseValue;

return result;

}

//imagine equals() method is present. Will show how to implement

//later in article

}

//Try running this snippet

FourWheeler fourWObj = new FourWheeler("Santro","333333);

map.put(fourWObj,"Hyundai);

fourWObj.setPurchaseValue("555555)

System.out.println(map.get(fourWObj));

//Output: null

We can see that inspite of passing the same object the value returned is null. This is because the hashcode() returns a different value since the purchaseValue is set to 555555 from 333333.

Though the above implementation is correct but it fails because for generating hashcode a changeable property (in this case price) is selected. To make above implementation correct it is recommended to has fields in hashcode which doesn't change.

One compatible, but not all that useful, way to define hashCode() is like this:

public int hashcode(){

return 0;

}

public class StringHelper {

private String inputString;

public StringHelper(String string) {

inputString=string;

}

@Override

public int hashCode() {

return inputString.length();

}

public static void main(String[] args) {

StringHelper helperObj = new StringHelper("string");

StringHelper helperObj1 = new StringHelper("string");

if(helperObj.hashCode() == helperObj1.hashCode()){

System.out.println("HashCode are equal");

}

if(helperObj.equals(helperObj1)){

System.out.println("Objects are equal");

}else{

System.out.println("Objects are not equal");

}

}

public String getInputString() {

return inputString;

}

// imagine equals() method is implemented. Will show how to implement

// later in article

}

// Output:

HashCode are equal

Objects are not equal

We can see that even though the StringHelper object contains the same value the equals method has returned false but the hashcode method has return true value.

To prevent this inconsistency, we should make sure that we override equals() method too such that the contract between both methods doesn't fail.

## How to implement equals() method

**Steps that need to be taken into consideration while implementing equals method.**

* **Use the == operator to check if the argument is a reference to this object.**If so, return true. This is just a performance optimization, but one that is worth doing if the comparison is potentially expensive.
* **Use the instanceof operator to check if the argument has the correct type**.
* If not, return false. Typically, the correct type is the class in which the method occurs. Occasionally, it is some interface implemented by this class. Use an interface if the class implements an interface that refines the equals contract to permit comparisons across classes that implement the interface. Collection interfaces such as Set, List, Map, and Map.Entry have this property.
* **Cast the argument to the correct type. Because this cast was preceded by an instanceof test, it is guaranteed to succeed.**
* **For each significant field in the class, checks if that field of the argument matches the corresponding field of this object**.If all these tests succeed, return true; otherwise, return false
* **When you are finished writing your equals method, ask yourself three questions: Is it symmetric? Is it transitive? Is it consistent?**

The correct implementation of equals method for the StringHelper class could be:

public class StringHelper {

private String inputString;

public StringHelper(String string) {

inputString = string;

}

@Override

public int hashCode() {

return inputString.length();

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

final StringHelper other = (StringHelper) obj;

if (inputString == null && other.inputString != null) {

return false;

} else if (!inputString.equals(other.inputString))

return false;

return true;

}

public static void main(String[] args) {

StringHelper helperObj = new StringHelper("string");

StringHelper helperObj1 = new StringHelper("string");

if(helperObj.hashCode() == helperObj1.hashCode()){

System.out.println("HashCode are equal");

}

if(helperObj.equals(helperObj1)){

System.out.println("Objects are equal");

}else{

System.out.println("Objects are not equal");

}

}

public String getInputString() {

return inputString;

}

}

// Output:

HashCode are equal

Objects are equal