**Why always override hashcode() if overriding equals()?**

In Java, every object has access to the *equals()* method because it is inherited from the *Object* class. However, this default implementation just simply compares the memory addresses of the objects. You can override the default implementation of the *equals()* method defined in *java.lang.Object*. If you override the *equals()*, you MUST also override *hashCode()*. Otherwise a violation of the general contract for Object.hashCode will occur, which can have unexpected repercussions when your class is in conjunction with all hash-based collections.

Here is the contract, copied from the *java.lang.Object* specialization:

**public int hashCode()**

Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by *java.util.Hashtable*.

The general contract of hashCode is:

* Whenever it 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.

As much as is reasonably practical, the *hashCode* method defined by class *Object* does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

The default implementation of *equals()* method checks to see if the two objects have the same identity. Similarly, the default implementation of the hashCode() method returns an integer based on the object's identity and is not based on the values of instance (and class) variables of the object. No matter how many times the values of its instance variables (data fields) change, the hash code calculated by the default hashCode implementation does not change during the life of the object.

Consider the following code, we have overridden *equals()* method to check if two objects are equal based on the values of their instance variables. Two objects may be stored at different memory addresses but may still be equal base on their instance variable.

public class CustomerID {  
 private long crmID;  
 private int nameSpace;  
  
 public CustomerID(long crmID, int nameSpace) {  
 super();  
 this.crmID = crmID;  
 this.nameSpace = nameSpace;  
 }  
  
 public boolean equals(Object obj) {  
 //null instanceof Object will always return false  
 if (!(obj instanceof CustomerID))  
 return false;  
 if (obj == this)  
 return true;  
 return this.crmID == ((CustomerID) obj).crmID &&  
 this.nameSpace == ((CustomerID) obj).nameSpace;  
 }  
  
 public static void main(String[] args) {  
 Map m = new HashMap();  
 m.put(new CustomerID(2345891234L,0),"Jeff Smith");  
 System.out.println(m.get(new CustomerID(2345891234L,0)));  
 }  
  
}

Compile and run the above code, the output result is

null

What is wrong? The two instances of CustomerID are logically equal according to the class's *equals* method. Because the *hashCode()* method is not overridden, these two instances' identities are not in common to the default *hashCode* implementation. Therefore, the *Object.hashCode* returns two seemingly random numbers instead of two equal numbers. Such behavior violates "Equal objects must have equal hash codes" rule defined in the hashCode contract.

Let's provide a simple *hashCode()* method to fix this problem:

public class CustomerID {  
 private long crmID;  
 private int nameSpace;  
  
 public CustomerID(long crmID, int nameSpace) {  
 super();  
 this.crmID = crmID;  
 this.nameSpace = nameSpace;  
 }  
  
 public boolean equals(Object obj) {  
 //null instanceof Object will always return false  
 if (!(obj instanceof CustomerID))  
 return false;  
 if (obj == this)  
 return true;  
 return this.crmID == ((CustomerID) obj).crmID &&  
 this.nameSpace == ((CustomerID) obj).nameSpace;  
 }  
  
 public int hashCode() {  
 int result = 0;  
 result = (int)(crmID/12) + nameSpace;  
 return result;  
 }  
  
 public static void main(String[] args) {  
 Map m = new HashMap();  
 m.put(new CustomerID(2345891234L,0),"Jeff Smith");  
 System.out.println(m.get(new CustomerID(2345891234L,0)));  
 }  
  
}

Compile and run the above code, the output result is

Jeff Smith

The hashcode distribution for instances of a class should be random. This is exactly what is meant by the third provision of the *hashCode* contract. Write a correct *hashCode* method is easy, but to write an effective *hashCode* method is extremely difficult.

For example, From [How to Avoid Traps and Correctly Override Methods From java.lang.Object](http://www.javaworld.com/javaworld/jw-01-1999/jw-01-object.html): If you are unsure how to implement hashCode(), just always return 0 in your implementations. So all of your custom objects will return the same hash code. Yes, it turns hashtable of your objects into one (possibly) long linked-list, but you have implemented hashCode() correctly!

public int hashCode(){  
 return 0;  
}

It's legal because it ensures that equal objects have the same hash code, but it also indicates that every object has the same hash code. So every object will be hashed into the same bucket, and hash tables degenerate to linked lists. The performance is getting worse when it needs to process a large number of objects. How to implement a good hash function is a big topic and we will not cover here.

**Further Reading**

[Java Theory and Practice: Hsahing It Out"](http://www-128.ibm.com/developerworks/java/library/j-jtp05273.html)

[Equals and Hash code](http://www.geocities.com/technofundo/tech/java/equalhash.html)

# See more samples here:

<http://stackoverflow.com/questions/11850929/hashcode-vs-equals>