**Deep in hashCode()**

**hashCode()** is a method that we override in our custom classes and its mandatory to override if the "equals" method is overridden. Hashcode is also used in hashing based collection classes such as Hashtable, HashMap, HashSet etc. Understanding hashCode is very simple.

*Let’s first see why and when we use hashCode?*

As we know, hashCode is a numeric value. It is used in retrieval of the respective object and it need not be unique which we see later. Now since its a number its always fast to retrieve an object using a number rather than an alphabetic key. How it will do? Assume we created a new object by passing some value which is already available in some other object. Now the new object will return the same hash value as of another object because the value passed is same. Once the same hash value is returned, JVM will go to the same memory address 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.

**Note:** A 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. I will explain this later with example.

*In what scenarios we override equals and hashCode methods.*

Here are few.  
In Hashtable, the hashCode method only gets invoked when you use the Object as the key to a Hashtable. It is not used when the Object is a Hashtable value. Most of the time your Hashtable keys are simple Strings, so you rarely need to write custom equals and hashCode methods. But in the case of HashSet, to help you check for duplicate Objects, we need a custom equals and hashCode method because here our Objects act as both key and value.

*Lets see how HashMap uses the hash code.*

In HashMap, when you try to get the object using get(key) method, the HashMap first gets the hash code of the key (remember that key is an object in the HashMap and it should have hashCode() method overridden) and generates again the hashcode to just defend against poor quality hash functions. Using this hashcode, an index for hash code will be created and this index is used to look up the entry table which is where all the hash entries are available. Now, once you get the Entry object from the Entry table using index, check if the key of the Entry object is same as the key we sent as parameter to get method. If yes, then return the value of the Entry object. If no, then go to next Entry object and continue the same.

Below is the code in the get method of HashMap

int hash = hash(key.hashCode());

for (Entry e = table[indexFor(hash, table.length)];

e != null;

e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

*Now lets go into some more details of hashCode.*

In the Java API documentation, the general contract of hashCode is given as:

* 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.

Mmmm fine. So what it says is equal objects must produce the same hash code anytime but unequal objects does not need to produce different hash code. They can produce same or different hashcode. Lets see an example.

public class Student {

private String name;

private int age;

private int totalMarks;

public Student(){}

public Student(String name, int age, int marks) {

this.name = name;

this.age = age;

this.marks = marks;

}

public void setMarks(int marks) {

this.marks = marks;

}

@Override

public boolean equals(Object obj)

{

if(this == obj)

return true;

if((obj == null) || (obj.getClass() != this.getClass()))

return false;

// object must be Test at this point

Student stud = (Student)obj;

return age == stud.age &&

(name != null && name.equals(stud.name));

}

@Override

public int hashCode() {

final int seed = 37;

int result = 1;

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

result = seed \* result + age;

return result;

}

}

In the above example I am overriding "equals" and "hashCode" methods. If you closely observe, whatever the fields I am comparing in "equals" method, the same fields I am using to generate hash code in "hashCode" method. Ofcourse, this is not mandatory. You can use subset of the variables that participate in the equals method comparison to improve performance of the hashCode method.

Lets check below SCJP question

Q: Given:

11. public class Person {

12. private String name, comment;

13. private int age;

14. public Person(String n, int a, String c) {

15. name = n; age = a; comment = c;

16. }

17. public boolean equals(Object o) {

18. if (! (o instanceof Person)) return false;

19, Person p = (Person)o;

20. return age == p.age && name.equals(p.name);

21. }

22. }

What is the appropriate definition of the hashCode method in class Person?

A. return super.hashCode();

B. return name.hashCode() + age \* 7;

C. return name.hashCode() + comment.hashCode() / 2;

D. return name.hashCode() + comment.hashCode() / 2 - age \* 3;

Can you guess? Yes, you are right. Its B. Why is it B? Before looking at hashCode method, look at equals method. In equals method the comparision is made between person's age and name to check if the two objects are equal. So we need to consider only person's age and name when generating hashCode.

Got it? Now again go back to previous example of Student class. In the "equals" method or "hashCode" method I didn't use "marks" field to compare or generate the hash code. This I have indicated in the note above 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.". Since "marks" field can change the state of the object, I didn't use marks field in generating hash code. See below example.

@Override

public int hashCode() {

final int seed = 37;

int result = 1;

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

result = seed \* result + age;

result = seed \* result + marks;

return result;

}

Student studObj = new Student("Julie",24,80);

map.put(studObj,"First Year");

studObj.setMarks(85);

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

The output for the above System.out.println is null.

Finally, it is incorrect to involve a random number directly while computing the hash code of the class object, as it would not consistently return the same hash code for multiple invocations during the same execution

**Dive in equals()**

Similar to hashCode(), **equals()** is a method that we override in our custom classes and its mandatory to override if the "hashCode()" method is overridden because hashCode() must generate equal values for equal objects. Its a general contract that we use the same set of fields for generating hashCode() and equals() methods. Note that equal objects must have the same hashcode but again note that unequal objects can have same hashcode too.

*What about equals() method in java.lang.Object?*

Ofcourse "java.lang.Object" provides an implementation of equals but it just tests referential equality which is nothing but "x == y" (objects addresses equality). This is called "shallow comparision". Basically, the equals() method should test the equality of objects (whether the states of two objects are equal or not), not identity of the objects and for this reason when we override equals() method in our class, we compare field by field and then return true or false. And this is called "deep comparision".

*Lets first see when and where we use equals?*

In simple words, equals() method is used to compare two objects. To elaborate it with example, it is used in Collections in java. The collections method "contains" use equals method to compare the objects and returns true or false as shown below. Also in any application when you want to perform any operation based on objects equality, the equals methos in the object is used.

Example e1 = new Example();

Set<Example> hashSet1 = new java.util.HashSet<Example>();

hashSet1.add(e1);

System.out.println(hashSet1.contains(e1)); // prints false

*In what scenarios we override equals method.*

When you have to compare two objects state equality but not identity equality, you override equals method. Say for example two car objects are said to be equal if they have same registration number. In the same way, two student objects are said to be equal if they have same student Id. So in this case, you will override equals method to compare the registration number and dont compare object properties that change frequently. For example you should not compare the car current speed in the equals method because the same car can have different speeds in different time and this difference does not mean it is a different car.

In the Java API documentation, the general contract of equals is given as, whenever it is invoked on the same object more than once during an execution of a Java application, the equals method must consistently return true, provided no information used in equals comparisons on the object is modified. This is explained in later part of this article with example.

*Lets see the common mistakes we make in overriding equals method.*

**Defining equals with the wrong signature:**When you override an equals method, you should exactly follow the signature in java.lang.Object else it wont be overriding but an overloaded alternative.   
Lets see the common mistake we make here - lets assume you have a class "Circle" that has centerX and centerY properties.

public class Circle {

private final int centerX;

private final int centerY;

public Circle(int centerX, int centerY) {

this.centerX = centerX;

this.centerY = centerY;

}

public int getCenterX() {

return centerX;

}

public int getCenterY() {

return centerY;

}

}

And generally you will define the equals method in the "Circle" as below and this is completely wrong. This method is not overriding the equals method in Object class.

public boolean equals(Circle other) {

return (this.getCenterX() == other.getCenterX()

&& this.getCenterY() == other.getCenterY());

}

So when you try to equate two objects as above, it will always written true but when you check the same in hashSet it will return false. The reason the above equals method is not following exact signature as in java.lang.Object class and so only when objects has referential equality (x == y), the equals return true else will return false.

Circle c1 = new Circle(100, 200);

Circle c2 = new Circle(100, 200);

HashSet coll = new HashSet();

coll.add(c1);

//This returns false as the equals method used is the equals

//method in the java.lang.Object which is referential equality

System.out.println(coll.contains(c2));

To correct this, just change the signature as below and all works perfectly fine.

@Override

public boolean equals(Object other) {

boolean result = false;

if (other instanceof Circle) {

Circle that = (Circle) other;

result = (this.getCenterX() == that.getCenterX()

&& this.getCenterY() == that.getCenterY());

}

return result;

}

**Overriding equals without overriding hashCode:** The next mistake that could happen is redefining equals without redefining hashCode. Lets see what will go wrong if you dont override both.

Circle c1 = new Circle(100, 200);

Circle c2 = new Circle(100, 200);

//This consistently returns true

System.out.println(c1.equals(c2));

HashSet coll = new HashSet();

coll.add(c1);

//This wont consistently returns true

System.out.println(coll.contains(c2));

Why "System.out.println(coll.contains(c2))" wont return true consistently? And the answer is you didnt override haashCode method. If you look closely, the collection in the example above is a HashSet which means the elements of the collection are determined by their hash code. Since you didnt override hashCode, the default implementation of hashCode in the java.lang.Object is used to retrieve the object from the HashSet and this default implementation may not be using the same fields for generating hashCode which caused the whole chaos. To correct this problem we need to override hashCode method.

@Override

public boolean equals(Object other) {

boolean result = false;

if (other instanceof Circle) {

Circle that = (Circle) other;

result = (this.getCenterX() == that.getCenterX()

&& this.getCenterY() == that.getCenterY());

}

return result;

}

@Override

public int hashCode() {

final int seed = 37;

int result = 1;

result = seed \* (result + this.getCenterX() + this.getCenterY());

return result;

}

**Changing equals without also changing hashCode:**This is one common mistake we generally make. And this results in the same issue as above one - the inconsistent results when the objects are compared in HashSet.

**Defining equals with the fields that actually does not define the object:**This must be confusing but lets see with an exmaple and its quite simple to understand. Lets assume we have a class called "Car" and it has properties "licenceNo", "registrationNo" and "currentSpeed" like below. So when you define the comparision in the equals method, you should use only the licenceNo and registrationNo fields but never "currentSpeed" because currentSpeed can change for the same car and when it is changed, it does not mean that the car has changed.

public class Car {

private int licenceNo;

private int registrationNo;

private int currentSpeed;

public Car(int licenceNo, int registrationNo) {

this.licenceNo = licenceNo;

this.registrationNo = registrationNo;

}

public int setLicenceNo() {

return licenceNo;

}

public int getLicenceNo() {

return licenceNo;

}

...

@Override

public boolean equals(Object other) {

boolean result = false;

if (other instanceof Car) {

Car that = (Car) other;

result = (this.getLicenceNo() == that.getLicenceNo()

&& this.getRegistrationNo() == that.getRegistrationNo());

//Never compare current speed here as it may change

//and will result in wrong results

}

return result;

}

}

**Failing to define equals as an equivalence relation:** Oops this is most important one. If you look at the java documentation for "equals" method, it defines few conditions that should be satisfield when you override equals method. mmmmm.. What are they?

* It is *reflexive*: for any non-null value x, the expression x.equals(x) should return true.
* It is *symmetric*: for any non-null values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
* It is *transitive*: for any non-null values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
* It is *consistent*: for any non-null values x and y, multiple invocations of x.equals(y) should consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
* For any non-null value x, x.equals(null) should return false.

Lets see this example which explains where the above conditions will fail.

public class ColoredCircle extends Circle {

private final String color;

public ColoredCircle(int centerX, int centerY, String color) {

super(centerX, centerY);

this.color = color;

}

@Override

public boolean equals(Object other) {

boolean result = false;

if (other instanceof ColoredCircle) {

ColoredCircle that = (ColoredCircle) other;

result = (this.color.equals(that.color) && super.equals(that));

}

return result;

}

}

Now check the equality.

Circle c = new Circle(100, 200);

ColoredCircle cc = new ColoredCircle(100, 200, "Red");

System.out.println(c.equals(cc)); // prints true

System.out.println(cc.equals(cc)); // prints false

This is violating the equality rule defined by Java. You may get a doubt on where this failure will affect. This will affect in collections. So in collection where you perform a comparision like using "contains" method, you get wrong results. See below.

Set<Circle> hashSet1 = new java.util.HashSet<Circle>();

hashSet1.add(c);

System.out.println(hashSet1.contains(cc)); // prints false

Set<Circle> hashSet2 = new java.util.HashSet<Circle>();

hashSet2.add(cc);

System.out.println(hashSet2.contains(c)); // prints true

One possible solution for this problem is, change your equals method to compare classes also as.....

@Override

public boolean equals(Object other) {

boolean result = false;

if (other instanceof Circle) {

Circle that = (Circle) other;

result = (this.getCenterX() == that.getCenterX()

&& this.getCenterY() == that.getCenterY()

&& this.getClass().equals(that.getClass()));

}

return result;

}

*Tips in overriding equals method.*

Finally I would like to mention couple of tips you can follow when you override equals method.

**First check for identity equality:** Here what I mean is in the equals method, first check for objects identity and then go for method euality. See below sample equals method.

@Override

public boolean equals(Object other) {

boolean result = false;

//Check the identity first and they are equal,

//then there is no need to compare fields

if (other instanceof Circle) {

if(this == other){

return true;

}else{

Circle that = (Circle) other;

result = (this.getCenterX() == that.getCenterX()

&& this.getCenterY() == that.getCenterY());

}

}

return result;

}

**Compare frequently changing fields first:** It is always good practice to order field comparisons such that the fields most likely to differ should be evaluated first. This allows the && "short-circuit" logical operator to minimize execution time which means better performance.