HASHCODE() - public native int hashCode()

EQUALS() - public boolean equals(Object)

1. When you are creating any object, for all the objects a hashcode will be assigned internally by the JVM. On the basis of that hashcode, JVM (tries to) identifies each object uniquely.
2. There is no hashCode() that uniquely identifies an instance of an object no matter what. The "hashcode" is designed to "narrow down" uniqueness (for Hashtables), but you must always follow it with "equals"
3. For each object, hashCode() method returns an integer hashcode.
4. The value returned by hashCode() is by no means guaranteed to be the memory address of the object.
5. Most classes will *override* ***hashCode()*** *such that two instances that are semantically equivalent but are not the same instance, i.e. reference is different but they have same values, say for variables, will hash to the same value*. This is especially important if the classes may be used within another data structure, such as Set, that *relies on hashCode() being consistent with equals()*.
6. For different DS and also Collections, storing user defined class (say Student) object, these make use of hashCode() and equals() method to check if the entity being inserted are same or not, it is very important to override the equals() method for the user defined class (Student) so that when storing Student objects in collection the compiler can correctly check if the objects being inserted are same or not.
7. If we don't override it, Object class's equals() will be used which will compare reference of the objects and this will not be a correct way because it is possible that two Student objects (so different references) can have same (say) id. So, in this case though it is the information for same Student, because the references are different, second object also gets stored in collection resulting in redundant data.
8. The above implementation will lead to duplicate data and say if the collection is a Set type which does not allow duplicate data, the integrity of the data and collection will be compromised as it is now allowing duplicated data.
9. This is the reason that anytime we are creating our own class and using it in Collections (or even if not) we should override the equals method so that duplicate entries cannot be inserted into the collection.
10. The part of the contract here which is important is: *objects which are .equals() MUST have the same .hashCode().* Also:
    1. If two objects are not equal by equals() method then there is no restriction on their hashcode, may be same or may not be same.
    2. If hashcode of 2 objects are equal then these objects may or may not be equal by equals().
    3. If hashcode of 2 objects are not equal, then these objects are always not equal by. equals().
11. Flow is like this – first hashcode is checked, if it is equal then the equals() method is checked and if this is also true then the object is duplicate.
12. Ideally, hashCode() generates hashcode based on the member variables (their values) of that object.
13. equals() should also ideally compare the values of the member variables of the two objects. This means that equals() will be true iff all corresponding member variables of both the objects are having the same values.
14. So, if equals() is true, meaning that all the corresponding variables in both the objects have same values, and because the hashCode() also uses variables to generate the hashcode, conclusion is – if equals() is true then hashCode() must return same hashcode for both the objects.

E.g. of hashCode override -

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + id;

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

**return** result;

}

E.g. of equals() override –

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

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

**return** **false**;

Student other = (Student) obj;

**if** (id != other.id)

**return** **false**;

**if** (name == **null**) {

**if** (other.name != **null**)

**return** **false**;

} **else** **if** (!name.equals(other.name))

**return** **false**;

**return** **true**;

}

Note the checks (apart from variables) for equals() method.