Hashing related data structure follow, the following fundamental rule

Two equivalent objects should be placed in same bucket, but all object present in the same bucket need not be equal.

Contract between equals () and hascode() method: -

- If two objects are equal by **equals () method** then **there hascode must be equal** i.e., **two equivalent objects should have same hashcode.** i.e., **if r1.equals (r2) is true then r1.hascode() == r2.hascode() is always true**
- Object class equals () methods and hashcode () method follow above contract hence whenever we overriding equals () method compulsory we should override hashcode() to satisfy above contract i.e. (two equivalent object should have same hashcode)
- if two object are not equal by equals () methods then there is no restriction on hashcode may be equal or may not be equals.
- if hashcode of two objects are equal then we can't conclude anything about equals () methods it may return true or false.
- if hashcode of two objects are not equal then these objects are always not equal by equals () methods

Note: - To satisfy contract between equals () and hashcode() method whenever we overriding equals () method then compulsory to override hashcode() method otherwise we would not get any compile time or runtime exception but it is not a good practice.

Based on which parameter we override equals () method it is highly recommended to use same parameter while overriding hascode() methods also.

In all collection classes, in all wrapper classes and in string class equals () method is overridden for content comparison hence it is highly recommended to override equals () method in our class also for content comparison.

equals (Object obj): a method provided by j**ava.lang.Object** that indicates whether some other object passed as an argument is "equal to" the current instance.

The default implementation provided by the JDK is based on memory location — two objects are equal if and only if they are stored in the same memory address.

hashcode(): a method provided by **java.lang.Object** that returns an integer representation of the object memory address.

By default, this method returns a random integer that is unique for each instance. This integer might change between several executions of the application and won't stay the same

How equals and hashcode contract works in java?

Answer: - The default implementation is not enough to satisfy business needs, especially if we're talking about a huge application that considers two objects as equal when some business fact happens. In some business scenarios, developers provide their own implementation in order to force their own equality mechanism regardless the memory addresses.

As per the Java documentation in perspective of equal and hashcode contract, developers should override both methods in order to achieve a fully working equality mechanism — it's not enough to just implement the **equals ()** method.

What is relation Between == operator and equals () methods.

Answer: - There are following relation between == operator and equals () method: -

- if two objects are equal by == operator then these objects are equal by equals () methods i.e., **if r1==r2 is true then r1.equals (r2) is always true.**

- if two objects are not equal by == operator then we cannot conclude anything about equals () method it may return True or False i.e., **if r1== r2 is false then r1.equals (r2) may return True or False, and we can't expect exactly.**
- if two objects are equal by .equals () methods then we can't conclude anything about == operator it may returns True of False i.e., **if r1.equals (r2) is true then r1== r2 may return True or False, and we can't expect exactly;**
- if two object are not equals by equals () method then these objects are not equal by == operator i.e., **if r1.equals (r2) is false then r1==r2 is always false.**

What is the difference between == operator and equals () methods?

Answer: - Answer in one line in general we can use == operator for reference comparison and equals () method for content comparison.

There is the following difference between == operator and equals () methods: -

== operator	**equals () methods**
It is an operator can be use with primitive as well as object comparison	It is Object class method and used with only Object type we can't use it with primitive type.
It is use for Reference Comparisions.	It can also use Reference comparison
Can't override for content comparison	We can override it.
When use == operator compulsory there should be some relationship between argument types (either child to parent or parent to child or same type) otherwise we will get compile time error saying incompatible types.	if there is no relation between argument types then equals () method won't any compile time or runtime error simply it returns false.

Note: - For any object refence r

r==null

r.equals (null))

always return false.

Example: -

Thread t =new Thread ();

sop(t==null) false

sop(t.equals (null)false

When you are writing equals () method, which other method or methods do you need to override?

Answer: - hashcode() method. Since equals and hashCode have their contract, so overriding one and not other will break the contract between them.

Can two objects which are not equal have the same hashCode?

Answer: - YES, two objects, which are not equal to equals () method can still return same hashCode.

Suppose your Class has an Id field, should you include in equals ()? Why?

Answer: - Well including id is not a good idea in equals () method because this method should check equality based upon content and business rules. Also including id, which is mostly a database identifier and not available to transient object until they are saved into the database.

What happens if equals () are not consistent with compareTo() method?

Answer: -Some java.util.Set implementation e.g., SortedSet or it's concrete implementation TreeSet uses compareTo() method for comparing objects. If compareTo() is not consistent means doesn't return zero, if equals () method returns true, it may break Set contract, which is not to avoid any duplicates.

What happens if you compare an object to null using equals ()?

Answer: - When a **null** object is passed as an argument to equals () method, it should return **false**, it must not throw **NullPointerException**, but if you call equals method on reference, which is **null** it will throw **NullPointerException**. That's why it's better to use == operator for comparing null e.g., **if(object != null) object.equals(anohterObject).** By the way, if you comparing String literal with another String object then you better call equals () method on the String literal rather than known object to avoid NullPointerException.

What is the difference in using instanceof and getClass() method for checking type inside equals?

Answer: - The key difference comes from the point that **instanceof operator returns true**, even if compared with subclass e.g., Subclass instanceof Superclass is true, but with **getClass() it's false.**

By using getClass() you ensure that your equals () implementation doesn't return true if compared with subclass object. While if you use instanceof operator, you end up breaking symmetry rule for equals which says that if a.equals(b) is true than b.equals(a) should also be true. Just replace a and b with an instance of Superclass and Subclass, and you will end up breaking symmetry rule for equals () method.

How do you avoid NullPointerException, while comparing two Strings in Java?

Answer: - Since when compared to null, equals return false and doesn't throw NullPointerException, you can use this property to avoid NullPointerException while using comparing String. Suppose you have a known String "abc" and you are comparing with an unknown String variable str, then you should call equals as "abc".equals(str), this will not throw Exception in thread Main: java.lang.NullPointerException, even if str is null.

On the other hand, if you call str.equals("abc"), it will throw NullPointerException.

How does get () method of HashMap works, if two keys have the same hashCode?

Answer: - When two key returns same hashcode, they end up in the same bucket. Now, in order to find the correct value, you used keys.equals() method to compare with key stored in each Entry of linked list there.

There Are Two Objects A and B With Same Hashcode. I Am Inserting These Two Objects Inside a Hashmap.

Hmap.put(a,a);

Hmap.put(b,b);

Where A.hashcode()==b.hashcode()

Now Tell Me How Many Objects Will Be There Inside the Hashmap?

Answer: - There can be two different elements with the same hashcode. When two elements have the same hashcode then Java uses the equals to further differentiation. So, there can be one or two objects depending on the content of the objects.

What Is the Use of Hashcode in Java?

Answer: - Hashcode is used for **bucketing** in Hash implementations like HashMap, HashTable, HashSet etc. The value received from hashcode() is used as **bucket number for storing elements**. This bucket number is the address of the element inside the set/map. when you do contains () then it will take the hashcode of the element, then look for the bucket where hashcode points to and if more than 1 element is found in the same bucket (multiple objects can have the same hashcode) then it uses the equals () method to evaluate if object is equal, and then decide if contain () is true or false, or decide if element could be added in the set or not.