**Contract between equals() and hashCode() methods :**

* If two objects are equal(according to ****equals()**** method) then the ****hashCode()**** method should return the same integer value for both the objects.
* But, it is not necessary that the****hashCode()**** method will return the distinct result for the objects that are not equal (according to ****equals()**** method).

**Why do we need to Override equals and hashcode methods in Java :** Java suggests to always override hashCode() method if the class overrides equals().

1. **Override only equals() without overriding hashCode() :** Overriding only equals() method without overriding hashCode() causes the two equal instances to have unequal hash codes, that is in violation of the hashCode contract (mentioned in Javadoc) that clearly says, 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.

Since the default hashCode implementation in the Object class return distinct integers for distinct objects, if only equals() method is overridden,

**2. Override only hashCode() without overriding equals() :** If we only override hashCode() method, both e1 and e2 will hash to the same bucket as they produces the same hash code. But since equals() method is not overridden, when the set hashes e2 and iterates through the bucket looking if there is an Employee e such that e2.equals(e) is true, it won’t find any as e2.equals(e1) will be false.

Please note that even though equal objects must have equal hash codes, the reverse is not true. It is perfectly valid to override hashCode() without overriding equals() as objects with equal hash codes need not be equal.

**Association, Composition and Aggregation in Java :**

**Association** is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.

Example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.

* It is a special form of Association where:
* It represents Has-A relationship.
* It is a unidirectional association i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation, both the entries can survive individually which means ending one entity will not effect the other entity

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

* It represents part-of relationship.
* In composition, both the entities are dependent on each other.
* When there is a composition between two entities, the composed object cannot exist without the other entity.

Aggregation vs Composition

* **Dependency:**Aggregation implies a relationship where the child can exist independently of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: Human and heart, heart don’t exist separate to a Human
* **Type of Relationship:** Aggregation relation is “has-a” and composition is “part-of” relation.
* **Type of association:**Composition is a strong Association whereas Aggregation is a weak Association.

**Is finally block always get executed in Java? :** Yes, the finally block is always get executed unless there is an abnormal program termination either resulting from a JVM crash or from a call to System.exit().

* A finally block is always get executed whether the exception has occurred or not.
* If an exception occurs like closing a file or DB connection, then the finally block is used to clean up the code.
* We cannot say the finally block is always executes because sometimes if any statement like System.exit() or some similar code is written into try block then program will automatically terminate and the finally block will not be executed in this case.
* A finally block will not execute due to other conditions like when JVM runs out of memory when our java process is killed forcefully from task manager or console when our machine shuts down due to power failure and deadlock condition in our try block.

try {

System.out.println("I am in try block");

System.exit(1);

} catch(Exception ex){

ex.printStackTrace();

} finally {

System.out.println("I am in finally block");

}

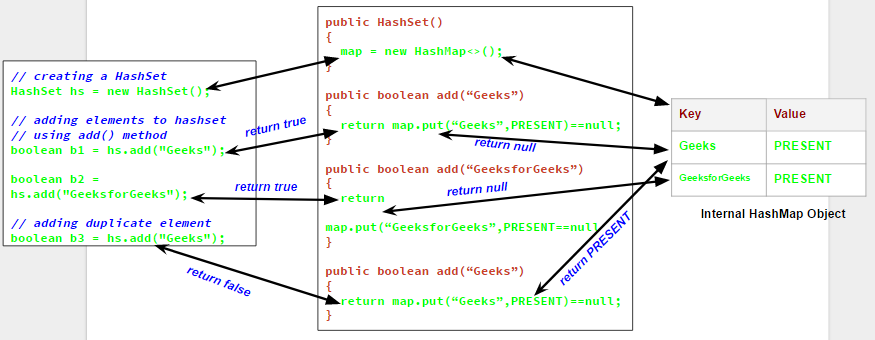
In the above example, the finally block will not execute due to the System.exit(1) condition in the try block.

**LinkedList class in Collection :**

* ***LinkedList***class extends ***AbstractSequentialList***and implements ***List***, ***Deque***and ***Queue*** interface.
* It can be used as ***List***, stack or ***Queue***as it implements all the related interfaces.
* It can contain duplicate elements and is not synchronized.
* maintains insertion order.
* not synchronized.
* No random access.
* manipulation fast because no shifting needs to be occurred.

**How LinkedList work Internally? :** Since a LinkedList acts as a dynamic array and we do not have to specify the size while creating it, the size of the list automatically increases when we dynamically add and remove items. And also, the elements are not stored in a continuous fashion. Therefore, there is no need to increase the size. Internally, the LinkedList is implemented using the [doubly linked list data structure](https://www.geeksforgeeks.org/doubly-linked-list/). The main difference between a normal linked list and a doubly LinkedList is that a doubly linked list contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

**Internal working of Set/HashSet in Java :** As we know that a set is a well-defined collection of distinct objects. Each member of a set is called an element of the set. So in other words, we can say that a set will never contain duplicate elements. But how in java Set interface implemented classes like HashSet, LinkedHashSet, TreeSet etc. achieve this uniqueness. we will discuss the hidden truth behind this uniqueness.



// creating a HashSet

        HashSet hs = new HashSet();

        // adding elements to hashset

        // using add() method

        boolean b1 = hs.add("Geeks");

        boolean b2 = hs.add("GeeksforGeeks");

           // adding duplicate element

        boolean b3 = hs.add("Geeks");

**Output :** b1 = true;b2 = true;b3 = false;[GeeksforGeeks, Geeks]

Now from the output, it is clear that when we try to add a duplicate element to a set using add() method, it returns false, and element is not added to hashset, as it is already present. Now the question comes, how add() method checks whether the set already contains the specified element or not. It will be more clear if we have a closer look on the add() method and default constructor in HashSet class.

// predefined HashSet class

public class **HashSet**{

// A HashMap object

private transient HashMap<e, object="" style="box-sizing: border-box;"> map;

// A Dummy value(PRESENT) to associate with an Object in the Map

private static final Object PRESENT = new Object();

// default constructor of HashSet class

// It creates a HashMap by calling default constructor of HashMap class

**public HashSet() {**

**map = new HashMap<>();**

**}**

// add method it calls put() method on map object and

//then compares it's return value with null

**public boolean add(E e) {**

**return map.put(e, PRESENT)==null;**

**}**

// Other methods in Hash Set

}

Now as you can see that whenever we create a HashSet, it internally creates a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) and if we insert an element into this HashSet using add() method, it actually call put() method on internally created HashMap object with element you have specified as it’s key and constant Object called **“PRESENT”** as it’s value. So we can say that **a Set achieves uniqueness internally through HashMap**.

As we know in a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) each key is unique and when we call put(Key, Value) method, it returns the previous value associated with key, or null if there was no mapping for key. So in add() method we check the return value of map.put(key, value) method with null value.

1. If map.put(key, value) returns null, then the statement “map.put(e, PRESENT) == null” will return true and element is added to the HashSet(internally HashMap).
2. If map.put(key, value) returns old value of the key, then the statement “map.put(e, PRESENT) == null” will return false and element is not added to the HashSet(internally HashMap).

As LinkedHashSet extends HashSet, so it internally calls constructors of HashSet using [super()](https://www.geeksforgeeks.org/super-keyword/). Similarly creating an object of [TreeSet](https://www.geeksforgeeks.org/treeset-class-java-examples/) class internally creates object of [Navigable Map](https://www.geeksforgeeks.org/navigablemap-interface-in-java-with-example/) as backing map.

<https://howtodoinjava.com/java/collections/hashmap/design-good-key-for-hashmap/>