Java Collections – hashCode() and equals()

`hashCode` and **`equals`** are closely related :

- if you override `equals`, you must override `hashCode`.
- `hashCode` must generate equal values for equal objects.
- `equals` and `hashCode` must depend on the `same set of significant fields`. You must use the same set of fields in both of these methods. You are not required to use all fields. For example, a calculated field that depends on others should very likely be omitted from `equals` and `hashCode`.

Implementing **`hashCode`** :

- if a class overrides `equals`, it must override `hashCode`
- when they are both overridden, `equals` and `hashCode` must use the same set of fields
- if two objects are equal, then their `hashCode` values must be equal as well
- if the object is immutable, then `hashCode` is a candidate for caching and lazy initialization

For the Best practice use below steps to implement your equals() method:

- Use this == that to check reference equality
- Use `instanceof` to test for correct argument type
- Cast the argument to the correct type
- Compare significant fields for equality

Collections:

The collections framework was designed to meet several goals.

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) are highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- Extending and/or adapting a collection had to be easy.

The Collection Interfaces:

1 The Collection Interface
This enables you to work with groups of objects; it is at the top of the collections hierarchy.

2 The List Interface
This extends **Collection** and an instance of List stores an ordered collection of elements.

3 The Set
This extends Collection to handle sets, which must contain unique elements

4 The SortedSet
This extends Set to handle sorted sets

5 The Map
This maps unique keys to values.

6 The Map.Entry
This describes an element (a key/value pair) in a map. This is an inner class of Map.

7 The SortedMap
This extends Map so that the keys are maintained in ascending order.

8 The Enumeration
This is legacy interface and defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator.

Collection classes:

Java provides a set of standard collection classes that implement Collection interfaces.

SN	Classes with Description
1	**AbstractCollection** Implements most of the Collection interface.
2	**AbstractList** Extends AbstractCollection and implements most of the List interface.
3	**AbstractSequentialList** Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
4	LinkedList Implements a linked list by extending AbstractSequentialList.
5	ArrayList Implements a dynamic array by extending AbstractList.
6	**AbstractSet** Extends AbstractCollection and implements most of the Set interface.
7	HashSet Extends AbstractSet for use with a hash table.
8	LinkedHashSet Extends HashSet to allow insertion-order iterations.

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed.

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

```
Deque d = new ArrayDeque();
```

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

java.util classes

SN	Classes with Description
1	Vector This implements a dynamic array. It is similar to ArrayList, but with some differences.
2	Stack Stack is a subclass of Vector that implements a standard last-in, first-out stack.
3	Dictionary Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.
4	Hashtable Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.
5	Properties Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.
6	BitSet A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed.

Java Iterator:

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

- **Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.**
- **Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.**
- **Within the loop, obtain each element by calling next( ).**

```
import java.util.*;
public class IteratorDemo {
  public static void main(String args[]) {
    // Create an array list
    ArrayList al = new ArrayList();
// add elements to the array list
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D")
al.add("F");
// Use iterator to display contents of al
System.out.print("Original contents of al: ");
Iterator itr = al.iterator();
while(itr.hasNext()) {
  Object element = itr.next();
  System.out.print(element + " ");
}
System.out.println();
```

```
    // Modify objects being iterated
    ListIterator litr = al.listIterator();
  while(litr.hasNext()) {
   Object element = litr.next();
   litr.set(element + "+");
  }
  System.out.print("Modified contents of al: ");
  itr = al.iterator();
  while(itr.hasNext()) {
   Object element = itr.next();
   System.out.print(element + " ");
  }
  System.out.println();
```

```
    // Now, display the list backwards
    System.out.print("Modified list backwards: ");
    while(litr.hasPrevious()) {
     Object element = litr.previous();
     System.out.print(element + " ");
    }
    System.out.println();
  }
}
```

The equals Method

The equals( ) method, shown here, tests whether an object equals the invoking comparator:

obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false

```
public class EqualsExample{
public static void main(String args[]){
String s1="javatpoint";
String s2="javatpoint";
String s3="JAVATPOINT";
String s4="python";
System.out.println(s1.equals(s2));//true because content and case is same
System.out.println(s1.equals(s3));//false because case is not same
System.out.println(s1.equals(s4));//false because content is not same
}}
```

The compare Method:

obj1 and obj2 are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned.

By overriding compare( ), you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

```
public class CompareToExample{
public static void main(String args[]){
String s1="hello";
String s2="hello";
String s3="meklo";
String s4="hemlo";
String s5="flag";
System.out.println(s1.compareTo(s2));//0 because both are equal
System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "m"
System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than "m"
System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater than "f"
}}
```

Set

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.

The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements.

add( ) - Adds an object to the collection.

clear( ) - Removes all objects from the collection.

contains( ) - Returns true if a specified object is an element within the collection.

isEmpty( ) - Returns true if the collection has no elements.

iterator( ) - Returns an Iterator object for the collection, which may be used to retrieve an object.

remove( ) - Removes a specified object from the collection.

size( ) - Returns the number of elements in the collection

```java
import java.util.*;
public class SetDemo {

  public static void main(String args[]) {
      int count[] = {34, 22,10,60,30,22};
      Set<Integer> set = new HashSet<Integer>();
      try {
         for(int i = 0; i < 5; i++) {
            set.add(count[i]);
         }
         System.out.println(set);

         TreeSet sortedSet = new TreeSet<Integer>(set);
         System.out.println("The sorted list is:");
         System.out.println(sortedSet);

         System.out.println("The First element of the set is: "+
(Integer)sortedSet.first());
         System.out.println("The last element of the set is: "+
(Integer)sortedSet.last());
      }
      catch(Exception e) {}
   }
}
```

Output:

```
[34, 22, 10, 60, 30]
The sorted list is:
[10, 22, 30, 34, 60]
The First element of the set is: 10
The last element of the set is: 60
```

Map:

- Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
- Several methods throw a NoSuchElementException when no items exist in the invoking map.
- A ClassCastException is thrown when an object is incompatible with the elements in a map.
- A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.
- An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map.

```
import java.util.*;
public class CollectionsDemo {

   public static void main(String[] args) {
      Map m1 = new HashMap();
      m1.put("Zara", "8");
      m1.put("Mahnaz", "31");
      m1.put("Ayan", "12");
      m1.put("Daisy", "14");

      System.out.println();
      System.out.println(" Map Elements");
      System.out.print("\t" + m1);
   }
}
```

Output:

```
Map Elements
    {Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}
```

Difference between TreeMap and TreeSet in Java

Interface : TreeMap implements Map interface while TreeSet implements Set interface.
Duplicates : TreeMap allows duplicate values while TreeSet does not allow duplicate objects.
Implementation : TreeMap is a red-black tree based NavigableMap implementation. TreeSet is a NavigableSet implementation based on a TreeMap.
Sorting : TreeMap is sorted based on keys while TreeSet is sorted based on objects.

Similarities between TreeMap and TreeSet in Java

1. Null values : Both TreeMap and TreeSet do not permit null values.
2. Sorting : Both TreeMap and TreeSet are sorted. Sorted order can be natural sorted order defined by Comparable interface or custom sorted order defined by Comparator interface.
3. Performance : Both TreeMap and TreeSet provides guaranteed log(n) time cost for operation like get, put, containsKey and remove. Both internally uses Red-Black tree.
4. Synchronization : Both TreeMap and TreeSet are not synchronized. Hence, they are not used in concurrent applications.
5. Fail-fast Iterator : The iterator returned by the iterator method of the TreeMap and TreeSet are fail-fast. You can find more about fail-fast and fail-safe iterator here.
6. Package : Both classes belong to the java.util package and are part of the Java Collections Framework

Recap : Difference between TreeMap and TreeSet in Java

	TreeMap	TreeSet
Interface	implements Map interface	implements Set interface
Duplicates	Allow duplicates values	No
Implementation	Red-black tree based Navigation Map implementation	NavigableSet implementation based on a TreeMap
Sorting	Based on Keys	Based on Objets