# Hashing :How hash map works in java or How get() method works internal

How Hashmap works in Java

HashMap works on the principle of Hashing . To understand Hashing , we should understand the three terms first i.e Hash Function , Hash Value and Bucket .

What is Hash Function , Hash Value and Bucket ?

hashCode() function which returns an integer value is the Hash function. The important point to note that , this method is present in Object class ( Mother of all class ) .

This is the code for the hash function(also known as hashCode method) in Object Class :

```
public native int hashCode();
```

The most important point to note from the above line : hashCode method return int value .

So the Hash value is the int value returned by the hash function .

What is bucket ?

A bucket is used to store key value pairs . A bucket can have multiple key-value pairs . In hash map, bucket used simple linked list to store objects .

After understanding the terms we are ready to move next step , How hash map works in java or How get() works internally in java .

Code inside Java Api (HashMap class internal implementation) for HashMap get(Obejct key) method

```
Public  V get(Object key)
 {
  if (key ==null)
  //Some code
     int hash = hash(key.hashCode());

  // if key found in hash table then  return value
 //    else return null
 }
```

Hash map works on the principle of hashing

HashMap get(Key k) method calls hashCode method on the key object and applies returned hashValue to its own static hash function to find a bucket location(backing array) where keys and values are stored in form of a nested class called Entry (Map.Entry) . So you have concluded that from the previous line that Both key and value is stored in the bucket as a form of Entry object . So thinking that Only value is stored in the bucket is not correct and will not give a good impression on the interviewer .

* Whenever we call get( Key k ) method on the HashMap object . First it checks that whether key is null or not . Note that there can only be one null key in HashMap .

If key is null , then Null keys always map to hash 0, thus index 0.

If key is not null then , it will call hashfunction on the key object , see line 4 in above method i.e. key.hashCode() ,so after key.hashCode() returns hashValue , line 4 looks like

```
int hash = hash(hashValue)
```

, and now ,it applies returned hashValue into its own hashing function .

We might wonder why we are calculating the hashvalue again using hash(hashValue). Answer is ,It defends against poor quality hash functions.

Now step 4 final hashvalue is used to find the bucket location at which the Entry object is stored . Entry object stores in the bucket like this (hash,key,value,bucketindex) .

Interviewer: What if when two different keys have the same hashcode ?

Solution, equals() method comes to rescue.Here candidate gets puzzled. Since bucket is one and we have two objects with the same hashcode .Candidate usually forgets that bucket is a simple linked list.

The bucket is the linked list effectively . Its not a LinkedList as in a java.util.LinkedList - It's a separate (simpler) implementation just for the map .

So we traverse through linked list , comparing keys in each entries using keys.equals() until it return true. Then the corresponding entry object Value is returned .

One of our readers Jammy asked a very good question

When the functions 'equals' traverses through the linked list does it traverses from start to end one by one...in other words brute method. Or the linked list is sorted based on key and then it traverses?

Answer is when an element is added/retrieved, same procedure follows:

a. Using key.hashCode() [ see above step 4],determine initial hashvalue for the key

b. Pass intial hashvalue as hashValue in hash(hashValue) function, to calculate the final hashvalue.

c. Final hash value is then passed as a first parameter in the indexFor(int ,int )method .

The second parameter is length which is a constant in HashMap Java Api , represented by DEFAULT_INITIAL_CAPACITY

The default value of DEFAULT_INITIAL_CAPACITY is 16 in HashMap Java Api .

indexFor(int,int) method returns the first entry in the appropriate bucket. The linked list in the bucket is then iterated over - (the end is found and the element is added or the key is matched and the value is returned )

Explanation about indexFor(int,int) is below :

```
/** *Returns index for hash code h.
*/
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

The above function indexFor() works because Java HashMaps always have a capacity, i.e. number of buckets, as a power of 2.

Let's work with a capacity of 256,which is 0x100, but it could work with any power of 2. Subtracting 1 from a power of 2 yields the exact bit mask needed to bitwise-and with the hash to get the proper bucket index, of range 0 to length - 1.

256 - 1 = 255

0x100 - 0x1 = 0xFF

E.g. a hash of 257 (0x101) gets bitwise-anded with 0xFF to yield a bucket number of 1.

Interviewer:  What if  when two  keys are same and have the same hashcode ?

If key needs to be inserted and already inserted hashkey's hashcodes are same, and keys are also same(via reference or using equals() method)  then override the previous key value pair with the current key value pair.

The other important point to note is that in Map ,Any class(String etc.) can serve as a key if and only if it overrides the equals() and hashCode() method .

Interviewer:  How will you measure the performance of HashMap?

According to Oracle Java docs,

An instance of HashMap has two parameters that affect its performance: initial capacity and load factor.

The capacity is the number of buckets in the hash table( HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.), and the initial capacity is simply the capacity at the time the hash table is created.

The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

In HashMap class, the default value of load factor is (.75) .

Interviewer : What is the time complexity of Hashmap get() and put() method ?

The hashmap implementation provides constant time performance for (get and put) basic operations

i.e the complexity of get() and put() is O(1) , assuming the hash function disperses the elements properly among the buckets.

# HashMap vs ConcurrentHashMap

1. Thread -Safe :

ConcurrentHashMap is thread-safe that is the code can be accessed by single thread at a time .

while HashMap is not thread-safe .

difference between hashmap and concurrenthashmap in java

2. Synchronization Method :

HashMap can be synchronized by using

synchronizedMap(HashMap) method . By using this

method we get a HashMap object which is equivalent

to the HashTable object . So every modification is performed

on Map is locked on Map object.

```
import java.util.*;
public class HashMapSynchronization {
    public static void main(String[] args) {
        // create map
        Map<String,String> map = new HashMap<String,String>();
         // populate the map
        map.put("1","ALIVE ");
        map.put("2","IS");
        map.put("3","AWESOME");
                // create a synchronized map
        Map<String,String> syncMap = Collections.synchronizedMap(map);
                System.out.println("Synchronized map :"+syncMap);
    }
}
```

ConcurrentHashMap synchronizes or locks on the certain portion of the Map . To optimize the performance of ConcurrentHashMap , Map is divided into different partitions depending upon the Concurrency level . So that we do not need to synchronize the whole Map Object.

3. Null Key

   ConcurrentHashMap does not allow NULL values . So the key can not be null in ConcurrentHashMap .While In HashMap there can only be one null key .

4. Performance

   In multiple threaded environment HashMap is usually faster than ConcurrentHashMap . As only single thread can access the certain portion of the Map and thus reducing the performance .

While in HashMap any number of threads can access the code at the same time .

# How TreeSet Works Internally in Java : TreeSet Interview Questions

What is TreeSet ?

TreeSet is like HashSet which contains the unique elements only but in a sorted manner. The major difference is that TreeSet provides a total ordering of the elements. The elements are ordered using their natural ordering, or by a Comparator typically provided at sorted set creation time. The set's iterator will traverse the set in ascending element order.

How TreeSet works in Java ?

If you look into the TreeSet Api in rt.jar , you will find the following code :

```
 public class TreeSet<E>
        extends AbstractSet<E>
        implements NavigableSet<E>, Cloneable, java.io.Serializable
{
   private transient NavigableMap<E,Object> map;
      // Dummy value to associate with an Object in the backing Map
      private static final Object PRESENT = new Object();
      public TreeSet() {
      this(new TreeMap<E,Object>());
   }
      // SOME CODE ,i.e Other methods in TreeSet
      public boolean add(E e) {
      return map.put(e, PRESENT)==null;
   }
   // SOME CODE ,i.e Other methods in TreeSet
}
```

According to TreeSet Oracle doc :

TreeSet is a NavigableSet implementation backed by TreeMap instance.

Hence , whenever you are adding element to the TreeSet object , it works just like HashSet , The only difference is that instead of HashMap here we have TreeMap object in the constructor.

As we know in TreeMap each key is unique as it internally uses HashMap . So what we do in the TreeSet is that we pass the argument in the add(Elemene E) that is E as a key in the TreeSet . Now we need to associate some value to the key , so what Java apis developer did is to pass the Dummy value that is ( new Object () ) which is referred by Object reference PRESENT .

So , actually when you are adding a line in TreeSet like treeset.add(3) what java does internally is that it will put that element E here 3 as a key in the TreeMap(created during TreeSet object creation) and some dummy value that is Object's object is passed as a value to the key .

Now if you see the code of the TreeMap put(Key k,Value V) method , you will find something like this

```
 public V put(K key, V value) {
//Some code
}
```

The main point to notice in above code is that put (key,value) will return

1. null , if key is unique and added to the map
2. Old Value of the key , if key is duplicate

So , in TreeSet add() method , we check the return value of map.put(key,value) method with null value

i.e.

```
 public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }
```

So , if map.put(key,value) returns null ,then

map.put(e, PRESENT)==null will return true and element is added to the TreeSet.

So , if map.put(key,value) returns old value of the key ,then

map.put(e, PRESENT)==null will return false and element is not added to the TreeSet .

How to find the index of  any element in the TreeSet ?

There are many ways to find out the index of element in the TreeSet. Below is the one liner :

set.headSet(element).size()

Note  :  headSet(element) method returns the sub TreeSet(portion of TreeSet) whose values are less than input element. Then we are calling size() method on the sub TreeSet , which returns the index of the element as sub TreeSet is already sorted.

Why and when we use TreeSet ?

We prefer TreeSet in order  to maintain the unique elements  in the sorted order .

What is the runtime performance of the add() method of the TreeSet and HashSet , where n represents the number of elements?

According to TreeSet Oracle doc :

TreeSet implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains ) method.

According to HashSet Oracle doc :

HashSet provides constant time performance O(1) for basic operations  (add, remove and contains) method assuming the hash  function disperses the elements properly among the buckets.

One-liner :  TreeSet : O(log(n))  HashSet : O(1)

What is natural ordering in TreeSet ?

"Natural" ordering is the ordering implied by the implementation of Comparable interface by the objects in the TreeSet . Essentially RBTree must be able to tell which object is smaller than other object , and there are two  ways to supply that logic to the RB Tree implementation :

1. We need to implement the Comparable interface in the class(es) used as objects in TreeSet.

2. Supply an implementation of the Comparator would do comparing outside the class itself.

Why do we need TreeSet when we already had SortedSet ?

sortedSet is an interface while TreeSet is the class implementing it. As we know, in java, we can not create the objects of the interface. The class implementing the interface must fulfill the contract of interface, i.e , concrete class must implement all the methods present in the interface. TreeSet is such an implementation.

Which data structure you will prefer  in your code : HashSet and TreeSet ?

TreeSet contains the elements in the sorted order while HashSet is faster. Thus , deciding which one to choose depends upon the conditions :

If you want to maintain the order of the elements then TreeSet should be used because the result is alphabetically sorted.

If you do not want to sort the elements and  avoid duplicate elements . Your task involves mainly insert and retrieve operations then prefer HashSet.While iterating HashSet there is no ordering of elements while TreeSet iterates in the natural order.

What happens if the TreeSet is concurrently modified while iterating the elements ?

The iterator's returned by the TreeSet class iterator method are fail-fast.  fail-fast means if the set is modified at any time after the iterator is created , in any way except the iterator's own remove method  , the iterator will throw a ConcurrentModificationException. Thus , in the face of concurrent modification , the iterator fails quickly and cleanly . You can find it here difference between fail-fast and fail-safe iterator in java with example.

Which copy technique (deep or shallow ) is used by TreeSet clone() method ?

According to Oracle docs , clone() method returns the shallow copy of the TreeSet instance. In shallow copy , Both objects A and B shared the same memory location .

How to convert HashSet to TreeSet object ?

One-liner :  Set  treeObject = new TreeSet( hashSetObject);

What is the difference between HashSet and TreeSet in Java ?

Write an example of TreeSet ?

```
import java.util.TreeSet;

public class TreeSetExample {

    public static void main(String[] args) {

    TreeSet<String> treesetobj = new TreeSet<String>();

    treesetobj.add("Alive is awesome");

    treesetobj.add("Love yourself");

    System.out.println("TreeSet object output :"+ treesetobj);    }  } Output :
```

TreeSet object output :[Alive is awesome, Love yourself]

# Difference between Iterator and Enumeration with example : Java Collections Question

What is the difference between iterator and enumeration.

Iterator

Iterator is the interface and found in the java.util package.

It has three methods

*hasNext()

*next()

*remove()

Enumeration

Enumeration is also an interface and found in the java.util package .

An enumeration is an object that generates elements one at a time. It is used for passing through a collection, usually of unknown size.

The traversing of elements can only be done once per creation.

It has following methods

*hasMoreElements()

*nextElement()

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework.

Iterators differ from enumerations in two ways:

Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

Method names have been improved.

```
import java.util.*;
public class Performance {
   public static void main(String[] args){
      Vector v=new Vector();
      Object element;
      Enumeration enum;
      Iterator iter;
      long start;
        for(int i=0; i<1000000; i++){
         v.add("New Element");
      }
      enum=v.elements();
      iter=v.iterator();
      //*****CODE BLOCK FOR ITERATOR**********************
      start=System.currentTimeMillis();
      while(iter.hasNext()){
         element=iter.next();
      }
      System.out.println("Iterator took " + (System.currentTimeMillis()-start));
      //**************END OF ITERATOR BLOCK*************************
            System.gc();   //request to GC to free up some memory
      //**************CODE BLOCK FOR ENUMERATION********************
      start=System.currentTimeMillis();
      while(enum.hasMoreElements()){
         element=enum.nextElement();
      }
      System.out.println("Enumeration took " + (System.currentTimeMillis()-start));
```

```
        //************END OF ENUMERATION BLOCK**********************
    }
}
```

*Why not use for(int i=0; i< v.size();i++){}?

For loops are expensive to the processor when the collection reaches large sizes, as many operations are done just to compute the first line:

int i = 0 is an assignment and creation (2 operations)

i get size, check value of i, and compare (3 operations)

i++ gets i then adds 1 to it [++i is only 2 operations] this one (3 operations)

*7/8 operations in total, each time the loop runs through

where an enumeration or iterator uses a while(){}

while(v.hasNext()) has next true or false (1 operation)

while(v.hasMoreElements()) has more true or false (1 operation)

*Only one operation per repeat of this loop

# Difference between HashSet and TreeSet : Java Collections Interview Question

HashSet and TreeSet are the leaves of the same branch , in java words they both implements the Set interface. The difference between HashSet and TreeSet is a popular interview question in java although it is not as popular as Arraylist vs Vector or Comparable vs Comparator but still can not be missed.

In this article we will see difference between HashSet and TreeSet ,as well as similarities and their examples.

Read Also : Internal working of HashSet or How HashSet works in java

Difference between HashSet and TreeSet

1. Ordering : HashSet stores the object in random order . There is no guarantee that the element we inserted first in the HashSet will be printed first in the output . For example

```
import java.util.HashSet;

public class HashSetExample {

public static void main(String[] args) {

    HashSet<String>  obj1= new HashSet<String>();

        obj1.add("Alive");

        obj1.add("is");

        obj1.add("Awesome");

        System.out.println(obj1);

    }

}
```

OUTPUT : [is, Awesome, Alive]

Elements are sorted according to the natural ordering of its elements in TreeSet. If the objects can not be sorted in natural order than use compareTo() method to sort the elements of TreeSet object .

```
import java.util.TreeSet;

public class TreeSetExample {

public static void main(String[] args) {

      TreeSet<String>  obj1= new TreeSet<String>();

      obj1.add("Alive");

      obj1.add("is");

      obj1.add("Awesome");

      System.out.println(obj1);

         }

   }
```

OUTPUT : [Alive, Awesome, is]

2. Null value :  HashSet can store null object while TreeSet does not allow null object. If one try to store null object in TreeSet object , it will throw Null Pointer Exception.

3. Performance : HashSet take constant time performance for the basic operations like add, remove contains and  size.While TreeSet guarantees log(n) time cost for the basic operations (add,remove,contains).

4. Speed : HashSet is much faster than TreeSet,as performance time of HashSet is constant against the log time of TreeSet for most operations (add,remove ,contains and size) . Iteration performance of HashSet mainly depends on the load factor and initial capacity parameters.

5. Internal implementation :  As we have already discussed How hashset internally works in java thus, in one line HashSet are internally backed by hashmap. While TreeSet is backed by a Navigable  TreeMap.

Difference between HashSet and TreeSet in Java with Example

6. Functionality :  TreeSet is rich in functionality as compare to HashSet. Functions like pollFirst(),pollLast(),first(),last(),ceiling(),lower() etc. makes TreeSet easier to use than HashSet.

7. Comparision : HashSet uses equals() method for comparison in java while TreeSet uses compareTo() method for maintaining ordering .

To whom priority is given TreeSet comparator or Comparable.compareTo() .

Suppose there are elements in TreeSet which can be naturally sorted by the TreeSet , but we also added our own sorting method by implementing Comparable interface compareTo() method .

Then to whom priority is given.

Answer to the above question is that the Comparator passed into the TreeSet constructor has been given priority.

According to Oracle Java docs

public TreeSet(Comparator comparator)

Constructs a new, empty tree set, sorted according to the specified comparator.

Parameters:

comparator - the comparator that will be used to order this set. If null, the natural ordering of the elements will be used.

Similarities Between HashSet and TreeSet

1. Unique Elements : Since HashSet and TreeSet both implements Set interface . Both are allowed to store only unique elements in their objects. Thus there can never be any duplicate elements inside the HashSet and TreeSet objects.

2. Not Thread Safe : HashSet and TreeSet both are not synchronized or not thread safe.HashSet and TreeSet, both implementations are not synchronized. If multiple threads access a hash set/ tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.

3. Clone() method copy technique: Both HashSet and TreeSet uses shallow copy technique to create a clone of their objects .

4. Fail-fast Iterators : The iterators returned by this class's method are fail-fast: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

When to prefer TreeSet over HashSet

1. Sorted unique elements are required instead of unique elements.The sorted list given by TreeSet is always in ascending order.

2. TreeSet has greater locality than HashSet.

If two entries are near by in the order , then TreeSet places them near each other in data structure and hence in memory, while HashSet spreads the entries all over memory regardless of the keys they are associated to.

As we know Data reads from the hard drive takes much more latency time than data read from the cache or memory. In case data needs to be read from hard drive than prefer TreeSet as it has greater locality than HashSet.

3. TreeSet uses Red- Black tree algorithm underneath to sort out the elements. When one need to perform read/write operations frequently , then TreeSet is a good choice.

Thats it for the difference between HashSet and TreeSet , if you have any doubts then please mention in the comments.

# How TreeMap works in java : 10 TreeMap Java Interview Questions

What is a Tree Map ?

Treemap class is like HashMap which stores key- value pairs . The major difference is that Treemap  sorts

the key in ascending order.

According to Java doc  :

Treemap is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

How TreeMap works in java ?

TreeMap is a Red-Black tree based NavigableMap implementation.In other words , it sorts the TreeMap object keys using Red-Black tree algorithm.

So we learned that TreeMap uses Red Black tree algorithm internally to sort the elements.

Red Black algorithm is a complex algorithm . We should read the pseudo code of Red Black algorithm in order to understand the internal implementation .

Red Black tree has the following properties :

1. As the name of the algorithm suggests ,color of every node in the tree is either red or black.

2. Root node must be Black in color.

3. Red node can not have a red color neighbor node.

4. All paths from root node to the null should consist the same number of black nodes .

Rotation in Red Black Tree :

how treemap works in java

Rotations maintains the inorder ordering of the keys(x,y,z).

A rotation can be maintained in O(1) time.

You can find more about the red black tree algorithm here

Interviewer : Why and when we use TreeMap ?

We need TreeMap  to get the sorted list of keys in ascending order.

Interviewer : What is the runtime performance of the get() method in TreeMap and HashMap  ,where n represents the number of elements ?

According to TreeMap Java doc, TreeMap implementation provides guaranteed log(n) time cost for the containsKey,get,put and remove operations.

According to HashMap Java doc :

HashMap implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets.

One liner : TreeMap : log(n)  HashMap : Constant time performance assuming elements disperses properly

Interviewer : What is "natural ordering" in TreeMap ?

"Natural" ordering is the ordering implied by the implementation of the Comparable interface by the objects used as keys in the TreeMap. Essentially, RBTree must be able to tell which key is smaller than the other key, and there are two ways to supply that logic to the RBTree implementation:

1.Implement Comparable interface in the class(es) used as keys to TreeMap, or

2.Supply an implementation of the Comparator that would do comparing outside the key class itself.

Natural ordering is the order provided by the Comparable interface .If somebody puts the key that do not implement natural order then it will throw ClassCastException.

Interviewer : Why do we need TreeMap when we have sortedMap ?

sortedMap is a interface and TreeMap is the class implementing it .As we know one can not create objects of the interface . Interface tells us which methods a sortedMap implementation should provide .TreeMap is such an implementation.

Interviewer : Which data structure you will prefer in your code : HashMap or TreeMap ?

HashMap is faster while  TreeMap is sorted .Thus we choose them according to their advantage.

If you do not want to sort the elements but just to insert and retrieve the elements then use HashMap .

But if you want to maintain the  order of the elements then TreeMap should be preferred because the result is alphabetically sorted .While iterating HashMap there is no ordering of the elements ,on the other hand , TreeMap iterates in the natural key order.

Interviewer : What happens if the TreeMap is concurrently modified while iterating the elements ?

The iterator fails fast and quickly if structurally modified at any time after the iterator is created (in any way except through the iterator's own remove method ). We already discussed the difference between Fail-fast and Fail safe iterators .

Interviewer : Which copy technique (deep or shallow ) is used by the TreeMap clone() method ?

According to docjar , clone() method returns the shallow copy of the TreeMap instance . In shallow copy object B points to object A location in memory . In other words , both object A and B are sharing the same elements .The keys and values  themselves are not cloned .

Interviewer : Why  java's  treemap does not allow an initial size ?

HashMap reallocates its internals as the new one gets inserted while TreeMap does not reallocate nodes on adding new ones. Thus , the size of the TreeMap  dynamically increases if needed , without shuffling the internals. So it is meaningless to set the initial size of the TreeMap .

# Internal implementation of Set/HashSet (How Set Ensures Uniqueness) :

Set Implementation Internally in Java

Each and every element in the set is unique . So that there is no duplicate element in set .

So in java if we want to add elements in the set then we write code like this

```
public class JavaHungry {
     public static void main(String[] args)
   {          // TODO Auto-generated method stub
      HashSet<Object> hashset = new HashSet<Object>();
      hashset.add(3);
      hashset.add("Java Hungry");
      hashset.add("Blogspot");
      System.out.println("Set is "+hashset);
   }
}
```

It will print the result : Set is [3, Java Hungry, Blogspot]

Now let add duplicate element in the above code

```
public class JavaHungry {
      public static void main(String[] args)
   {   HashSet<Object> hashset = new HashSet<Object>();
      hashset.add(3);
      hashset.add("Java Hungry");
      hashset.add("Blogspot");
      hashset.add(3);                  // duplicate elements
      hashset.add("Java Hungry");            // duplicate elements
      System.out.println("Set is "+hashset);
   } }
```

It will print the result : Set is [3, Java Hungry, Blogspot]

Now , what happens internally when you pass duplicate elements in the add() method of the Set object , It will return false and do not add to the HashSet , as the element is already present .So far so good .

But the main problem arises that how it returns false . So here is the answer

When you open the HashSet implementation of the add() method in Java Apis that is rt.jar , you will find the following code in it

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, java.io.Serializable
{
    private transient HashMap<E,Object> map;
    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();
     public HashSet() {
        map = new HashMap<>();
    }
        // SOME CODE ,i.e Other methods in Hash Set
     public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }
    // SOME CODE ,i.e Other methods in Hash Set
}
```

So , we are achieving uniqueness in Set,internally in java through HashMap . Whenever you create an object of HashSet it will create an object of HashMap as you can see in the italic lines in the above code .

We already discussed How HashMap works internally in java .

As we know in HashMap each key is unique . So what we do in the set is that we pass the argument in the add(Elemene E) that is E as a key in the HashMap . Now we need to associate some value to the key , so what Java apis developer did is to pass the Dummy value that is ( new Object () ) which is referred by Object reference PRESENT .

So , actually when you are adding a line in HashSet like hashset.add(3) what java does internally is that it will put that element E here 3 as a key in the HashMap(created during HashSet object creation) and some dummy value that is Object's object is passed as a value to the key .

Now if you see the code of the HashMap put(Key k,Value V) method , you will find something like this

```
 public V put(K key, V value) {
//Some code
}
```

The main point to notice in above code is that put (key,value) will return

1. null , if key is unique and added to the map
2. Old Value of the key , if key is duplicate

So , in HashSet add() method , we check the return value of map.put(key,value) method with null value

i.e.

```
   public boolean add(E e) {
          return map.put(e, PRESENT)==null;
      }
```

So , if map.put(key,value) returns null ,then

map.put(e, PRESENT)==null will return true and element is added to the HashSet.

So , if map.put(key,value) returns old value of the key ,then

map.put(e, PRESENT)==null will return false and element is not added to the HashSet .

# Difference between Comparable and Comparator Interface along with Example In Java : Collection

Difference between Comparable and Comparator with example is probably one of the most important question on java collections topic ,which are mostly used in Java as the sorting tools for the Collection classes such as the Arraylist ,Hashset ,etc.

First we need to understand what are comparable and comparator interfaces .

Read Also : Difference between ArrayList and Vector

What is Comparable Interface ? Where we use it ?

Comparable Interface : Comparable is an public interfaces which is used to impose an natural ordering (if numbers then 1,2,3 or in alphabetical order 'a','b','c' ) of the class that implements it.

Now here the total ordering defines as the natural ordering which means in JVM that when we compare two objects using the comparable interfaces they are actually compared through their ASCII values which is the natural ordering. This means that the comparable by default uses the sorting technique of JVM i.e. Of sorting by the ASCII values.Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort).

What is Comparator Interface ? Where we use it ?

Comparator Interface : A comparison function, which is used to impose ordering on some collection of objects. To allow precisely control over the sort order , Comparators can be passed to a sort method (e.g Collections.sort()). Certain type of data structures such as TreeSet or TreeMap can also be sorted using Comparator.

Read Also : How Hash Map works in Java

Differences between the Comparator and the Comparable interfaces :

1. Sort sequence : In comparable ,Only one sort sequence can be created while in comparator many sort sequences can be created .

2. Methods Used : Comparator interface in Java has method public int compare (Object o1, Object o2) which returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. While Comparable interface has method public int compareTo(Object o) which returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

3.  Objects needed for Comparision : If you see then logical difference between these two is Comparator in Java compare two objects provided to it , while Comparable interface compares "this" reference with the object specified. So only one object is provided which is then compared to "this" reference.

4 Modify Classes :One has to modify the class whose instances you want to sort while in comparator one build a class separate from the class whose instances one want to sort .

5. Package : Comparator in Java is defined in java.util package while Comparable interface in Java is defined in java.lang package, which very much says that Comparator should be used as an utility to sort objects which Comparable should be provided by default.

Points to Remember regarding Comparable and Comparator :

1. Comparable in Java is used to implement natural ordering of object. In Java API String, Date and wrapper classes implements Comparable interface.Its always good practice to override compareTo() for value objects.

2. If any class implement Comparable interface in Java then collection of that object either list or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and object will be sorted based on there natural order defined by CompareTo method.

3. Objects which implement Comparable in Java can be used as keys in a SortedMap like treemap or elements in a SortedSet for example TreeSet, without specifying any Comparator.

Read Also :  Difference between Iterator and Enumeration with example

Before Moving onto the examples of Comparable and Comparator we need to learn more about the few important functions

Functions with the Comparable Interface :

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is less than, equal to, or greater than the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The String method compareTo( ) serves this purpose. It has this general form:

1) public int compareTo(Object o) :

Compares this object with the specified object for order. Returns a negative integer,  zero, or a positive integer as this object is less than, equal to, or greater than the  specified object .

Here , str is the String being compared with the invoking String. The result of the comparison is returned and is interpreted as shown here:

| Value | Meaning |
| --- | --- |
| Less than zero | The invoking string is less . |
| Greater than zero | The invoking string is greater . |
| Zero | The two strings are equal. |

Functions with the Comparator Interface :

1) public int compare (Object o1,Object o2)

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

2) public boolean equals(Object obj)

Indicates whether some other object is "equal to" this Comparator. This method must obey the general contract of Object.equals(Object). Additionally, this method can return true only if the specified Object is also a comparator and it imposes the same ordering as this comparator. Thus, comp1.equals(comp2) implies that sgn(comp1.compare(o1, o2))==sgn(comp2.compare(o1, o2)) for every object reference o1 and o2 .

Example of Comparable Interface :

Comparable interface:

Class whose objects to be sorted must implement this interface.In this,we have to implement compareTo(Object) method.

For example:

Country.java

```
//If this.countryId < country.countryId:then compare method will return -1
//If this.countryId > country.countryId:then compare method will return 1
//If this.countryId==country.countryId:then compare method will return 0
public class Country implements Comparable<Country>{
    int countryId;
    String countryName;
      public Country(int countryId, String countryName) {
        super();
        this.countryId = countryId;
        this.countryName = countryName;
    }
   @Override
    public int compareTo(Country country) {
  return (this.countryId < country.countryId ) ? -1: (this.countryId > country.countryId ) ? 1:0 ;
    }
      public int getCountryId() {
        return countryId;
    }
       public void setCountryId(int countryId) {
        this.countryId = countryId;
    }
        public String getCountryName() {
        return countryName;
    }
        public void setCountryName(String countryName) {
        this.countryName = countryName;       }      }
```

ComparableMain.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class ComparableMain {
	public static void main(String[] args) {
	Country indiaCountry=new Country(1, "India");
	Country chinaCountry=new Country(4, "USA");
	Country nepalCountry=new Country(3, "Russia");
	Country bhutanCountry=new Country(2, "Japan");
		List<Country> listOfCountries = new ArrayList<Country>();
	listOfCountries.add(indiaCountry);
	listOfCountries.add(usaCountry);
	listOfCountries.add(russiaCountry);
	listOfCountries.add(japanCountry);
	System.out.println("Before Sort  : ");
	for (int i = 0; i < listOfCountries.size(); i++) {
	   Country country=(Country) listOfCountries.get(i);
 System.out.println("Country Id: "+country.getCountryId()+"||"+"Country
name:"+country.getCountryName());
	}
	Collections.sort(listOfCountries);
		System.out.println("After Sort  : ");
	for (int i = 0; i < listOfCountries.size(); i++) {
	   Country country=(Country) listOfCountries.get(i);
 System.out.println("Country Id: "+country.getCountryId()+"|| "+"Country name:
"+country.getCountryName());
	}	}	}	OUTPUT :  India,Japan,Russia,USA
```

Example of Comparator Interface:

We will create class country having attribute id and name and will create another class CountrySortByIdComparator which will implement Comparator interface and implement compare method to sort collection of country object by id and we will also see how to use anonymous comparator.

Country.java

```
public class Country{
   int countryId;
   String countryName;
   public Country(int countryId, String countryName) {
      super();
      this.countryId = countryId;
      this.countryName = countryName;
   }
   public int getCountryId() {
      return countryId;
   }
   public void setCountryId(int countryId) {
      this.countryId = countryId;
   }
   public String getCountryName() {
      return countryName;
   }
   public void setCountryName(String countryName) {
      this.countryName = countryName;
   }
   }
```

CountrySortbyIdComparator.java

```
import java.util.Comparator;
//If country1.getCountryId() < country2.getCountryId():then compare method will return -1
//If country1.getCountryId() > country2.getCountryId():then compare method will return 1
//If country1.getCountryId()==country2.getCountryId():then compare method will return 0
public class CountrySortByIdComparator implements Comparator<Country>{
    @Override
   public int compare(Country country1, Country country2) {
       return (country1.getCountryId() < country2.getCountryId() ) ? -1:
(country1.getCountryId() > country2.getCountryId() ) ? 1:0 ;
   }
   }
```

ComparatorMain.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
public class ComparatorMain {
      public static void main(String[] args) {
       Country indiaCountry=new Country(1, "India");
       Country chinaCountry=new Country(3, "USA");
       Country nepalCountry=new Country(4, "Russia");
       Country bhutanCountry=new Country(2, "Japan");
      List<Country> listOfCountries = new ArrayList<Country>();
       listOfCountries.add(indiaCountry);
       listOfCountries.add(usaCountry);
       listOfCountries.add(russiaCountry);
       listOfCountries.add(japanCountry);
```

```
    System.out.println("Before Sort by id : ");
      for (int i = 0; i < listOfCountries.size(); i++) {
         Country country=(Country) listOfCountries.get(i);
         System.out.println("Country Id: "+country.getCountryId()+"||"+"Country name: "+country.getCountryName());
      }
      Collections.sort(listOfCountries,new CountrySortByIdComparator());
       System.out.println("After Sort by id: ");
      for (int i = 0; i < listOfCountries.size(); i++) {
         Country country=(Country) listOfCountries.get(i);
         System.out.println("Country Id: "+country.getCountryId()+"|| "+"Country name: "+country.getCountryName());
      }
            //Sort by countryName
      Collections.sort(listOfCountries,new Comparator<Country>() {
         @Override
         public int compare(Country o1, Country o2) {
            return o1.getCountryName().compareTo(o2.getCountryName());
         }
      });
            System.out.println("After Sort by name: ");
      for (int i = 0; i < listOfCountries.size(); i++) {
         Country country=(Country) listOfCountries.get(i);
         System.out.println("Country Id: "+country.getCountryId()+"|| "+"Country name: "+country.getCountryName());
      }     }       }
```

OUTPUT :  India,Japan,USA,Russia    // sort according to id

Functional Java 8 Use of Comparator

difference between comparable and comparator

Situations when to use Comparable and Comparator

1. If there is a natural or default way of sorting Object already exist during development of Class than use Comparable. This is intuitive and you given the class name people should be able to guess it correctly like Strings are sorted chronically, Employee can be sorted by there Id etc. On the other hand if an Object can be sorted on multiple ways and client is specifying on which parameter sorting should take place than use Comparator interface. for example Employee can again be sorted on name, salary or department and clients needs an API to do that. Comparator implementation can sort out this problem.

2. Some time you write code to sort object of a class for which you are not the original author, or you don't have access to code. In these cases you can not implement Comparable and Comparator is only way to sort those objects.

3. Beware with the fact that How those object will behave if stored in SorteSet or SortedMap like TreeSet and TreeMap If an object doesn't implement Comparable than while putting them into SortedMap, always provided corresponding Comparator which can provide sorting logic.

4. Order of comparison is very important while implementing Comparable or Comparator interface. for example if you are sorting object based upon name than you can compare first name or last name on any order, so decide it judiciously. I have shared more detailed tips on compareTo on my post how to implement CompareTo in Java.

5. Comparator has a distinct advantage of being self descriptive for example if you are writing Comparator to compare two Employees based upon there salary than name that comparator as SalaryComparator, on the other hand compareTo()

Note :

So in Summary if you want to sort objects based on natural order then use Comparable in Java and if you want to sort on some other attribute of object then use Comparator in Java.

# Difference Between String , StringBuilder and StringBuffer Classes with Example : Java

Today we are going to understand the difference between String , StringBuilder and StringBuffer . As you will find that there are minor differences between the above mentioned classes.

String

String is immutable ( once created can not be changed )object . The object created as a String is stored in the Constant String Pool .

Every immutable object in Java is thread safe ,that implies String is also thread safe . String can not be used by two threads simultaneously.

String once assigned can not be changed.

String demo = " hello " ;

// The above object is stored in constant string pool and its value can not be modified.

demo="Bye" ; //new "Bye" string is created in constant pool and referenced by the demo variable

// "hello" string still exists in string constant pool and its value is not overrided but we lost reference to the "hello"string

StringBuffer

StringBuffer is mutable means one can change the value of the object . The object created through StringBuffer is stored in the heap . StringBuffer has the same methods as the StringBuilder , but each method in StringBuffer is synchronized that is StringBuffer is thread safe .

Due to this it does not allow two threads to simultaneously access the same method . Each method can be accessed by one thread at a time .

But being thread safe has disadvantages too as the performance of the StringBuffer hits due to thread safe property . Thus StringBuilder is faster than the StringBuffer when calling the same methods of each class.

StringBuffer value can be changed , it means it can be assigned to the new value . Nowadays its a most common interview question ,the differences between the above classes .

String Buffer can be converted to the string by using toString() method.

StringBuffer demo1 = new StringBuffer("Hello") ;

// The above object stored in heap and its value can be changed .

demo1=new StringBuffer("Bye");

// Above statement is right as it modifies the value which is allowed in the StringBuffer

StringBuilder

StringBuilder is same as the StringBuffer , that is it stores the object in heap and it can also be modified . The main difference between the StringBuffer and StringBuilder is that StringBuilder is also not thread safe.

StringBuilder is fast as it is not thread safe .

StringBuilder demo2= new StringBuilder("Hello");

// The above object too is stored in the heap and its value can be modified

demo2=new StringBuilder("Bye");

// Above statement is right as it modifies the value which is allowed in the StringBuilder

| | String | StringBuffer | StringBuilder |
|---|---|---|---|
| Storage Area | Constant String Pool | Heap | Heap |
| Modifiable | No (immutable) | Yes( mutable ) | Yes( mutable ) |
| Thread Safe | Yes | Yes | No |
| Performance | Fast | Very slow | Fast |

# Difference between Arraylist and Vector : Core Java Interview Collection Question

1. Synchronization and Thread-Safe

Vector is synchronized while ArrayList is not synchronized . Synchronization and thread safe means at a time only one thread can access the code .In Vector class all the methods are synchronized .Thats why the Vector object is already synchronized when it is created .

2. Performance

Vector is slow as it is thread safe . In comparison ArrayList is fast as it is non synchronized . Thus in ArrayList two or more threads can access the code at the same time , while Vector is limited to one thread at a time.

3. Automatic Increase in Capacity

A Vector defaults to doubling size of its array . While when you insert an element into the ArrayList , it increases its Array size by 50% .

By default ArrayList size is 10 . It checks whether it reaches the last element then it will create the new array ,copy the new data of last array to new array ,then old array is garbage collected by the Java Virtual Machine (JVM) .

4. Set Increment Size

ArrayList does not define the increment size . Vector defines the increment size .

You can find the following method in Vector Class

public synchronized void setSize(int i) { //some code }

There is no setSize() method or any other method in ArrayList which can manually set the increment size.

5. Enumerator

Other than Hashtable ,Vector is the only other class which uses both Enumeration and Iterator .While ArrayList can only use Iterator for traversing an ArrayList .

6. Introduction in Java

java.util.Vector class was there in java since the very first version of the java development kit (jdk).

java.util.ArrayList was introduced in java version 1.2 , as part of Java Collections framework . In java version 1.2 , Vector class has been refactored to implement the List Inteface .