**Quick Reference to**

**Java Collections**

Index

[1. Java Collection Diagram 4](#_Toc468889361)

[2. How to create an ArrayList and what happened internally? 5](#_Toc468889362)

[3. Which one is correct? 6](#_Toc468889363)

[4. Is ArrayList al = new ArrayList(); is valid statement? 6](#_Toc468889364)

[5. Explain ArrayList size increment mechanism. 6](#_Toc468889365)

[6. Which copy technique internally used by the ArrayList class clone() method? 6](#_Toc468889366)

[7. How to create ArrayList from Array (Object[]) ? 7](#_Toc468889367)

[8. How to convert List to Array? 7](#_Toc468889368)

[9. What happens if ArrayList is concurrently modified while iterating the elements? 7](#_Toc468889369)

[10. Can we add element in an ArrayList at any arbitrary index (e.g. directly adding an element at 2nd place in place of starting with 0?) 7](#_Toc468889370)

[11. What is the difference between Iterator and ListIterator? 7](#_Toc468889371)

[12. What is the difference between Iterator and Enumeration? 8](#_Toc468889372)

[13. What is the difference between ArrayList and Vector? 9](#_Toc468889373)

[14. What is the difference between ArrayList and LinkedList? 9](#_Toc468889374)

[15. Are there any differences between different methods in each of the following groups of element retrieve operations in LinkedList? 10](#_Toc468889375)

[16. What is the difference between Collection and Collections? 10](#_Toc468889376)

[17. Hash Table - Data Structure and Algorithm (To understand Hash Table Data Structure concepts only) 10](#_Toc468889377)

[18. What is meant by number of buckets in the HashMap? 12](#_Toc468889378)

[19. Get size/length of a value in a HashMap 13](#_Toc468889379)

[20. HashMap Method Summary 13](#_Toc468889380)

[21. How HashMap works in Java? 14](#_Toc468889381)

[Have you used HashMap before or  What is HashMap? Why do you use it? 15](#_Toc468889382)

[Do you Know how HashMap works in Java or How does get () method of HashMap works in Java? 15](#_Toc468889383)

[What will happen if two different objects have the same hashcode? 16](#_Toc468889384)

[How will you retrieve Value object  if two Keys will have the same hashcode? 16](#_Toc468889385)

[Some more Hashtable and HashMap Questions 18](#_Toc468889386)

[1) Why String, Integer and other wrapper classes are considered good keys? 18](#_Toc468889387)

[2) Can we use any custom object as a key in HashMap? 19](#_Toc468889388)

[3) Can we use ConcurrentHashMap in place of Hashtable? 19](#_Toc468889389)

[4) Difference between HashMap and Collections.synchronizedMap(HashMap) 19](#_Toc468889390)

[5) Difference between ConcurrentHashMap and Collections.synchronizedMap( HashMap ) 20](#_Toc468889391)

[6) Difference between HashMap and HashTable 20](#_Toc468889392)

[7) Difference between HashTable and Collections.synchronized(HashMap) 20](#_Toc468889393)

[How HashMap works in Java 21](#_Toc468889394)

[What will happen if two different HashMap  key objects have the same hashcode? 21](#_Toc468889395)

[How null key is handled in HashMap? Since equals() and hashCode() are used to store and retrieve values, how does it work in case of the null key? 21](#_Toc468889396)

[HashMap Changes in JDK 1.7 and JDK 1.8 22](#_Toc468889397)

[22. What is the significance of load factor in HashMap? 22](#_Toc468889398)

[23. Aggregating values in hashmap by keys 23](#_Toc468889399)

[24. What is the difference between HashMap and Hashtable? 24](#_Toc468889400)

[25. What is hash-collision in Hashtable and how it is handled in Java? 24](#_Toc468889401)

[26. On what bases TreeMap and TreeSet is sorted? 24](#_Toc468889402)

[27. What is difference between ordered and sorted collection? 25](#_Toc468889403)

[28. What is the difference between List and Set? 25](#_Toc468889404)

[29. HashSet vs. TreeSet vs. LinkedHashSet 25](#_Toc468889405)

[30. Are HashSet, TreeSet , & LinkedHashSet allow null key? 26](#_Toc468889406)

[31. Are HashSet, TreeSet , & LinkedHashSet synchronized? 27](#_Toc468889407)

[32. How to synchronize List, Set and Map elements? 27](#_Toc468889408)

[33. What is a default capacity of ArrayList, Vector, HashMap, Hashtable and Hashset? 27](#_Toc468889409)

[34. What is the difference between Comparable and Comparator? 28](#_Toc468889410)

[35. What is the use of Comparator/Comparable interfaces? 29](#_Toc468889411)

[36. for-each vs Iterator. Which will be the better option? 29](#_Toc468889412)

[37. HashMap Vs ConcurrentHashMap 29](#_Toc468889413)

[38. Difference between ConcurrentHashMap and Hashtable 30](#_Toc468889414)

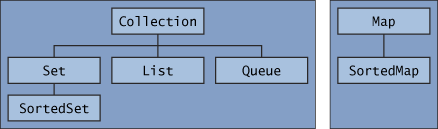
[Other Concepts 31](#_Toc468889415)

[i) Fail Fast Vs Fail Safe Iterator in Java 31](#_Toc468889416)

[ii) Legacy Classes and Interfaces 35](#_Toc468889417)

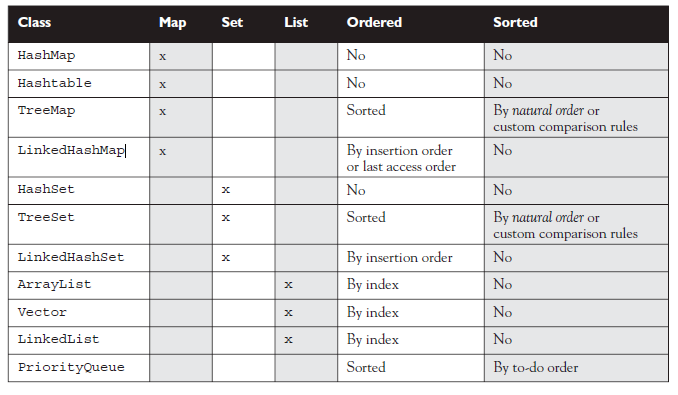
[iii) Deep Copy vs Shallow Copy 35](#_Toc468889418)

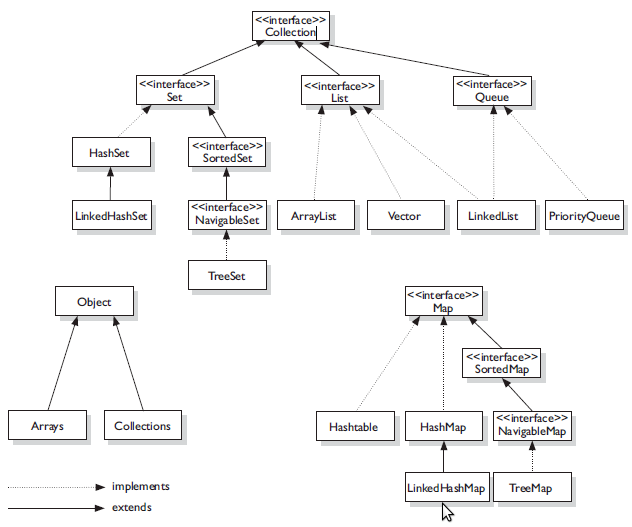
# Java Collection Diagram



|  | | **Implementations** | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Hash Table** | **Resizable Array** | **Balanced Tree** | **Linked List** | **Hash Table + Linked List** |
| **Interfaces** | **Set** | [HashSet](http://download.oracle.com/javase/6/docs/api/java/util/HashSet.html) |  | [TreeSet](http://download.oracle.com/javase/6/docs/api/java/util/TreeSet.html) |  | [LinkedHashSet](http://download.oracle.com/javase/6/docs/api/java/util/LinkedHashSet.html) |
| **List** |  | [ArrayList](http://download.oracle.com/javase/6/docs/api/java/util/ArrayList.html)  [Vector](http://download.oracle.com/javase/6/docs/api/index.html) |  | [LinkedList](http://download.oracle.com/javase/6/docs/api/java/util/LinkedList.html) |  |
| **Map** | [HashMap](http://download.oracle.com/javase/6/docs/api/java/util/HashMap.html)  [Hashtable](http://download.oracle.com/javase/6/docs/api/index.html) |  | [TreeMap](http://download.oracle.com/javase/6/docs/api/java/util/TreeMap.html) |  | [LinkedHashMap](http://download.oracle.com/javase/6/docs/api/java/util/LinkedHashMap.html) |

(<http://pedrocardoso.eu/scjp-java-collections-cheat-sheet/>)





# How to create an ArrayList and what happened internally?

- There are two ways to create an ArrayList object.

**a. Create the empty list with initial capacity**

* 1. **List arrlstobj = new ArrayList(); (or ArrayListarrlstobj = new ArrayList();)**

When we create ArrayList this way, the default constructor of the ArrayList class is invoked. It will create internally an array of Object with default size set to 10.

* 1. **List arrlstobj = new ArrayList(20);**

When we create ArrayList this way, the ArrayList will invoke the constructor with the integer argument. It will create internally an array of Object. The size of the Object[] will be equal to the argument passed in the constructor. Thus when above line of code is executed,it creates an Object[] of capacity 20.

Thus, above ArrayList constructors will create an empty list .Their initial capacity can be 10 or equal to the value of the argument passed in the constructor.

**b. Create the non-empty list containing the elements of the specified collection.**

List arrlstobj = new ArrayList(Collection c);

The above ArrayList constructor will create a non-empty list containing the elements of the collection passed in the constructor.

# Which one is correct?

**List<String>arrlstobj= new ArrayList<>(); OR**

**List<String>arrlstobj= new ArrayList<String>();**

* Both are correct but first one is valid only after JAVA SE 7 or later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called the diamond.

# Is ArrayList al = new ArrayList(); is valid statement?

* Yes it is a valid statement and will make ArrayList of Object Class. Even it is same as **ArrayList al = new ArrayList<>();**

# Explain ArrayList size increment mechanism.

* While new element is being added it will try to determine what the current size of occupied elements is and what the maximum size of the array is. If size of the filled elements (including the new element to be added to the ArrayList class) is greater than the maximum size of the array then increase the size of array. But the size of the array cannot be increased dynamically. So what happens internally is new Array is created with capacity (almost 50% of old size)

**Till Java 6**

intnewCapacity = (oldCapacity \* 3)/2 + 1;

**(Update) From Java 7**

intnewCapacity = oldCapacity + (oldCapacity>> 1);

Also, data from the old array is copied into the new array.

# Which copy technique internally used by the ArrayList class clone() method?

* There are two copy techniques present in the object oriented programming language, deep copy and shallow copy. (For more info **Deep Copy vs Shallow Copy**)

Just like HashSet, ArrayList also returns the shallow copy of the HashSet object. It means elements themselves are not cloned. In other words, shallow copy is made by copying the reference of the object.

# How to create ArrayList from Array (Object[]) ?

* One liner answer: List arraylistobj = Arrays.asList(arrayobj);

# How to convert List to Array?

* String[] a = list\_1.toArray(new String[0]);

If the list\_1 is of size of 3 and if we pass new String[5] the it will print null two times. But if we pass new String[x] where x is less than or equal to list\_1.size() then it will print the list.

# What happens if ArrayList is concurrently modified while iterating the elements?

* According to ArrayList Oracle Java docs, the iterators returned by the ArrayList class's iterator and listiterator method are fail-fast. So it will throw ConcurrentModificationException, which can be happened in following two scenarios:

**Single Threaded Environment**

After the creation of the iterator, structure is modified at any time by any method other than iterator's own remove method.

**Multiple Threaded Environment**

If one thread is modifying the structure of the collection while other thread is iterating over it.

(Read more **Fail Fast Vs Fail Safe Iterator in Java**)

# Can we add element in an ArrayList at any arbitrary index (e.g. directly adding an element at 2nd place in place of starting with 0?)

* It will give runtime error (java.lang.IndexOutOfBoundsException ->java.util.ArrayList.rangeCheckForAdd)

# What is the difference between Iterator and ListIterator?

|  |  |
| --- | --- |
| **Iterator** | **List Iterator** |
| Iterator is used for traversing List, Set & Queue. | We can use ListIterator to traverse List only, we cannot traverse Set using ListIterator. |
| We can traverse in only forward direction using Iterator. | Using ListIterator, we can traverse a List in both the directions (forward and Backward). |
| We cannot obtain indexes while using Iterator | We can obtain indexes at any point of time while traversing a list using ListIterator. The methods nextIndex() and previousIndex() are used for this purpose. |
| We cannot add element to collection while traversing it using Iterator, it throws ConcurrentModificationException when you try to do it. | We can add element at any point of time while traversing a list using ListIterator. |
| We cannot replace the existing element value when using Iterator. | By using set(E e) method of ListIterator we can replace the last element returned by next() or previous() methods. |
| Methods of Iterator:  hasNext() next() remove() | Methods of ListIterator:  add(E e) hasNext() hasPrevious() next() nextIndex() previous() previousIndex() remove() set(E e) |
| **Example:**  ArrayList<String> a1= new ArrayList<>();  a1.add("Zero");  a1.add("One");  a1.add("Two");  a1.add("Three");  // Iterator<String> i = a1.iterator(); or  Iterator i = a1.iterator();  while(i.hasNext()){  System.out.println(i.next());  }  **Output:**  Zero ++  One  Two  Three | **Example:**  ArrayList<String> a1= new ArrayList<>();  String s;  a1.add("Zero");  a1.add("One");  a1.add("Two");  a1.add("Three");  ListIterator<String> i1 = a1.listIterator();  i1.next();  i1.set("Zero ++");  ListIterator<String> i2 = a1.listIterator();  while(i2.hasNext()){  System.out.println(i2.next());  }  **Output:**  Zero ++  One  Two  Three |
| Iterator uses Iterator Design Pattern internally. (Provides a way to access the elements of an aggregate object without exposing its underlying represenation.) |  |

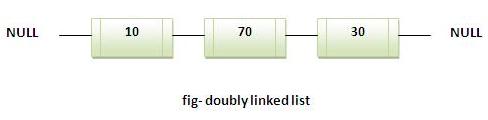
# What is the difference between Iterator and Enumeration?

|  |  |
| --- | --- |
| **Iterator** | **Enumeration** |
| Iterator can traverse legacy and non-legacy elements.  (Check out: **Legacy Classes and Interfaces**) | Enumeration can traverse only legacy elements. |
| Iterator is fail-fast. | Enumeration is not fail-fast. |
| Iterator is slower than Enumeration. | Enumeration is faster than Iterator. |

# What is the difference between ArrayList and Vector?

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| ArrayList is not synchronized. | Vector is synchronized (Thread safe). |
| ArrayList is not a legacy class. | Vector is a legacy class. (All legacy classes were re-engineered to support generic in JDK5.) |
| Initial capacity (size) of an Arraylist is 10 | Initial capacity (size) of an Vector is also 10 |
| ArrayList increases its size by 50% of the array size. | Vector increases its size by doubling the array size. |
| **Example:**  ArrayList<String> a1= new ArrayList<>();  a1.add("Zero");  a1.add("One");  a1.add("Two");  a1.add("Three");  Iterator<String>i = a1.iterator();  while(i.hasNext()){  System.out.println(i.next());  } | **Example:**  Vector<String> a1= new Vector<>();  a1.add("Zero");  a1.add("One");  a1.add("Two");  a1.add("Three");  Iterator<String>i = a1.iterator();  while(i.hasNext()){  System.out.println(i.next());  } |

# What is the difference between ArrayList and LinkedList?



LinkedList has almost all methods as ArrayList like void add(int index, E element). It also has some other methods related to Linked and Queue concepts like addFirst, addLast, offerFirst, offerLast, peek, pop, push etc.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| ArrayList internally uses dynamic array to store the elements. | LinkedList internally uses doubly linked list to store the elements. |
| Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| ArrayList class can act as a list only because it implements List only. | LinkedList class can act as a list and queue both because it implements List and Deque interfaces. |
| ArrayList is better for storing and accessing data. | LinkedList is better for manipulating data. |

# Are there any differences between different methods in each of the following groups of element retrieve operations in LinkedList?

**(Are there any differences between different methods in each of the following groups of element retrieve operations in LinkedList?**

**Returning null + removing operations: poll(), pollFirst().**

**Returning null + not removing operations: peek(), peekFirst().**

**Throwing exception + removing operations: pop(), remove(), removeFirst().**

**Throwing exception + not removing operations: element(), getFirst().**

**Similar duplications exist in insertion methods. If there is no such difference, I would expect it to be mentioned in the javadoc of the methods (something like the good old "This is exactly like calling ..."). Is it only a sloppy documentation, or am I missing anything?)**

**(**[**http://stackoverflow.com/questions/14851367/java-linkedlist-differences-between-retrieve-operations**](http://stackoverflow.com/questions/14851367/java-linkedlist-differences-between-retrieve-operations)**)**

* There is no difference between them, and it is listed in the documentation too, but you have to do some recursive searching to get there. LinkedList implements two interfaces - Queue and Deque. And Deque extends from Queue. Now, Deque has defined the method - Deque#pollFirst() and inherited the method - Queue#poll(). So, LinkedList has basically these two methods defined for the two interfaces it implements.

# What is the difference between Collection and Collections?

* Collection is an interface whereas Collections is a class. Collection interface provides normal functionality of data structure to List, Set and Queue. But, Collections class is to sort and synchronize collection elements.

# Hash Table - Data Structure and Algorithm (To understand Hash Table Data Structure concepts only)

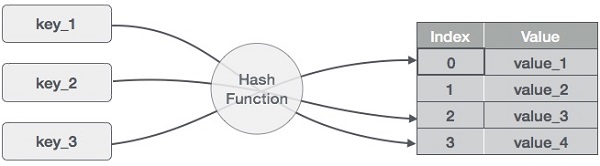
* (<https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm>)

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

**Hashing**

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



* (1,20)
* (2,70)
* (42,80)
* (4,25)
* (12,44)
* (14,32)
* (17,11)
* (13,78)
* (37,98)

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Key** | **Hash** | **Array Index** |
| 1 | 1 | 1 % 20 = 1 | 1 |
| 2 | 2 | 2 % 20 = 2 | 2 |
| 3 | 42 | 42 % 20 = 2 | 2 |
| 4 | 4 | 4 % 20 = 4 | 4 |
| 5 | 12 | 12 % 20 = 12 | 12 |
| 6 | 14 | 14 % 20 = 14 | 14 |
| 7 | 17 | 17 % 20 = 17 | 17 |
| 8 | 13 | 13 % 20 = 13 | 13 |
| 9 | 37 | 37 % 20 = 17 | 17 |

**Linear Probing**

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No.** | **Key** | **Hash** | **Array Index** | **After Linear Probing, Array Index** |
| 1 | 1 | 1 % 20 = 1 | 1 | 1 |
| 2 | 2 | 2 % 20 = 2 | 2 | 2 |
| 3 | 42 | 42 % 20 = 2 | 2 | 3 |
| 4 | 4 | 4 % 20 = 4 | 4 | 4 |
| 5 | 12 | 12 % 20 = 12 | 12 | 12 |
| 6 | 14 | 14 % 20 = 14 | 14 | 14 |
| 7 | 17 | 17 % 20 = 17 | 17 | 17 |
| 8 | 13 | 13 % 20 = 13 | 13 | 13 |
| 9 | 37 | 37 % 20 = 17 | 17 | 18 |

\*In Java Bucket term used here is actually an index of array, that array is called table in HashMap implementation. Thus table[0] is referred as bucket0, table[1] as bucket1 and so on.

# [What is meant by number of buckets in the HashMap?](http://stackoverflow.com/questions/18636576/what-is-meant-by-number-of-buckets-in-the-hashmap)

* (<http://stackoverflow.com/questions/18636576/what-is-meant-by-number-of-buckets-in-the-hashmap>)

Yes, exactly, each bucket can have multiple key-value pairs.

The object's hashCode() determines which bucket it goes into, via this expression: object.hashCode() % n, where n = the total number of buckets and % is the modulus operator.

Most often the objects will be well distributed across buckets, but you have no guarantee where they go. This depends on the data and the hashCode function.

Obviously, when the hashCode implementation is poor, the performance of the hashmap will go down.

Also read up on the equals / hashcode contract, which is relevant.

# Get size/length of a value in a HashMap

(<http://stackoverflow.com/questions/12483457/get-size-length-of-a-value-in-a-hashmap>)

(Map<Integer, String> map = new HashMap<Integer, String>();

How do I get the size/length of matching value's at the String?

example:

1 , Red 2 , Red 3 , Blue 4 , Blue 5 , Red

Size of the String of RED = 3)

* Use [.values()](http://docs.oracle.com/javase/6/docs/api/java/util/Map.html#values%28%29) to get a collection containing all the values of the hash map, and then use [Collections.frequency()](http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#frequency%28java.util.Collection,%20java.lang.Object%29) to count the number of objects in the collection.

return Collections.frequency(map.values(), "red");

# HashMap Method Summary

* Below method summary based on Java SE 7

(<https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html>)

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| void | [**clear**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#clear())()  Removes all of the mappings from this map. |
| [**Object**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) | [**clone**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#clone())()  Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| boolean | [**containsKey**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#containsKey(java.lang.Object))([**Object**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) key)  Returns true if this map contains a mapping for the specified key. |
| boolean | [**containsValue**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#containsValue(java.lang.Object))([**Object**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) value)  Returns true if this map maps one or more keys to the specified value. |
| [**Set**](https://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<[**Map.Entry**](https://docs.oracle.com/javase/7/docs/api/java/util/Map.Entry.html)<[**K**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html),[**V**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html)>> | [**entrySet**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#entrySet())()  Returns a [**Set**](https://docs.oracle.com/javase/7/docs/api/java/util/Set.html) view of the mappings contained in this map. |
| [**V**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) | [**get**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#get(java.lang.Object))([**Object**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) key)  Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. |
| boolean | [**isEmpty**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#isEmpty())()  Returns true if this map contains no key-value mappings. |
| [**Set**](https://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<[**K**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html)> | [**keySet**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#keySet())()  Returns a [**Set**](https://docs.oracle.com/javase/7/docs/api/java/util/Set.html) view of the keys contained in this map. |
| [**V**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) | [**put**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#put(K,%20V))([**K**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) key, [**V**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) value)  Associates the specified value with the specified key in this map. |
| void | [**putAll**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#putAll(java.util.Map))([**Map**](https://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<? extends [**K**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html),? extends [**V**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html)> m)  Copies all of the mappings from the specified map to this map. |
| [**V**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) | [**remove**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#remove(java.lang.Object))([**Object**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) key)  Removes the mapping for the specified key from this map if present. |
| int | [**size**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#size())()  Returns the number of key-value mappings in this map. |
| [**Collection**](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<[**V**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html)> | [**values**](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html#values())()  Returns a [**Collection**](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html) view of the values contained in this map. |

Java SE 8 has some extra methods checkout it [here](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html).

# How HashMap works in Java?

* (<http://javarevisited.blogspot.in/2011/02/how-hashmap-works-in-java.html>)

HashMap in Java works on hashing principle. It is a data structure which allows us to store object and retrieve it in constant time O(1) provided we know the key. In hashing, hash functions are used to link key and value in HashMap. Objects are stored by calling put(key, value) method of HashMap and retrieved by calling get(key) method. When we call put method, hashcode() method of the key object is called so that hash function of the map can find a bucket location to store value object, which is actually an index of the internal array, known as the table. HashMap internally stores mapping in the form of **Map.Entry** object which contains both key and value object. When you want to retrieve the object, you call [the get() method](http://java67.blogspot.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) and again pass the key object. This time again key object generate same hash code (it's mandatory for it to do so to retrieve the object and that's why HashMap keys are immutable e.g. String) and we end up at same bucket location. If there is only one object then it is returned and that's your value object which you have stored earlier. Things get little [tricky](http://java67.blogspot.com/2012/09/top-10-tricky-java-interview-questions-answers.html) when collisions occur. It's easy to answer this question if you have read good books on data structure and algorithms like [this](http://www.amazon.com/Data-Structures-Algorithm-Analysis-Edition/dp/0132576279?tag=javamysqlanta-20) one. If you know how hash table data structure works then this is a piece of cake.  
  
Since the internal array of HashMap is of fixed size, and if you keep storing objects, at some point of time hash function will return same bucket location for two different keys, this is called collision in HashMap. In this case, a linked list is formed at that bucket location and a new entry is stored as next node.  
  
If we try to retrieve an object from this linked list, we need an extra check to search correct value, this is done by equals() method. Since each node contains an entry, HashMap keeps comparing entry's key object with the passed key using equals() and when it return true, Map returns the corresponding value.

**How HashMap Internally Works in Java**

Questions start with simple statement:

## Have you used HashMap before or  What is HashMap? Why do you use it?

Almost everybody answers this with yes and then interviewee keep talking about common facts about HashMap like HashMap accept null while Hashtable doesn't, [HashMap is not synchronized](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html), HashMap is fast and so on along with basics like its stores key and value pairs etc. This shows that person has used HashMap and quite familiar with the functionality it offers, but interview takes a sharp turn from here and next set of follow-up questions gets more detailed about fundamentals involved with HashMap in Java. Interviewer strike back with questions like:

## Do you Know how HashMap works in Java or How does get () method of HashMap works in Java?

And then you get answers like,  I don't bother its standard Java API, you better look code on Java source or Open JDK; I can find it out in Google at any time etc. But some interviewee definitely answers this and will say **HashMap works on the principle of hashing**, we have put(key, value) and get(key) method for storing and retrieving Objects from HashMap. When we pass Key and Value object  to put() method on Java HashMap, HashMap implementation calls [hashCode method](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) on Key object and applies returned hashcode into its own hashing function to find a bucket location for storing Entry object, important point to mention is that HashMap in Java stores both key and value object as Map.Entry in a bucket which is essential to understand the retrieving logic.   
  
If people fail to recognize this and say it only stores Value in the bucket they will fail to explain the retrieving logic of any object stored in Java HashMap. This answer is very much acceptable and does make sense that interviewee has a fair bit of knowledge on how hashing works and how HashMap  works in Java. But this is just start of story and confusion increases when you put interviewee on scenarios faced by Java developers on day by day basis. Next question could be about collision detection and collision resolution in Java HashMap  e.g.

## What will happen if two different objects have the same hashcode?

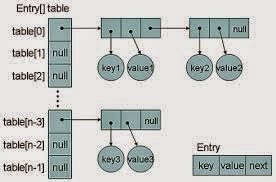
Now from here onwards real confusion starts, sometime candidate will say that since hashcode is equal, both objects are equal and HashMap  will throw exception or not store them again etc, Then you might want to remind them about [equals() and hashCode() contract](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) that two unequal objects in Java can have same hashcode. Some will give up at this point and few will move ahead and say "Since hashcode is same, bucket location would be same and collision will occur in HashMap Since HashMap uses LinkedList to store object, this entry (object of Map.Entry comprise key and value )  will be stored in [LinkedList](http://javarevisited.blogspot.sg/2012/02/difference-between-linkedlist-vs.html). Great this answer make sense though there are many collision resolution methods available  like linear probing and chaining, this is simplest and HashMap in Java does follow this. But story does not end here and interviewer asks

## How will you retrieve Value object  if two Keys will have the same hashcode?

Interviewee will say we will call get() method and then HashMap uses Key Object's hashcode to find out bucket location and retrieves Value object but then you need to remind him that there are two Value objects are stored in same bucket , so they will say about [traversal in LinkedList](http://javarevisited.blogspot.sg/2010/10/how-do-you-find-length-of-singly-linked.html) until we find the value object , then you ask *how do you identify value object because you don't  have value object to compare* ,Until they know that HashMap  stores both Key and Value in LinkedList node or as Map.Entry they won't be able to resolve this issue and will try and fail.

But those bunch of people who remember this key information will say that after finding bucket location, we will **call keys.equals() method** to identify a correct node in LinkedList and return associated value object for that key in Java HashMap. Perfect this is the correct answer.

In many cases interviewee fails at this stage because they get confused between[hashCode()](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) and equals(**)** or keys and values object in Java HashMap  which is pretty obvious because they are dealing with the hashcode() in all previous questions and equals() come in picture only in case of retrieving value object from HashMap in Java. Some good developer point out here that using immutable, [final object](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) with proper equals() and hashcode() implementation would act as perfect Java HashMap  keys and **improve the performance of Java HashMap  by reducing collision**. Immutability *also allows caching their hashcode of different keys* which makes overall retrieval process very fast and suggest that [String](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) and various wrapper classes e.g. Integer very good keys in Java HashMap.

[](http://4.bp.blogspot.com/-adRczhctozE/VD_eimhTQbI/AAAAAAAACCg/lfA1G5GZXyM/s1600/How+HashMap+works+in+Java+(1).jpg)

Now if you clear this entire Java HashMap interview,  You will be surprised by this very interesting question "**What happens On HashMap in Java if the size of the HashMap  exceeds a given threshold defined by load factor ?"**. Until you know how HashMap  works exactly you won't be able to answer this question. If the size of the Map exceeds a given threshold defined by load-factor e.g. if the load factor is .75 it will act to re-size the map once it filled 75%. Similar to other collection classes like [ArrayList](http://javarevisited.blogspot.sg/2011/05/example-of-arraylist-in-java-tutorial.html),  Java HashMap re-size itself by creating a new bucket array of size twice of the previous size of HashMap and then start putting every old element into that new bucket array. This process is called rehashing because it also applies the hash function to find new bucket location.

If you manage to answer this question on HashMap in Java you will be greeted by **"do you see any problem with resizing of HashMap  in Java"** , you might not be able to pick the context and then he will try to give you hint about multiple thread accessing the Java HashMap and potentially looking for **race condition on HashMap  in Java**.

So the answer is Yes there is potential [race condition](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) exists while resizing HashMap in Java, if two [thread](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html) at the same time found that now HashMap needs resizing and they both try to resizing. on the process of resizing of HashMap in Java, the element in the bucket which is stored in linked list get reversed in order during their migration to new bucket because Java HashMap  doesn't append the new element at tail instead it append new element at the head *to avoid tail traversing*. If race condition happens then you will end up with an infinite loop. Though this point, you can potentially argue that what the hell makes you think to use HashMap  in multi-threaded environment to interviewer :)

## Some more Hashtable and HashMap Questions

Few more question on HashMap in Java which is contributed by readers of Javarevisited blog:

## 1) Why String, Integer and other wrapper classes are considered good keys?

String, Integer and other wrapper classes are natural candidates of HashMap key, and String is most frequently used key as well because [String is immutable and final](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html), and overrides equals and hashcode() method. Other wrapper class also shares similar property. Immutability is required, in order to prevent changes on fields used to calculate hashCode() because if key object returns different hashCode during insertion and retrieval than it won't be possible to get an object from HashMap.   
  
Immutability is best as it offers other advantages as well like [thread-safety](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), If you can keep your hashCode same by only making certain fields final, then you go for that as well. Since equals() and hashCode() method is used during retrieval of value object from HashMap, it's important that key object correctly override these methods and follow contact. If unequal object returns different hashcode than chances of collision will be less which subsequently improve the performance of HashMap.

## 2) Can we use any custom object as a key in HashMap?

This is an extension of previous questions. Of course you can use any Object as key in Java HashMap provided it follows equals and hashCode contract and its hashCode should not vary once the object is inserted into [Map](http://javarevisited.blogspot.sg/2011/12/how-to-traverse-or-loop-hashmap-in-java.html). If the custom object is Immutable than this will be already taken care because you can not change it once created.

## 3) Can we use ConcurrentHashMap in place of Hashtable?

This is another question which getting popular due to increasing popularity of ConcurrentHashMap. Since we know Hashtable is synchronized but ConcurrentHashMap provides better concurrency by only locking portion of map determined by concurrency level. ConcurrentHashMap is certainly introduced as Hashtable and can be used in place of it, but Hashtable provides stronger thread-safety than ConcurrentHashMap. See my post [difference between Hashtable and ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) for more details.

Personally, I like this question because of its depth and number of concept it touches indirectly if you look at questions asked during interview this HashMap  questions has verified

* The concept of hashing
* Collision resolution in HashMap
* Use of equals () and hashCode () and their importance in HashMap?
* The benefit of the immutable object?
* Race condition on HashMap  in Java
* Resizing of Java HashMap

Just to summarize here are the answers which do make sense for above questions

(Below 4,5,6,7 are from <http://howtodoinjava.com/core-java/collections/popular-hashmap-and-concurrenthashmap-interview-questions/>)

## ****4) Difference between HashMap and Collections.synchronizedMap(HashMap)****

It’s easy question, right !! HashMap is non-synchronized and Collections.synchronizedMap() returns a wrapped instance of HashMap which has all get, put methods synchronized.

Essentially, **Collections.synchronizedMap() returns the reference of internally created inner-class “SynchronizedMap”**, which contains key-value pairs of input HashMap, passed as argument.

This instance of inner class has nothing to do with original parameter HashMap instance and is completely independent.

## ****5) Difference between ConcurrentHashMap and Collections.synchronizedMap( HashMap )****

This one is slightly tougher. Both are synchronized version of HashMap, with difference in their core functionality and internal structure.

As stated above, ConcurrentHashMap is consist of internal segments which can be viewed as independent HashMaps, conceptually. All such segments can be locked by separate threads in high concurrent executions. In this way, **multiple threads can get/put key-value pairs from ConcurrentHashMap without blocking/waiting for each other**.

In Collections.synchronizedMap(), we get a synchronized version of HashMap and **it is accessed in blocking manner**. This means if multiple threads try to access synchronizedMap at same time, they will be allowed to get/put key-value pairs one at a time in synchronized manner.

## ****6) Difference between HashMap and HashTable****

It is also very easy question. The major difference is that **HashTable is synchronized and HashMap is not**.

If asked for other reasons, tell them, **HashTable is legacy class** (part of JDK 1.0) which was promoted into collections framework by implementing Map interface later. It still has some **extra features like Enumerator** with it, which HashMap lacks.

Another minor reason can be: **HashMap supports null key** (mapped to zero bucket), HashTable does not support null keys and throws NullPointerException on such attempt.

## ****7) Difference between HashTable and Collections.synchronized(HashMap)****

So far you must have got the core idea of the similarities between them. Both are synchronized version of collection. Both have synchronized methods inside class. Both are blocking in nature i.e. multiple threads will need to wait for getting the lock on instance before putting/getting anything out of it.

So what is the difference. Well, **NO major difference** for above said reasons. Performance is also same for both collections.

Only thing which separates them is the fact **HashTable is legacy** class promoted into collection framework. It got its own extra features like enumerators.

## How HashMap works in Java

HashMap  works on the principle of hashing, we have put() and get() method for storing and retrieving object from HashMap.When we pass both key and value to put() method to store on HashMap, it uses key object hashcode() method to calculate hashcode and them by applying hashing on that hashcode it identifies bucket location for storing value object. While retrieving it uses key object equals method to find out correct key value pair and return value object associated with that key. HashMap  uses linked list in case of collision and object will be stored in next node of linked list. Also, [HashMap stores both key and value tuple](http://java67.blogspot.com/2013/02/10-examples-of-hashmap-in-java-programming-tutorial.html) in every node of linked list in the form of Map.Entry object.

## What will happen if two different HashMap  key objects have the same hashcode?

They will be stored in the same bucket but no next node of linked list. And keys equals () method will be used to identify correct key value pair in HashMap.

How null key is handled in HashMap? Since equals() and hashCode() are used to store and retrieve values, how does it work in case of the null key?  
The null key is handled specially in HashMap, there are two separate methods for that putForNullKey(V value) and getForNullKey(). Later is offloaded version of get() to look up null keys.  Null keys always map to index 0.  This null case is split out into separate methods for the sake of performance in the two most commonly used operations (get and put), but incorporated with conditionals in others. In short, equals() and hashcode() method are not used in case of null keys in HashMap.  
  
here is how nulls are retrieved from HashMap  
  
   **private** V **getForNullKey**() {

**if** (size == **0**) {

**return** **null**;

}

**for** (Entry<K,V> e = table[**0**]; e != **null**; e = e.next) {

**if** (e.key == **null**)

**return** e.value;

}

**return** **null**;

}

In terms of usage, Java HashMap is very versatile and I have mostly used HashMap as cache in an electronic trading application I have worked. Since finance domain used Java heavily and due to performance reason we need caching HashMap and ConcurrentHashMap  comes as very handy there. You can also check following articles from Javarevisited to learn more about

## HashMap Changes in JDK 1.7 and JDK 1.8

There is some [performance improvement done on HashMap and ArrayList from JDK 1.7](http://javarevisited.blogspot.com/2014/07/java-optimization-empty-arraylist-and-Hashmap-cost-less-memory-jdk-17040-update.html), which reduce memory consumption. Due to this empty Map are lazily initialized and will cost you less memory. Earlier, when you create HashMap e.g. new HashMap() it automatically creates an array of default length e.g. 16. After some research, Java team found that most of this Map are temporary and never use that many elements, and only end up wasting memory. Also, From JDK 1.8 onwards HashMap has introduced an improved strategy to deal with high collision rate. Since a poor hash function e.g. which always return location of same bucket, can turn a HashMap into linked list, i.e. converting get() method to perform in O(n) instead of O(1) and someone can take advantage of this fact, Java now internally replace linked list to a binary true once certain threshold is breached. This ensures performance or order O(log(n)) even in the worst case where a hash function is not distributing keys properly.

# What is the significance of load factor in HashMap?

* (<http://stackoverflow.com/questions/10901752/what-is-the-significance-of-load-factor-in-hashmap>)

**What is load factor?**

The amount of capacity which is to be exhausted for the HashMap to increase its capacity?

**Why load factor?**

Load factor is by default 0.75 of the initial capacity (16) therefore 25% of the buckets will be free before there is an increase in the capacity & this makes many new buckets with new hashcodes pointing to them to exist just after the increase in the number of buckets.

**Now why should you keep many free buckets & what is the impact of keeping free buckets on the performance?**

If you set the loading factor to say 1.0 then something very interesting might happen.

Say you are adding an object x to your hashmap whose hashCode is 888 & in your hashmap the bucket representing the hashcode is free , so the object x gets added to the bucket, but now again say if you are adding another object y whose hashCode is also 888 then your object y will get added for sure BUT at the end of the bucket (because the buckets are nothing but linkedList implementation storing key,value & next) now this has a performance impact ! Since your object y is no longer present in the head of the bucket if you perform a lookup the time taken is not going to be O(1) this time it depend on how many items are there in the same bucket. This is called hash collision by the way & this even happen when your loading factor is less than 1.

**Correlation between performance , hash collision & loading factor ?**

**Lower load factor** = more free buckets **= less chances of collision** = high performance = high space requirement.

Default initial capacity of the HashMap takes is 16 and load factor is 0.75f (i.e 75% of current map size). The load factor represents at what level the HashMap capacity should be doubled.

**For example** product of capacity and load factor as 16 \* 0.75 = 12. This represents that after storing the 12th key – value pair into the HashMap , its capacity becomes 32

# Aggregating values in hashmap by keys

* (<http://stackoverflow.com/questions/21544103/aggregating-values-in-hashmap-by-keys>)

List<Integer> values = new ArrayList<>();

for (Map.Entry<Integer, Integer> entry : map.entrySet()) {

if (entry.getKey() % 5 == 0) {

values.add(entry.getValue());

}

}

FWIW, a comparable Java 8 approach might look like (Lambda Expression)

map.entrySet().stream()

.filter(entry -> entry.getKey() % 5 == 0)

.map(Entry<Integer, Integer>::getValue)

.collect(toLi

# What is the difference between HashMap and Hashtable?

|  |  |
| --- | --- |
| **HashMap** | **HashTable** |
| HashMap is not synchronized. | Hashtable is synchronized. |
| HashMap can contain one null key and multiple null values. | Hashtable cannot contain any null key or null value. |

# What is hash-collision in Hashtable and how it is handled in Java?

* Two different keys with the same hash value is known as hash-collision.

If two keys have same hashCode then first one isn't overridden just because the second one has the same hashCode. It will be overridden only if it is also equal (as said by equals). If not, both values will be kept in the linked list. When fetching a key, all nodes with the same hashCode will be compared to the provided key until one is equal then its value will be returned (using the equals method). If no key in the map is equal, you'll get null. The only problem you have if many objects have the same hashCode (or more precisely the same hashCode modulo the size of the internal Entry[] table) is that the linked list will always be read, which is slower (and defeats the purpose of any hash table). That's why it's important when designing a hashcode method to ensure the generated integers are well distributed.

# On what bases TreeMap and TreeSet is sorted?

* A TreeMap is always sorted based on its keys, however if you want to sort it based on its values then you can build a logic to do this using comparator.

A TreeSet is sorted on the value (not index).

Note that the ordering maintained by a sorted set (whether or not an explicit comparator is provided) must be consistent with equals if the sorted set is to correctly implement the Set interface. (See the Comparable interface or Comparator interface for a precise definition of consistent with equals.)

It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

(<http://stackoverflow.com/questions/7229977/java-why-it-is-implied-that-objects-are-equal-if-compareto-returns-0>)

# What is difference between ordered and sorted collection?

* An ordered collection means that the elements of the collection have a specific order. The order is independent of the value. A List is an example. A sorted collection means that not only does the collection have order, but the order depends on the value of the element. A SortedSet (TreeSet) is an example. In contrast, a collection without any order can maintain the elements in any order. A Set is an example.

# What is the difference between List and Set?

* List can contain duplicate elements whereas Set contains only unique elements.

You cannot access set using index because sets have no ordering. Some implementations do (particularly those implementing the java.util.SortedSet interface), but that is not a general property of sets. If you're trying to use sets this way, you should consider using a list instead

**Example of Set:**

Set<String> a1= new HashSet<>();

a1.add("Zero");

a1.add("One");

a1.add("Two");

a1.add("Three");

a1.add("Three"); //This will be avoided

Iterator<String>i = a1.iterator();

while(i.hasNext()){

System.out.println(i.next());

}

**Output:**

Zero

One

Two

Three

# HashSet vs. TreeSet vs. LinkedHashSet

* HashSet is Implemented using a hash table. Elements are not ordered. The add, remove, and contains methods have constant time complexity O(1).

TreeSet is implemented using a tree structure (red-black tree in algorithm book). TreeSet is significantly slower than LinkedHashSet and HashSet because of this sorting overhead. TreeSet sorts all object based upon their natural ordering by using compareTo() method, or custom order by using compare() method. The elements in a set are sorted, but the add, remove, and contains methods has time complexity of O(log (n)). It offers several methods to deal with the ordered set like first(), last(), headSet(), tailSet(), etc.

LinkedHashSet is between HashSet and TreeSet. It is implemented as a hash table with a linked list running through it, so it provides the order of insertion. The time complexity of basic methods is O(1).

In summary, HashSet is fastest for common operation e.g. add, search and remove, LinkedHashSet is close second, as it suffers a little drop in performance due to overhead of maintaining doubly linked list when an element is inserted or deleted. TreeSet is much slower than these two because it needs to perform sorting every time there is change in TreeSet. It means by default you should use HashSet, until and unless you really need LinkedHashSet or TreeSet.

**Interface Implemented:**

HashSet - Serializable, Cloneable, Iterable<E>, Collection<E>, Set<E>

LinkedHashSet - Serializable, Cloneable, Iterable<E>, Collection<E>, Set<E>

TreeSet - Serializable, Cloneable, Iterable<E>, Collection<E>, Set<E>, SortedSet<E>, NavigableSet<E>

# Are HashSet, TreeSet , & LinkedHashSet allow null key?

* This property (null key) allowance is derived from HashMap, LinkedHashMap, and TreeMap since HashSet internally uses HashMap, LinkedHashSet internally uses LinkedHashMap and TreeSet internally uses TreeMap. Both HashMap and LinkedHashMap allow one null key and so are these two Set implementations. On the other hand, since TreeMap doesn't allow null keys, TreeSet doesn't allow null elements and throws java.lang.NullPointerException when you try to add a null object. The main reason of this is the use of compareTo() and compare() method, which throws NullPointerException if one element is null, but it truly depends on implementation.

# Are HashSet, TreeSet , & LinkedHashSet synchronized?

* All three i.e. HashSet, TreeSet, and LinkedHashSet are not synchronized. They cannot be shared between multiple threads until specifically synchronized. It's easy to create synchronized Set, though, all you need to do is use java.util.Collections utility class as shown below :

**Synchronizing HashSet in Java**

Set s = Collections.synchronizedSet(new HashSet(...));

**Synchronizing LinkedHashSet in Java**

Set s = Collections.synchronizedSet(new LinkedHashSet(...));

**Synchronizing TreeSet in Java**

Set s = Collections.synchronizedSet(new TreeSet(...));

# How to synchronize List, Set and Map elements?

* Yes, Collections class provides methods to make List, Set or Map elements as synchronized: public static List synchronizedList(List l){}

public static Set synchronizedSet(Set s){}

public static SortedSet synchronizedSortedSet(SortedSet s){}

public static Map synchronizedMap(Map m){}

public static SortedMap synchronizedSortedMap(SortedMap m){}

# What is a default capacity of ArrayList, Vector, HashMap, Hashtable and Hashset?

ArrayList-10

Vector-10

HashSet-16

HashMap-16

HashTable-11

HashSet-16

**Explanation:**

**ArrayList:**

Constructs an empty list with an initial capacity of 10.

**Vector:**

Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.

**HashMap:**

Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).

**Hashtable:**

Constructs a new, empty hashtable with a default initial capacity (11) and load factor (0.75).

**Hashset:**

Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).

# What is the difference between Comparable and Comparator?

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| A comparable object is capable of comparing itself with another object. | A comparator object is capable of comparing two different objects. The class is not comparing its instances, but some other class’s instances. |
| It provides one method named compareTo(). | It provides one method named compare(). |
| It is found in java.lang package. | It is found in java.util package. |
| If we implement Comparable interface, actual class is modified. | Actual class is not modified. |
| Comparable is in the implementation and not visible from the interface, so when you sort you don't really know what is going to happen. | Comparator gives you reassurance that the ordering will be well defined. |
| An object should implement Comparable if that is the clear natural way to sort the class, and anyone would need to sort the class would generally want to do it that way. | However, the sorting was an unusual use of the class, or the sorting only makes sense for a specific use case, or multiple sorting logic then a Comparator is a better option. |
| **Example:**  public class HDTV implements Comparable<HDTV> {  private int size;  private String brand;  public int getSize() {  return size;  }  public void setSize(int size) {  this.size = size;  }  public String getBrand() {  return brand;  }  public void setBrand(String brand) {  this.brand = brand;  }  @Override  public int compareTo(HDTV tv) {  if (this.getSize() > tv.getSize()) {  return 1;  } else if (this.getSize() < tv.getSize()) {  return -1;  } else {  return 0;  }  }  } | **Example:**  public class HDTV1 implements Comparator<HDTV1> {  private int size;  private String brand;  public int getSize() {  return size;  }  public void setSize(int size) {  this.size = size;  }  public String getBrand() {  return brand;  }  public void setBrand(String brand) {  this.brand = brand;  }  @Override  public int compare(HDTV1 tv1, HDTV1 tv2) {    if (tv1.getSize() > tv2.getSize()) {  return 1;  } else if (tv1.getSize() < tv2.getSize()) {  return -1;  } else {  return 0;  }  }  } |
| **Sorting using Comparable:**  Collections.sort(l); | **Sorting using Comparator:**  Collections.sort(al1, new HDTV1()); |

# What is the use of Comparator/Comparable interfaces?

* Many times we need to sort a collection of objects by a certain property or rule, especially if you are working on a domain where people are your main entities (e.g. HR), you will encounter this requirement more frequently. Also it is useful to provide custom sorting on TreeMap and TreeSet.

# for-each vs Iterator. Which will be the better option?

* Internally the for-each loop creates an Iterator to iterate through the collection. for-each is syntactic sugar for using iterators (approach 2).

You might need to use iterators if you need to modify collection in your loop. First approach will throw exception.

for (String i : list) {

System.out.println(i);

list.remove(i); // throws java.util.ConcurrentModificationException exception but

} // if we break the loop on removal then it won’t

Iterator it=list.iterator();

while (it.hasNext()){

System.out.println(it.next());

it.remove(); // valid here

}

# 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 .

**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.

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 in ConcurrentHashMap 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.

# Difference between ConcurrentHashMap and Hashtable

* The difference between Hashtable and ConcurrentHashMap, both can be used in the multithreaded environment but once the size of Hashtable becomes considerable large performance degrade because for iteration it has to be locked for a longer duration.

Since ConcurrentHashMap introduced the concept of segmentation, how large it becomes only certain part of it get locked to provide thread safety so many other readers can still access map without waiting for iteration to complete.

In Summary, ConcurrentHashMap only locked certain portion of Map while Hashtable locks full map while doing iteration. This will be clearer by looking at this diagram which explains the internal working of ConcurrentHashMap in Java.

# Other Concepts

## Fail Fast Vs Fail Safe Iterator in Java

(<http://javahungry.blogspot.com/2014/04/fail-fast-iterator-vs-fail-safe-iterator-difference-with-example-in-java.html>)

* Difference between Fail fast and fail safe iterator or Fail fast vs Fail Safe iterator is one of those questions which are used to test your knowledge about the topic Concurrency.  
  Before we discuss in detail about fail safe iterator and fail fast iterator in addition to theircomparison, we should understand the term *Concurrent Modification*.  
    
  **What is Concurrent Modification?**  
    
  When one or more thread is iterating over the collection, in between, one thread changes the structure of the collection (either adding the element to the collection or by deleting the element in the collection or by updating the value at particular position in the collection) is known as Concurrent Modification  
    
  **Difference between Fail Fast iterator and Fail Safe iterator**  
    
  **Fail fast Iterator**  
  Fail fast iterator while iterating through the collection , instantly throws Concurrent Modification Exception if there is structural modification  of the collection . 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.   
    
  Fail-fast iterator can throw ConcurrentModificationException in two scenarios:

***Single Threaded Environment***   
After the creation of the iterator, structure is modified at any time by any method other than iterator's own remove method.   
   
***Multiple Threaded Environment***  
  
 If one thread is modifying the structure of the collection while other thread is iterating over it.  
  
According to [Oracle docs](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) , **the fail-fast behavior of an iterator cannot be guaranteed** as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness:**the fail-fast behavior of iterators should be used only to detect bugs.**  
  
**Interviewer: How  Fail  Fast Iterator  come to know that the internal structure is modified ?**  
Iterator read internal data structure (object array) directly. The internal data structure(i.e object array) should not be modified while iterating through the collection. To ensure this it maintains an internal  flag *"mods" .*Iterator checks the *"mods" flag*whenever it gets the next value (using hasNext() method and next() method). Value of *mods* flag changes whenever there is an structural modification. Thus indicating iterator to throw ConcurrentModificationException.  
  
  
**Fail Safe Iterator:**  
  
Fail Safe Iterator makes copy of the internal data structure (object array) and iterates over the copied data structure.Any structural modification done to the iterator affects the copied data structure.  So , original data structure remains  structurally unchanged .Hence , no ConcurrentModificationException throws by the fail safe iterator.  
  
Two issues associated with Fail Safe Iterator are:  
  
1. Overhead of maintaining the copied data structure i.e memory.  
  
2.  Fail safe iterator does not guarantee that the data being read is the data currently in the original data structure.   
  
According to [Oracle docs](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html) , fail safe iterator is ordinarily too costly, but may be more efficient than alternatives when traversal operations vastly outnumber mutations, and is useful when you cannot or don’t want to synchronize traversals, yet need to preclude interference among concurrent threads. The "snapshot" style iterator method uses a reference to the state of the array at the point that the iterator was created. This **array never changes during the lifetime of the iterator, so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException**.The iterator will not reflect additions, removals, or changes to the list since the iterator was created. Element-changing operations on iterators themselves (remove(), set(), and add()) are not supported. These methods throw UnsupportedOperationException.  
  
  
  
**Example of Fail Fast Iterator and Fail Safe Iterator**

**importjava.util.HashMap**;

**importjava.util.Iterator**;

**importjava.util.Map**;

**publicclassFailFastExample**

{

**publicstaticvoidmain**(String[]args)

{

Map<String,String>premiumPhone=**new**HashMap<String,String>();

premiumPhone.put("Apple","iPhone");

premiumPhone.put("HTC","HTC one");

premiumPhone.put("Samsung","S5");

Iterator iterator=premiumPhone.keySet().iterator();

**while**(iterator.hasNext())

{

System.out.println(premiumPhone.get(iterator.next()));

premiumPhone.put("Sony","Xperia Z");

}

}

}

**Output :**

iPhone

Exception in thread "main"java.util.ConcurrentModificationException

atjava.util.HashMap$HashIterator.nextEntry(Unknown Source)

atjava.util.HashMap$KeyIterator.next(Unknown Source)

atFailFastExample.main(FailFastExample.java:**20**)

**Fail Safe Iterator Example :**

**importjava.util.concurrent.ConcurrentHashMap**;

**importjava.util.Iterator**;

**publicclassFailSafeExample**

{

**publicstaticvoidmain**(String[]args)

{

ConcurrentHashMap<String,String>premiumPhone=

**new**ConcurrentHashMap<String,String>();

premiumPhone.put("Apple","iPhone");

premiumPhone.put("HTC","HTC one");

premiumPhone.put("Samsung","S5");

Iterator iterator=premiumPhone.keySet().iterator();

**while**(iterator.hasNext())

{

System.out.println(premiumPhone.get(iterator.next()));

premiumPhone.put("Sony","Xperia Z");

}

}

}

**Output :**

S5

HTC one

iPhone

**Recap: Difference between Fail Fast Iterator and Fail Safe Iterator**

|  |  |  |
| --- | --- | --- |
|  | **Fail Fast Iterator** | **Fail Safe Iterator** |
| Throw ConcurrentModification Exception | Yes | No |
| Clone object | No | Yes |
| Memory Overhead | No | Yes |
| Examples | HashMap,Vector,ArrayList,HashSet | CopyOnWriteArrayList, ConcurrentHashMap |

## Legacy Classes and Interfaces

* Legacy classes and interfaces are the classes and interfaces that formed the collections framework in the earlier versions of Java and how now been restructured or re-engineered. They are fully compatible with the framework.

## Deep Copy vs Shallow Copy

* class Foo {
* private Bar myBar;
* ...
* public Foo shallowCopy() {
* Foo newFoo = new Foo();
* newFoo.myBar = myBar;
* return newFoo;
* }
* public Foo deepCopy() {
* Foo newFoo = new Foo();
* newFoo.myBar = myBar.clone(); //or new Bar(myBar) or myBar.deepCopy or ...
* return newFoo;
* }
* }
* Foo myFoo = new Foo();
* Foo sFoo = myFoo.shallowCopy();
* Foo dFoo = myFoo.deepCopy();
* myFoo.myBar == sFoo.myBar => true
* myFoo.myBar.equals(sFoo.myBar) => true
* myFoo.myBar == dFoo.myBar => \*\*false\*\*

myFoo.myBar.equals(dFoo.myBar) => true