**Quick Reference to**

**Java Collections**

**Index**

[**1.** **Java Collection Diagram** 4](#_Toc466646162)

[**2.** **How to create an ArrayList and what happened internally?** 5](#_Toc466646163)

[**3.** **Which one correct?** 6](#_Toc466646164)

[**List<String>arrlstobj= new ArrayList<>(); OR** 6](#_Toc466646165)

[**List<String>arrlstobj= new ArrayList<String>();** 6](#_Toc466646166)

[**4.** **Is ArrayList al = new ArrayList(); is valid statement?** 6](#_Toc466646167)

[**5.** **Explain ArrayList size increment mechanism.** 6](#_Toc466646168)

[**6.** **Which copy technique internally used by the ArrayList class clone() method?** 6](#_Toc466646169)

[**7.** **How to create ArrayList from Array (Object[]) ?** 7](#_Toc466646170)

[**8.** **How to convert List to Array?** 7](#_Toc466646171)

[**9.** **What happens if ArrayList is concurrently modified while iterating the elements?** 7](#_Toc466646172)

[**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](#_Toc466646173)

[**11.** **What is the difference between Iterator and ListIterator?** 7](#_Toc466646174)

[**12.** **What is the difference between Iterator and Enumeration?** 8](#_Toc466646175)

[**13.** **What is the difference between ArrayList and Vector?** 9](#_Toc466646176)

[**14.** **What is the difference between ArrayList and LinkedList?** 9](#_Toc466646177)

[**15.** **Are there any differences between different methods in each of the following groups of element retrieve operations in LinkedList?** 10](#_Toc466646178)

[**16.** **What is the difference between Collection and Collections?** 10](#_Toc466646179)

[**17.** **What is difference between ordered and sorted collection?** 10](#_Toc466646180)

[**18.** **What is the difference between List and Set?** 10](#_Toc466646181)

[**19.** **What is hashCode() and how it is useful in Collection?** 11](#_Toc466646182)

[**20.** **HashSet vs. TreeSet vs. LinkedHashSet** 12](#_Toc466646183)

[**21.** **Are HashSet, TreeSet , & LinkedHashSet allow null key?** 12](#_Toc466646184)

[**22.** **Are HashSet, TreeSet , & LinkedHashSet synchronized?** 13](#_Toc466646185)

[**23.** **How to synchronize List, Set and Map elements?** 13](#_Toc466646186)

[**24.** **What is the difference between HashMap and Hashtable?** 13](#_Toc466646187)

[**25.** **What is hash-collision in Hashtable and how it is handled in Java?** 13](#_Toc466646188)

[**26.** **On what bases TreeMap and TreeSet is sorted?** 14](#_Toc466646189)

[**27.** **What is a default capacity of ArrayList, Vector, HashMap, Hashtable and Hashset?** 14](#_Toc466646190)

[**28.** **What is the difference between Comparable and Comparator?** 15](#_Toc466646191)

[**29.** **What is the use of Comparator/Comparable interfaces?** 16](#_Toc466646192)

[**30.** **for-each vs Iterator. Which will be the better option?** 16](#_Toc466646193)

[**31.** **HashMap Vs ConcurrentHashMap** 16](#_Toc466646194)

[**32.** **Difference between ConcurrentHashMap and Hashtable** 17](#_Toc466646195)

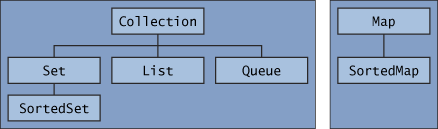
[**Other Concepts** 18](#_Toc466646196)

[**i)** **Fail Fast Vs Fail Safe Iterator in Java** 18](#_Toc466646197)

[**ii)** **Legacy Classes and Interfaces** 22](#_Toc466646198)

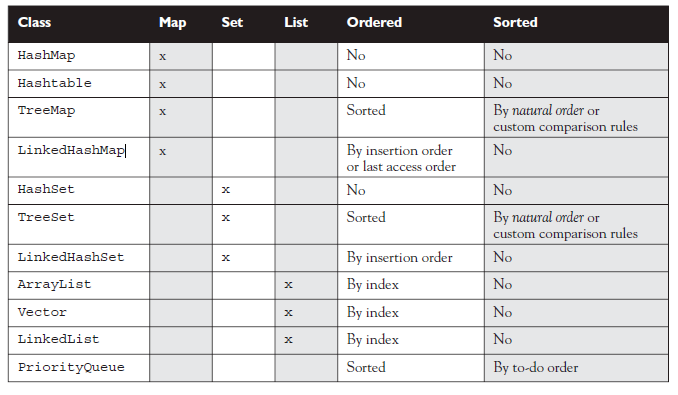
[**iii)** **Deep Copy vs Shallow Copy** 22](#_Toc466646199)

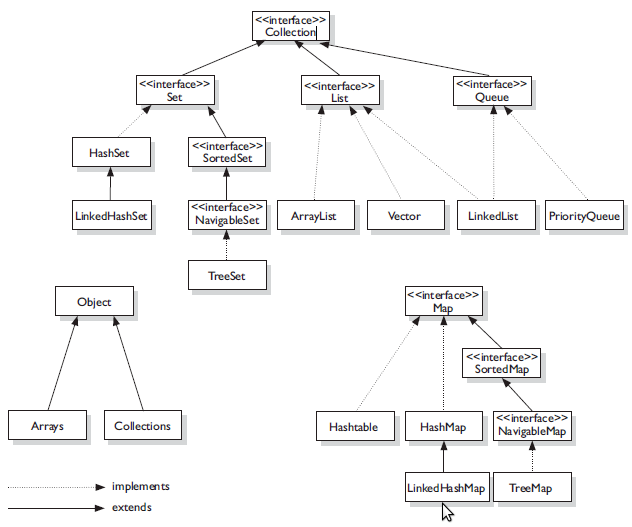
# **Java Collection Diagram**



|  | | **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 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 )

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.) |  |

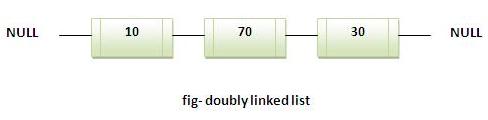
# **What is the difference between Iterator and Enumeration?**

|  |  |
| --- | --- |
| **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.

# **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:**

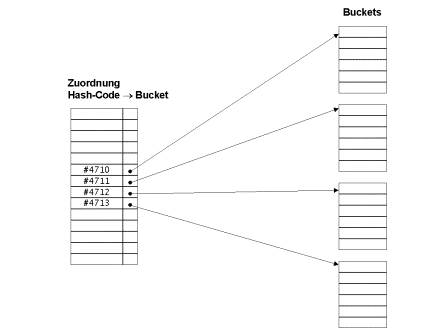
Zero

One

Two

Three

# **What is hashCode() and how it is useful in Collection?**

* 

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

# **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 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 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