**Auto boxing and auto unboxing**

* Integer x=new Interger(10);

Integer y=new Interger(10);

Syso(x==y) 🡪 false

* Integer x=new Interger(10);

Integer y=10;

Syso(x==y) 🡪 false

* Integer x=10

Integer y=10;

Syso(x==y) 🡪 true

* Integer x=10

Integer y=Integer.valueOf(10);

Syso(x==y) 🡪 true

* Integer x= Integer.valueOf(10);

Integer y=Integer.valueOf(10);

Syso(x==y) 🡪 true

* Integer x=1000

Integer y=1000;

Syso(x==y) 🡪 false (outside buffer range)

Integer {

Static{

Buffer array of size 1 byte is maintained

}

}

Internally to provide support for auto boxing a buffer of wrapper objects will be created at the time of wrapper class loading .By auto boxing if an object is required to be created jvm will first check in the buffer that object is present .If it is already available in the buffer it will reused if not present in the buffer new object will be created.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| -128  Integer Object1 | -127  Integer Object2 | ------------ | ----------- | ------------ | 126  Integer Object2 | 127  Integer Object2 |

The range of buffer for different data types

|  |  |  |
| --- | --- | --- |
| Data types | Start range | End range |
| Byte | always |  |
| Short | -128 | 127 |
| Integer | -128 | 127 |
| Long | -128 | 127 |
| Character | 0 | 127 |
| Boolean | Always |  |
|  |  |  |

Buffer concept is not there for Float and Double.

Internally auto boxing is implemented using valueOf() method so buffer concept is applicable to valueOf() also.

**Autoboxing vs widening**

Class Test{ o/p widening becoz it’s a older concept

Public static void m1(long i){

Syso(“autoboxing”);

}

Public static void m1(Integer i){

Syso(“widening”);

}

Public static void main(String[] args){

Int i=10;

m1(10)

}

}

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc. | Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc. |
| 2) Comparable **affects the original class** i.e. actual class is modified. | Comparator **doesn't affect the original class** i.e. actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is found in **java.lang** package. | Comparator is found in **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method. |

**Use Comparable:**

If the object is in your control.

If the comparing behaviour is the main comparing behaviour. Whenever we want to store only homogeneous elements and default natural sorting order required, we can go for class implementing comparable interface.

**Use Comparator :**

If the object is outside your control and you cannot make them implement Comparable.

When you want comparing behaviour different from the default (which is specified by Comparable) behaviour.

Whenever we want to store homogeneous and heterogeneous elements and we want to sort in default customized sorting order, we can go for comparator interface.

public class ObjectComparator implements Comparator{

public int compare(Object obj1, Object obj2) {

MyObject myObj1 = (MyObject) obj1;

MyObject myObj2 = (MyObject) obj2;

stringResult = myObj1.getString().compareTo(myObj2.getString());

if (stringResult == 0) {

// Strings are equal, sort by date

return myObj1.getDate().compareTo(myObj2.getDate());

}

else {

return stringResult;

}

}

}

The following points help you in deciding in which situations one should use Comparable and in which Comparator:

1) Code Availabilty

2) Single Versus Multiple Sorting Criteria

3) Arrays.sort() and Collection.sort()

4) As keys in SortedMap and SortedSet

5) More Number of classes Versus flexibility

6) Interclass comparisions

7) Natural Order

1) Code Availabilty  
  
The first thing to note is that while using Comparable you have to implement it in your class i.e *you need to change your class.*Example  
  
  public class Book implements Comparable{  
   ...  
  }  
  
For this, code of that class should be availaible to you . If you dont have access to the code of that class (say class belongs to third party), then there is no choice but to use Comparator because *Comparator does not need to change the original class.*  
  
2) Single Versus Multiple Sorting Criteria  
  
If you have only single sorting criteria to sort your elements then you can use Comparable but if you have more than one sorting criterias then you have to go for Comparator *also*.  
  
3) Arays.sort() and Collection.sort()  
  
Using Comparable has a advantage over Comparator. If your class implements Comparable then Arrays.sort() and Collections.sort() can sort its instances automatically. You do not need to write *separate comparators* and pass them to *overloaded*sort() as shown [here](http://java-journal.blogspot.in/2011/01/comparable-and-comparator-part-3.html).   
  
4) As keys in SortedMap and SortedSet  
  
This is another advantage of Comparable over Comparator. Objects which implement Comparable interface can be used as keys in a SortedMap( like TreeMap) or as elements in a SortedSet  (like TreeSet). Otherwise you have to write separate Comparator and pass it in the constructor of TreeMap.  
  
5) More Number of classes Versus flexibilty  
  
Use of Comparable does not require creation of extra classes while use of Comparator requires writing of *separate comparators*i.e *more number of classes*.   
  
But this has a advantage also. You can add as many sorting criteria later as you want or modify the existing ones without changing the class whose instances you are sorting.  
  
 Thus comparators provides flexibility while Comparable avoids extra classes.  
  
Note that you can also write Comparators as anonymous classes. In that case you can avoid separate comparators also.  
  
6) Interclass comparisions  
  
If you are going to compare *instances of same class* then you *should* use Comparable. Though we can also compare objects of different types while using Comparable as shown [here](http://java-journal.blogspot.in/2011/01/using-comparable-interface-to-compare.html) but we should avoid it.  
  
 If you are going to compare instances of different classes then you should use Comparator. But this was valid upto  pre Java 5, before the introduction of generics.  
  
With the introduction of generics syntax of Comparator has been changed from :  
  
public interface Comparator  to   public interface Comparator <T>  
  
and of compare() from:  
  
public int compare(Object o1, Object o2)  to  public int compare(T o1,T o2)  
  
As you are seeing in new syntax both o1 and o2 are of type T. If their types would be different as:  
  
compare(Integer o1, String o2)   
  
then this will give compile time error.  
  
Therefore if generic form of compare() is used then it compares objects of only same types.  
  
With non generic form of compare() you can still compare objects of different types.  
  
7) Natural Order  
  
If you are going to sort elements according to their natural order then you should use Comparable and for any other order different from natural order Comparator should be used.  
  
**Thus to answer above given questions:-**  
  
1) When to use Comparable and When to use Comparator?  
  
Ans:- When you are going to sort according to natural order, have single sort criteria and  have access to the class you would use Comparable.  
  
Otherwise,  
  
If you cannot change the class and have multiple sorting criteria use Comparator.  
  
  
2) Interviewer may give you some class and  some criteria  and ask you to write code to sort instances of that class based on that criteria.You have to decide there whether to use Comparable or to use Comparator. or even, not use any of them?  
  
Ans: For example,

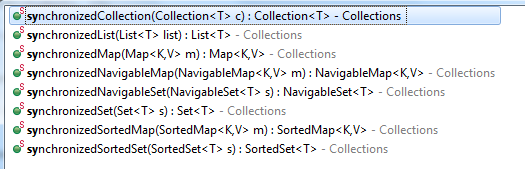
If you are given a class named Employee and asked to sort it's instances by employees*id*, then you should go Comparable as it seems its natural order.

But if you have to sort them by its salary or Date of Joining () etc.(which does not seem to be natural order of employee) then you can go for either Comparable or Comparator. Both are legal. If you would use Comparator then you have to write extra classes as comparator.  
  
And if you have to sort employees on id and *also* on salary and Date of Joining, then use Comparable for *id* ad create comparators for salary and  Date of joining.  
  
Suppose you are given list of objects of type *Integer*. Then which interface you will use? None, as Integer already implements Comparable. Just pass its array (or List) to Arrays.sort() (or Collections.sort()).  
  
3) What is the difference between Comparable and Comparator?  
  
  
Check here for this: [Difference between Comparable and Comparator Interfaces.](http://java-journal.blogspot.in/2010/12/difference-between-comparable-and.html)  
  
 4) Which is preferred Comparable or Comparator?  
  
Comparable has advantage that it avoids the creation of more number of classes and also elements implementing Comparable can be used directly in utility functions like  Arrays.sort()  and Collections.sort().   
  
While Comparator is more flexible. It has advantage that it avoids the changing the class you are going to sort and more sorting criterias can be added later.   
  
So in my view one should go for Comparable first for that sorting criteria *which is not going to change in future* and for additional criterias we can use Comparator in addition also.  
  
  
5) Why do you need Comparable if  Comparator can also sort things?  
  
Because Comparable :-  
  
1) makes it easy to use the elements implementing it in some utility functions and classes like TreeSet and TreeMap.  
  
2) Avoids creation of new classes.

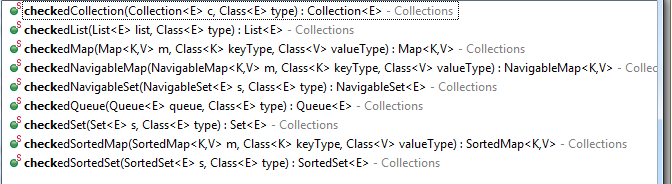
**Collections class**

Returns a synchronized (thread-safe) collection backed by the specified collection. In order to guarantee serial access, it is critical that **all** access to the backing collection is accomplished through the returned collection.

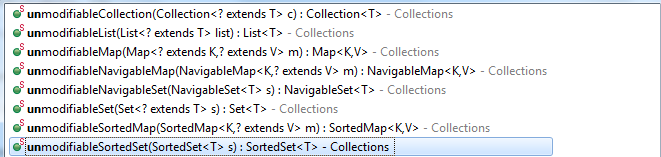
Returns a synchronized (thread-safe) map backed by the specified map. In order to guarantee serial access, it is critical that **all** access to the backing map is accomplished through the returned map.



Returns a dynamically typesafe view of the specified collection. Any attempt to insert an element of the wrong type will result in an immediate [ClassCastException](eclipse-javadoc:%E2%98%82=CoreJabasics/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_131%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections~checkedCollection~Ljava.util.Collection%5C%3CTE;%3E;~Ljava.lang.Class%5C%3CTE;%3E;%E2%98%82ClassCastException). Assuming a collection contains no incorrectly typed elements prior to the time a dynamically typesafe view is generated, and that all subsequent access to the collection takes place through the view, it is *guaranteed* that the collection cannot contain an incorrectly typed element.



Returns an unmodifiable view of the specified collection. This method allows modules to provide users with "read-only" access to internal collections. Query operations on the returned collection "read through" to the specified collection, and attempts to modify the returned collection, whether direct or via its iterator, result in an UnsupportedOperationException.



Sorts the specified list into ascending order, according to the [natural ordering](eclipse-javadoc:%E2%98%82=CoreJabasics/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_131%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections~sort~Ljava.util.List%5C%3CTT;%3E;%E2%98%82Comparable) of its elements. All elements in the list must implement the [Comparable](eclipse-javadoc:%E2%98%82=CoreJabasics/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_131%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections~sort~Ljava.util.List%5C%3CTT;%3E;%E2%98%82Comparable) interface. Furthermore, all elements in the list must be *mutually comparable* (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the list).

Sorts the specified list according to the order induced by the specified comparator. All elements in the list must be *mutually comparable* using the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the list).

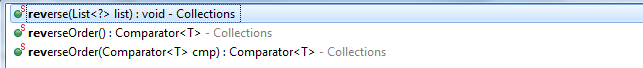


Reverses the order of the elements in the specified list.

Returns a comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the Comparable interface. (The natural ordering is the ordering imposed by the objects' own compareTo method.) This enables a simple idiom for sorting (or maintaining) collections (or arrays) of objects that implement the Comparable interface in reverse-natural-order. For example, suppose a is an array of strings. Then:

Arrays.sort(a, Collections.reverseOrder());

Returns a comparator that imposes the reverse ordering of the specified comparator. If the specified comparator is null, this method is equivalent to [reverseOrder()](eclipse-javadoc:%E2%98%82=CoreJabasics/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_131%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections~reverseOrder~Ljava.util.Comparator%5C%3CTT;%3E;%E2%98%82%E2%98%82reverseOrder%E2%98%82) (in other words, it returns a comparator that imposes the reverse of the natural ordering on a collection of objects that implement the Comparable interface).



Swaps the elements at the specified positions in the specified list. (If the specified positions are equal, invoking this method leaves the list unchanged.)



Copies all of the elements from one list into another. After the operation, the index of each copied element in the destination list will be identical to its index in the source list. The destination list must be at least as long as the source list. If it is longer, the remaining elements in the destination list are unaffected.

Checks whether two lists have any common element. It calls equals method of the object to check common elements.



Count of the specified object in the collection



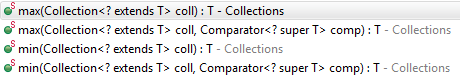
Searches the specified list for the specified object using the binary search algorithm. The list must be sorted into ascending order according to the [natural ordering](eclipse-javadoc:%E2%98%82=CoreJabasics/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_131%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections~binarySearch~Ljava.util.List%5C%3C+Ljava.lang.Comparable%5C%3C-TT;%3E;%3E;~TT;%E2%98%82Comparable) of its elements (as by the [sort(List)](eclipse-javadoc:%E2%98%82=CoreJabasics/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_131%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections~binarySearch~Ljava.util.List%5C%3C+Ljava.lang.Comparable%5C%3C-TT;%3E;%3E;~TT;%E2%98%82%E2%98%82sort%E2%98%82List) method) prior to making this call. If it is not sorted, the results are undefined. If the list contains multiple elements equal to the specified object, there is no guarantee which one will be found.

Searches the specified list for the specified object using the binary search algorithm. The list must be sorted into ascending order according to the specified comparator (as by the [sort(List, Comparator)](eclipse-javadoc:%E2%98%82=CoreJabasics/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_131%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections~binarySearch~Ljava.util.List%5C%3C+TT;%3E;~TT;~Ljava.util.Comparator%5C%3C-TT;%3E;%E2%98%82%E2%98%82sort%E2%98%82List%E2%98%82Comparator) method), prior to making this call. If it is not sorted, the results are undefined. If the list contains multiple elements equal to the specified object, there is no guarantee which one will be found.



Returns the maximum element of the given collection, according to the *natural ordering* of its elements. All elements in the collection must implement the Comparable interface. Furthermore, all elements in the collection must be *mutually comparable* (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the collection).

Returns the maximum element of the given collection, according to the order induced by the specified comparator. All elements in the collection must be *mutually comparable* by the specified comparator (that is, comp.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the collection).



**Failfast and Failsafe iterators**

### Brief Introduction To Fail Fast And Fail Safe Systems :

A system is called fail-fast if it is shut down immediately when an error occurred. These systems don’t carry on with the errors. They immediately stop operating when a fault is occurred in the system. The errors in the fail-fastsystems are immediately exposed. But, fail-safe systems are not like that. They don’t stop operating even when a fault is occurred in the system. They continue the operation by hiding the errors. They don’t expose the errors immediately. They carry on with the errors. Which one is the best system is always the most discussed topic in the system design field. In this post, we limit our discussion to Fail Fast and Fail Safe Iterators in java.

### Fail Fast And Fail Safe Iterators In Java :

Iterators in java give us the facility to traverse over the Collection objects. Iterators returned by the Collection are either fail-fast in nature or fail-safe in nature. Fail-Fast iterators immediately throw ConcurrentModificationException if a collection is modified while iterating over it. Where as Fail-Safe iterators don’t throw any exceptions if a collection is modified while iterating over it. Because, they operate on the clone of the collection, not on the actual collection. Let’s see Fail-Fast and Fail-Safe Iterators in detail.

### Fail-Fast Iterators In Java :

Fail-Fast iterators, returned by most of the collection types, doesn’t tolerate any structural modifications to a collection while iterating over it. (Structural modifications means add, remove or updating an element in the collection) They throw ConcurrentModificationException if a collection is structurally modified while iteration is going on the collection. But, they don’t throw any exceptions if the collection is modified by the iterator’s own methods like remove().

**How Fail-Fast Iterators Work?**

All Collection types maintain an internal array of objects ( Object[] ) to store the elements. Fail-Fast iterators directly fetch the elements from this array. They always consider that this internal array is not modified while iterating over its elements. To know whether the collection is modified or not, they use an internal flag called modCount which is updated each time a collection is modified. Every time when an Iterator calls the next() method, it checks the modCount. If it finds the modCount has been updated after this Iterator has been created, it throws ConcurrentModificationException.

The iterators returned by the ArrayList, Vector, HashMap etc are all Fail-Fast in nature.

[?](http://javaconceptoftheday.com/fail-fast-and-fail-safe-iterators-in-java-with-examples/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | import java.util.ArrayList;  import java.util.Iterator;    public class FailFastIteratorExample  {      public static void main(String[] args)      {          //Creating an ArrayList of integers            ArrayList<Integer> list = new ArrayList<Integer>();            //Adding elements to list            list.add(1452);            list.add(6854);            list.add(8741);            list.add(6542);            list.add(3845);            //Getting an Iterator from list            Iterator<Integer> it = list.iterator();            while (it.hasNext())          {              Integer integer = (Integer) it.next();                list.add(8457);      //This will throw ConcurrentModificationException          }      }  } |

**Output :**

[?](http://javaconceptoftheday.com/fail-fast-and-fail-safe-iterators-in-java-with-examples/)

|  |  |
| --- | --- |
| 1  2  3  4 | Exception in thread "main" java.util.ConcurrentModificationException      at java.util.ArrayList$Itr.checkForComodification(Unknown Source)      at java.util.ArrayList$Itr.next(Unknown Source)      at pack1.MainClass.main(MainClass.java:32) |

### Fail-Safe Iterators In Java :

Fail-Safe iterators don’t throw any exceptions if the collection is modified while iterating over it. Because, they iterate on the clone of the collection not on the actual collection. So, any structural modifications done on the actual collection goes unnoticed by these iterators. But, these iterators have some drawbacks. One of them is that it is not always guaranteed that you will get up-to-date data while iterating. Because any modifications to collection after the iterator has been created is not updated in the iterator. One more disadvantage of these iterators is that there will be additional overhead of creating the copy of the collection in terms of both time and memory.

Iterator returned by ConcurrentHashMap is a fail-safe iterator.

[?](http://javaconceptoftheday.com/fail-fast-and-fail-safe-iterators-in-java-with-examples/)

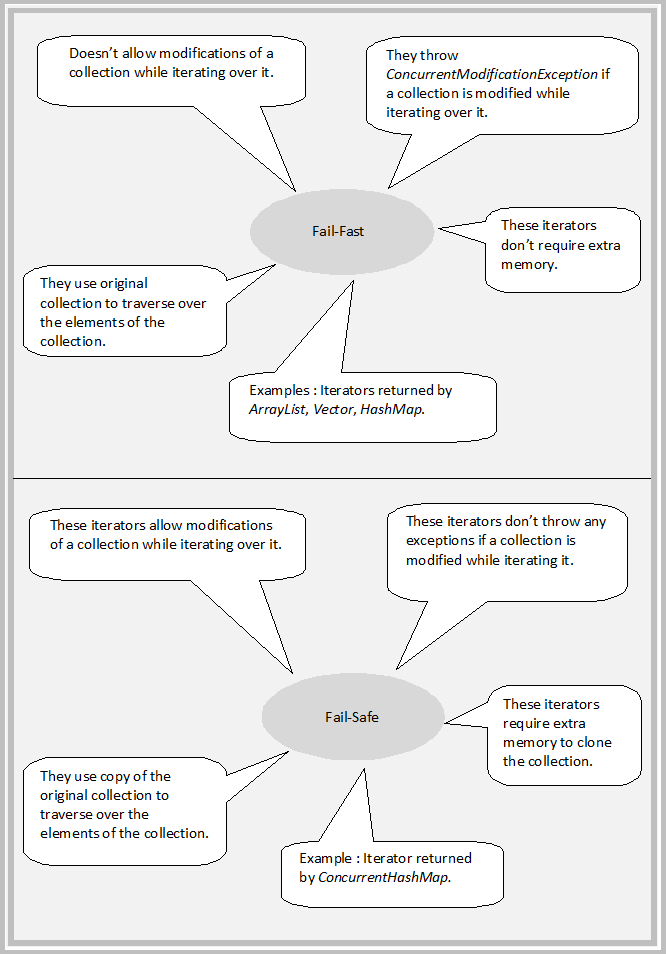
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | import java.util.Iterator;  import java.util.concurrent.ConcurrentHashMap;    public class FailSafeIteratorExample  {      public static void main(String[] args)      {          //Creating a ConcurrentHashMap            ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<String, Integer>();            //Adding elements to map            map.put("ONE", 1);            map.put("TWO", 2);            map.put("THREE", 3);            map.put("FOUR", 4);            //Getting an Iterator from map            Iterator<String> it = map.keySet().iterator();            while (it.hasNext())          {              String key = (String) it.next();                System.out.println(key+" : "+map.get(key));                map.put("FIVE", 5);     //This will not be reflected in the Iterator          }      }  } |

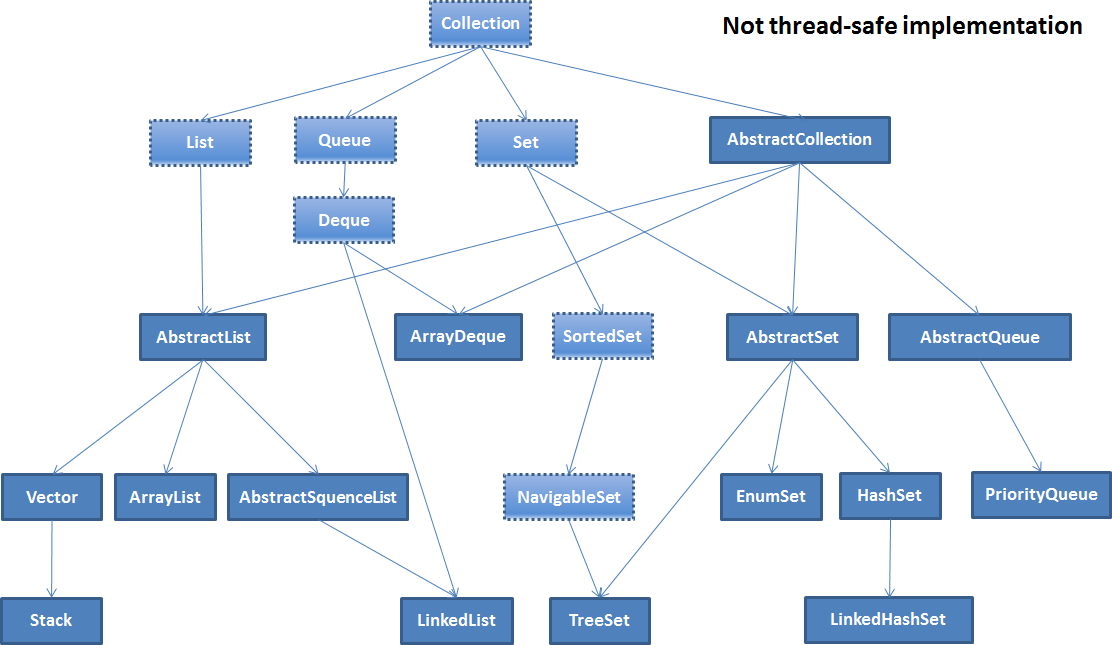
**Output :**

TWO : 2  
FOUR : 4  
ONE : 1  
THREE : 3

### Fail Fast Vs Fail Safe Iterators In Java :

|  |  |
| --- | --- |
| **Fail-Fast Iterators** | **Fail-Safe Iterators** |
| Fail-Fast iterators doesn’t allow modifications of a collection while iterating over it. | Fail-Safe iterators allow modifications of a collection while iterating over it. |
| These iterators throw ConcurrentModificationException if a collection is modified while iterating over it. | These iterators don’t throw any exceptions if a collection is modified while iterating over it. |
| They use original collection to traverse over the elements of the collection. | They use copy of the original collection to traverse over the elements of the collection. |
| These iterators don’t require extra memory. | These iterators require extra memory to clone the collection. |
| Ex : Iterators returned by ArrayList, Vector, HashMap. | Ex : Iterator returned by ConcurrentHashMap. |





ArrayList-10 0.5 load factor

Vector-10

HashSet-16

HashMap-16

HashTable-11

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).

**Arraylist and Vector**

|  |  |  |
| --- | --- | --- |
| **Property** | **Arraylist** | **Vector** |
| Synchronization | Non-synchronized | Synchronized |
| Legacy | Not legacy | legacy |
| Iterate | Iterator | Enumerator/Iterator |
| Performance | Fast becoz not synchronized | slow |
| Size increment | Increase by 50% | Doubles the size |
| Thread safe | No | yes |

**HashMap and HashTable**

|  |  |  |
| --- | --- | --- |
| **Property** | **HashMap** | **HashTable** |
| Synchronization | Non-synchronized | synchronized |
| Legacy | Not legacy | legacy |
| Iterate | Iterator | Enumerator/Iterator |
| Performance | Fast as not synchronized | slow |
| Thread safe | No | yes |
| Null key and value | One null key and multiple null value are allowed | Null key and values are not allowed throws null pointer exception |
| Inherits | AbstractMap | Dictonary |

**HashSet vs LinkedHashSet vs TreeSet**

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| Duplicate element | No | No | No |
| Synchronized | No | No | No |
| Clonable and serializable | Yes | Yes | yes |
| Type of iterators returned by these classes | Fail fast | Fail fast | Fail fast |
| How internally works? | Uses HashMap internally to store its elements | Uses LinkedHashMap | Uses TreeMap |
| Order of elements | Does not maintain any order of elements | It maintains the insertion order of the elements | TreeSet orders the element according to supplied comparator. If no comparator is supplied then elements will be sorted based on natural ascending order. |
| Performance | Better than LinkedHashSet and TreeSet | It’s slower than HashSet as it maintains linkedList to maintain the insertion order of the elements. | Less performance than both HashSet and LinkedHashSet as it needs to sort the elements after each insertion or deletion of elements. |

**List and Set**

|  |  |  |
| --- | --- | --- |
| **Property** | **List** | **Set** |
| Insertion order | Ordered Collection and maintains insertion order. I.e. upon displaying the list content it will display the elements in the same order in which they got inserted in to the list. | Only few implementation of set maintains insertion order.  I.e. LinkedHashSet |
| Allow duplicate values | Yes | No |
| Allow any number of null values | yes | Max it can contain one null value. In TreeSet it’s not even allowed. |
| Iteration | Can be iterated using Iterator/ListIterator/Enumerator  Enumerator for legacy class like Vector  Using ListIterator we can traverse both forward and reverse direction. | Iterator only |
| Has Legacy class | Vector | No |

**ArrayList and HashSet**

|  |  |  |
| --- | --- | --- |
| **Property** | **ArrayList** | **HashSet** |
| Interface | List | Set |
| Duplicate | Allowed | Not allowed |
| Insertion order | Maintains | Doesn’t maintain |
| Backed by | Array | HashMap |
| Retrieve object or remove by index | remove(index)  remove(object)  get(index)  add(object)  add(index,object)  set(index,object) | Not there. Only below method is present  Remove(object)  Add(object) |

**ArrayList and LinkedList**

Both are non-synchronized

Clonable are serializable

Both preserve insertion order

|  |  |  |
| --- | --- | --- |
| **Property** | **ArrayList** | **LinkedList** |
| Structure | Index based architecture where each element is associated with an index. | Elements in a LinkedList are stored as nodes each node contains three things   1. Previous node reference 2. Next node reference 3. The current element |
| Insertion and removal | Insertion or removal of element (in middle of ArrayList) is a costly process in as all other elements needs to be shifted.  If there is no space left in the current array while inserting new element then all the elements needs to be copied to the newer array with new size. | Insertion or deletion of element from any position is faster than ArrayList because there no need to shift the elements after every insertion or removal.  Only reference of previous and next elements are to be changed. |
| Retrieval or searching | Retrieval is faster as all elements are index based. | Retrieval of elements in LinkedList is very slow because it has to traverse through the elements either from the beginning or from the end to reach the element. |
| Random Access | ArrayList is of type Random Access. i.e. Elements can be access randomly. | LinkedList is not of type random access. Elements cannot be access randomly. You have to traverse from beginning or end to access the element. |
| Usage | Cannot be used as stack or Queue | Can be used as stack or Queue. |
| Memory usage | Less memory as elements are stored based on index. | More memory each node of linked list need to store the reference previous and next node along with the data. |
| Internal data structure | Dynamic array | Doubly linked list |
| Where to use? | If requirement is for more retrieval than insertion or deletion, then use ArrayList. | If requirement is for more insertion or deletion then retrieval then use linkedList. |

**Comparable and Comparator**

|  |  |  |
| --- | --- | --- |
| **Property** | **Comparable** | **Comparator** |
| Package | Java.lang | Java.util |
| Method | Int compareTo(Object o);  1) +ve integer if this is greater than o.  2) 0 if this object is equal to o  3) -ve integer if this is less than o. | Int compare(Object o1,Object o2)   1. +ve integer if o1 is greater than o2. 2. 0 if both are equal 3. –ve integer if o1 is less than o2. |
| Sorting logic | Sorting logic should be inside the class whose objects needs to be sorted.  Hence this is called natural ordering sorting. | Sorting logic in separate class. Hence the sorting logic can be written based on different attributes of the object as it is not natural ordering of objects. |
| Sorting | Collections.sort(list); | Collections.sort(list,new MyComparator()) |
|  |  |  |

**Cloning**

# **Difference between Deep and Shallow Copy in Java Object Cloning**

Shallow copy and deep copy is related with cloning process so before go into the deep of shallow and deep copy we need to understand what is clone in java. Clone is nothing but the process of copying one object to produce the exact object, which is not guaranteed. We all know in Java object is referred by reference we cannot copy one object directly to another object. So we have cloning process to achieve this objective. Now one question arises in mind why we need this process so the answer is whenever we need a local copy of the object to modify the object in some method but not in method caller.  So we can define Cloning as “**create a copy of object** “  .I think now we are somehow clear about the cloning but there  is more to it depending on how we are doing this copy, we can divide cloning into two types.

* Shallow Copy
* Deep Copy

Before going into the deep of shallow and deep copy we need to understand how we achieve cloning in java.

### How to Clone in java?

### In Java, everything is achieved through class, object and interface .By default no Java class support cloning but Java provide one interface called Cloneable, which is a [marker interface](http://javarevisited.blogspot.com/2012/01/what-is-marker-interfaces-in-java-and.html) and by implementing this interface we can make the duplicate copy of our object by calling clone() method of  java.lang.Object class.

This Method is protected inside the object class and Cloneable interface is  a marker interface and this method also throw  **CloneNotSupportedException**if we have not implemented thisinterface and try to call clone() method of Object class.  By default any clone () method gives the **shallow copy** of the object i.e. if we invoke **super.clone()** then it’s a shallow copy but if we want to **deep copy** we have to override the clone() method and make it public and give own definition of making copy of object. Now we let’s see  what is shallow and deep copy of object in Java programming language.

### Shallow Copy

Whenever we use default implementation of clone method we get shallow copy of object means it create new instance and copy all the field of object to that new instance and return it as **object type** we need to explicitly cast it back to our original object. This is shallow copy of the object. clone() method of the object class support shallow copy of the object. If the object contains primitive as well as non primitive or reference type variable In  shallow copy, the cloned object also refers to the same object to which the original object refers as only the object references gets copied and not the referred objects themselves. That's why the name shallow copy or shallow cloning in Java. If only primitive type fields or [Immutable objects](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html) are there then there is no difference between shallow and deep copy in Java.

### Deep Copy

Whenever we need own meaning of copy not to use default implementation we call it as deep copy, whenever we need deep copy of the object we need to implement according to our need. So for deep copy we need to ensure all the member class also implement the Clonable interface and override the clone() method of the object class. After that we override the clone() method in all those classes even in the classes where we have only primitive type members otherwise we would not be able to call the protected clone() method of Object class on the instances of those classes inside some other class. It’s typical restriction of the protected access.

### Difference between Shallow and Deep Copy in Java

I think now we know what is deep and shallow copy of object in Java, let see some difference between them so that we can get some more clarity on them.

* When we call Object.clone(), this method performs a shallow copy of object, by copying data field by field, and if we override this method and by convention first call super.clone(), and then modify some fields to "deep" copy, then we get deep copy of object. This modification is done to ensure that original and cloned object are independent to each other.
* In shallow copy main or parent object is copied, but they share same fields or children if fields are modified in one parent object other parent fields have automatic same changes occur, but in deep copy this is not the case.
* If our parent object contains only primitive value then shallow copy is good for making clone of any object because in new object value is copied but if parent object contains any other object then only reference value is copied in new parent object and both will point to same object so in that case according to our need we can go for deep copy.
* Deep copy is expensive as compare to shallow copy in terms of object creation, because it involves recursive copying of data from other mutable objects, which is part of original object.

This is all about deep copy and shallow copy of objects in Java. Now the question comes when we use shallow copy and when go for deep copy , so answer would be  simple that if the object has only primitive fields or Immutable objects, then obviously we will go for shallow copy, but if the object has references to other mutable objects, then based on the requirement, shallow copy or deep copy can be chosen. Means if the references are not modified anytime, then there is no point in going for deep copy, we can go for shallow copy. But if the references are modified often, then you need to go for deep copy. Again there is no hard and fast rule, it all depends on the requirement.

Hope this article will help to make clear about deep and shallow copy of cloning process.