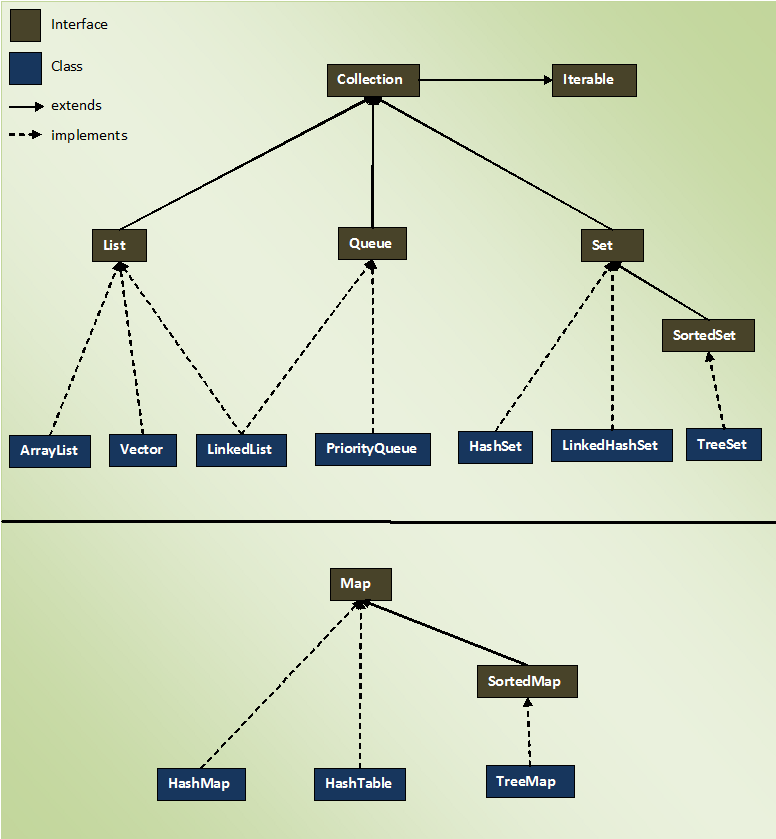
Class Hierarchy Of Collection Framework :

All classes and interfaces related to Collection Framework are placed in **java.util** package. **java.util.Collection** class is at the top of class hierarchy of Collection Framework. Below diagram shows the class hierarchy of collection framework.



The entire collection framework is divided into four interfaces.

**1) List**  —> It handles sequential list of objects. **ArrayList**, **Vector** and **LinkedList** classes implement this interface.

**2) Queue**  —> It handles special list of objects in which elements are removed only from the head. **LinkedList** and **PriorityQueue** classes implement this interface.

**3) Set**  —> It handles list of objects which must contain unique element. This interface is implemented by **HashSet**and **LinkedHashSet** classes and extended by **SortedSet** interface which in turn, is implemented by **TreeSet**.

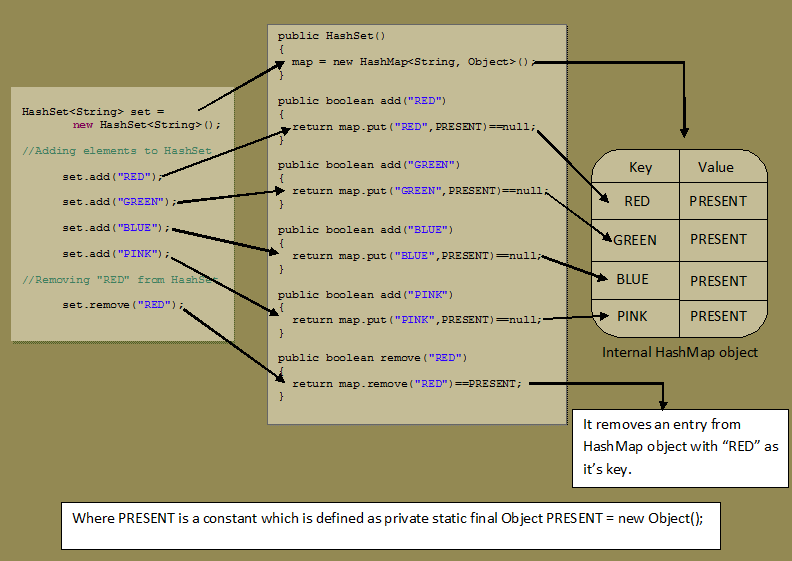
**4) Map**  —> This is the one interface in Collection Framework which is not inherited from Collection interface. It handles group of objects as Key/Value pairs. It is implemented by **HashMap** and **HashTable** classes and extended by **SortedMap** interface which in turn is implemented by **TreeMap**.

Three of above interfaces (List, Queue and Set) inherit from Collection interface. Although, Map is included in collection framework it does not inherit from Collection interface.

# [How HashSet Works Internally In Java?](http://javaconceptoftheday.com/how-hashset-works-internally-in-java/)

**HashSet** uses HashMap internally to store it’s objects. Whenever you create a HashSet object, one **HashMap** object associated with it is also created. This HashMap object is used to store the elements you enter in the HashSet. The elements you add into HashSet are stored as **keys** of this HashMap object. The value associated with those keys will be a **constant**.

See the below picture how above program works internally. You can observe that internal HashMap object contains elements of HashSet as keys and constant “PRESENT” as their value.



In the same manner, all methods of HashSet class process internally backing HashMap object to get the desired result. If you know how HashMap works, it will be easy for you to understand how HashSet works. You go through the source code of HashSet class once, you will get a clear picture about how HashSet works internally in Java.

**Set Implementation Internally in Java**  
Each and every element in the set is unique .  So that there is no duplicate element in set .  
  
So in java if we want to add elements in the set then we write code like this

**public** **class** **JavaHungry** {

**public** **static** **void** **main**(String[] args)

{

// TODO Auto-generated method stub

HashSet<Object> hashset = **new** HashSet<Object>();

hashset.add(**3**);

hashset.add("Java Hungry");

hashset.add("Blogspot");

System.out.println("Set is "+hashset);

}

}

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

Now let add duplicate element in the above code

**public** **class** **JavaHungry** {

**public** **static** **void** **main**(String[] args)

{

HashSet<Object> hashset = **new** HashSet<Object>();

hashset.add(**3**);

hashset.add("Java Hungry");

hashset.add("Blogspot");

hashset.add(**3**); // duplicate elements

hashset.add("Java Hungry"); // duplicate elements

System.out.println("Set is "+hashset);

}

}

*It will print the result* :       Set is [3, Java Hungry, Blogspot]  
  
  
Now , what happens internally when you pass duplicate elements in the  add() method of the Set object , It will return false and do not add to the HashSet , as the element is already present .So far so good .  
  
But the main problem arises that how it returns false . So here is the answer  
  
When you open the HashSet implementation of the add() method in Java Apis that is rt.jar , you will find the following code in it

**public** **class** **HashSet**<E>

**extends** AbstractSet<E>

**implements** Set<E>, Cloneable, java.io.Serializable

{

**private** **transient** HashMap<E,Object> map;

// Dummy value to associate with an Object in the backing Map

**private** **static** **final** Object PRESENT = **new** Object();

**public** **HashSet**() {

*map =* ***new*** *HashMap<>();*

}

// SOME CODE ,i.e Other methods in Hash Set

**public** **boolean** **add**(E e) {

**return** map.put(e, PRESENT)==**null**;

}

// SOME CODE ,i.e Other methods in Hash Set

}

So , we are achieving uniqueness in Set,internally in java  through HashMap . Whenever you create an object of HashSet it will create an object of HashMap as you can see in the italic lines in the above code .  
We already discussed   [How HashMap works internally  in java](http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html) .  
  
As we know in HashMap each key is unique . So what we do in the set is that we pass the argument in the add(Elemene E) that is E as a key in the HashMap . Now we need to associate some value to the key , so what Java apis developer did is to pass the Dummy  value that is ( new Object () ) which is referred by Object reference PRESENT .  
  
So , actually when you are adding a line in HashSet like  hashset.add(3)   what java does internally is that it will put that element E here 3 as a key in the HashMap(created during HashSet object creation) and some dummy value that is Object's object is passed as a value to the key .  
  
Now if you see the code of the HashMap put(Key k,Value V) method , you will find something like this  
  
 public V put(K key, V value) {  
//Some code  
}  
  
The main point to notice in above code is that put (key,value) will return  
  
1.  null , if key is unique and added to the map  
2.  Old Value of the key , if key is duplicate  
  
So , in HashSet add() method ,  we check the return value of map.put(key,value) method with null value   
i.e.  
  
   public boolean add(E e) {  
            return map.put(e, PRESENT)==null;  
       }  
  
So , if map.put(key,value) returns null ,then  
map.put(e, PRESENT)==null      will return true and element is added to the HashSet.  
  
  
  
So , if map.put(key,value) returns old value of the key ,then  
map.put(e, PRESENT)==null      will return false and element is  not added to the HashSet .

# 20 Java Collections Interview Questions

In java, collection interview questions are mostly asked by the interviewers. Here is the list of mostly asked collections interview questions with answers.

### 1) What is the difference between ArrayList and Vector?

|  |  |  |
| --- | --- | --- |
| **No.** | **ArrayList** | **Vector** |
| 1) | ArrayList is not synchronized. | Vector is synchronized. |
| 2) | ArrayList is not a legacy class. | Vector is a legacy class. |
| 3) | ArrayList increases its size by 50% of the array size. | Vector increases its size by doubling the array size. |

### 2) What is the difference between ArrayList and LinkedList?

|  |  |  |
| --- | --- | --- |
| **No.** | **ArrayList** | **LinkedList** |
| 1) | ArrayList uses a dynamic array. | LinkedList uses doubly linked list. |
| 2) | ArrayList is not efficient for manipulation because a lot of shifting is required. | LinkedList is efficient for manipulation. |
| 3) | ArrayList is better to store and fetch data. | LinkedList is better to manipulate data. |

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

* **Link** − Each link of a linked list can store a data called an element.
* **Next** − Each link of a linked list contains a link to the next link called Next.
* **Prev** − Each link of a linked list contains a link to the previous link called Prev.
* **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

### 3) What is the difference between Iterator and ListIterator?

Iterator traverses the elements in forward direction only whereas ListIterator traverses the elements in forward and backward direction.

|  |  |  |
| --- | --- | --- |
| **No.** | **Iterator** | **ListIterator** |
| 1) | Iterator traverses the elements in forward direction only. | ListIterator is an interface in **Collection API**. It extends Iterator interface. ListIterator traverses the elements in backward and forward directions both. |
| 2) | Iterator can be used in List, Set and Queue. | ListIterator can be used in List only. |
| 3) | Methods of Iterator:   * hasNext() * next() * remove() | Methods of ListIterator:   * add(E e) * hasNext() * hasPrevious() * next() * nextIndex() * previous() * previousIndex() * remove() * set(E e) () |
| 4) | 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. |
| 5) | 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. |

### 4) What is the difference between Iterator and Enumeration?

|  |  |  |
| --- | --- | --- |
| **No.** | **Iterator** | **Enumeration (Interface)** |
| 1) | Iterator can traverse legacy and non-legacy elements. | Enumeration can traverse only legacy elements. java.util.Enumeration |
| 2) | Iterator is fail-fast. | Enumeration is not fail-fast. |
| 3) | Iterator is slower than Enumeration. | Enumeration is faster than Iterator. |
| 4) | Using Iterator, you can remove an element of the collection while traversing it. | Using Enumeration, you can only traverse the collection. You can’t do any modifications to collection while traversing it. |
| 5) | Iterator is introduced from JDK 1.2 | Enumeration is introduced in JDK 1.0 |
| 6) | Iterator is safer and secured than Enumeration. | Enumeration is not safe and secured due to it’s fail-safe nature |
| 7) | Iterator is used to iterate most of the classes in the collection framework like ArrayList, HashSet, HashMap, LinkedList etc. | Enumeration is used to traverse the legacy classes like Vector, Stack and HashTable. |
| 8) | Iterator is fail-fast in nature. | Enumeration is fail-safe in nature. |
| 9) | : hasNext(), next() and remove() | Methods  o  [**hasMoreElements**](https://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/api/java.util.Enumeration.html#hasMoreElements())()  Tests if this enumeration contains more elements.  o  **[nextElement](https://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/api/java.util.Enumeration.html" \l "nextElement())**()  Returns the next element of this enumeration. |

### 5) What is the difference between List and Set?

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

### 6) What is the difference between HashSet and TreeSet?

1) [HashSet](https://beginnersbook.com/2013/12/hashset-class-in-java-with-example/" \o "HashSet Class in Java with example" \t "_blank) gives better performance (faster) than [TreeSet](https://beginnersbook.com/2013/12/treeset-class-in-java-with-example/" \o "TreeSet Class in Java with example" \t "_blank) for the operations like add, remove, contains, size etc. HashSet offers constant time cost while TreeSet offers log(n) time cost for such operations.

2) HashSet does not maintain any order of elements while TreeSet elements are sorted in ascending order by default.

**Similarities**:

1) Both HashSet and TreeSet does not hold duplicate elements, which means both of these are duplicate free.

2) If you want a sorted Set then it is better to add elements to HashSet and then [convert it into TreeSet](https://beginnersbook.com/2014/08/how-to-convert-a-hashset-to-a-treeset/) rather than creating a TreeSet and adding elements to it.

3) Both of these classes are non-synchronized that means they are not thread-safe and should be synchronized explicitly when there is a need of thread-safe operations.

### 7) What is the difference between Set and Map?

Set contains values only whereas Map contains key and values both.

### 8) What is the difference between HashSet and HashMap?

HashSet contains only values whereas HashMap contains entry(key,value). HashSet can be iterated but HashMap need to convert into Set to be iterated.

### 9) What is the difference between HashMap and TreeMap?

HashMap maintains **no order** but TreeMap maintains **ascending order**.

### 10) What is the difference between HashMap and Hashtable?

|  |  |  |
| --- | --- | --- |
| **No.** | **HashMap** | **Hashtable** |
| 1) | HashMap is not synchronized. | Hashtable is synchronized. |
| 2) | HashMap can contain one null key and multiple null values. | Hashtable cannot contain any null key or null value. |

### 11) 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.

### 12) What is the difference between Comparable and Comparator?

|  |  |  |
| --- | --- | --- |
| **No.** | **Comparable** | **Comparator** |
| 1) | Comparable provides only one sort of sequence. | Comparator provides multiple sort of sequences. |
| 2) | It provides one method named compareTo(). | It provides one method named compare(). |
| 3) | It is found in java.lang package. | it is found in java.util package. |
| 4) | If we implement Comparable interface, actual class is modified. | Actual class is not modified. |

### 13) What is the advantage of Properties file?

If you change the value in properties file, you don't need to recompile the java class. So, it makes the application easy to manage.

### 14) What does the hashCode() method?

The hashCode() method returns a hash code value (an integer number).

The hashCode() method returns the same integer number, if two keys (by calling equals() method) are same.

But, it is possible that two hash code numbers can have different or same keys.

### 15) Why we override equals() method?

The equals method is used to check whether two objects are same or not. It needs to be overridden if we want to check the objects based on property.

For example, Employee is a class that has 3 data members: id, name and salary. But, we want to check the equality of employee object on the basis of salary. Then, we need to override the equals() method.

### 16) 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){} |

### 17) What is the advantage of generic collection?

# Generics in Java

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

#### Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

**1) Type-safety :** We can hold only a single type of objects in generics. It doesn’t allow to store other objects.

**2) Type casting is not required:** There is no need to typecast the object.

If we use generic class, we don't need typecasting. It is typesafe and checked at compile time.

### 18) 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. Two different entries will be kept in a single hash bucket to avoid the collision.

### 19) What is the Dictionary class?

The Dictionary class provides the capability to store key-value pairs.

### 20) What is the default size of load factor in hashing based collection?

The default size of load factor is **0.75**. The default capacity is computed as initial capacity \* load factor. For example, 16 \* 0.75 = 12. So, 12 is the default capacity of Map.

# How to make ArrayList Read Only?

The read-only means unmodifiable view of Collection in which we can not perform any operation which will change the collection through add(), remove() or set() method. We can obtain read-only collection from the existing collection by calling Collections.unmodifiableCollection() method.

1. **public** **class** UnmodifiableArrayList {
2. **public** **static** **void** main(String[] args) {
3. List<String>fruitList = **new** ArrayList<String>();
5. fruitList.add("Mango");
6. fruitList.add("Banana");
7. fruitList.add("Apple");
8. fruitList.add("Strawberry");
9. fruitList.add("Pineapple");
11. List<String>unmodifiableList= Collections.unmodifiableList(fruitList);
12. unmodifiableList.add("INDIA");
13. System.out.println(fruitList);
14. }
15. }

Output:

Exception in thread "main"java.lang.UnsupportedOperationException

at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)

at collectionInterview.list.UnmodifiableArrayList.main(UnmodifiableArrayList.java:20)

Here we have converted the existing list fruitList to unmodifiable List. If we alter the "unmodifiableList", it will cause UnsupportedOperationException. We can still change the list with fruitList reference. like calling:

1. fruitList.add("INDIA");

# How to remove duplicates from ArrayList in Java?

To remove dupliates from ArrayList, we can convert it into Set. Since Set doesn't contain duplicate elements, it will have only unique elements.

Let's see an example to remove duplicates from ArrayList:

1. **public** **class** RemoveDuplicateArrayList {
2. **public** **static** **void** main(String[] args) {
3. List<String> l = **new** ArrayList<String>();
4. l.add("Mango");
5. l.add("Banana");
6. l.add("Mango");
7. l.add("Apple");
8. System.out.println(l.toString());
9. Set<String> s = **new** LinkedHashSet<String>(l);
10. System.out.println(s);
11. }
12. }

Output:

Before converting to set

[Mango, Banana, Mango, Apple]

After converting to set

[Mango, Banana, Apple]

# How to Synchronize ArrayList in Java?

We can use Collections.synchronizedList(List<T>) method to synchronize collections in java. The synchronizedList(List<T>) method is used to return a synchronized (thread-safe) list backed by the specified list.

1. **import** java.util.\*;
2. **public** **class** SyncronizeArrayList {
3. **public** **static** **void** main(String args[]) {
4. // Non Synchronized ArrayList
5. List<String> fruitList = **new** ArrayList<String>();
7. fruitList.add("Mango");
8. fruitList.add("Banana");
9. fruitList.add("Apple");
10. fruitList.add("Strawberry");
11. fruitList.add("Pineapple");
13. // Synchronizing ArrayList in Java
14. furitList = Collections.synchronizedList(fruitList);
16. // we must use synchronize block to avoid non-deterministic behavior
17. **synchronized** (fruitList) {
18. Iterator<String> itr = fruitList.iterator();
19. **while** (itr.hasNext()) {
20. System.out.println(itr.next());
21. }
22. }
23. }
24. }

Output:

Mango

Banana

Apple

Strawberry

Pineapple

# When to use ArrayList and LinkedList in Java

ArrayList provides constant time for search operation, so it is better to use ArrayList if searching is more frequent operation than add and remove operation. The LinkedList provides constant time for add and remove operations. So it is better to use LinkedList for manipulation.

ArrayList has O(1) time complexity to access elements via the get and set methods.

LinkedList has O(n/2) time complexity to access the elements.

LinkedLinked class implements Deque interface also, so you can get the functionality of double ended queue in LinkedList. The ArrayList class doesn't implement Deque interface.

In sort, ArrayList is better to access data wherease LinkedList is better to manipulate data. Both classes implements List interface.

### ArrayList Example

1. **import** java.util.\*;
2. **public** **class** ListExample {
3. **public** **static** **void** main(String[] args) {
4. //ArrayList is better to store and view data
5. List<String> list=**new** ArrayList<>();
6. list.add("ankit");
7. list.add("peter");
8. list.add("mayank");
10. System.out.println("Traversing ArrayList...");
11. **for**(String s:list){
12. System.out.println(s);
13. }
14. }
15. }

Output:

Traversing ArrayList...

ankit

peter

mayank

### LinkedList Example

1. **import** java.util.\*;
2. **public** **class** ListExample2 {
3. **public** **static** **void** main(String[] args) {
4. //LinkedList is better to manipulate data
5. List<String> list=**new** LinkedList<>();
6. list.add("ankit");
7. list.add("peter");
8. list.add("mayank");
9. System.out.println("After adding: "+list);
10. list.remove("peter");
11. System.out.println("After removing: "+list);
12. list.set(1,"vivek");
13. System.out.println("After changing: "+list);
14. }
15. }

Output:

After adding: [ankit, peter, mayank]

After removing: [ankit, mayank]

After changing: [ankit, vivek]

# How to Sort Java ArrayList in Descending Order

By using Collections.reverseOrder(Comparator<T>cmp) method, we can sort the collection in reverse order. The reverseOrder() method does the reversing on the basis of given Comparator. In case of null, it will reverse collection in natural ordering.

Let's see a simple example to sort the ArrayList in descending order.

**SmartPhone.java**

1. **import** java.util.Comparator;
2. **public** **class** SmartPhone {
3. String brand;
4. String model;
5. intprice;
6. intrating;
7. SmartPhone(String brand,String model,intprice, intrating){
8. **this**.brand = brand;
9. **this**.model = model;
10. **this**.price = price;
11. **this**.rating = rating;
13. }
14. **public** String getBrand() {
15. returnbrand;
16. }
17. **public** **void** setBrand(String brand) {
18. **this**.brand = brand;
19. }
20. **public** String getModel() {
21. returnmodel;
22. }
23. **public** **void** setModel(String model) {
24. **this**.model = model;
25. }
26. **public** **int** getPrice() {
27. returnprice;
28. }
29. **public** **void** setPrice(intprice) {
30. **this**.price = price;
31. }
32. **public** **int** getRating() {
33. returnrating;
34. }
35. **public** **void** setRating(intrating) {
36. **this**.rating = rating;
37. }
38. **public** String toString() {
39. **return**"SmartPhone [brand=" + brand + ", model=" + model + ", price=" + price + ", rating=" + rating + "]";
40. }
41. **public** **int** compareTo(SmartPhone sp) {
42. returnthis.price - sp.price;
44. }
45. }
46. **class** RatingComparator **implements** Comparator<SmartPhone> {
47. @Override
48. **public** **int** compare(SmartPhone obj1, SmartPhone obj2) {
49. **return** (obj1.rating<obj2.rating) ? -1 : (obj1.rating>obj2.rating) ? 1 : 0;
50. }
51. }

**ArrayListLearning.java**

1. **public** **class** ArrayListLearning {
2. @SuppressWarnings("unchecked")
3. **public** **static** **void** main(String[] args) {
5. List<SmartPhone> phoneList = **new** ArrayList<>();
6. SmartPhone ph1 = **new** SmartPhone("Apple", "6s", 50000, 10);
7. SmartPhone ph2 = **new** SmartPhone("lg", "pro2", 40000, 9);
8. SmartPhone ph3 = **new** SmartPhone("MI", "3s", 10000, 6);
9. SmartPhone ph4 = **new** SmartPhone("Letv", "le2", 12000, 7);
11. phoneList.add(ph1);
12. phoneList.add(ph2);
13. phoneList.add(ph3);
14. phoneList.add(ph4);
15. System.out.println("Actual List");
16. System.out.println(phoneList);
17. System.out.println("Sorting the list as comparator");
18. Collections.sort(phoneList, **new** RatingComparator());
20. System.out.println(phoneList);
21. System.out.println("Reversing the Comparator sorting");
22. Comparator<SmartPhone> cmp = Collections.reverseOrder(**new** RatingComparator());
24. Collections.sort(phoneList, cmp);
25. System.out.println("Printing the reverse list");
26. System.out.println(phoneList);
27. }
29. }

Output:

Actual List

[SmartPhone [brand=Apple, model=6s, price=50000, rating=10],

SmartPhone [brand=lg, model=pro2, price=40000, rating=9],

SmartPhone [brand=MI, model=3s, price=10000, rating=6],

SmartPhone [brand=Letv, model=le2, price=12000, rating=7]]

Sorting the list as comparator

[SmartPhone [brand=MI, model=3s, price=10000, rating=6],

SmartPhone [brand=Letv, model=le2, price=12000, rating=7],

SmartPhone [brand=lg, model=pro2, price=40000, rating=9],

SmartPhone [brand=Apple, model=6s, price=50000, rating=10]]

Reversing the Comparator sorting

Printing the reverse list

[SmartPhone [brand=Apple, model=6s, price=50000, rating=10],

SmartPhone [brand=lg, model=pro2, price=40000, rating=9],

SmartPhone [brand=Letv, model=le2, price=12000, rating=7],

SmartPhone [brand=MI, model=3s, price=10000, rating=6]]

# Difference between length of array and size() of ArrayList in Java

ArrayList doesn't have length() method, the size() method of ArrayList provides the number of objects available in the collection.

Array has length property which provides the length or capacity of the Array. It is the total space allocated during the intialization of the array.

1. **import** java.util.ArrayList;
2. **public** **class** LengthVsSize {
3. **public** **static** **void** main(String[] args) {
4. //creating array of 10 elements
5. **int** arr[]=**new** **int**[10];
6. //storing 2 elements
7. arr[0]=10;
8. arr[1]=12;
9. //printing length of array
10. System.out.println(arr.length);//10
12. //Creating ArrayList
13. ArrayList<String> list=**new** ArrayList<String>();
14. //storing 2 elements
15. list.add("ankit");
16. list.add("nippun");
17. //printing size of ArrayList
18. System.out.println(list.size());//2
19. }
20. }

Output:

10

2