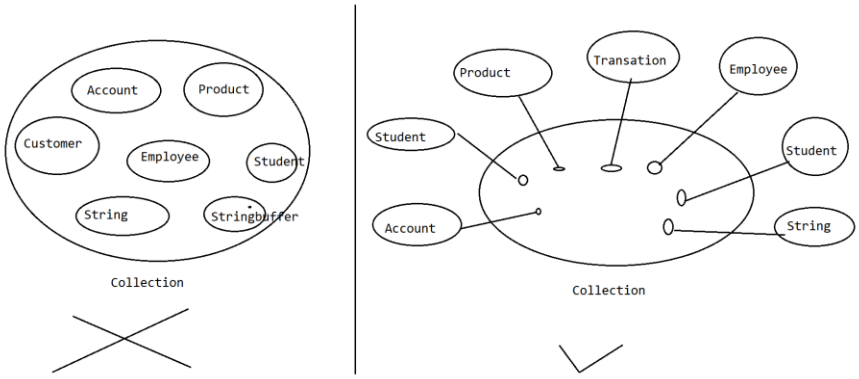
Collection Framework
—----------------------
Collection is an object, it is able to manage a group of other objects as a single unit.



Note: Collection objects are able to allow only Object reference values, they are not able to allow primitive values directly, if we provide primitive values to the Collection objects , internally JVM will convert these primitive values to the Wrapper class object and JVM will add these wrapper class objects to the Collection object.

Q)In Java applications, to represent a group of other objects already we have arrays then what is the requirement to use Collections?
—---
Ans:
—---
1. Arrays are available in fixed size nature, arrays are unable to allow the elements over its size, if we add any element over the array size then JVM will raise an exception like java.lang.ArrayIndexOutOfBoundsException.
EX:

```
Student[] stds = new Student[3];
stds[0] = new Student();
stds[1] = new Student();
stds[2] = new Student();
stds[3] = new Student(); —-> ArrayIndexOutOufBoundsException
```

Collections are available in dynamically growable nature, Collections are able to allow the elements over its size by increasing its size dynamically.

```
ArrayList al = new ArrayList(3);
```

```
al.add(new Student());
al.add(new Student());
al.add(new Student());
al.add(new Student()); –> No Exception
```

2. Arrays are able to allow only Homogeneous elements, arrays are not allowing heterogeneous elements. If we add heterogeneous elements to the arrays then we are able to get the compilation error like Incompatible Types.

EX:

```
Student[] stds = new Student[3];
stds[0] = new Student();
stds[1] = new Employee(); –-> Incompatible Types Error
```

Collections are able to allow heterogeneous elements.

EX:

```
Arraylist al = new ArrayList(3);
al.add(new Student());
al.add(new Employee()); –---> No Error
al.add(new Product()); –-----> No Error
```

3. Arrays do not have predefined library support to perform the operations like sorting and searching,...., where in arrays we have to write logic explicitly to perform these operations.

EX:

```
int[] smarks = {89, 56, 76, 98, 59, 77};
–-- Explicit Logic to sort all the elements–--
```

Collections have predefined support to perform the operations like Sorting and searching, where it is not required to write logic explicitly.

EX:

```
TreeSet ts = new TreeSet();
ts.add(89);
ts.add(56);
ts.add(76);
ts.add(98);
ts.add(59);
ts.add(77);
Sopln(ts); [56, 59, 76, 77, 89, 98]
```

4. Collections are more flexible than arrays.

5. Arrays are less API dependent, it is very simple to perform debugging and testing.

Collections are more API dependent, it is difficult to perform debugging and testing.
6. Arrays are able to represent the same type of elements, it improves Typed ness in java applications and it allows Type Safe operations.

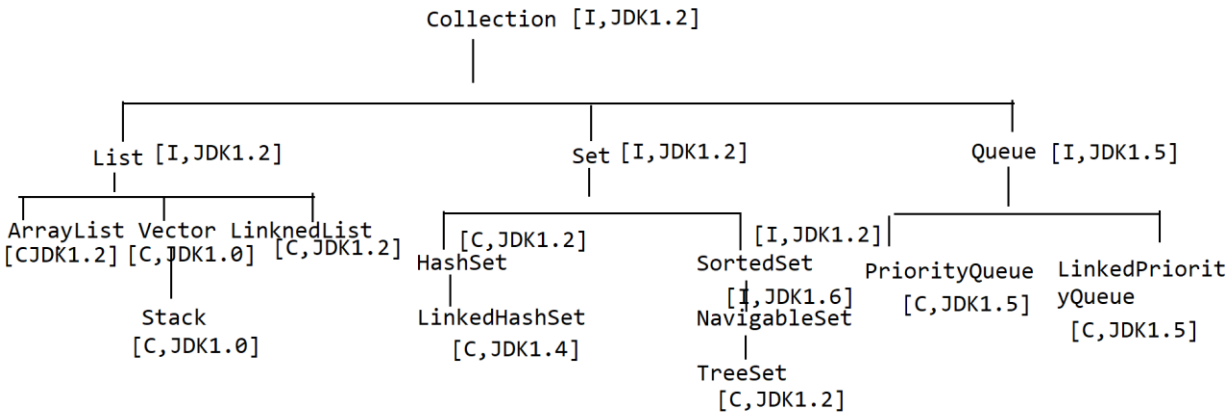
Collections are able to allow different types of elements, it reduces typed ness in java applications and it allows type unsafe operations.

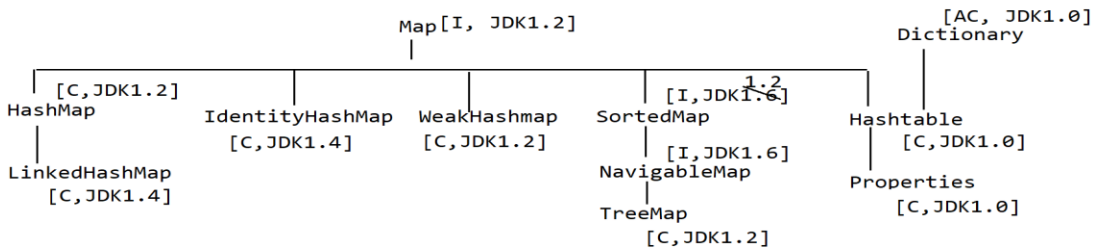
7. If we know the number of elements which we are representing in java applications, it is suggestible to use Arrays instead of Collections, because Arrays are able to provide very good performance when compared with the Collections.

To represent Collection objects , JAVA has provided a set of predefined classes and interfaces called “Collection Framework”.

As part of the Collection Framework, JAVA has provided all the predefined classes and interfaces in a package “java.util”.

To provide all the Collections in java applications, Java has provided the following list of predefined classes and interfaces in java.util package.





Q)What are the differences between Collection and Map?

—----------

Ans:

—---

Collection is an object, it is able to manage all other objects as individual elements.

EX: To represent a set of Employee objects individually there we must use a Collection object.

Map is an object, it is able to represent all the other objects in the form of Key-Value pairs.

EX: To represent Phone Number and Customer Name in Telephone directory/Dictionary we will use Map object.

EX: To represent Student register that contains Student roll number and Student Name we will use Map object.

Q)What are the differences between List and Set?

—----------

Ans:

—---

1. List is index based, where all the elements are arranged as per the indexes.

Set is not index based, where all the elements are not arranged as per the indexes, but all the elements are arranged as per the element’s Hashcode values.

2. Lists are following Insertion order.

Sets are not following Insertion Order.

Note:In Set implementations, LinkedHashSet is following Insertion order.

3. Lists are not following Sorting order.

By default Sets are not following Sorting order.
Note: In Set implementations, SortedSet, NavigableSet and TreeSet are following Sorting Order.

4. Lists are able to allow duplicate elements.

Sets are not allowing Duplicate elements.

5. Lists are able to allow any number of null elements.

Sets are able to allow only one null element.
Note: The Set implementations like SortedSet, NavigableSet and TreeSet are not allowing even single null element, If we add any null value to the SortedSet, NavigableSet and TreeSet then JVM will raise an exception like java.lang.NullPointerException.

6. Lists are able to allow heterogeneous elements.

By default Sets are able to allow heterogeneous elements.
Note: The Set implementations like SortedSet, NavigableSet and TreeSet are not allowing heterogeneous elements, they are able to allow only Homogeneous elements. If we add any Heterogeneous element then JVM will raise an exception like java.lang.ClassCastException.

Collection:
—----------

1. It represents a group of other objects as a single entity.
2. It is a root interface for all the collection objects.
3. It was provided by JAVA in its JDK1.2 version.
4. It has defined a number of methods in order to implement these methods by all the Collection implementation classes.

Methods:

1. public boolean add(Object obj)
   It can be used to add the specified element to the Collection object. If the element is added to the Collection object successfully then add() method will return true value, if the element is not added to the Collection object then add() method will return false value.

EX:

```
import java.util.HashSet;
import java.util.Set;

public class Main {
public static void main(String[] args) {
Set set = new HashSet();
System.out.println(set.add("AAA"));
System.out.println(set.add("BBB"));
System.out.println(set.add("CCC"));
System.out.println(set.add("DDD"));
System.out.println(set);
System.out.println(set.add("AAA"));
System.out.println(set.add("BBB"));
System.out.println(set.add("CCC"));
System.out.println(set.add("DDD"));
System.out.println(set);

}
}
```

```
true
true
true
true
[AAA, CCC, BBB, DDD]
false
false
false
false
[AAA, CCC, BBB, DDD]
```

2. public boolean addAll(Collection c):
   It is able to add all the elements of the Specified Collection to the present Collection object, if at least one element is added then it will return true value, if no element is added then it will return false value.

EX:

```
import java.util.Collection;
import java.util.HashSet;

public class Main {
public static void main(String[] args) {

Collection collection1 = new HashSet();
collection1.add("AAA");
collection1.add("BBB");
collection1.add("CCC");
collection1.add("DDD");
System.out.println("Collection-1 :
"+collection1);

Collection collection2 = new HashSet();
System.out.println("Collection-2 Before ADD :
"+collection2);
System.out.println(collection2.addAll(collection1
));
System.out.println(collection2.addAll(collection1
));
System.out.println("Collection-2 After ADD :
"+collection2);

Collection collection3 = new HashSet();
collection3.add("AAA");
collection3.add("BBB");
collection3.add("XXX");
collection3.add("YYY");
System.out.println(collection2.addAll(collection3
));
System.out.println("Collection-2 After Add :
"+collection2);
```

```
}
}
```

```
Collection-1 : [AAA, CCC, BBB, DDD]
Collection-2 Before ADD : []
true
false
Collection-2 After ADD : [AAA, CCC, BBB, DDD]
true
Collection-2 After Add : [AAA, CCC, BBB, DDD, YYY, XXX]
```

3. public boolean remove(Object obj):
   It can be used to remove the specified element element from the Collection. It will return true value when the element is removed successfully, it will return false value when the element is not removed.
   EX:

```
import java.util.ArrayList;
import java.util.Collection;

public class Main {
public static void main(String[] args) {
Collection collection = new ArrayList();
collection.add("AAA");
collection.add("BBB");
collection.add("CCC");
collection.add("DDD");
System.out.println(collection);
System.out.println(collection.remove("BBB"));
System.out.println(collection.remove("CCC"));
System.out.println(collection);
System.out.println(collection.remove("BBB"));
System.out.println(collection.remove("CCC"));
System.out.println(collection);

}
```

```
}
```

```
[AAA, BBB, CCC, DDD]
true
true
[AAA, DDD]
false
false
[AAA, DDD]
```

4. public boolean removeAll(Collection c)
   It can be used to remove all the elements of the specified collection from the present Collection object. If at least one element is removed from the Collection then removeAll() method will return true value, if no element is removed from the Collection then it will return false value.
   EX:

```
import java.util.ArrayList;
import java.util.Collection;

public class Main {
public static void main(String[] args) {
Collection collection1 = new ArrayList();
collection1.add("AAA");
collection1.add("BBB");
collection1.add("CCC");
collection1.add("DDD");
System.out.println(collection1);
Collection collection2 = new ArrayList();
collection2.add("AAA");
collection2.add("BBB");
System.out.println(collection2);
System.out.println(collection1.removeAll(collecti
on2));
System.out.println(collection1.removeAll(collecti
on2));
Collection collection3 = new ArrayList();
```

```
collection3.add("CCC");
collection3.add("XXX");
System.out.println(collection3);
System.out.println(collection1.removeAll(collecti
on3));


}
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB]
true
false
[CCC, XXX]
true
```

5. public boolean contains(Object obj)
   It can be used to check whether the specified object is available or not in the present Collection. If the specified element is available in the Collection object then it will return true value, if the specified element is not available in the Collection object then it will return false value.
   EX:

```
import java.util.ArrayList;
import java.util.Collection;

public class Main {
public static void main(String[] args) {
Collection collection1 = new ArrayList();
collection1.add("AAA");
collection1.add("BBB");
collection1.add("CCC");
collection1.add("DDD");
System.out.println(collection1);
System.out.println(collection1.contains("BBB"));
System.out.println(collection1.contains("ZZZ"));
```

```
}
}
```

```
[AAA, BBB, CCC, DDD]
true
false
```

6. public boolean containsAll(Collection c)
   It can be used to check whether all the elements of the specified Collection are available or not available in the present Collection object, If all the elements of the Specified Collection are available in the present Collection then it will return true value, if no element of the specified Collection is not available in the present Collection then it will return false value, If at least one element of the specified Collection is not matched with the Present Collection then it will return false value.

   EX:

```
import java.util.ArrayList;
import java.util.Collection;

public class Main {
public static void main(String[] args) {
Collection collection1 = new ArrayList();
collection1.add("AAA");
collection1.add("BBB");
collection1.add("CCC");
collection1.add("DDD");
System.out.println(collection1);
Collection collection2 = new ArrayList();
collection2.add("AAA");
collection2.add("BBB");
System.out.println(collection2);
System.out.println(collection1.containsAll(collec
tion2));
Collection collection3 = new ArrayList();
```

```
collection3.add("XXX");
collection3.add("YYY");
System.out.println(collection3);
System.out.println(collection1.containsAll(collec
tion3));
Collection collection4 = new ArrayList();
collection4.add("AAA");
collection4.add("BBB");
collection4.add("XXX");
collection4.add("YYY");
System.out.println(collection4);
System.out.println(collection1.containsAll(collec
tion4));

}
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB]
true
[XXX, YYY]
false
[AAA, BBB, XXX, YYY]
false
```

7. public boolean retainAll(Collection c):
   It can be used to remove all elements from the present Collection except the elements which are matched with the specified Collection elements. If no element from the present Collection is removed then it will return false value, if at least one element is removed from the present Collection then it will return true value.
   EX:

```
import java.util.ArrayList;
import java.util.Collection;

public class Main {
public static void main(String[] args) {
```

```
Collection collection1 = new ArrayList();
collection1.add("AAA");
collection1.add("BBB");
collection1.add("CCC");
collection1.add("DDD");
collection1.add("EEE");
collection1.add("FFF");
System.out.println(collection1);
Collection collection2 = new ArrayList();
collection2.add("BBB");
collection2.add("CCC");
collection2.add("DDD");
System.out.println(collection2);
System.out.println(collection1.retainAll(collecti
on2));
System.out.println(collection1);
}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
[BBB, CCC, DDD]
true
[BBB, CCC, DDD]
```

EX:

```
import java.util.ArrayList;
import java.util.Collection;

public class Main {
public static void main(String[] args) {
Collection collection1 = new ArrayList();
collection1.add("AAA");
collection1.add("BBB");
collection1.add("CCC");
collection1.add("DDD");
```

```
collection1.add("EEE");
collection1.add("FFF");
System.out.println(collection1);
Collection collection2 = new ArrayList();
collection2.add("BBB");
collection2.add("CCC");
collection2.add("DDD");
collection2.add("XXX");
collection2.add("YYY");
System.out.println(collection2);
System.out.println(collection1.retainAll(collecti
on2));
System.out.println(collection1);
}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
[BBB, CCC, DDD, XXX, YYY]
true
[BBB, CCC, DDD]
```

EX:

```
import java.util.ArrayList;
import java.util.Collection;

public class Main {
public static void main(String[] args) {
Collection collection1 = new ArrayList();
collection1.add("AAA");
collection1.add("BBB");
collection1.add("CCC");
collection1.add("DDD");
collection1.add("EEE");
collection1.add("FFF");
System.out.println(collection1);
Collection collection2 = new ArrayList();
collection2.add("AAA");
```

```
collection2.add("BBB");
collection2.add("CCC");
collection2.add("DDD");
collection2.add("EEE");
collection2.add("FFF");
System.out.println(collection2);
System.out.println(collection1.retainAll(collection2));
System.out.println(collection1);
}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
[AAA, BBB, CCC, DDD, EEE, FFF]
false
[AAA, BBB, CCC, DDD, EEE, FFF]
```

8. public int size():
   It will return the size of the Collection.

9. public void clear():
   It will remove all elements from the Collection object.

10. public boolean isEmpty():
    It is able to check whether the Collection is empty or not, if the Collection is empty then it will return true value, if the Collection is not empty then it will return false value.

EX:

```
import java.util.ArrayList;
import java.util.Collection;

public class Main {
public static void main(String[] args) {
Collection collection1 = new ArrayList();
collection1.add("AAA");
collection1.add("BBB");
collection1.add("CCC");
collection1.add("DDD");
collection1.add("EEE");
collection1.add("FFF");
```

```
System.out.println(collection1);
System.out.println(collection1.size());
System.out.println(collection1.isEmpty());
collection1.clear();
System.out.println(collection1.isEmpty());
}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
6
false
true
```

11. public Object[] toArray():

It can be used to convert all the elements from the Collection object to the Object[].

EX:

```
import java.util.ArrayList;
import java.util.Collection;

public class Main {
public static void main(String[] args) {
Collection collection1 = new ArrayList();
collection1.add("AAA");
collection1.add("BBB");
collection1.add("CCC");
collection1.add("DDD");
collection1.add("EEE");
collection1.add("FFF");
System.out.println(collection1);
Object[] objArray = collection1.toArray();
for(Object obj: objArray){
System.out.println(obj);
}
}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
AAA
```

BBB
CCC
DDD
EEE
FFF

List:
------
1. It is a child interface to Collection interface.
2. It was introduced in JDK1.2 version.
3. It is able to represent a group of elements as a single entity.

4. It is index based.
5. It allows Duplicate elements.
6. It follows insertion order.
7. It does not follow Sorting Order.
8. It allows any number of null elements.
9. It allows Heterogeneous elements.

Methods:
1. public void add(int index, Object obj)
   It can be used to add the specified element at the specified index in the List object.
   EX:

```
import java.util.ArrayList;
import java.util.List;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
System.out.println(list);
list.add(3, "DDD");
list.add(4, "EEE");
list.add(5, "FFF");
System.out.println(list);
}
}
```

[AAA, BBB, CCC]
[AAA, BBB, CCC, DDD, EEE, FFF]

2. public boolean addAll(int index, Collection c):
   It can be used to add all elements of the Specified Collection to the present List object at the specified index.If at least one element is added then it will return true value, if no element is added then it will return false value.
   Note: If we provide an index value which is outside the range of the List then it will raise an exception like "java.lang.IndexOutOfBoundsException".

   EX:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
list.add("DDD");
list.add("EEE");
System.out.println(list);
Collection collection = new HashSet();
collection.add("XXX");
collection.add("YYY");
collection.add("ZZZ");
System.out.println(collection);
System.out.println(list.addAll(2, collection));
System.out.println(list);

}
}
```

   [AAA, BBB, CCC, DDD, EEE]
   [YYY, XXX, ZZZ]

true
[AAA, BBB, YYY, XXX, ZZZ, CCC, DDD, EEE]

3. public Object get(int index):
   It is able to return an element in the form of Object which is available at the specified index value, here if the specified index value does not exist in the List then it will raise an exception like java.lang.indexOutofBoundsException.
   EX:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
list.add("DDD");
list.add("EEE");
System.out.println(list);
System.out.println(list.get(2));
System.out.println(list.get(3));
//System.out.println(list.get(5)); --->
IndexOutOfBoundsException
//System.out.println(list.get(20)); -->
IndexOutOfboundsException

}
}
```

[AAA, BBB, CCC, DDD, EEE]
CCC
DDD

4. public Object set(int index, Object obj):
   It can be used to add the specified element at the specified index value.If any element exists at the specified index then it will replace the existing element with the specified element. If no element exists at the specified index then it will raise an exception.

   EX:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
list.add("DDD");
list.add("EEE");
System.out.println(list);
System.out.println(list.set(2,"XXX"));
System.out.println(list);


}
}
```

   [AAA, BBB, CCC, DDD, EEE]
   CCC
   [AAA, BBB, XXX, DDD, EEE]

Q)What are the differences between add() method and set() method?
—--
Ans:
—----

1. add() method is mainly for inserting a new element in the list at the specified index value, if any element exists at the specified index then that existing element will be adjusted to the next index and the new element will be added to the specified index, if we provide the

index value which is after the last index then add() method will add the specified element as last element without raising any exception.
EX:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
list.add("DDD");
list.add("EEE");
System.out.println(list);
list.add(2, "XXX");
System.out.println(list);
list.add(6,"YYY");
System.out.println(list);


}
}
```

```
[AAA, BBB, CCC, DDD, EEE]
[AAA, BBB, XXX, CCC, DDD, EEE]
[AAA, BBB, XXX, CCC, DDD, EEE, YYY]
```

set() method is mainly for override the existed element at the specified index, if any element is existed at the specified index then set() method will remove the existing element and add the new element at the specified index, if no element is existed at the specified index then set() method will raise an exception like IndexOutOfBoundsException.
EX:

```
import java.util.ArrayList;
import java.util.Collection;
```

```
import java.util.HashSet;
import java.util.List;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
list.add("DDD");
list.add("EEE");
System.out.println(list);
System.out.println(list.set(2, "XXX"));
//System.out.println(list.set(5,"YYY"));--
>IndexOutOfBoundsException
System.out.println(list);


}
}
```

```
[AAA, BBB, CCC, DDD, EEE]
CCC
[AAA, BBB, XXX, DDD, EEE]
```

5. public int indexOf(Object obj):
It can be used to return an index value where the first occurrence of the specified element.

6. public int lastindexOf(Object obj):
It can be used to return an index value where the last occurrence of the specified element.

Note: in the above two methods, if the specified element does not exist then indexOf() and lastindexOf() methods will return -1 value.

EX:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
```

```
import java.util.List;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
list.add("DDD");
list.add("BBB");
list.add("EEE");
list.add("FFF");
list.add("BBB");
System.out.println(list);
System.out.println(list.indexOf("BBB"));
System.out.println(list.lastIndexOf("BBB"));
}
}
```

```
[AAA, BBB, CCC, DDD, BBB, EEE, FFF, BBB]
1
7
```

EX:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
list.add("DDD");
list.add("BBB");
list.add("EEE");
list.add("FFF");
```

```
list.add("BBB");
System.out.println(list);
System.out.println(list.indexOf("ZZZ"));
System.out.println(list.lastIndexOf("ZZZ"));
}
}
```

```
[AAA, BBB, CCC, DDD, BBB, EEE, FFF, BBB]
-1
-1
```

ArrayList:

———————

1. It was introduced in JDK1.2 version.
2. It is not a Legacy Collection.
3. It is a direct implementation class to List interface.

4. It is index based.
5. It allows duplicate elements.
6. It follows insertion order.
7. It does not follow Sorting Order.
8. It allows heterogeneous elements.
9. It allows any number of null elements.

10. Its initial capacity is 10 elements.
11. Its incremental capacity value is currentCapacity*3/2+1
12. Its internal data structure is “Resizable Array”.
13. It is suggestible for the frequent retrieval operations.

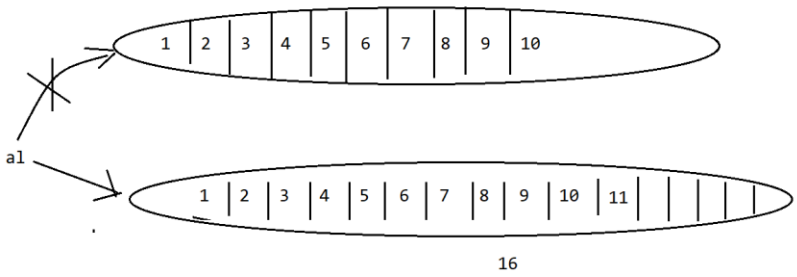
14. It is not a Synchronized Resource.
15. No method is synchronized in ArrayList.
16. It allows multiple threads at a time.
17. It follows parallel execution.
18. It reduced execution time.
19. It improves Application Performance.
20. It is not guaranteed for the Data Consistency.
21. It is not a Threadsafe Resource.

Resizable Array:

When we create an ArrayList in Java applications , its initial capacity is 10 elements, that is, ArrayList will create an array of 10 size, if we add elements to the ArrayList, automatically all these elements will come to the internal Array, if we add 11th [Exceeding max capacity]

element to the ArrayList then ArrayList will create another new array with the capacity 16 [CurrentCapacity*3/2+1], ArrayList will copy all the elements from the existing array to new array and ArrayList will refer new Array, where the existed array is eligible for Garbage Collection.

```
ArrayList al = new Arraylist();
for(int i = 1; i <= 10; i++){
al.add(i);
}

al.add(11);      10*3/2+1 = 16
```



Constructors:

—----------

1. public ArrayList():

It can be used to create an empty ArrayList object with the initial capacity 10 .

```
import java.util.ArrayList;

public class Main {
public static void main(String[] args) {
ArrayList al = new ArrayList();
System.out.println(al);
}
}
```

[]

2. public ArrayList(int capacity):

It can be used to create an empty ArrayList object with the specified capacity value.

```
import java.util.ArrayList;

public class Main {
public static void main(String[] args) {
ArrayList al = new ArrayList(20);
```

```
System.out.println(al);
}
}
```

[]

3. public ArrayList(Collection c):
It can be used to create an ArrayList object with all the elements of the specified Collection.

In the above Constructor, in place of Collection we can pass any Collection implementation class object as parameter, it will be used mainly for converting elements from a Collection[List, Set, Queue] to the ArrayList.

EX:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.PriorityQueue;

public class Main {
public static void main(String[] args) {
Collection collection1 = new ArrayList();
collection1.add("AAA");
collection1.add("BBB");
collection1.add("CCC");
ArrayList al1 = new ArrayList(collection1);
System.out.println(al1);

Collection collection2 = new HashSet();
collection2.add(100);
collection2.add(200);
collection2.add(300);
ArrayList al2 = new ArrayList(collection2);
System.out.println(al2);

Collection collection3 = new PriorityQueue();
collection3.add("XXX");
collection3.add("YYY");
collection3.add("ZZZ");
ArrayList al3 = new ArrayList(collection3);
```

```
System.out.println(al3);

}
}
```

```
[AAA, BBB, CCC]
[100, 200, 300]
[XXX, YYY, ZZZ]
```

EX:

```
import java.util.ArrayList;

public class Main {
public static void main(String[] args) {
ArrayList al = new ArrayList();
al.add("AAA");
al.add("BBB");
al.add("CCC");
al.add("DDD");
System.out.println(al);
al.add("BBB");
al.add("CCC");
System.out.println(al);
al.add(10);
al.add(22.22f);
System.out.println(al);
al.add(null);
al.add(null);
System.out.println(al);
}
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD, BBB, CCC]
[AAA, BBB, CCC, DDD, BBB, CCC, 10, 22.22]
[AAA, BBB, CCC, DDD, BBB, CCC, 10, 22.22, null, null]
```

Vector:
—------

1. It was provided by Java in its JDK1.0 version.
2. It is a Legacy Collection.
3. It is a direct implementation class to List interface.

4. It is index based.
5. It allows Duplicate elements.
6. It follows Insertion order.
7. It does not follow Sorting order.
8. It allows Heterogeneous elements.
9. It allows any number of null elements.

10. Its initial capacity is 10 elements.
11. Its internal Incremental capacity is 2*CurrentCapacity
12. Its internal Data Structure is “Resizable Array”.
13. It is suggestible when we have frequent retrieval operations.

14. It is a synchronized Collection.
15. It has synchronized methods.
16. It allows only one thread at a time.
17. It follows Sequential execution.
18. It increases application execution time.
19. It reduces application performance.
20. It gives guarantee for the Data Consistency.
21. It is a Threadsafe resource.

Constructors:

1. public Vector():
   It can be used to create an empty Vector object with the initial capacity value 10.
   EX:

```
import java.util.ArrayList;
import java.util.Vector;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector();
System.out.println(vector);
System.out.println(vector.capacity());
}
}
```

```
[]
10
```

2. public Vector(int capacity):
   It can be used to create an empty Vector object with the specified capacity value.

   EX:

```java
import java.util.ArrayList;
import java.util.Vector;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector(20);
System.out.println(vector);
System.out.println(vector.capacity());
}
}
```

```
[]
20
```

3. public Vector(int capacity, int incrementalRatio)
   It can be used to create an empty Vector object with the specified initial capacity value and with the specified incrementalCapacity value.
   EX:

```java
import java.util.ArrayList;
import java.util.Vector;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector(5,5);
System.out.println(vector);
System.out.println(vector.capacity());
}
}
```

```
[]
5
```

EX:

```
import java.util.ArrayList;
import java.util.Vector;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector();
System.out.println(vector.capacity());
for(int i = 1; i <= 10; i++){
vector.add(i);
}
vector.add(11);
System.out.println(vector.capacity());
for(int i = 12; i <= 20; i++){
vector.add(i);
}
vector.add(21);
System.out.println(vector.capacity());
}
}
```

```
10
20
40
```

EX:

```
import java.util.ArrayList;
import java.util.Vector;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector(5,5);
System.out.println(vector.capacity());
for(int i = 1; i <= 5; i++){
vector.add(i);
}
vector.add(6);
System.out.println(vector.capacity());
for(int i = 7; i <= 10; i++){
```

```
vector.add(i);
}
vector.add(11);
System.out.println(vector.capacity());
}
}
```

```
5
10
15
```

4. public Vector(Collection c)
   It can be used to create a Vector object with all the elements of the Specified Collection object.
   Note: It can be used to convert the elements from all the types of Collections like List, Set and Queue to the Vector.

   EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
Vector vector1 = new Vector(list);
System.out.println(vector1);

Set set = new HashSet();
set.add(100);
set.add(200);
set.add(300);
Vector vector2 = new Vector(set);
System.out.println(vector2);

Queue queue = new PriorityQueue();
queue.add("XXX");
queue.add("YYY");
```

```
queue.add("ZZZ");
Vector vector3 = new Vector(queue);
System.out.println(vector3);
}
}
```

```
[AAA, BBB, CCC]
[100, 200, 300]
[XXX, YYY, ZZZ]
```

Methods:

—---------

1. public void addElement(Object obj)
   It can be used to add the specified element to the vector object.
   EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector();
vector.addElement("AAA");
vector.addElement("BBB");
vector.addElement("CCC");
vector.addElement("DDD");
System.out.println(vector);

}
}
```

```
[AAA, BBB, CCC, DDD]
```

2. public Object elementAt(int index)
   It can be used to get an element that exists at the specified index value.

   EX:

```
import             *;

public class Main {
public static void main(String[] args) {
```

```
Vector vector = new Vector();
vector.addElement("AAA");
vector.addElement("BBB");
vector.addElement("CCC");
vector.addElement("DDD");
System.out.println(vector);
System.out.println(vector.elementAt(2));
System.out.println(vector.elementAt(3));

}
}
```

3. public void removeElementAt(int index)
   It can be used to remove an element that existed at the specified index value.
   EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector();
vector.addElement("AAA");
vector.addElement("BBB");
vector.addElement("CCC");
vector.addElement("DDD");
System.out.println(vector);
vector.removeElementAt(2);
System.out.println(vector);

}
}
```

   [AAA, BBB, CCC, DDD]
   [AAA, BBB, DDD]

4. public boolean removeElement(Object obj):
   It can be used to remove the specified element from the Vector. If the specified element exists in the Vector then it will move the specified element and it will return true value, if the

specified element does not exist in the Vector then it will return false value.
EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector();
vector.addElement("AAA");
vector.addElement("BBB");
vector.addElement("CCC");
vector.addElement("DDD");
System.out.println(vector);
System.out.println(vector.removeElement("BBB"));
System.out.println(vector);
System.out.println(vector.removeElement("ZZZ"));
System.out.println(vector);

}
}
```

```
[AAA, BBB, CCC, DDD]
true
[AAA, CCC, DDD]
false
[AAA, CCC, DDD]
```

5. public void removeAllElements():
   It can be used to remove all elements from the Vector.
   EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector();
vector.addElement("AAA");
vector.addElement("BBB");
vector.addElement("CCC");
vector.addElement("DDD");
System.out.println(vector);
```

```
vector.removeAllElements();
System.out.println(vector);

}
}
```

```
[AAA, BBB, CCC, DDD]
[]
```

6. public Object firstElement():
   It can be used to get the First element from the Vector.

7. public Object lastElement():
   It can be used to get the last element from the Vector.

   EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector();
vector.addElement("AAA");
vector.addElement("BBB");
vector.addElement("CCC");
vector.addElement("DDD");
System.out.println(vector);
System.out.println(vector.firstElement());
System.out.println(vector.lastElement());

}
}
```

```
[AAA, BBB, CCC, DDD]
AAA
DDD
```

8. public void insertElementAt(Object element, int index):
   It can be used to insert the specified element at the specified index value. If the specified index value is immediately after the last index value then it will raise an exception, it will add the element as the last element to the Vector. If the index value

is ion outside range of the index of the Vector then it will raise an exception like java.lang.ArrayIndexOutOfBoundsException

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector();
vector.addElement("AAA");
vector.addElement("BBB");
vector.addElement("CCC");
vector.addElement("DDD");
System.out.println(vector);
vector.insertElementAt("XXX", 2);
System.out.println(vector);
//vector.insertElementAt("YYY", 10); --->
ArrayIndexOutOfBoundsException


}
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, XXX, CCC, DDD]
```

Q)What are the differences between ArrayList and Vector?

—----------

Ans:

—---

1. ArrayList was introduced in JDK1.2 version.
   Vector was introduced in JDK1.0 version

2. ArrayList is not a Legacy Collection.
   Vector is a Legacy Collection

3. In the case of ArrayList, it is not possible to get capacity value explicitly, because ArrayList does not have a capacity() method.

   In the case of Vector , there is a capacity() method to find capacity value of the Vector explicitly.

4. ArrayList has the incremental capacity ratio CurrentCapacity*3/2+1
   Vector has the incremental capacity ratio 2*currentCapacity

5. ArrayList does not allow us to provide our own incremental capacity value.
   Vector allows us to provide our own incremental capacity value.

6. ArrayList is not a synchronized Collection.
   Vector is a Synchronized Collection.

7. ArrayList does not have synchronized methods.
   Vector has synchronized methods.

8. ArrayList allows more than one thread at a time.
   Vector allows only one thread at a time.

9. ArrayList follows parallel execution of the threads.
   Vector allows sequential execution of the threads.

10. Arraylist reduces the application execution time.
    Vector increases application execution time.

11. ArrayList improves application performance.
    Vector reduces application performance.

12. ArrayList is not giving guarantees for the data consistency.
    Vector is giving guarantee for the data Consistency.

13. ArrayList is not Thread Safe
    Vector is Thread safe.

Note: In Java applications, it is suggestible to ArrayList when compared with the Vector.

Stack:
—-----
Stack is a Collection, it is a subclass to the Vector, It is able to arrange all the elements as per positions starts from 1 to size , it follows LIFO[Last In First Out] to retrieve all the elements.

Constructor:
public Stack():
It is able to create an empty Stack.

```java
import java.util.*;

public class Main {
public static void main(String[] args) {
Stack stack = new Stack();
System.out.println(stack);
}
}
```

[]

Methods:

1. public void push(Object element):
   It inserted the specified element at the TOP of the stack.

2. public Object pop():
   It is able to read and remove the top of the stack.

3. public Object peek():
   It is able to read the top of the stack, it will not remove the top of the stack.

4. public int search(Object obj):
   It can be used to search for the specified element in the stack, if the element exists in the stack then search() method will return its position, if the element does not exist in the stack then search() method will return -1 value.

   EX:

```java
import java.util.*;

public class Main {
public static void main(String[] args) {
Stack stack = new Stack();
stack.push("AAA");
stack.push("BBB");
stack.push("CCC");
stack.push("DDD");
System.out.println(stack);
System.out.println(stack.peek());
System.out.println(stack.pop());
```

```
        System.out.println(stack);
        System.out.println(stack.search("BBB"));
        System.out.println(stack.search("ZZZ"));

        }
        }
```

```
[AAA, BBB, CCC, DDD]
DDD
DDD
[AAA, BBB, CCC]
2
-1
```

LinkedList:

—----------

1. It was introduced in JDK1.2 version.
2. It is not a Legacy Collection.
3. It is a direct implementation class to List.

4. It is index based.
5. It allows duplicate elements
6. It follows insertion order.
7. It does not follow Sorting Order.
8. It allows Heterogeneous elements.
9. It allows any number of null values.

10. Its internal data Structure is Double LinkedList.
11. It is suggestible for the frequent insertions and deletions.

12. It is not a synchronized Collection.
13. It does not have synchronized methods.
14. It allows more than one thread at a time.
15. It follows parallel execution.
16. It reduces application execution time.
17. It improves application performance.
18. It is not giving guarantees for the Data Consistency.
19. It is not Thread safe.

Constructors:

1. Public LinkedList():
   It can be used to create an empty LinkedList object.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
LinkedList ll = new LinkedList();
System.out.println(ll);

}
}
```

[]

2. public LinkedList(Collection c) :
   It can be used to create a LinkedList object with all the elements of the specified Collection.

   Note: It is mainly for the internal conversion purposes.
   EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add(10);
list.add(22.22f);
LinkedList ll1 = new LinkedList(list);
System.out.println(ll1);

Set set = new HashSet();
set.add("BBB");
set.add(20);
set.add(33.33f);
LinkedList ll2 = new LinkedList(set);
System.out.println(ll2);

Queue queue = new PriorityQueue();
```

```
queue.add("CCC");
queue.add("DDD");
queue.add("EEE");
LinkedList ll3 = new LinkedList(queue);
System.out.println(ll3);

}
}
```

```
[AAA, 10, 22.22]
[BBB, 20, 33.33]
[CCC, DDD, EEE]
```

Methods:

1. public void addFirst(Object element):
   It can be used to add the specified element as the first element.

2. public void addLast(Object element):
   It can be used to add the specified element as Last element.

3. public Object getFirst():
   It can be used to get the first element.

4. public Object getLast():
   It can be used to get the last element.

5. public void removeFirst():
   It can be used to remove the first element from the linkedList.

6. public void removeLast():
   It can be used to remove the last element from the LinkedList.

7. public boolean removeFirstOccurrence(Object element):
   It can be used to remove the first occurrence of the specified element.

8. public boolean removeLastOccurrence(Object element):
   It can be used to remove the last occurrence of the specified element.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
LinkedList ll = new LinkedList();
ll.add("AAA");
ll.add("BBB");
ll.add("CCC");
ll.add("DDD");
System.out.println(ll);
ll.addFirst("XXX");
ll.addLast("YYY");
System.out.println(ll);
ll.removeFirst();
ll.removeLast();
System.out.println(ll);
System.out.println(ll.getFirst());
System.out.println(ll.getLast());
}
}
```

```
[AAA, BBB, CCC, DDD]
[XXX, AAA, BBB, CCC, DDD, YYY]
[AAA, BBB, CCC, DDD]
AAA
DDD
```

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
LinkedList ll = new LinkedList();
ll.add("AAA");
```

```
    ll.add("BBB");
    ll.add("CCC");
    ll.add("DDD");
    ll.add("BBB");
    ll.add("FFF");
    ll.add("BBB");
    System.out.println(ll);
    ll.removeFirstOccurrence("BBB");
    System.out.println(ll);
    ll.removeLastOccurrence("BBB");
    System.out.println(ll);
    }
    }
```

```
[AAA, BBB, CCC, DDD, BBB, FFF, BBB]
[AAA, CCC, DDD, BBB, FFF, BBB]
[AAA, CCC, DDD, BBB, FFF]
```

Q) What are the differences between ArrayList and LinkedList?

—----------

Ans:

—---

1. ArrayList has Resizable Array as an internal Data Structure.
   LinkedList has Double LinkedList as an internal Data Structure.

2. Arraylist is suggestible for the frequent retrieval operations.
   LinkedList is suggestible for the frequent insertion and deletion operations.

Cursors / Iterators in Collection Framework:

—----------

Consider the following program,

```
ArrayList al = new ArrayList();
al.add(“AAA”);
al.add(“BBB”);
al.add(“CCC”);
al.add(“DDD”);
System.out.println(al);
```

OP: [AAA,BBB,CCC,DDD]
If we pass any Collection object reference variable as a parameter to System.out.println() method then JVM will access toString() method over the provided reference variable.

In Java , all the Collection classes have overridden the Object class provided toString() method in such a way that to return a String contains all the elements of the respective collection by enclosing with [].

```
import java.sql.Array;
import java.util.*;

public class Main {
public static void main(String[] args) {
ArrayList al = new ArrayList();
al.add("AAA");
al.add("BBB");
al.add("CCC");
al.add("DDD");
System.out.println(al);
}
}
```

[AAA, BBB, CCC, DDD]

In the Collection Framework, we don't want to display all the elements of the Collection object at a time through toString() method, we want to read all the elements of the Collections individually [one after another, not all at a time]

In the Collections, if we want to read all the elements individually then Collection Framework has provided an alternative in the form of CURSORS or Iterators.

Collection Framework has provided the following three types of CURSORs.

1. Enumeration
2. Iterator
3. ListIterator

Enumeration:
—----------

1. It was introduced in the JDK1.0 version.
2. It is a Legacy Cursor, it will be used for only Legacy Collections.
3. It is able to perform only read operations while iterating elements, it is not able to perform the operations like remove, insert and replace,...
4. It is represented in the form of an interface like java.util.Enumeration

To use Enumeration in Java applications we have to use the following steps.

1. Create Enumeration Object:
   To create an Enumeration Object we have to use the following method from all the Legacy Collections like java.util.Vector.

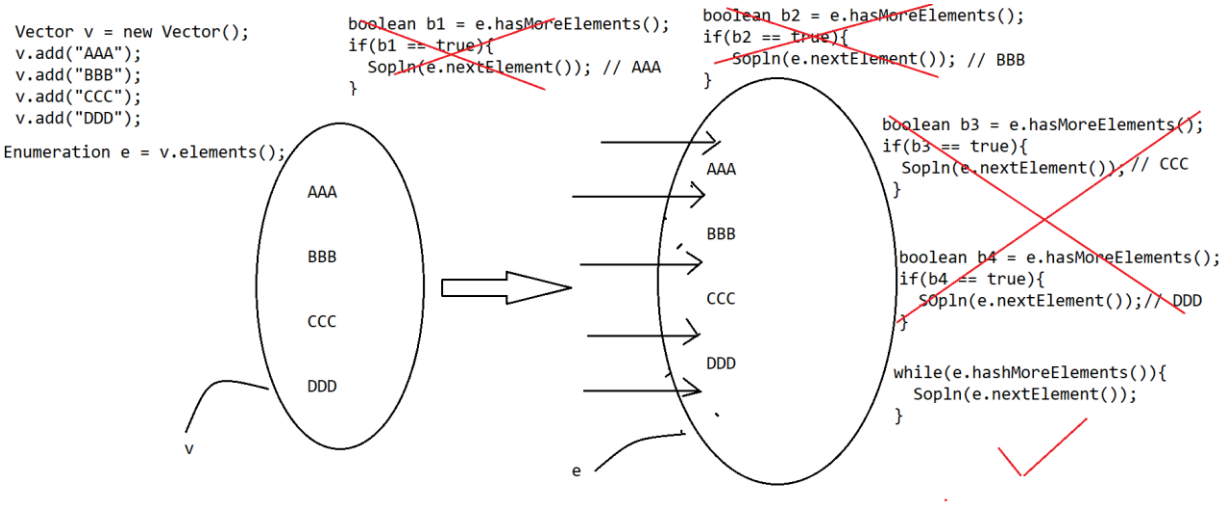
   public Enumeration elements()
   EX: Enumeration e = v.elements();

   When an Enumeration object is created, all the elements of the Collection object will be copied to the Enumeration object and a cursor will be created before the first element in order to iterate the elements.

2. Read Elements from Enumeration object:
   a. Check whether the next element is available or not from the current cursor position.
      public boolean hasMoreElements()
      ⇒ It will return true value if the next element is existed.
      ⇒ It will return a false value if the next element does not exist.

   b. If the next element exists then read the next element and move the cursor to the next position.
      public Object nextElement()

   Note: By repeating the above two steps we are able to read all the elements from Enumeration.




```java
import java.sql.Array;
import java.util.*;

public class Main {
public static void main(String[] args) {
Vector vector = new Vector();
vector.add("AAA");
vector.add("BBB");
vector.add("CCC");
vector.add("DDD");
System.out.println(vector);
Enumeration enumeration = vector.elements();
while(enumeration.hasMoreElements()){
System.out.println(enumeration.nextElement());
}

}
}
```

```
[AAA, BBB, CCC, DDD]
AAA
BBB
CCC
DDD
```

Note: Enumeration allows only read operation while iterating elements, it does not allow the operations like remove, insert, replace,.....

Iterator:

1. Iterator was provided in JDK1.2 version.
2. It is not a legacy Cursor.
3. It is applicable for all the types of Collections, so it is also called "Universal Cursor".
4. It allows both read and remove operations while iterating elements.
5. It is represented by JAVA in the form of an interface "java.util.Iterator".
6. To create an Iterator object we have to use the following method from all the Collection classes.

```
public Iterator iterator()
EX: Iterator it = c.iterator();
```

To get elements from the Iterator object we have to use the following methods.

1. public boolean hasNext()
   → It will return true if the next element exists.
   → It will return a false value if the next element does not exist.
2. Public Object next()
   It will read the next element and move Cursor to the next position.
3. public void remove():
   It will remove the current element from the Iterator.

EX:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
public static void main(String[] args) {
ArrayList arrayList = new ArrayList();
arrayList.add("AAA");
arrayList.add("BBB");
arrayList.add("CCC");
arrayList.add("DDD");
System.out.println(arrayList);
```

```
Iterator iterator = arrayList.iterator();
while (iterator.hasNext()){
String element = (String) iterator.next();
System.out.println(element);
if(element.equals("BBB")){
iterator.remove();
}
}
System.out.println(arrayList);

}
}
```

```
[AAA, BBB, CCC, DDD]
AAA
BBB
CCC
DDD
[AAA, CCC, DDD]
```

Drawbacks:

1. It allows us to perform only read and remove operations, it does not allow insert and replace operations.
2. It allows us to read elements in only Forward Direction, not in backward direction.

ListIterator:

---

1. It was introduced in JDK1.2 version.
2. It is not a Legacy Cursor.
3. It is applicable for only List implementations.
4. It allows the user to perform the operations like read, remove, insert and replace while iterating elements.
5. It allows you to read all the elements in both Forward direction and Backward direction.
6. It is represented in the form of an interface java.util.ListIterator.

To create a ListIterator object we have to use the following method from the List implementation classes.

```
public ListIterator listIterator()
```

To read elements in Forward direction we have to use the following methods.

public boolean hasNext()
public Object next()
public int nextIndex()

To read elements in Backward direction we have to use the following methods.
public boolean hasPrevious()
public Object previous()
public int previousIndex()

To perform the operations like remove, insert and replace we have to use the following methods.

public void remove()
public void add(Object obj)
public Object set(Object obj)

EX:

```
import java.util.LinkedList;
import java.util.ListIterator;

public class Main {
public static void main(String[] args) {

LinkedList linkedList = new LinkedList();
linkedList.add("AAA");
linkedList.add("BBB");
linkedList.add("CCC");
linkedList.add("DDD");
System.out.println(linkedList);
System.out.println();
ListIterator listIterator = linkedList.listIterator();
System.out.println("Elements in Forward Direction");
while (listIterator.hasNext()){
System.out.println(listIterator.next());
}
System.out.println();
System.out.println("Elements in Backward Direction");
while (listIterator.hasPrevious()){
System.out.println(listIterator.previous());
```

```
}

}
}
```

```
[AAA, BBB, CCC, DDD]

Elements in Forward Direction
AAA
BBB
CCC
DDD

Elements in Backward Direction
DDD
CCC
BBB
AAA
```

EX:

```
import java.util.LinkedList;
import java.util.ListIterator;

public class Main {
public static void main(String[] args) {

LinkedList linkedList = new LinkedList();
linkedList.add("AAA");
linkedList.add("BBB");
linkedList.add("CCC");
linkedList.add("DDD");
linkedList.add("EEE");
linkedList.add("FFF");
System.out.println(linkedList);
System.out.println();
ListIterator listIterator = linkedList.listIterator();
while (listIterator.hasNext()){
String element = (String) listIterator.next();
```

```
if(element.equals("BBB")){
listIterator.remove();
}
if(element.equals("DDD")){
listIterator.add("XXX");
}
if (element.equals("EEE")){
listIterator.set("YYY");
}

}
System.out.println(linkedList);


}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
[AAA, CCC, DDD, XXX, YYY, FFF]
```

Q)What are the differences between Enumeration, Iterator and ListIterator?

—----------

Ans:

—--

1. Enumeration was introduced in JDK1.0 version.
   Iterator and ListIterator were introduced in JDK1.2 version.

2. Enumeration is applicable for the Legacy Collection.
   Iterator is applicable for all the types of Collections.
   ListIterator is applicable for only List implementations.

3. Enumeration allows only read operation while iterating elements.
   Iterator allows read and remove operations while iterating elements.
   ListIterator allows read, remove, insert and replace operations while iterating elements.

4. Enumeration and Iterator allows reading elements in only Forward direction.
   ListIterator allows reading elements in both Forward Direction and Backward direction.

Set:

—---

1. It was introduced in JDK1.2 version.
2. It is a Child interface to Collection interface.
3. It is not Index based, it is able to arrange all the elements as per their hashcode values.
4. It does not allow duplicate elements.
5. It does not follow insertion order.
   Note: LinkedHashSet is able to follow insertion order.
6. It does not follow Sorting order.
   Note: SortedSet, NavigableSet and TreeSet are able to follow Sorting order.
7. It is able to allow only one null element.
   Note: SortedSet, NavigableSet and TreeSet are not allowing null elements.
8. It is able to allow Heterogeneous elements.
   Note: SortedSet, NavigableSet and TreeSet are not allowing Heterogeneous elements, they are able to allow only Homogeneous elements.

HashSet:

—--------

1. It was provided by JAVA in its JDK1.2 version.
2. It is not a Legacy Collection.
3. It is a direct implementation class to Set interface.
4. It is not index based.
5. It does not allow duplicate elements.
6. It does not follow insertion order.
7. It does not follow Sorting order.
8. It allows only one null element.
9. It allows heterogeneous elements.
10. Its initial capacity value is 16 elements.
11. Its load factor or fill ratio is 75%.
    Note: When HashSet is filled up to 75% then HashSet will increase its capacity dynamically.
12. Its internal data structure is “Hashtable”.
13. It is suggestible for the frequent search operations.
14. It is not a synchronized Collection.
15. It does not have synchronized methods.
16. It allows more than one thread at a time to access data.
17. It follows parallel execution of the threads.

18. It reduces application execution time.
19. It improves application performance.
20. It is not giving guarantee for the Data Consistency.
21. It is not Thread safe.

Constructors:

1. public HashSet():
   It is able to create an empty hashSet object with the initial capacity 16 elements and with the default fill ratio is 75%.

   EX:

```
import java.util.HashSet;

public class Main {
public static void main(String[] args) {
HashSet hashSet = new HashSet();
System.out.println(hashSet);
}
}
```

   []

2. public HashSet(int capacity)
   It can be used to create an empty HashSet object with the specified initial capacity and with the default fill ratio 75%.

   EX:

```
import java.util.HashSet;

public class Main {
public static void main(String[] args) {
HashSet hashSet = new HashSet(10);
System.out.println(hashSet);
}
}
```

   []

3. public HashSet(int capacity, float fillRatio):

It can be used to create an empty HashSet object with the specified initial capacity and with the specified fill ratio.

EX:

```
import java.util.HashSet;

public class Main {
public static void main(String[] args) {
HashSet hashSet = new HashSet(10, 0.85f);
System.out.println(hashSet);
}
}
```

[]

4. public HashSet(Collection c):
   It can be used to create a HashSet object with all the elements of the Specified Collection.

   Note: This constructor is used mainly for the internal conversions purposes.

   EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
HashSet hashSet1 = new HashSet(list);
System.out.println(hashSet1);

Set set = new HashSet();
set.add(111);
set.add(222);
set.add(333);
HashSet hashSet2 = new HashSet(set);
```

```
System.out.println(hashSet2);

Queue queue = new PriorityQueue();
queue.add("xxx");
queue.add("yyy");
queue.add("zzz");
HashSet hashSet3 = new HashSet(queue);
System.out.println(hashSet3);



}
}
```

```
[AAA, CCC, BBB]
[333, 222, 111]
[yyy, xxx, zzz]
```

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
HashSet hashSet = new HashSet();
hashSet.add("AAA");
hashSet.add("BBB");
hashSet.add("CCC");
hashSet.add("DDD");
System.out.println(hashSet);
hashSet.add("BBB");
System.out.println(hashSet);
hashSet.add(null);
hashSet.add(null);
System.out.println(hashSet);
hashSet.add(10);
hashSet.add(22.22f);
hashSet.add(33.333d);
```

```
System.out.println(hashSet);
}
}
```

```
[AAA, CCC, BBB, DDD]
[AAA, CCC, BBB, DDD]
[null, AAA, CCC, BBB, DDD]
[null, AAA, CCC, BBB, DDD, 10, 33.333, 22.22]
```

LinkedHashSet:
—-------------
Q)What are the differences between HashSet and LinkedHashSet?
—---
Ans:
—---
1. HashSet was introduced in JDK1.2 version
   LinkedHashSet was introduced in JDK1.4 version.

2. HashSet has the internal data structure Hashtable.
   LinkedHashSet has an internal Data Structure like “Hashtable+LinkedList”.

3. HashSet does not follow Insertion order.
   LinkedHashSet follows Insertion order.

4. HashSet is a superclass of LinkedHashSet.

EX:

```
import java.util.HashSet;
import java.util.LinkedHashSet;

public class Main {
public static void main(String[] args) {
HashSet hashSet = new HashSet();
hashSet.add("FFF");
hashSet.add("EEE");
hashSet.add("DDD");
hashSet.add("CCC");
hashSet.add("BBB");
hashSet.add("AAA");
```

```
System.out.println(hashSet);
LinkedHashSet linkedHashSet = new LinkedHashSet();
linkedHashSet.add("FFF");
linkedHashSet.add("EEE");
linkedHashSet.add("DDD");
linkedHashSet.add("CCC");
linkedHashSet.add("BBB");
linkedHashSet.add("AAA");
System.out.println(linkedHashSet);
}
}
```

```
[AAA, CCC, BBB, EEE, DDD, FFF]
[FFF, EEE, DDD, CCC, BBB, AAA]
```

SortedSet:

---

1. It was introduced in JDK1.2 version
2. It is a direct child interface to Set interface.
3. It is not index based.
4. It does not allow duplicate elements.
5. It does not follow the Insertion order.
6. It is mainly for Sorting Order.
7. It does not allow null elements.If we add any null element then JVM will raise an exception like java.lang.NullPointerException.
8. It does not allow heterogeneous elements, it allows only Homogeneous elements, if we add any heterogeneous elements then JVM will raise an exception like java.lang.ClassCastException.
9. If we want to add any element to the SortedSet then that element must be Comparable Element, that is the element related class must implement java.lang.Comparable interface, If the element is not Comparable and if we add that element to the SortedSet then JVM will raise an exception like java.lang.ClassCastException.
   Note: If we want to add a non comparable element to the SortedSet then we must use Comparator.

Methods:

1. public SortedSet headSet(Object element):
   It is able to return a SortedSet object which contains all the elements which are less than the Specified Element.

2. public SortedSet tailSet(Object element):
   It is able to return a SortedSet object which contains all the elements which are greater than or equal to the specified element.

3. public SortedSet subSet(Object fromElement, Object toElement):
   It is able to return a SortedSet object which contains all the elements which are greater than or equal to the specified fromElement and which are less than the specified toElement.

4. public Object first():
   It is able to return the First element from the SortedSet.

5. public Object last():
   It is able to return the last element from the SortedSet.

EX:

```
import java.util.SortedSet;
import java.util.TreeSet;

public class Main {
public static void main(String[] args) {
SortedSet sortedSet = new TreeSet();
sortedSet.add("FFF");
sortedSet.add("AAA");
sortedSet.add("EEE");
sortedSet.add("BBB");
sortedSet.add("DDD");
sortedSet.add("CCC");
System.out.println(sortedSet);
System.out.println(sortedSet.headSet("DDD"));
System.out.println(sortedSet.tailSet("DDD"));
System.out.println(sortedSet.subSet("BBB", "EEE"));
System.out.println(sortedSet.first());
System.out.println(sortedSet.last());
```

```
}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
[AAA, BBB, CCC]
[DDD, EEE, FFF]
[BBB, CCC, DDD]
AAA
FFF
```

NavigableSet:
—----------
NavigableSet was introduced in Java 1.6 version, NavigableSet is a child interface to SortedSet interface, it is almost all the same as the SortedSet but it has defined some methods to perform navigation over the elements like Getting Descending order of the elements, Finding ceiling element, floor element , higher element, lower elements for a particular element.

Methods:

1. public NavigableSet descendingSet():
   It can be used to return a NavigableSet object that contains all the elements in descending order.

2. public Object ceiling(Object element):
   It will return the lowest element among all the elements which are greater than or equal to the specified element.

3. public Object higher(Object element):
   It will return the lowest element among all the elements which are greater than the specified element.

4. public Object floor(Object element):
   It will return the highest element among all the elements that are less than or equal to the specified element.

5. public Object lower(Object element):
   It will return the highest element among all the elements that are less than the specified element.

6. public Object pollFirst():
   It will return and remove the First element.
7. public Object pollLast():

It will return and remove the last element from the NavigableSet.

EX:

```
import java.util.NavigableSet;
import java.util.TreeSet;

public class Main {
public static void main(String[] args) {
NavigableSet navigableSet = new TreeSet();
navigableSet.add("FFF");
navigableSet.add("AAA");
navigableSet.add("EEE");
navigableSet.add("BBB");
navigableSet.add("DDD");
navigableSet.add("CCC");
System.out.println(navigableSet);
System.out.println(navigableSet.descendingSet());
System.out.println(navigableSet.ceiling("DDD"));
System.out.println(navigableSet.higher("DDD"));
System.out.println(navigableSet.floor("DDD"));
System.out.println(navigableSet.lower("DDD"));
System.out.println(navigableSet.pollFirst());
System.out.println(navigableSet.pollLast());
System.out.println(navigableSet);
}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
[FFF, EEE, DDD, CCC, BBB, AAA]
DDD
EEE
DDD
CCC
AAA
FFF
[BBB, CCC, DDD, EEE]
```

TreeSet:

—-------

1. It was introduced in JDK1.2 version.
2. It is not a legacy Collection.
3. It is an implementation class to the NavigableSet interface , that is it has provided the implementations for all the methods of the interfaces like Collection, Set, SortedSet and NavigableSet.

4. It is not index based.
5. It does not allow duplicate elements.
6. It does not follow insertion order.
7. It follows Sorting order.
8. It does not allow null elements, if we add null elements to TreeSet then JVM will raise an exception like java.lang.NullPointerException.
9. It does not allow heterogeneous elements, it allows only Homogeneous elements. If we add heterogeneous elements to the TreeSet then JVM will raise an exception like java.lang.ClassCastException.
10. It is able to allow only Comparable elements , that is the elements which are implementing java.lang.Comparable interface. If we add non comparable elements to the TreeSet then JVM will raise an exception like java.lang.ClassCastException.
11. If we want to add Non Comparable elements to the TreeSet then we have to use the java.util.Comparator interface.

12. It uses “Balanced Tree” as an internal data Structure.
13. It is not a synchronized Collection.
14. It does not have the SYnchronized methods.
15. It allows more than one thread at a time.
16. It follows Parallel execution of the threads.
17. It reduces application execution time.
18. It improves application Performance.
19. It is not guaranteed for data Consistency.
20. It is not Thread Safe.

Constructors:

—------------

1. public TreeSet():
   It is able to create an empty TreeSet object.
   EX:

```
import java.util.NavigableSet;
import java.util.TreeSet;

public class Main {
```

```
public static void main(String[] args) {
TreeSet ts = new TreeSet();
System.out.println(ts);
}
}
```

[]

2. public TreeSet(SortedSet ss):
   It is able to create a TreeSet object with all the elements of the provided SortedSet.
   EX:

```
import java.util.NavigableSet;
import java.util.SortedSet;
import java.util.TreeSet;

public class Main {
public static void main(String[] args) {
SortedSet sortedSet = new TreeSet();
sortedSet.add("DDD");
sortedSet.add("AAA");
sortedSet.add("EEE");
sortedSet.add("BBB");
sortedSet.add("FFF");
sortedSet.add("CCC");
System.out.println(sortedSet);
TreeSet ts = new TreeSet(sortedSet);
System.out.println(ts);
}
}
```

[AAA, BBB, CCC, DDD, EEE, FFF]
[AAA, BBB, CCC, DDD, EEE, FFF]

3. public TreeSet(Collection c):
   It is able to create aTreeSet object with all the elements of the specified Collection.

   Note: It will be used to convert all the elements from the collection types like List, Set and Queue to TreeSet.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("DDD");
list.add("BBB");
list.add("CCC");
TreeSet ts1 = new TreeSet(list);
System.out.println(ts1);

Set set = new HashSet();
set.add(200);
set.add(100);
set.add(400);
set.add(300);
TreeSet ts2 = new TreeSet(set);
System.out.println(ts2);

Queue queue = new PriorityQueue();
queue.add("XXX");
queue.add("YYY");
queue.add("ZZZ");
TreeSet ts3 = new TreeSet(queue);
System.out.println(ts3);

}
}
```

```
[AAA, BBB, CCC, DDD]
[100, 200, 300, 400]
[XXX, YYY, ZZZ]
```

EX:

```
import java.util.*;
```

```java
class Employee{

}
class Customer implements Comparable{

@Override
public int compareTo(Object o) {
return 100;
}

@Override
public String toString() {
return "Customer";
}
}

public class Main {
public static void main(String[] args) {
TreeSet ts = new TreeSet();
ts.add("DDD");
ts.add("AAA");
ts.add("EEE");
ts.add("BBB");
ts.add("FFF");
ts.add("CCC");
System.out.println(ts);
ts.add("BBB");
ts.add("CCC");
System.out.println(ts);
//ts.add(null); ---> NullPointerException
//ts.add(100); ----> ClassCastException
TreeSet ts1 = new TreeSet();
//ts1.add(new Employee()); ---> ClassCastException
ts1.add(new Customer());
System.out.println(ts1);


}
```

```
}
```

OP:
[AAA, BBB, CCC, DDD, EEE, FFF]
[AAA, BBB, CCC, DDD, EEE, FFF]
[Customer]

Importance of the compareTo() method from String and Wrapper classes:
—--
Initially compareTo() was declared in java.lang.Comparable interface, its main intention is to compare elements in sorting order[By Default natural ascending order in alphabetical order], it was implemented in the predefined classes like String, all Wrapper classes like Byte, Short, Integer, Long,.....

public int compareTo(Object obj)

str1.compareTo(str2):

1. If str1 comes first when compared with str2 in dictionary order then compareTo() method will return -ve value.
2. If str2 comes first when compared with str1 in dictionary order then the compareTo() method will return +ve value.
3. If str1 and str2 are at the same position in the dictionary order then the compareTo() method will return 0 value.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {

String str1 = new String("abc");
String str2 = new String("def");
String str3 = new String("abc");

System.out.println(str1.compareTo(str2));//abc.cTo(def)
==> -3
System.out.println(str2.compareTo(str3));//def.cTo(abc)
==> +3
System.out.println(str3.compareTo(str1));//abc.cTo(abc)
==> 0
```

```
    }
    }
```

```
-3
3
0
```

Internal Working Process of TreeSet:
—----------------------------------

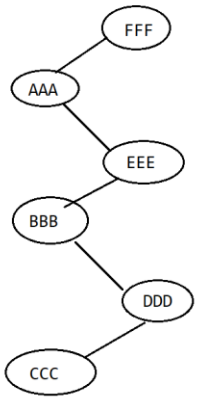
1. Create a Balanced Tree with all the elements of TreeSet.
2. Get all the elements as per In Order Traversal.

Create a Balanced Tree with all the elements of TreeSet:

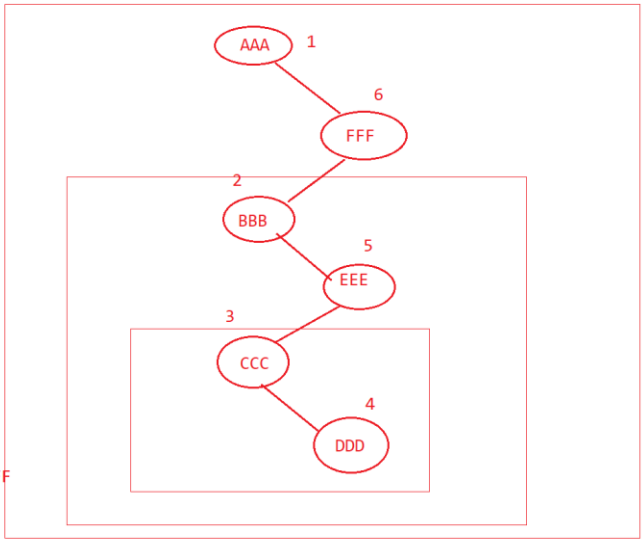
1. If we add the first element in TreeSet then it will be the root node in the Balanced Tree.
2. If we add an element which is not first element:
    a. Access compareTo() method on the new element by passing the existing elements of the TreeSet right from the Root node.
    b. If compareTo() method returns +ve value then go and compare with the right child by accessing compareTo() method again and by passing the right node. If no Right Node exists then adds the new element as Right Node to the Root.
    c. If compareTo() method returns -ve value then go and compare with left child by accessing compareTo() method again and by passing left child as parameter. If no left child exists then add the new element as left child.
    d. If compareTo() method returns 0 value then discard that element and declare that element as a duplicate element.

Get all the elements as per In Order Traversal:
Inorder Traversal : left-Root-Right
Follow Inorder traversal to get all elements in Sorting order.

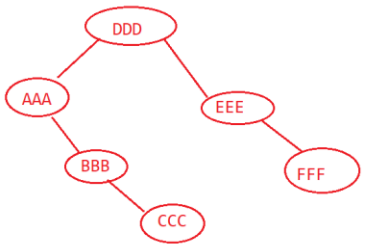
```
TreeSet ts = new TreeSet();
ts.add("FFF");

ts.add("AAA");  "AAA".cTo("FFF"); -ve

ts.add("EEE");  "EEE".cTo("FFF");-ve
                "EEE".cTo("AAA");+ve

ts.add("BBB");  "BBB".cTo("FFF");-ve
                "BBB".cTo("AAA");+ve
                "BBB".cTo("EEE");-ve

ts.add("DDD");  "DDD".cTo("FFF");-ve
                "DDD".cTo("AAA");+ve
                "DDD".cTo("EEE");-ve
                "DDD".cTo("BBB");+ve

ts.add("CCC");  "CCC".cTo("FFF");-ve
                "CCC".cTo("AAA");+ve
                "CCC".cTo("EEE");-ve
                "CCC".cTo("BBB");+ve
                "CCC".cTo("DDD");-ve

Sopln(ts);      Left-Root-Right        AAA  BBB  CCC  DDD  EEE  FFF
```




```
TreeSet ts = new TreeSet();
ts.add("AAA");

ts.add("FFF");  "FFF".cTo("AAA");+ve

ts.add("BBB");  "BBB".cTo("AAA");+ve
                "BBB".cTo("FFF");-ve

ts.add("EEE");  "EEE".cTo("AAA");+ve
                "EEE".cTo("FFF");-ve
                "EEE"cTo("BBB");+ve

ts.add("CCC");  "CCC".cTo("AAA");+ve
                "CCC".cTo("FFF");-ve
                "CCC".cTo("BBB");+ve
                "CCC".cTo("EEE");-ve

ts.add("DDD");  "DDD".cTo("AAA");+ve
                "DDD".cTo("FFF");-ve
                "DDD".cTo("BBB");+ve
                "DDD".cTo("EEE");-ve
                "DDD".cTo("CCC");+ve

Sopln(ts);      AAA  BBB  CCC  DDD  EEE  FFF
```



```
TreeSet ts = new TreeSet();
ts.add("DDD");
ts.add("AAA");   "AAA".cTo("DDD");-ve
ts.add("EEE");   "EEE".cTo("DDD");+ve
ts.add("BBB");   "BBB".cTo("DDD");-ve
                 "BBB".cTo("AAA");+ve
ts.add("FFF");   "FFF".cTo("DDD");+ve
                 "FFF".cTo("EEE");+ve
ts.add("CCC");   "CCC"c.To("DDD");-ve
                 "CCC".cTo("AAA");+ve
                 "CCC".cTo("BBB");+ve

Sopln(ts);     AAA   BBB   CCC   DDD   EEE   FFF
```



Note: To sort all the elements in TreeSet, all the elements must have compareTo() method with sorting logic, to get compareTo() method in all the elements we must implement java.lang.Comparable interface in all the elements.

If we want to sort any user defined elements like Employee, Student, Account,..... as per a particular field values then we have to use the following steps.

1. Declare an User defined class.
2. Implement java.lang.Comparable interface in user defined class.
3. Provide implementation for compareTo() method in User defined class with the sorting logic that must be accessed by TreeSet internally.
4. In the Main class, in the main() method, create User defined class Objects and add these user defined class objects to TreeSet.
5. Display all elements of TreeSet.

EX:

```
import java.util.*;
class Employee implements Comparable{

private int eno;
private String ename;
private float esal;
```

```
private String eaddr;

public Employee(int eno, String ename, float esal, String
eaddr) {
this.eno = eno;
this.ename = ename;
this.esal = esal;
this.eaddr = eaddr;
}


@Override
public int compareTo(Object o) {
Employee emp = (Employee) o;
int val = this.ename.compareTo(emp.ename);
return val;
}

@Override
public String toString() {
return "Employee{" +
"eno=" + eno +
", ename='" + ename + '\'' +
", esal=" + esal +
", eaddr='" + eaddr + '\'' +
'}'+"\n";
}
}
public class Main {
public static void main(String[] args) {
Employee emp1 = new Employee(111, "Durga", 5000, "Hyd");
Employee emp2 = new Employee(222, "Anil", 6000, "Chennai");
Employee emp3 = new Employee(333, "Raju", 7000, "Pune");
Employee emp4 = new Employee(444, "Pardhu", 85000,
"Delhi");
Employee emp5 = new Employee(555, "Balu", 9000, "Mumbai");
Employee emp6 = new Employee(666, "Gopi", 5000, "Goa");
```

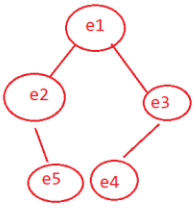
```
TreeSet ts = new TreeSet();
ts.add(emp1);
ts.add(emp2);
ts.add(emp3);
ts.add(emp4);
ts.add(emp5);

System.out.println(ts);


}
}
```

```
[Employee{eno=222, ename='Anil', esal=6000.0, eaddr='Chennai'},
Employee{eno=555, ename='Balu', esal=9000.0, eaddr='Mumbai'},
Employee{eno=111, ename='Durga', esal=5000.0, eaddr='Hyd'},
Employee{eno=444, ename='Pardhu', esal=85000.0, eaddr='Delhi'},
Employee{eno=333, ename='Raju', esal=7000.0, eaddr='Pune'}
]
```

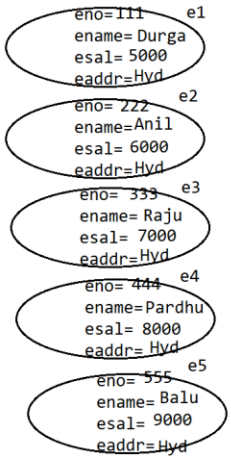
```
class Employee implements Comparable{
private int eno;
private String ename;
private float esal;
private String eaddr;
-----
-----
public int compareTo(Employee emp){
        e5 Balu.cTo(Anil)     e2
  return this.ename.compareTo(emp.ename);
}
-----
-----
}
```

```
TreeSet ts = new TreeSet();
ts.add(e1);
ts.add(e2);
ts.add(e3);
ts.add(e4);
ts.add(e5);

Sopln(ts);
```

e2.cTo(e1);-ve
e3.cTo(e1);+ve
e4.cTo(e1);+ve
e4.cTo(e3);-ve
e5.cTo(e1);-ve
e5.cTo(e2);+ve



e2	e5	e1	e4	e3
Anil	Balu	Durga	Pardhu	Raju



EX:

```
import java.util.*;
class Employee implements Comparable{

private int eno;
private String ename;
private float esal;
private String eaddr;

public Employee(int eno, String ename, float esal, String
eaddr) {
this.eno = eno;
this.ename = ename;
this.esal = esal;
this.eaddr = eaddr;
}


@Override
public int compareTo(Object o) {
Employee emp = (Employee) o;
int val = this.eaddr.compareTo(emp.eaddr);
return -val;
}

@Override
public String toString() {
return "Employee{" +
"eno=" + eno +
", ename='" + ename + '\'' +
", esal=" + esal +
", eaddr='" + eaddr + '\'' +
'}'+"\n";
}
}
public class Main {
public static void main(String[] args) {
```

```
Employee emp1 = new Employee(111, "Durga", 5000, "Hyd");
Employee emp2 = new Employee(222, "Anil", 6000, "Chennai");
Employee emp3 = new Employee(333, "Raju", 7000, "Pune");
Employee emp4 = new Employee(444, "Pardhu", 85000,
"Delhi");
Employee emp5 = new Employee(555, "Balu", 9000, "Mumbai");
Employee emp6 = new Employee(666, "Gopi", 5000, "Goa");

TreeSet ts = new TreeSet();
ts.add(emp1);
ts.add(emp2);
ts.add(emp3);
ts.add(emp4);
ts.add(emp5);

System.out.println(ts);


}
}
```

```
[Employee{eno=333, ename='Raju', esal=7000.0, eaddr='Pune'}
, Employee{eno=555, ename='Balu', esal=9000.0, eaddr='Mumbai'}
, Employee{eno=111, ename='Durga', esal=5000.0, eaddr='Hyd'}
, Employee{eno=444, ename='Pardhu', esal=85000.0, eaddr='Delhi'}
, Employee{eno=222, ename='Anil', esal=6000.0, eaddr='Chennai'}
]
```

If we want to sort all the elements by using Comparable then each and every element must have sorting logic, that is each and every element must have compareTo() method, that is each and every element must implement java.lang.Comparable interface.

To the TreeSet, if we add Non Comparable elements then JVM will raise an exception like java.lang.ClassCastException.

When we add non comparable elements to the TreeSet then TreeSet will not have implicit Sorting logic due to the unavailability of compareTo() method ,in the above context, to sort Non comparable elements in TreeSet we must provide sorting logic explicitly, here to provide sorting logic explicitly to the TreeSet we must use java.util.Comparator.

java.util.Comparator contains the following two methods.

```
public int compare(Object obj1, Object obj2)
public boolean equals(Object obj)
```

Where the compare() method is the same as compareTo() of the Comparable interface, it is able to compare two elements as per the dictionary order.

```
ts.compare(str1,str2)
```

If str1 comes first when compared with str2 in dictionary order then compare() method will return -ve value.

If str2 comes first when compared with str1 in dictionary order then compare() method will return +ve value.

If str1 and str2 are at the same position in the dictionary order then compare() method will return 0 value.

In Java applications, if we want to use java.util.Comparator then we have to use the following steps.

1. Declare an user defined class.
2. Implement java.util.Comparator interface in user defined class.
3. Provide implementation for Comparator interface methods in the user defined class.
   Note: Here we need to provide implementation for only compare() method, not for equals() method, because equals() method is coming from the default superclass java.lang.Object class.
4. In the main class, in the main() method provide Comparator object to the TreeSet by using the following constructor.

   ```
   public TreeSet(Comparator c)
   ```

5. Create Elements which we want to sort.
6. Add elements to TreeSet.
7. Display all the elements.

EX:

```
class MyComparator implements Comparator{
  public int compare(Object obj1, Object obj2){
      –-----
      return val;
   }
}

class Test{
   public static void main(String[] args){
        MyComparator mc = new MyComparator();
        TreeSet ts = new TreeSet(mc);
        ts.add(e1);
        ts.add(e2);
        ts.add(e3);
        ts.add(e4);
        –---
        System.out.println(ts);
   }
}
```

EX:

```
import java.util.Comparator;
import java.util.TreeSet;
class MyComparator implements Comparator{
@Override
public int compare(Object o1, Object o2) {

StringBuffer sb1 = (StringBuffer) o1;
StringBuffer sb2 = (StringBuffer) o2;

String str1 = sb1.toString();
String str2 = sb2.toString();

int val = str1.compareTo(str2);
return val;
```

```
}
}
public class Main {
public static void main(String[] args) {
StringBuffer sb1 = new StringBuffer("AAA");
StringBuffer sb2 = new StringBuffer("BBB");
StringBuffer sb3 = new StringBuffer("CCC");
StringBuffer sb4 = new StringBuffer("DDD");
StringBuffer sb5 = new StringBuffer("EEE");
MyComparator mc = new MyComparator();
TreeSet treeSet = new TreeSet(mc);
treeSet.add(sb5);
treeSet.add(sb1);
treeSet.add(sb4);
treeSet.add(sb2);
treeSet.add(sb3);
System.out.println(treeSet);


}
}
```

[AAA, BBB, CCC, DDD, EEE]

EX:

```
import java.util.Comparator;
import java.util.TreeSet;
class MyComparator implements Comparator{
@Override
public int compare(Object o1, Object o2) {

StringBuffer sb1 = (StringBuffer) o1;
StringBuffer sb2 = (StringBuffer) o2;

int length1 = sb1.length();
```

```
int length2 = sb2.length();

int val = 0;
if(length1 > length2){
val = 100;
}else{
val = -100;
}
return val;
}
}
public class Main {
public static void main(String[] args) {
StringBuffer sb1 = new StringBuffer("AAAA");
StringBuffer sb2 = new StringBuffer("BBBBB");
StringBuffer sb3 = new StringBuffer("C");
StringBuffer sb4 = new StringBuffer("DDD");
StringBuffer sb5 = new StringBuffer("EE");
MyComparator mc = new MyComparator();
TreeSet treeSet = new TreeSet(mc);
treeSet.add(sb5);
treeSet.add(sb1);
treeSet.add(sb4);
treeSet.add(sb2);
treeSet.add(sb3);
System.out.println(treeSet);


}
}
```

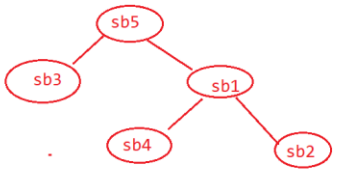
```
[C, EE, DDD, AAAA, BBBBB]
```

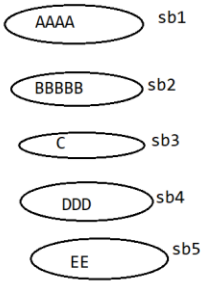
```
class MyComp implements Comparator{
  public int compare(Object o1, Object o2){
    StrBuff sb1 = (StrBuff)o1;
    StrBuff sb2 = (StrBuff)o2;

    int len1 = sb1.length(); 1
    int len2 = sb2.length(); 2
    int val = 0;
    if(len1 > len2){
      val = 100;
    }else{
      val = -100;
    }
    return val;
  }
}
```

```
MyComp mc = new MyComp();
TreeSet ts = new TreeSet(mc);
ts.add(sb5);
ts.add(sb1);
ts.add(sb4);
ts.add(sb2);
ts.add(sb3);

Sopln(ts);
```

ts.compare(sb1,sb5) +ve
ts.compare(sb4,sb5); +ve
ts.compare(sb4,sb1); -ve
ts.compare(sb2,sb5); +ve
ts.compare(sb2,sb1); +ve
ts.compare(sb3,sb5) -ve

sb3	sb5	sb4	sb1	sb2
1	2	3	4	5





EX:

```
import java.util.Comparator;
import java.util.TreeSet;

class StudentComparator implements Comparator{
@Override
public int compare(Object o1, Object o2) {

Student s1 = (Student) o1;
Student s2 = (Student) o2;

int smarks1 = s1.getSmarks();
int smarks2 = s2.getSmarks();

return smarks1 - smarks2;
}
}
class Student{
private String sid;
private String sname;
private int smarks;

public Student(String sid, String sname, int smarks) {
this.sid = sid;
```

```
this.sname = sname;
this.smarks = smarks;
}

public int getSmarks() {
return smarks;
}

@Override
public String toString() {
return "Student{" +
"sid='" + sid + '\'' +
", sname='" + sname + '\'' +
", smarks=" + smarks +
'}'+"\n";
}
}

public class Main {
public static void main(String[] args) {
Student std1 = new Student("S-111", "AAA", 78);
Student std2 = new Student("S-222", "BBB", 65);
Student std3 = new Student("S-333", "CCC", 89);
Student std4 = new Student("S-444", "DDD", 96);
Student std5 = new Student("S-555", "EEE", 82);

TreeSet ts = new TreeSet(new StudentComparator());
ts.add(std1);
ts.add(std2);
ts.add(std3);
ts.add(std4);
ts.add(std5);

System.out.println(ts);

}
}
```

```
[Student{sid='S-222', sname='BBB', smarks=65}
, Student{sid='S-111', sname='AAA', smarks=78}
, Student{sid='S-555', sname='EEE', smarks=82}
```

```
, Student{sid='S-333', sname='CCC', smarks=89}
, Student{sid='S-444', sname='DDD', smarks=96}
]
```

Q)If we provide both implicit sorting through Comparable and explicit sorting through Comparator to the TreeSet then Which Sorting logic would be preferred by TreeSet while sorting elements?

Ans:

If we provide both implicit Sorting through Comparable and explicit sorting through Comparator to the TreeSet then TreeSet will search for the explicitly sorting through Comparator , if the explicit sorting does not exist then TreeSet will search for the implicit sorting through Comparable.

EX:

```java
import java.util.Comparator;
import java.util.TreeSet;

class StudentComparator implements Comparator{
@Override
public int compare(Object o1, Object o2) {

Student s1 = (Student) o1;
Student s2 = (Student) o2;

int smarks1 = s1.getSmarks();
int smarks2 = s2.getSmarks();

return -(smarks1 - smarks2);
}
}
class Student implements Comparable{
private String sid;
private String sname;
private int smarks;

public Student(String sid, String sname, int smarks) {
this.sid = sid;
```

```
this.sname = sname;
this.smarks = smarks;
}

public int getSmarks() {
return smarks;
}

@Override
public int compareTo(Object o) {
Student std = (Student)o;
return this.smarks - std.smarks;
}

@Override
public String toString() {
return "Student{" +
"sid='" + sid + '\'' +
", sname='" + sname + '\'' +
", smarks=" + smarks +
'}'+"\n";
}
}

public class Main {
public static void main(String[] args) {
Student std1 = new Student("S-111", "AAA", 78);
Student std2 = new Student("S-222", "BBB", 65);
Student std3 = new Student("S-333", "CCC", 89);
Student std4 = new Student("S-444", "DDD", 96);
Student std5 = new Student("S-555", "EEE", 82);

TreeSet ts = new TreeSet(new StudentComparator());
ts.add(std1);
ts.add(std2);
ts.add(std3);
```

```
ts.add(std4);
ts.add(std5);

System.out.println(ts);

}
}
```

```
[Student{sid='S-444', sname='DDD', smarks=96}
, Student{sid='S-333', sname='CCC', smarks=89}
, Student{sid='S-555', sname='EEE', smarks=82}
, Student{sid='S-111', sname='AAA', smarks=78}
, Student{sid='S-222', sname='BBB', smarks=65}
]
```

Q)What are the differences between Comparable and Comparator?

—----------

Ans:

—---

1. Comparable exists in the java.lang package.
   Comparator exists in the java.util package.

2. Comparable has only one method.
   public int compareTo(Object obj)

   Comparator has two methods.
   public int compare(Object obj1, Object obj2)
   public boolean equals(Object obj)

3. The main purpose of the Comparable is to provide implicit sorting to the TreeSet.

   The main purpose of the Comparator is to provide explicit Sorting to the TreeSet.

4. In the case of Comparable, each and every element of the TreeSet must have the sorting logic.

   In the case of Comparator, every element of TreeSet is not required to manage Sorting logic.

5. Some predefined classes like String, wrapper classes,... are implementing Comparable interface.

   No predefined class is implementing Comparator interface.

Note: In Older Versions of Java like Java7, Java8,.. StringBuffer is not Comparable, but in the latest versions of the java like Java17 StringBuffer is Comparable.

Map:
—----

1. It was introduced in JDK1.2 version.
2. It is not a child interface to the Collection interface.
3. It is able to represent all the elements in the form of Key-value pairs.
4. In Key-Value pairs, both Keys and values are Objects.
5. In key-value pairs, keys must be unique and values may be duplicated.
6. In Key-Value pairs, keys are not allowing duplicates.
7. In Key-Value pairs, both keys and values are allowing null elements, where keys are able to allow only one null element, but values are able to allow more than one null element.
8. In Key-Value pairs, both Keys and values are able to allow heterogeneous elements.

Methods:

1. public void put(Object key, Object value):
   It is able to add the provided key-value in the Map object.
   EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map stdRegister = new HashMap();
stdRegister.put("101", "Durga");
stdRegister.put("102", "Anil");
stdRegister.put("103", "Rakesh");
stdRegister.put("104", "Raju");
stdRegister.put("105", "Ramesh");
System.out.println(stdRegister);

}
```

```
}
```

{101=Durga, 102=Anil, 103=Rakesh, 104=Raju, 105=Ramesh}

2. public void putAll(Map map):
   It is able to add all key-value pairs[Entries] of the provided Map to the present Map object.

   EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);

Map map1 = new HashMap();
System.out.println(map1);
map1.putAll(map);
System.out.println(map1);


}
}
```

{A=AAA, B=BBB, C=CCC, D=DDD}
{}
{A=AAA, B=BBB, C=CCC, D=DDD}

3. public Object get(Object key):
   It can be used to get the value of a Key-Value pair on the basis of the provided Key.
   EX:

```
import java.util.HashMap;
import java.util.Map;
```

```
public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
System.out.println(map.get("A"));
System.out.println(map.get("B"));
System.out.println(map.get("C"));



}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
AAA
BBB
CCC
```

4. public Object remove(Object key):
   It is able to remove a particular key-Value pair on the basis of the provided key.
   EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
System.out.println(map.remove("B"));
System.out.println(map.remove("C"));
System.out.println(map);
```

```
}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
BBB
CCC
{A=AAA, D=DDD}
```

5. public boolean containsKey(Object key):
   It can be used to check whether the provided key exists or not at keys side.
   EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
System.out.println(map.containsKey("B"));
System.out.println(map.containsKey("Z"));


}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
true
false
```

6. public boolean containsValue(Object values):
   It can be used to check whether the provided value exists at the values side or not.
   EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
System.out.println(map.containsValue("BBB"));
System.out.println(map.containsValue("ZZZ"));


}
}
```

   {A=AAA, B=BBB, C=CCC, D=DDD}
   true
   false

7. public int size():
   It can be used to return the size of the Map that is the number of entries which exist in the map.
   EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
```

```
System.out.println(map);
System.out.println(map.size());


}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
4
```

8. public boolean isEmpty():
   It can be used to check whether the Map is empty or not, if the Map is empty then isEmpty() method will return true value, if the map is not empty then isEmpty() method will return false value.
   EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
System.out.println(map.isEmpty());
Map map1 = new HashMap();
System.out.println(map1.isEmpty());


}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
false
true
```

9. public void clear():
   It can be used to remove all key-value from the Map.

EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
map.clear();
System.out.println(map);


}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
{}
```

10. public Set keySet():

It can be used to get all Keys from the Entries of a Map object in the form of a Set.

EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
System.out.println(map.keySet());

```

```
}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
[A, B, C, D]
```

11. public Collection values():

It can be used to get all values of the entries in a Map object in the form of Collection.

EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
System.out.println(map.values());


}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
[AAA, BBB, CCC, DDD]
```

HashMap:

—-------

1. It was introduced in JDK1.2 version.
2. It is not a legacy Map.
3. It is a direct implementation class to Map interface.
4. It is able to allow all the elements in the form of Key-Value pairs.
5. In Key-value pairs, both Keys and Values are Objects.

6. In Key-Values pairs, keys must be unique and values may be duplicated.
7. In Key-value pairs, Keys are not allowing duplicates but values are able to allow duplicates.
8. It does not follow the key's Insertion order.
9. It does not follow the key's Sorting order.
10. It allows only one null element at the keys side and any number of null elements at the values side.
11. It allows heterogeneous elements at both keys and values.

12. Its initial capacity is 16 entries.
13. Its initial load factor is 75%.
14. Its internal data Structure is Hashtable.
15. It is suggestible for the frequent search operations.

16. It is not a synchronized Map.
17. No method is synchronized in HashMap.
18. It allows more than one thread at a time to access data.
19. It follows parallel execution of the threads.
20. It will reduce application execution time.
21. It will increase application performance.
22. It is not giving guarantees for the data consistency.
23. It is not thread safe.

Constructors:
—----------

1. public HashMap():
   It can be used to create an empty HashMap object with the initial capacity 16 elements and with the initial load factor 75%.
   EX:
   import ;

```
import java.util.Map;

public class Main {
public static void main(String[] args) {
HashMap hashMap = new HashMap();
System.out.println(hashMap);


}
```

```
}
```

2. public HashMap(int capacity)
   It can be used to create an empty HashMap object with the specified Capacity value and with the default load factor 75%.
   EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
HashMap hashMap = new HashMap(20);
System.out.println(hashMap);


}
}
```

   {}

3. public HashMap(int capacity, float loadFactor):
   It can be used to create an empty HashMap object with the specified initial capacity and with the specified load Factor.
   EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
HashMap hashMap = new HashMap(20, 0.85f);
System.out.println(hashMap);


}
}
```

   {}

4. public HashMap(Map map):
   It can be used to create a HashMap object with all the entries of the specified Map object.
   EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
HashMap hashMap = new HashMap(map);
System.out.println(hashMap);


}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
{A=AAA, B=BBB, C=CCC, D=DDD}
```

EX:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
map.put("B", "EEE");
System.out.println(map);
```

```
map.put(null, "FFF");
map.put(null, "XXX");
System.out.println(map);
map.put("E", "EEE");
map.put("F","EEE");
System.out.println(map);
map.put("G", null);
map.put("H", null);
System.out.println(map);


}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
{A=AAA, B=EEE, C=CCC, D=DDD}
{null=XXX, A=AAA, B=EEE, C=CCC, D=DDD}
{null=XXX, A=AAA, B=EEE, C=CCC, D=DDD, E=EEE, F=EEE}
{null=XXX, A=AAA, B=EEE, C=CCC, D=DDD, E=EEE, F=EEE, G=null, H=null}
```

Internal Working of HashMap:

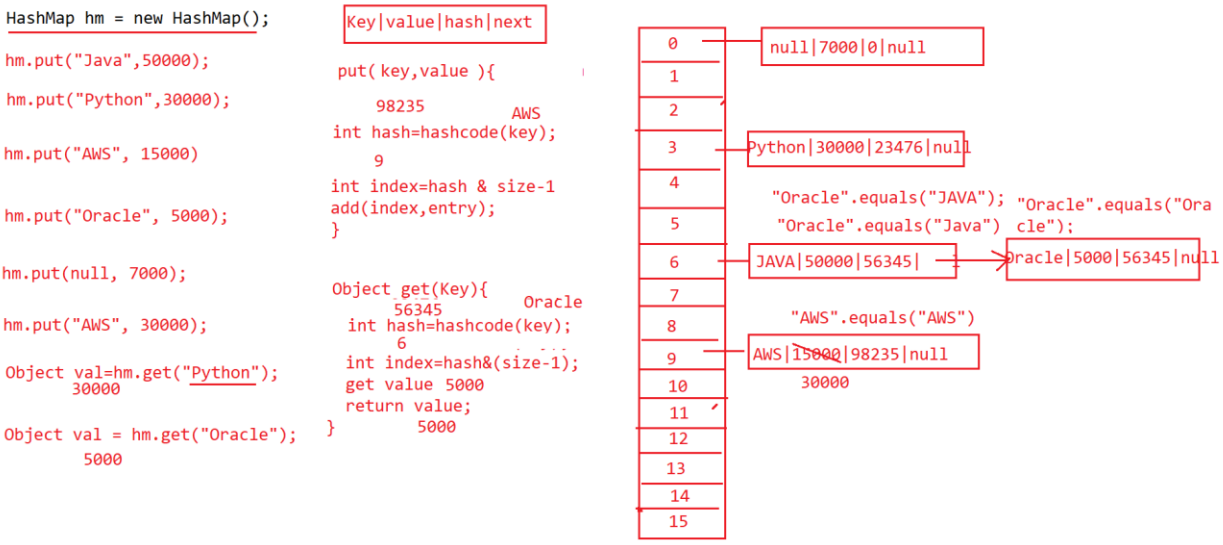
—----------

1. When we create a HashMap object , automatically a HashMap is created with the 16 buckets size, where each Bucket is able to manage a linked list and it is able to store entries with the following format.

   |Key|Value|hash|next|

2. When we add an entry to the HashMap by using put() method then put() will perform the following actions.
   a. Calculate Hash value for the key by using hashcode() method.
   b. Calculate the bucket position by using the following formula.
      int position = hash & (size-1)
   c. Add the new entry at the generated position.
   d. If we have already an entry at the generated position value then it will compare the key of the new entry with the key of the existing entry by using equals() method, if the equals() method returns true value then the put() method will replace the existing value with the new value. If the equals() method returns false value then the put() method will add the new Entry as a Second entry at the same position as a link to the first entry.
   e. If the key is a null then its entry will be added at 0th position

3. When we search for an entry by using get() then the get() method will perform the following actions.
   a. It will calculate the hash value for the provided key.
   b. It will calculate the bucket index/Position value for the key
   c. It will goto the bucket position and check whether it is a single entry or multiple entries , if we have only one entry at the bucket position then the get() method will read the value from the entry and it will return that value. If we have multiple entries then get() method will compare the provided key with the existing entry key by using equals() method, if it return true then it will return value from the current entry, if it returns false value then get() method will goto the next entry as per the next field reference at current entry.



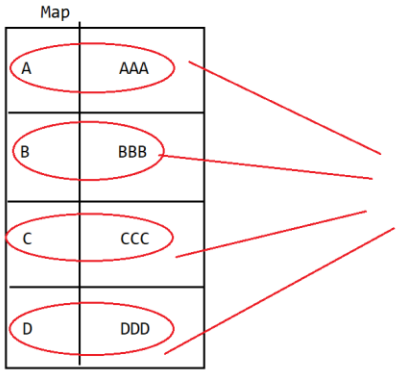
Entry:
—------
Each and every Key-Value pair in Map is called an Entry.

To represent Entry in Java applications Collection Framework has provided an inner interface in the form of Map.Entry.

Map.Entry has provided the following methods to get key, value,...

1. public Object getKey(): It is able to return the key existing in the Entry object.
2. public Object getValue(): It is able to return value object existing the Entry object.
3. public void setValue(Object obj): It is able to override the value of the Entry object with the provided new Value.

```
Map map = new HashMap();
map.put("A","AAA");
map.put("B","BBB");
map.put("C","CCC");
map.put("D","DDD");
```




```
public interface Map{
   public interface Entry{
      public Object getKey();
      public Object getValue();
      public void setValue(Object obj);
      -------
   }
}
```

EX:

```java
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
Set entrySet = map.entrySet();
Iterator iterator = entrySet.iterator();
```

```
while (iterator.hasNext()){
Map.Entry entry = (Map.Entry) iterator.next();
System.out.println(entry.getKey()+"------
>"+entry.getValue());
}
}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
A------>AAA
B------>BBB
C------>CCC
D------>DDD
```

EX:

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("C", "CCC");
map.put("D", "DDD");
System.out.println(map);
Set entrySet = map.entrySet();
Iterator iterator = entrySet.iterator();
while (iterator.hasNext()){
Map.Entry entry = (Map.Entry) iterator.next();
String key = (String) entry.getKey();
if(key.equals("B")){
entry.setValue("XXX");
}
```

```
}
System.out.println(map);
}
}
```

LinkedHashMap:
—--------------
Q)What are the differences between HashMap and LinkedHashMap?
—---
Ans:
—----

1. HashMap was introduced in JDK1.2 version.
   LinkedHashMap was introduced in JDK1.4 version

2. HashMap has the internal data structure Hashtable.
   LinkedHashMap has the internal data structure Hashtable+LinkedList.

3. HashMap does not follow insertion order.
   LionkedHashMap follows the Insertion order.

4. LinkedHashMap is a child class to HashMap.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
HashMap hm = new HashMap();
hm.put("F", "FFF");
hm.put("A", "AAA");
hm.put("E", "EEE");
hm.put("B", "BBB");
hm.put("D", "DDD");
hm.put("C", "CCC");
System.out.println(hm);
```

```
LinkedHashMap lhm = new LinkedHashMap();
lhm.put("F", "FFF");
lhm.put("A", "AAA");
lhm.put("E", "EEE");
lhm.put("B", "BBB");
lhm.put("D", "DDD");
lhm.put("C", "CCC");
System.out.println(lhm);
}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD, E=EEE, F=FFF}
{F=FFF, A=AAA, E=EEE, B=BBB, D=DDD, C=CCC}
```

IdentityHashMap:

—---

Q)What is the difference between equals() method and == operator?

—---

Ans:

—---

== operator is a comparison operator , it is able to compare the provided two operands whether the provided operands are the same or not, here the provided operands may be two primitive values or may be two object reference values.

Initially, equals() method was defined in the java.lang.Object class, it was implemented in such a way that to compare two object reference values , later on the Object class equals() method was overridden in the some of the predefined classes like String, wrapper classes,.....

In String class and in the wrapper classes, equals() method was implemented to perform content comparison instead of the references comparison.

EX:

```
import java.util.*;
class A{

}
```

```java
public class Main {
public static void main(String[] args) {
int a = 10;
int b = 10;

A a1 = new A();
A a2 = new A();

String str1 = new String("abc");
String str2 = new String("abc");

System.out.println(a == b);// true
System.out.println(a1 == a2);// false
System.out.println(str1 == str2);// false
System.out.println();

System.out.println(a1.equals(a2));// false
System.out.println(str1.equals(str2));// true

}
}
```

```
true
false
false

false
true
```

Q)What are the differences between HashMap and IdentitHashMap?

Ans:

---

1. HashMap was introduced in JDK1.2 version.
   IdentityHashMap was introduced in JDK1.4 version.

2. HashMap is able to use equals() method to check duplicate keys.
IdentityHashMap is able to use == operator to check duplicate keys.

EX:

```
import java.util.*;
class A{

}
public class Main {
public static void main(String[] args) {
Integer in1 = new Integer(10);
Integer in2 = new Integer(10);

HashMap hm = new HashMap();
hm.put(in1, "AAA");// {10=AAA}
hm.put(in2, "BBB");//in2.equals(in1);=>{10=BBB}
System.out.println(hm);

IdentityHashMap ihm = new IdentityHashMap();
ihm.put(in1, "AAA");//{10=AAA}
ihm.put(in2, "BBB");//in2==in1 ==> false ==> {10=AAA,
10=BBB}
System.out.println(ihm);
}
}
```

```
{10=BBB}
{10=AAA, 10=BBB}
```

WeakHashMap:
—----------
In Java programming , when we create objects as per the requirement, internally JVM uses a separate component in the form of Garbage Collector to destroy objects.

In Java, the garbage collector is able to destroy objects as per our requirement also.

Steps:

1. Nullify the object reference to make eligible an object for the Garbage Collection.
   A a = new A();
   a = null;
2. Activate Garbage Collector and make the Garbage Collector to destroy the objects which are not having references.
   System.gc();

Note: When we access System.gc() method, Garbage COllector will be activated, Garbage Collector will search for the objects which are not having references, Garbage Collector will access finalize() method just before destroying the object inorder to give final intimation about to destroy the objects.

EX:

```
import java.util.*;
class A{
A(){
System.out.println("Object Creating......");
}

@Override
protected void finalize() throws Throwable {
System.out.println("Object Destroying.......");
}
}
public class Main {
public static void main(String[] args) {
A a = new A();
a = null;
System.gc();
}
```

```
}
```

```
Object Creating......
Object Destroying.......
```

Q)What is the difference between HashMap and WeakHashMap?

—----------

Ans:

—---

HashMap is able to protect its elements once the element is associated with the HashMap, where HashMap will not allow Garbage Collector to destroy the associated elements.

WeakHashMap is unable to provide protection for its associated elements, it allows Garbage Collector to destroy its associated elements.

EX:

```
import java.util.*;
class A{
@Override
public String toString() {
return "A";
}
}
public class Main {
public static void main(String[] args) {
A a = new A();
HashMap hm = new HashMap();
hm.put(a, "AAA");
System.out.println("Before GC: HashMap : "+hm);
a = null;
System.gc();
System.out.println("After GC: HashMap : "+hm);
System.out.println();

A a1 = new A();
WeakHashMap whm = new WeakHashMap();
whm.put(a1, "AAA");
System.out.println("Before GC: WeakHashMap : "+whm);
a1 = null;
System.gc();
System.out.println("After GC: WeakHashMap : "+whm);
```

```
}
}
```

```
Before GC: HashMap  : {A=AAA}
After GC:  HashMap  : {A=AAA}

Before GC: WeakHashMap : {A=AAA}
After GC:  WeakHashMap : {}
```

SortedMap:

—----------

1. It was introduced in JDK1.2 version.
2. It is not a legacy Map.
3. It is a direct child interface to the Map interface.

4. It is able to manage all the elements in the form of Key-Value pairs, where both Keys and Values are objects.
5. In Key-Value pairs, keys must be unique but values may be duplicated.
6. It does not follow insertion order.
7. It follows Keys Sorting order.
8. It does not allow null elements at the keys side, but we can provide null elements at the values side.If we add any null element at the keys side then JVM will raise java.lang.NullPointerException.
9. It allows only Homogeneous elements at the keys side, if we add heterogeneous elements at the keys side then JVM will raise an exception like java.lang.ClassCastException.
10. It allows only Comparable Elements at the Keys side, if we provide non comparable elements at the keys side then JVM will raise an exception like java.lang.ClassCastException.

Methods:

1. public SortedMap headMap(Object obj)
   It is able to return a SortedMap object containing all the key-values whose keys are less than the specified key.

2. public SortedMap tailMap(Object key):
   It is able to return a SortedMap object containing all the key-value pairs whose keys are greater than or equal to the specified key.

3. public SortedMap subMap(Object fromKey, Object toKey):

It is able to return a SortedMap object containing all the key-value pairs whose keys must be greater than or equal to the specified fromKey and less than the specified toKey.

4. public Object firstKey():
   It is able to return the first elements at the keys side.

5. public Object lastKey():
   It is able to return the last element at the Keys side.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
SortedMap sortedMap = new TreeMap();
sortedMap.put("F", "FFF");
sortedMap.put("A", "AAA");
sortedMap.put("E", "EEE");
sortedMap.put("B", "BBB");
sortedMap.put("D", "DDD");
sortedMap.put("C", "CCC");
System.out.println(sortedMap);
System.out.println(sortedMap.headMap("D"));
System.out.println(sortedMap.tailMap("D"));
System.out.println(sortedMap.subMap("B", "E"));
System.out.println(sortedMap.firstKey());
System.out.println(sortedMap.lastKey());
}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD, E=EEE, F=FFF}
{A=AAA, B=BBB, C=CCC}
{D=DDD, E=EEE, F=FFF}
{B=BBB, C=CCC, D=DDD}
A
F
```

NavigableMap:
—————————
It was introduced in JDK1.6 version, it is a child interface to the SOrtedMap, it is the same as the SortedMap and it has defined some methods to perform navigation over the elements.

Methods

1. public NavigableMap descendingMap():
   It is able to return a NavigableMap object containing all the key-value pairs in the keys descending order.
2. public Map.Entry ceilingEntry(Object key):
   It is able to return a lower entry among all the entries whose keys are greater than or equal to the specified key.
3. public Map.Entry higherEntry(Object key):
   It is able to return a lower entry among all the entries whose keys are greater than the specified key.

4. Public Map.Entry floorEntry(Object key):
   It is able to return a higher Entry among all the entries whose keys are less than or equal to the specified key.

5. Public Map.Entry lowerEntry(Object key):
   It is able to return a higher Entry among all the entries whose keys are less than the specified key.
6. public Map.Entry pollFirstEntry():
   It removes the first entry from the NavigableMap.

7. Public Map.Entry pollLastEntry():
   It removes the last entry from the NavigableMap.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
NavigableMap navigableMap = new TreeMap();
navigableMap.put("F", "FFF");
navigableMap.put("A", "AAA");
navigableMap.put("E", "EEE");
navigableMap.put("B", "BBB");
navigableMap.put("D", "DDD");
navigableMap.put("C", "CCC");
System.out.println(navigableMap);
System.out.println(navigableMap.descendingMap());
```

```
System.out.println(navigableMap.ceilingEntry("D"));
System.out.println(navigableMap.higherEntry("D"));
System.out.println(navigableMap.floorEntry("D"));
System.out.println(navigableMap.lowerEntry("D"));
System.out.println(navigableMap.pollFirstEntry());
System.out.println(navigableMap.pollLastEntry());
System.out.println(navigableMap);

}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD, E=EEE, F=FFF}
{F=FFF, E=EEE, D=DDD, C=CCC, B=BBB, A=AAA}
D=DDD
E=EEE
D=DDD
C=CCC
A=AAA
F=FFF
{B=BBB, C=CCC, D=DDD, E=EEE}
```

TreeMap:

—-------

1. It was introduced in JDK1.2 version.
2. It is not a legacy Map.
3. It has provided implementations for all methods of the interfaces like NavigableMap, SortedMap and Map.
4. It allows the elements in the form of Key-Value pairs, where both keys and values are objects.
5. In key-Value pairs, keys must be unique, keys must not be duplicated, but values may be duplicated.
6. It does not follow Insertion.
7. It follows Sorting Order.
8. It does not allow null elements at the keys side, but values are able to allow any number of null elements, if we add any null element at the keys side then JVM will raise an exception like java.lang.NullPointerException.
9. It allows only Homogeneous elements at the keys side, but values are able to allow heterogeneous elements, if we add heterogeneous elements at the keys side then JVM will raise an exception like java.lang.ClassCastException.

10. It is able to allow only Comparable elements at the keys side, but values are able to allow both comparable and non comparable elements, if we add a non comparable element at keys side then JVM will raise an exception like java.lang.ClassCastException.

    Note: If we want to add non comparable elements at the keys side then we must use java.util.Comparator.

11. It has an internal data structure like “Red-Black Tree”.

12. It is a non synchronized map.
13. No method is synchronized in TreeMap.
14. It allows more than one thread at a time to access data.
15. It follows parallel execution of the threads.
16. It will reduce application execution time.
17. It will improve application performance.
18. It is not guaranteed for data consistency.
19. It is not a Threadsafe.

Constructors:
—----------

1. public TreeMap():
   It is able to create an empty TreeMap object.
   EX:

```
import java.util.*;
public class Main {
public static void main(String[] args) {
TreeMap treeMap = new TreeMap();
System.out.println(treeMap);
}
}
```

   {}

2. public TreeMap(SortedMap sm):
   It is able to create a TreeMap object with all the entries of the specified SortedMap.
   EX:

```
import java.util.*;
public class Main {
public static void main(String[] args) {
```

```
SortedMap sortedMap = new TreeMap();
sortedMap.put("D", "DDD");
sortedMap.put("A", "AAA");
sortedMap.put("C", "CCC");
sortedMap.put("B", "BBB");
System.out.println(sortedMap);
TreeMap treeMap = new TreeMap(sortedMap);
System.out.println(treeMap);
}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
{A=AAA, B=BBB, C=CCC, D=DDD}
```

3. public TreeMap(Map map):
   It is able to create a TreeMap object with all entries of the specified Map object.
   EX:

```
import java.util.*;
public class Main {
public static void main(String[] args) {
Map map = new HashMap();
map.put("A", "AAA");
map.put("B", "BBB");
map.put("F", "FFF");
map.put("E","EEE");
map.put("C","CCC");
map.put("D", "DDD");
System.out.println(map);
TreeMap treeMap = new TreeMap(map);
System.out.println(treeMap);
}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD, E=EEE, F=FFF}
{A=AAA, B=BBB, C=CCC, D=DDD, E=EEE, F=FFF}
```

4. public TreeMap(Comparator c):
   It is able to create a TreeMap object with the explicit sorting through the provided Comparator.

EX:

```
import java.util.*;
class A{

}
public class Main {
public static void main(String[] args) {
TreeMap tm = new TreeMap();
tm.put("D", "DDD");
tm.put("A", "AAA");
tm.put("C", "CCC");
tm.put("B", "BBB");
System.out.println(tm);
tm.put("B", "XXX");
System.out.println(tm);
tm.put("E","CCC");
System.out.println(tm);
//tm.put(null, "EEE"); -->
java.lang.NullPointerException
tm.put("F", null);
System.out.println(tm);
//tm.put(10, 100); ----> java.lang.ClassCastException
A a = new A();
//tm.put(a, "YYY"); ---> java.lang.ClassCastException
System.out.println(tm);
}
}
```

```
{A=AAA, B=BBB, C=CCC, D=DDD}
{A=AAA, B=XXX, C=CCC, D=DDD}
{A=AAA, B=XXX, C=CCC, D=DDD, E=CCC}
```

{A=AAA, B=XXX, C=CCC, D=DDD, E=CCC, F=null}
{A=AAA, B=XXX, C=CCC, D=DDD, E=CCC, F=null}

EX: Comparable

```
import java.util.*;
class Address{
private String hno;
private String street;
private String city;
private String state;

public Address(String hno, String street, String
city, String state) {
this.hno = hno;
this.street = street;
this.city = city;
this.state = state;
}

@Override
public String toString() {
return "Address{" +
"hno='" + hno + '\'' +
", street='" + street + '\'' +
", city='" + city + '\'' +
", state='" + state + '\'' +
'}'+"\n";
}
}
class Employee implements Comparable{
private int eno;
private String ename;
private float esal;
```

```
public Employee(int eno, String ename, float esal) {
this.eno = eno;
this.ename = ename;
this.esal = esal;

}

@Override
public String toString() {
return "Employee{" +
"eno=" + eno +
", ename='" + ename + '\'' +
", esal=" + esal +
'}';
}

@Override
public int compareTo(Object o) {
Employee emp = (Employee)o;
return -this.ename.compareTo(emp.ename);
}
}
public class Main {
public static void main(String[] args) {
Employee emp1 = new Employee(111, "Durga", 15000);
Employee emp2 = new Employee(222, "Anil", 5000);
Employee emp3 = new Employee(333, "Rakesh", 10000);
Employee emp4 = new Employee(444, "Pardhu", 30000);

Address addr1 = new Address("123/3rt", "Z Street",
"Hyd", "Telangana");
Address addr2 = new Address("423/1sr", "MG Road",
"Chennai", "Tamilnadu");
```

```
Address addr3 = new Address("1145/s-23", "PS Road",
"Pune", "Maharastra");
Address addr4 = new Address("567/e432", "X Street",
"New Delhi", "Delhi");

TreeMap tm = new TreeMap();
tm.put(emp1, addr1);
tm.put(emp2, addr2);
tm.put(emp3, addr3);
tm.put(emp4, addr4);
System.out.println(tm);

}
}
```

```
{Employee{eno=333, ename='Rakesh', esal=10000.0}=Address{hno='1145/s-23', street='PS Road', city='Pune',
state='Maharastra'}
, Employee{eno=444, ename='Pardhu', esal=30000.0}=Address{hno='567/e432', street='X Street', city='New
Delhi', state='Delhi'}
, Employee{eno=111, ename='Durga', esal=15000.0}=Address{hno='123/3rt', street='Z Street', city='Hyd',
state='Telangana'}
, Employee{eno=222, ename='Anil', esal=5000.0}=Address{hno='423/1sr', street='MG Road', city='Chennai',
state='Tamilnadu'}
}
```

EX: Comparator:

```
import java.util.*;
class Address{
private String hno;
private String street;
private String city;
private String state;

public Address(String hno, String street, String
city, String state) {
this.hno = hno;
this.street = street;
this.city = city;
```

```
this.state = state;
}

@Override
public String toString() {
return "Address{" +
"hno='" + hno + '\'' +
", street='" + street + '\'' +
", city='" + city + '\'' +
", state='" + state + '\'' +
'}'+"\n";
}
}
class Employee {
private int eno;
private String ename;
private float esal;

public int getEno() {
return eno;
}

public String getEname() {
return ename;
}

public float getEsal() {
return esal;
}

public Employee(int eno, String ename, float esal) {
this.eno = eno;
this.ename = ename;
this.esal = esal;
```

```
}

@Override
public String toString() {
return "Employee{" +
"eno=" + eno +
", ename='" + ename + '\'' +
", esal=" + esal +
'}';
}

}
class EmployeeComparator implements Comparator{

@Override
public int compare(Object o1, Object o2) {
Employee emp1 = (Employee) o1;
Employee emp2 = (Employee) o2;
return (int)(emp1.getEsal() - emp2.getEsal());
}
}
public class Main {
public static void main(String[] args) {
Employee emp1 = new Employee(111, "Durga", 15000);
Employee emp2 = new Employee(222, "Anil", 5000);
Employee emp3 = new Employee(333, "Rakesh", 10000);
Employee emp4 = new Employee(444, "Pardhu", 30000);

Address addr1 = new Address("123/3rt", "Z Street",
"Hyd", "Telangana");
Address addr2 = new Address("423/1sr", "MG Road",
"Chennai", "Tamilnadu");
```

```
Address addr3 = new Address("1145/s-23", "PS Road",
"Pune", "Maharastra");
Address addr4 = new Address("567/e432", "X Street",
"New Delhi", "Delhi");

EmployeeComparator employeeComparator = new
EmployeeComparator();
TreeMap tm = new TreeMap(employeeComparator);
tm.put(emp1, addr1);
tm.put(emp2, addr2);
tm.put(emp3, addr3);
tm.put(emp4, addr4);
System.out.println(tm);

}
}
```

```
{Employee{eno=222, ename='Anil', esal=5000.0}=Address{hno='423/1sr',
street='MG Road', city='Chennai', state='Tamilnadu'}
, Employee{eno=333, ename='Rakesh', esal=10000.0}=Address{hno='1145/s-23',
street='PS Road', city='Pune', state='Maharastra'}
, Employee{eno=111, ename='Durga', esal=15000.0}=Address{hno='123/3rt',
street='Z Street', city='Hyd', state='Telangana'}
, Employee{eno=444, ename='Pardhu', esal=30000.0}=Address{hno='567/e432',
street='X Street', city='New Delhi', state='Delhi'}
}
```

Q)If we provide both Implicit Sorting through Comparable and Explicit Sorting through Comparator to the TreeMap then which Sorting logic is used by the TreeMap to sort out all the elements in TreeMap?

—----------

Ans:

—---

If we provide both Implicit Sorting through Comparable and Explicit Sorting through Comparator to the TreeMap then TreeMap will use Explicit Sorting which is provided through Comparator.

EX:

```
import java.util.*;
```

```
class Address{
private String hno;
private String street;
private String city;
private String state;

public Address(String hno, String street, String
city, String state) {
this.hno = hno;
this.street = street;
this.city = city;
this.state = state;
}

@Override
public String toString() {
return "Address{" +
"hno='" + hno + '\'' +
", street='" + street + '\'' +
", city='" + city + '\'' +
", state='" + state + '\'' +
'}'+"\n";
}
}
class Employee implements Comparable{
private int eno;
private String ename;
private float esal;

public int getEno() {
return eno;
}

public String getEname() {
```

```
return ename;
}

public float getEsal() {
return esal;
}

public Employee(int eno, String ename, float esal) {
this.eno = eno;
this.ename = ename;
this.esal = esal;

}

@Override
public String toString() {
return "Employee{" +
"eno=" + eno +
", ename='" + ename + '\'' +
", esal=" + esal +
'}';
}

@Override
public int compareTo(Object o) {
Employee emp = (Employee)o;
return this.ename.compareTo(emp.ename);
}
}
class EmployeeComparator implements Comparator{

@Override
public int compare(Object o1, Object o2) {
Employee emp1 = (Employee) o1;
```

```
Employee emp2 = (Employee) o2;
return (int)(emp1.getEsal() - emp2.getEsal());
}
}
public class Main {
public static void main(String[] args) {
Employee emp1 = new Employee(111, "Durga", 15000);
Employee emp2 = new Employee(222, "Anil", 5000);
Employee emp3 = new Employee(333, "Rakesh", 10000);
Employee emp4 = new Employee(444, "Pardhu", 30000);

Address addr1 = new Address("123/3rt", "Z Street",
"Hyd", "Telangana");
Address addr2 = new Address("423/1sr", "MG Road",
"Chennai", "Tamilnadu");
Address addr3 = new Address("1145/s-23", "PS Road",
"Pune", "Maharastra");
Address addr4 = new Address("567/e432", "X Street",
"New Delhi", "Delhi");

EmployeeComparator employeeComparator = new
EmployeeComparator();
TreeMap tm = new TreeMap(employeeComparator);
tm.put(emp1, addr1);
tm.put(emp2, addr2);
tm.put(emp3, addr3);
tm.put(emp4, addr4);
System.out.println(tm);

}
}
```

```
{Employee{eno=222, ename='Anil', esal=5000.0}=Address{hno='423/1sr',
street='MG Road', city='Chennai', state='Tamilnadu'}
```

, Employee{eno=333, ename='Rakesh', esal=10000.0}=Address{hno='1145/s-23', street='PS Road', city='Pune', state='Maharastra'}
, Employee{eno=111, ename='Durga', esal=15000.0}=Address{hno='123/3rt', street='Z Street', city='Hyd', state='Telangana'}
, Employee{eno=444, ename='Pardhu', esal=30000.0}=Address{hno='567/e432', street='X Street', city='New Delhi', state='Delhi'}
}

Hashtable:
—---------
Q)What are the differences between HashMap and Hashtable?
—---
Ans:
—----

1. HashMap was introduced in JDK1.2 version.
   Hashtable was introduced in JDK1.0 version.

2. HashMap is not a legacy map.
   Hashtable is a Legacy Map.

3. HashMap uses Hashtable as an internal data Structure.
   Hashtable uses the same Hashtable as an internal data Structure.

4. HashMap allows only one null element at the keys side and any number of null elements at the values side.

   Hashtable does not allow null elements at both Keys and Values side. If we add null elements at both Keys and values sides then JVM will raise an exception like java.lang.NullPointerException.

5. HashMap is not a synchronized Map.
   Hashtable is a synchronized Map.

6. HashMap does not have synchronized methods.
   Hashtable has a number of synchronized methods.

7. HashMap is able to allow more than one thread at a time to access data.
   Hashtable is able to allow only one thread at a time to access data.

8. HashMap follows parallel execution of the Threads.
   Hashtable follows sequential execution of the threads.

9. HashMap is able to reduce application execution time.
   Hashtable is able to increase application execution time.

10. HashMap is able to improve application performance.
    Hashtable is able to reduce application performance.

11. HashMap is not giving guarantees for the data consistency.
    Hashtable is giving guarantees for the data consistency.

12. HashMap is not a Threadsafe resource.
    Hashtable is a Threadsafe resource.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
HashMap hm = new HashMap();
hm.put("A", "AAA");
hm.put("B", "BBB");
System.out.println(hm);
hm.put(null, "CCC");
hm.put("C", null);
hm.put("D", null);
System.out.println(hm);

Hashtable ht = new Hashtable();
ht.put("A", "AAA");
ht.put("B", "BBB");
//ht.put(null, "CCC"); --->
java.lang.NullPointerException
//ht.put("C", null); ---->
java.lang.NullPointerException
System.out.println(ht);
}
}
```

{A=AAA, B=BBB}
{null=CCC, A=AAA, B=BBB, C=null, D=null}
{A=AAA, B=BBB}

Properties:
—----------
The main purpose of java.util.Properties is to represent the data of a particular properties file.

properties file is a normal file with the .properties extension , it is able to represent the data in the form of key-value pairs.

abc.properties
—-------------
eno=111
ename=AAA
esal=5000
eaddr=Hyd

The main purpose of the properties files in the java applications is

1. In any Java application, if we want to change the data frequently then we must perform recompilation for every change in the java application, it is not suggestible in the enterprise applications, here to avoid the number of recompilations we have to use the properties file, where we have to provide the frequently changed data and we have to get the data from the properties file to the java application as per the requirement.
2. To manage the Jdbc parameters like driver class names, driver url, database user names, database password , connection pooling mechanisms ,.... in Jdbc applications We need to use properties files .
3. To manage all the GUI components labels in GUI applications we need a properties file.
4. To manage all the local users' respective messages in the Internationalization applications we need properties files.
5. To manage user defined exception messages in the Exception handling we need properties files.
6. To manage all the validation messages in the Data Validations in the enterprise applications we need properties files.
7. To manage security messages in the data security we need properties files.
8. To provide server configuration details like server port numbers, server ip addresses,.... in web applications we need properties files.
   —-----

If we want to use properties files in Java applications then we have to use the following steps.

1. Create properties file and send data from kJava application to the properties file.
2. Get Data from the properties file to the Java application.

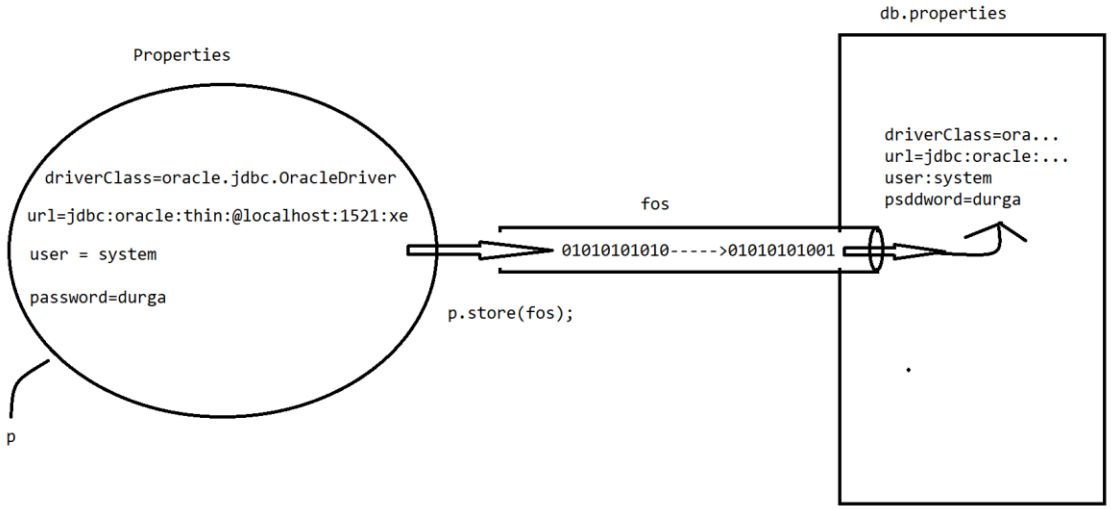
Sending data from Java application to the properties file:

1. Create properties file by using FileOutputStream:
   FileOutputStream fos = new FileOutputStream("E:/abc/db.properties");

2. Create java.util.Properties class Object and keep data in the Properties Object:

   Properties p = new Properties();
   p.setProperty("driverClass","oracle.jdbc.OracleDriver");
   p.setProperty("url","jdbc:oracle:thin:@localhost:1521:xe");
   p.setProperty("user","system");
   p.setProperty("password","durga");

3. Send data from the Properties object to the properties file.
   p.store(fos, "JDBC Parameters");



EX:

```
import java.io.FileOutputStream;
import java.util.*;
```

```
public class Main {
public static void main(String[] args) throws
Exception{
FileOutputStream fos = new
FileOutputStream("E:/abc/xyz/db.properties");

Properties properties = new Properties();
properties.setProperty("driverClass",
"oracle.jdbc.OracleDriver");
properties.setProperty("url","jdbc:oracle:thin:@local
host:1521:xe");
properties.setProperty("user","system");
properties.setProperty("password","durga");

properties.store(fos, "Jdbc Parameters");

System.out.println("Data sent to
E:/abc/xyz/db.properties");
fos.close();

}
}
```

Reading data from the properties file to Java application:

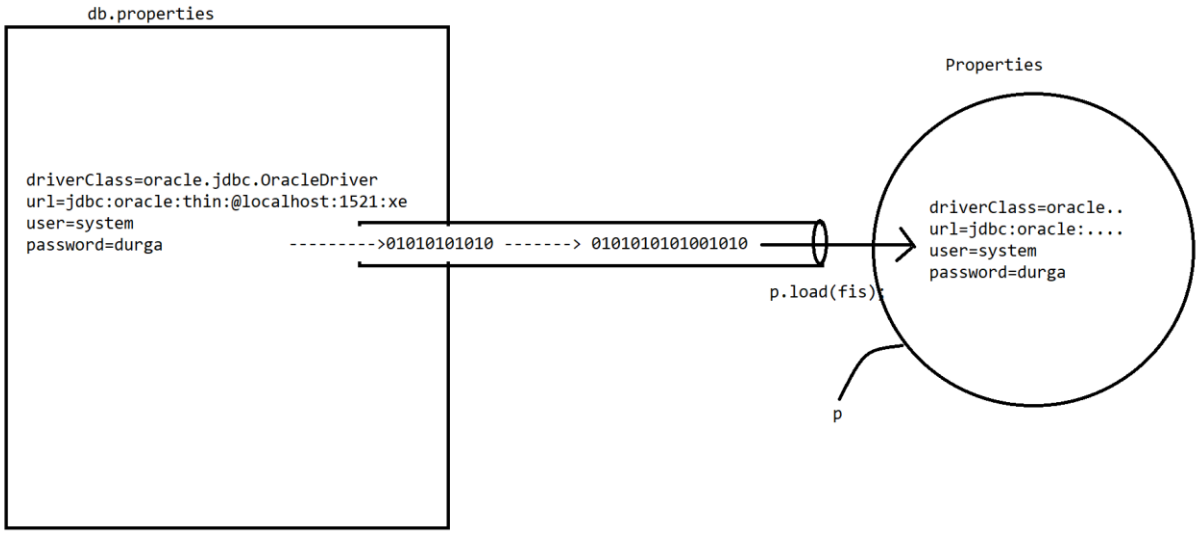
1. Create FileInputStream to read data from the properties file.
   FileInputSTream fis = new FileInputStream(“E:/abc/xyz/db.properties”);

2. Create Properties class object:
   Properties p = new Properties();

3. Load data from the FileInputStream to the Properties object:
   p.load(fis);

4. Read and display data from the Properties object:
   Sopln(p.getProperty(“driverClass”);
   Sopln(p.getProperty(“url”);

Sopln(p.getProperty(“user”);
Sopln(p.getProperty(“password”);



EX:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.*;

public class Main {
public static void main(String[] args) throws
Exception{
FileInputStream fis = new
FileInputStream("E:/abc/xyz/db.properties");
Properties p = new Properties();
p.load(fis);
System.out.println("JDBC Parameters");
System.out.println("--------------------------");
System.out.println("DriverClass :
"+p.getProperty("driverClass"));
System.out.println("Driver URL :
"+p.getProperty("url"));
System.out.println("Db User Name :
"+p.getProperty("user"));
```

```
System.out.println("DB Password :
"+p.getProperty("password"));

}
}
```

JDBC Parameters

DriverClass : oracle.jdbc.OracleDriver
Driver URL : jdbc:oracle:thin:@localhost:1521:xe
Db User Name : system
DB Password : durga

Queue:
—---

1. It was introduced in JDK1.5 version.
2. It is a child interface to Collection interface.
3. It is able to follow the FIFO algorithm to manage the elements, we can change this algorithm in the implementation classes as per the requirement.
4. It is not index based.
5. It is able to manage the elements as per prior to the process.
6. It allows duplicate elements.
7. It does not follow the Insertion order.
8. It does not follow Sorting order.
9. It does not allow null elements.
10. It allows only Homogeneous elements.
11. It allows only Comparable Elements, if we add non comparable elements to Queue then JVM will raise an exception like java.lang.ClassCastException.
    Note: If we want to add Non comparable elements to the Queue then we must use Comparator.

Methods:

1. public void offer(Object obj):
   It is able to add the provided element to the Queue.

2. public void add(Object obj):
   It is able to add the provided element to the Queue through offer() method.
3. public Object peek():
   It is able to return the Head Element of the Queue.

4. public Object element():
   It is able to return the Head Element of the Queue.

Q)What is the difference between peek() and element() method?
—--
Ans:
—---
If we access peek() method on an empty queue then JVM will return null value , but if we access element() method on an empty Queue then JVM will raise an exception like java.util.NoSuchElementException

5. public Object poll():
   It is able to read and remove the Head element of the Queue.

6. public Object remove():
   It is able to read and remove the Head Element of the Queue.

Q)What is the difference between poll() method and remove() method?
—--
Ans:
—---
If we access poll() method on an empty Queue then poll() method will return null value, but if we access remove() method on an empty Queue then JVM will raise an exception like java.util.NoSuchElementException.

EX:

```
import java.util.PriorityQueue;
import java.util.Queue;

public class Main {
public static void main(String[] args) {
Queue queue = new PriorityQueue();
queue.offer("AAA");
queue.offer("BBB");
queue.offer("CCC");
queue.offer("DDD");
System.out.println(queue);
queue.add("EEE");
queue.add("FFF");
System.out.println(queue);
```

```
System.out.println(queue.peek());
System.out.println(queue.element());
System.out.println(queue.poll());
System.out.println(queue);
System.out.println(queue.remove());
System.out.println(queue);
/*
Queue queue1 = new PriorityQueue();
System.out.println(queue1.peek()); ---> Null
System.out.println(queue1.element()); ---->
java.util.NoSuchElementException
*/
/*
Queue queue2 = new PriorityQueue();
System.out.println(queue2.poll()); ----> Null
System.out.println(queue2.remove()); --->
java.util.NoSuchElementException
*/
}
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD, EEE, FFF]
AAA
AAA
AAA
[BBB, DDD, CCC, FFF, EEE]
BBB
[CCC, DDD, EEE, FFF]
```

EX:

```
import java.util.PriorityQueue;
import java.util.Queue;

class A{

}
public class Main {
```

```
public static void main(String[] args) {
Queue queue = new PriorityQueue();
queue.add(6);
queue.add(2);
queue.add(8);
queue.add(5);
queue.add(2);
queue.add(10);
//queue.add(null); ---> NullPointerException
//queue.add("abc"); --> ClassCastException
System.out.println(queue);
Queue q1 = new PriorityQueue();
//q1.add(new A());----> ClassCastException
System.out.println(q1);
}
}
```

PriorityQueue:

---

1. It was introduced in JDK1.5 version.
2. It is not a legacy Collection.
3. It is an implementation class to Queue interface.

4. It is not index based.
5. It is able to manage all the elements prior to processing as per the priorities.
6. It allows duplicate elements.
7. It does not follow the Insertion order.
8. It does not follow Sorting order.
9. It does not allow null elements.
10. It allows only Homogeneous elements.
11. It allows only Comparable Elements, if we add non comparable elements to Queue then JVM will raise an exception like java.lang.ClassCastException.
    Note: If we want to add Non comparable elements to the Queue then we must use Comparator.

12. Its initial capacity is 11 elements.
13. It is not a Synchronized Collection.
14. No method is Synchronized in PriorityQueue.
15. It allows more than one thread at a time to access data.

16. It follows parallel execution of the threads.
17. It reduces application execution time.
18. It improves application performance.
19. It is not giving guarantee for the Data consistency.
20. It is not a Threadsafe.

Constructors:
—————————
1. public PriorityQueue():
It is able to create an empty PriorityQueue object with the default initial capacity of 11 elements.
EX:

```
import java.io.PrintStream;
import java.util.PriorityQueue;

public class Main {
public static void main(String[] args) {
PriorityQueue pq = new PriorityQueue();
System.out.println(pq);
}
}
```

2. public PriorityQueue(int capacity):
It is able to create an empty PriorityQueue with the specified capacity value.
EX:

```
import java.util.PriorityQueue;

public class Main {
public static void main(String[] args) {
PriorityQueue pq = new PriorityQueue(20);
System.out.println(pq);
}
}
```

3. public PriorityQueue(PriorityQueue pq):
It is able to create a PriorityQueue object with all elements of the Specified PriorityQueue.

EX:

```
import java.util.PriorityQueue;

public class Main {
public static void main(String[] args) {
PriorityQueue pq1 = new PriorityQueue();
pq1.add("AAA");
pq1.add("BBB");
pq1.add("CCC");
pq1.add("DDD");
System.out.println(pq1);

PriorityQueue priorityQueue = new PriorityQueue(pq1);
System.out.println(priorityQueue);
}
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
```

4.public PriorityQueue(SortedSet ss):
It is able to create a Priority Queue with all elements of the specified SortedSet.

Note: It is able to convert all elements from SortedSet to PriorityQueue
EX:

```
import java.util.PriorityQueue;
import java.util.SortedSet;
import java.util.TreeSet;

public class Main {
public static void main(String[] args) {
SortedSet sortedSet = new TreeSet();
sortedSet.add("AAA");
sortedSet.add("BBB");
sortedSet.add("CCC");
sortedSet.add("DDD");
System.out.println(sortedSet);
```

```
PriorityQueue priorityQueue = new PriorityQueue(sortedSet);
System.out.println(priorityQueue);
}
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
```

5. public PriorityQueue(Collection c):
It can be used to create a PriorityQueue object with all elements of the provided Collection.

Note: It can be used to convert elements from any type of Collection[List, Set, Queue] to PriorityQueue.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add(10);
list.add(20);
list.add(30);
list.add(40);
System.out.println(list);
PriorityQueue priorityQueue1 = new PriorityQueue(list);
System.out.println(priorityQueue1);

Set set = new HashSet();
set.add("AAA");
set.add("BBB");
set.add("CCC");
set.add("DDD");
System.out.println(set);
PriorityQueue priorityQueue2 = new PriorityQueue(set);
System.out.println(priorityQueue2);

Queue queue = new PriorityQueue();
```

```
queue.add("XXX");
queue.add("YYY");
queue.add("ZZZ");
System.out.println(queue);
PriorityQueue priorityQueue3 = new PriorityQueue(queue);
System.out.println(priorityQueue3);
}
}
```

```
[10, 20, 30, 40]
[10, 20, 30, 40]
[AAA, CCC, BBB, DDD]
[AAA, CCC, BBB, DDD]
[XXX, YYY, ZZZ]
[XXX, YYY, ZZZ]
```

6. public PriorityQueue(Comparator c):
It is able to create a PriorityQueue with the Comparator provided Sorting logic inorder to arrange all the elements.

In PriorityQueue, when we add elements , automatically PriorityQueue will find highest priority element among all the elements and PriorityQueue will keep that highest Priority Element as Head Element, it will be processed immediately, in this case all the remaining elements are provided without having any particular order.

The same process will be repeated when the Head element is processed.

EX:

```
import javax.xml.stream.events.ProcessingInstruction;
import java.util.*;

public class Main {
public static void main(String[] args) {
PriorityQueue priorityQueue = new PriorityQueue();
priorityQueue.add(10);
priorityQueue.add(6);
priorityQueue.add(8);
priorityQueue.add(2);
priorityQueue.add(5);
```

```
priorityQueue.add(3);
System.out.println(priorityQueue);
System.out.println(priorityQueue.poll());
System.out.println(priorityQueue);
System.out.println(priorityQueue.poll());
System.out.println(priorityQueue);
System.out.println(priorityQueue.poll());
System.out.println(priorityQueue);
System.out.println(priorityQueue.poll());
System.out.println(priorityQueue);
System.out.println(priorityQueue.poll());
System.out.println(priorityQueue);
}
}
```

EX:

```
import javax.xml.stream.events.ProcessingInstruction;
import java.util.*;

public class Main {
public static void main(String[] args) {
PriorityQueue priorityQueue = new
PriorityQueue(Comparator.reverseOrder());
priorityQueue.add(10);
priorityQueue.add(6);
priorityQueue.add(8);
priorityQueue.add(2);
priorityQueue.add(5);
priorityQueue.add(3);
System.out.println(priorityQueue);
System.out.println(priorityQueue.poll());
System.out.println(priorityQueue);
System.out.println(priorityQueue.poll());
System.out.println(priorityQueue);
System.out.println(priorityQueue.poll());
System.out.println(priorityQueue);
```

```
System.out.println(priorityQueue.poll());
System.out.println(priorityQueue);
System.out.println(priorityQueue.poll());
System.out.println(priorityQueue);
}
}
```

```
[10, 6, 8, 2, 5, 3]
10
[8, 6, 3, 2, 5]
8
[6, 5, 3, 2]
6
[5, 2, 3]
5
[3, 2]
3
[2]
```

Deque/ Double Ended Queue:

—--------------------------

Deque was introduced in JDK1.6 version, it is a child interface to Queue interface, it is the same as the Queue but it allows insertions and deletions at both sides and it is following FIFO and LIFO algorithms, it is faster than PriorityQueue and Stack , it has both the Collections Methods.

Methods:

—-------

1. public void addFirst(Object obj):
   It is able to add the specified element as First element.

2. public void offerFirst(Object obj):
   It is able to add the specified element as First element.

EX:

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.concurrent.BlockingDeque;
import java.util.concurrent.LinkedBlockingDeque;

public class Main {
```

```
public static void main(String[] args) {
Deque deque = new ArrayDeque(3);
deque.addFirst("AAA");
deque.addFirst("BBB");
deque.addFirst("CCC");
System.out.println(deque);
deque.addFirst("DDD");
System.out.println(deque);

Deque deque1 = new ArrayDeque(3);
deque1.offerFirst("AAA");
deque1.offerFirst("BBB");
deque1.offerFirst("CCC");
System.out.println(deque1);
deque1.offerFirst("DDD");
System.out.println(deque1);

}
}
```

Q)What is the difference between addFirst() method and offerFirst() method?

Ans:

If we use addFirst() method to add an element to the BoundedDeque over its size then JVM will raise an exception like java.lang.IllegalStateException.

If we use the offerFirst() method to add an element to the BoundedDeque over its size then JVM will not raise any exception and the BoundedDeque will not allow that element.

Note: BoundedDeque is a Deque , it is able to allow the elements up to its size, it will not allow the elements over its size.

```
import java.util.concurrent.BlockingDeque;
import java.util.concurrent.LinkedBlockingDeque;

public class Main {
public static void main(String[] args) {
```

```
BlockingDeque blockingDeque = new LinkedBlockingDeque(3);
blockingDeque.addFirst("AAA");
blockingDeque.addFirst("BBB");
blockingDeque.addFirst("CCC");
System.out.println(blockingDeque);
//blockingDeque.addFirst("DDD"); --->
java.lang.IllegalStateException
blockingDeque.offerFirst("DDD");
System.out.println(blockingDeque);

}
}
```

```
[CCC, BBB, AAA]
[CCC, BBB, AAA]
```

3. public void addLast(Object obj)
   It is able to add the specified element as the last element.

4. public void offerLast(Object obj)
   It is able to add the specified element as Last element.

   Note: If we access addLast() method to add an element over the max size of BoundedDeque then JVM will raise an exception like java.lang.IllegalStateException, but if we use offerLast() method to add an element to the BoundedDeque then JVM will not raise any exception.

   EX:

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
public static void main(String[] args) {
Deque deque = new ArrayDeque(3);
deque.addLast("AAA");
deque.addLast("BBB");
deque.offerLast("CCC");
deque.offerLast("DDD");
System.out.println(deque);

}
```

```
}
```

[AAA, BBB, CCC, DDD]

5. public Object getFirst():
   It will return the first element from the Deque.

6. public Object peekFirst():
   It will return the first element from the Deque.

EX:

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
public static void main(String[] args) {
Deque deque = new ArrayDeque(3);
deque.addLast("AAA");
deque.addLast("BBB");
deque.offerLast("CCC");
deque.offerLast("DDD");
System.out.println(deque);
System.out.println(deque.getFirst());
System.out.println(deque.peekFirst());

}
}
```

[AAA, BBB, CCC, DDD]
AAA
AAA

Note: If we access getFirst() method on an empty Deque then JVM will raise an exception like java.util.NoSuchElementException, but if we access peekFirst() method over an empty Deque then JVM will not raise any exception, where JVM will return null value.

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
```

```
public static void main(String[] args) {
Deque deque = new ArrayDeque(3);
//System.out.println(deque.getFirst()); --->
java.util.NoSuchElementException
System.out.println(deque.peekFirst());

}
}
```

null

7. public Object getLast():
   It is able to return the last element from Deque.

8. public Object peekLast():
   It is able to return the last element from Deque.

   Note: If we access getLast() method over an empty Deque then JVM will raise an exception like java.util.NoSuchElementException, but if we access peekLast() method over an empty Deque then JVM will not raise any exception and JVM will return null value.

EX:

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
public static void main(String[] args) {
Deque deque = new ArrayDeque(3);
deque.addLast("AAA");
deque.addLast("BBB");
deque.offerLast("CCC");
deque.offerLast("DDD");
System.out.println(deque);
System.out.println(deque.getLast());
System.out.println(deque.peekLast());

}
}
```

9. public Object removeFirst():
   It is able to remove the First Element from the Deque.

10. public Object pollFirst():

It is able to remove the first element from the Deque.

Note: If we access removeFirst() method on an empty Deque then JVM will raise an exception like java.util.NoSuchElementException, but if we access pollFirst() method over an empty Deque then JVM will not raise any exception, JVM will return null value.

EX:

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
public static void main(String[] args) {
Deque deque = new ArrayDeque(3);
deque.addLast("AAA");
deque.addLast("BBB");
deque.offerLast("CCC");
deque.offerLast("DDD");
System.out.println(deque);
System.out.println(deque.removeFirst());
System.out.println(deque);
System.out.println(deque.pollFirst());
System.out.println(deque);
}
}
```

```
[AAA, BBB, CCC, DDD]
AAA
[BBB, CCC, DDD]
BBB
[CCC, DDD]
```

11. public Object removeLast():

It is able to remove the last element from the Deque.

12. public Object pollLast():

It is able to remove the last element from Deque.

Note: If we access removeLast() method on an empty Deque then JVM will raise an exception like java.util.NoSuchElementException, but if we

access pollLast() method on an empty Deque then JVM will not raise any exception, where JVM will return null value.

13. public boolean removeFirstOccurence(Object obj):
It is able to remove the specified element at its first occurrence in the Deque.

14. public boolean removeLastOccurence(Object obj):
It is able to remove the specified element at its last occurrence in the Deque.

Note: In the above cases, if the specified element does not exist then JVM will return false value from both the methods.

EX:

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
public static void main(String[] args) {
Deque deque = new ArrayDeque(3);
deque.add("AAA");
deque.add("BBB");
deque.add("CCC");
deque.add("DDD");
deque.add("BBB");
deque.add("EEE");
deque.add("BBB");
System.out.println(deque);
System.out.println(deque.removeFirstOccurrence("BBB"));
System.out.println(deque);
System.out.println(deque.removeLastOccurrence("BBB"));
System.out.println(deque);

}
}
```

```
[AAA, BBB, CCC, DDD, BBB, EEE, BBB]
true
[AAA, CCC, DDD, BBB, EEE, BBB]
true
[AAA, CCC, DDD, BBB, EEE]
```

EX:

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
public static void main(String[] args) {
Deque deque = new ArrayDeque(3);
deque.add("AAA");
deque.add("BBB");
deque.add("CCC");
deque.add("DDD");
deque.add("BBB");
deque.add("EEE");
deque.add("BBB");
System.out.println(deque);
System.out.println(deque.removeFirstOccurrence("ZZZ"));
System.out.println(deque);
System.out.println(deque.removeLastOccurrence("ZZZ"));
System.out.println(deque);

}
}
```

```
[AAA, BBB, CCC, DDD, BBB, EEE, BBB]
false
[AAA, BBB, CCC, DDD, BBB, EEE, BBB]
false
[AAA, BBB, CCC, DDD, BBB, EEE, BBB]
```

15. public Iterator iterator():
It is able to return an Iterator object with all the elements of the Deque.

16. public Iterator descendingIterator():
It is able to return an Iterator object with all elements of the Deque in reverse order.

EX:

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Iterator;
```

```
public class Main {
public static void main(String[] args) {
Deque deque = new ArrayDeque(3);
deque.add("AAA");
deque.add("BBB");
deque.add("CCC");
deque.add("DDD");
deque.add("EEE");
deque.add("FFF");
deque.add("GGG");
System.out.println(deque);
Iterator iterator = deque.iterator();
while (iterator.hasNext()){
System.out.println(iterator.next());
}
System.out.println();

Iterator iterator1 = deque.descendingIterator();
while (iterator1.hasNext()){
System.out.println(iterator1.next());
}

}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF, GGG]
AAA
BBB
CCC
DDD
EEE
FFF
GGG

GGG
FFF
EEE
DDD
CCC
BBB
AAA
```

ArrayDeque:
—----------

1. It was introduced in JDK1.6 version.
2. It is not a Legacy Collection.
3. It is an implementation class to the Deque interface, it has provided implementation for all methods of the Collection interface, Queue interface and Deque interface.
4. It is not index based.
5. It allows duplicate elements.
6. It follows Insertion order.
7. It does not follow Sorting order.
8. It allows heterogeneous elements.
9. It does not allow null elements.
10. Its initial capacity is 16 elements.
11. Its internal data structure is “Resizable Array”.
12. It is not a synchronized Collection.
13. No Method is synchronized in ArrayDeque.
14. It allows more than one thread at a time to access data.
15. It follows parallel execution.
16. It reduces application execution time.
17. It improves application performance.
18. It is not giving guarantees for the data consistency.
19. It is not a threadsafe resource.

Constructors:
—-------------

1. public ArrayDeque():
   It is able to create an empty ArrayDeque object with the initial capacity of 16 elements.
   EX:

```
import java.util.ArrayDeque;
public class Main {
public static void main(String[] args) {
ArrayDeque arrayDeque = new ArrayDeque();
System.out.println(arrayDeque);

}
}
```

   []
2. public ArrayDeque(int capacity):

It is able to create an empty ArrayDeque with the specified initial capacity.

EX:

```
import java.util.ArrayDeque;
public class Main {
public static void main(String[] args) {
ArrayDeque arrayDeque = new ArrayDeque(20);
System.out.println(arrayDeque);
}
}
```

[]

3. public ArrayDeque(Collection c):
   It can be used to create an ArrayDeque object with all elements of the specified Collection object.

   EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
ArrayDeque arrayDeque1 = new ArrayDeque(list);
System.out.println(arrayDeque1);

Set set = new HashSet();
set.add(111);
set.add(222);
set.add(333);
ArrayDeque arrayDeque2 = new ArrayDeque(set);
System.out.println(arrayDeque2);

Queue queue = new PriorityQueue();
queue.add("XXX");
queue.add("YYY");
queue.add("ZZZ");
ArrayDeque arrayDeque3 = new ArrayDeque(queue);
System.out.println(arrayDeque3);
}
```

```
}
```

```
[AAA, BBB, CCC]
[333, 222, 111]
[XXX, YYY, ZZZ]
```

EX:

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Iterator;

public class Main {
public static void main(String[] args) {
ArrayDeque deque = new ArrayDeque(3);
deque.add("AAA");
deque.add("GGG");
deque.add("BBB");
deque.add("FFF");
deque.add("CCC");
deque.add("EEE");
deque.add("DDD");
deque.add(10);
//deque.add(null); ---> NullPointerException

System.out.println(deque);

}
}
```

Utility Classes in Collection Framework:

1. Collections
2. Arrays
   —----
   —----

Collections:
—-----------
Q)What is the difference between Collection and Collections?
—---
Ans:
—---
Collection is an interface , it is able to represent a group of other objects as a single entity.

Collections is an utility class, it has static methods to perform the utility operations over the Collection elements like searching, sorting, copying , making Collection as a Synchronized collection,.....

Methods:
—--------
1. public static void addAll(Collection, Object … obj):
   It is able to add all the specified elements to the Specified Collection.
   EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
ArrayList arrayList = new ArrayList();
arrayList.add("AAA");
arrayList.add("BBB");
arrayList.add("CCC");
arrayList.add("DDD");
System.out.println(arrayList);

ArrayList arrayList1 = new ArrayList();
Collections.addAll(arrayList1, "AAA", "BBB", "CCC", "DDD");
System.out.println(arrayList1);

}
}
```

2. public static void copy(List dest , List source):

It is able to copy all elements from the source List to destination List when the destination List size is equal or greater than the source list size, if the destination list size is less than the source list size then JVM will raise an exception like java.lang.IndexOutofBoundsException.

Note: In the Destination List , existing elements will be overridden with the Source List elements.

3. public static void sort(List list):
   It is able to sort all the elements in the Specified List object.

4. Public static void reverse(List list):
   It is able to sort all the elements in the Specified List in descending order.

5. public static void shuffle(List list):
   It is able to arrange all the elements in the specified List in random order.
   EX:

```
import             *;

public class Main {
public static void main(String[] args) {
List<String> arrayList1 = new ArrayList<>();
arrayList1.add("AAA");
arrayList1.add("FFF");
arrayList1.add("BBB");
arrayList1.add("EEE");
arrayList1.add("CCC");
arrayList1.add("DDD");
System.out.println(arrayList1);
Collections.sort(arrayList1);
System.out.println(arrayList1);
Collections.reverse(arrayList1);
System.out.println(arrayList1);
Collections.shuffle(arrayList1);
System.out.println(arrayList1);

}
```

```
[AAA, FFF, BBB, EEE, CCC, DDD]
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
[FFF, EEE, DDD, CCC, BBB, AAA]
[BBB, AAA, DDD, FFF, EEE, CCC]
```

6. public static int binarySearch(List list, Object key)
It is able to search the specified element in the specified List , where all the elements must be in Sorting order. It is able to return the element's index value if it exists in the List, if it does not exist in the List then it will return a value like “-size-1”.

EX:

```java
import java.util.*;

public class Main {
public static void main(String[] args) {
List arrayList = new ArrayList();
arrayList.add("AAA");
arrayList.add("FFF");
arrayList.add("BBB");
arrayList.add("EEE");
arrayList.add("CCC");
arrayList.add("DDD");
System.out.println(arrayList);
Collections.sort(arrayList);
System.out.println(Collections.binarySearch(arrayList, "EEE"));
}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
4
```

EX:

```java
import java.util.*;

public class Main {
public static void main(String[] args) {
List arrayList = new ArrayList();
arrayList.add("AAA");
arrayList.add("BBB");
```

```
arrayList.add("CCC");
arrayList.add("DDD");
arrayList.add("EEE");
arrayList.add("FFF");
System.out.println(arrayList);
//Collections.sort(arrayList);
System.out.println(Collections.binarySearch(arrayList, "EEE"));
}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
4
```

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
List arrayList = new ArrayList();
arrayList.add("AAA");
arrayList.add("BBB");
arrayList.add("CCC");
arrayList.add("DDD");
arrayList.add("EEE");
arrayList.add("FFF");
System.out.println(arrayList);
//Collections.sort(arrayList);
System.out.println(Collections.binarySearch(arrayList, "ZZZ"));
}
}
```

```
[AAA, BBB, CCC, DDD, EEE, FFF]
-7
```

7. public static void fill(List list, Object element):

It is able to replace all the elements of the specified List with the specified element.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
List list = new ArrayList(3);
list.add("AAA");
list.add("BBB");
list.add("CCC");
System.out.println(list);
Collections.fill(list, "ZZZ");
System.out.println(list);

}
}
```

```
[AAA, BBB, CCC]
[ZZZ, ZZZ, ZZZ]
```

8. public static List unmodifiableList(List list):

It is able to fix the number of elements inside a List, if we perform modifications on the list after accessing this method then JVM will raise an exception like java.lang.UnsupportedOperationException.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
list.add("DDD");
System.out.println(list);
List list1 = Collections.unmodifiableList(list);
```

```
System.out.println(list1);
list1.add("EEE");
System.out.println(list1);

}
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
java.lang.UnsupportedOperationException
```

Note: Similarly we can use unmodifiableSet() , unmodifiableMap() methods for Set of elements and for Map of elements.

9. public static List synchronizedList(List list):

It is able to make a List as Synchronized List, it allows only one thread at a time and it is able to achieve Data Consistency and it is now Threadsafe.

EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
list.add("DDD");
System.out.println(list);
List list1 = Collections.synchronizedList(list);
System.out.println(list1);

}
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
```

Note: Similarly we are able to use synchronizedSet(), synchronizedMap(),... for set elements, for Map elements,.....

Arrays:

It is an utility class in Collection Framework, it is able to perform some utility operations over the elements of the Collections.

Methods:
1. public static List asList(Object … elements):
It is able to allow number of elements as input and it will return all these elements as a List.
EX:

```
import java.util.*;

public class Main {
public static void main(String[] args) {
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");
list.add("DDD");
System.out.println(list);

List list1 = new ArrayList();
Collections.addAll(list1, "AAA", "BBB", "CCC", "DDD");
System.out.println(list1);

List list2 = Arrays.asList("AAA", "BBB", "CCC", "DDD");
System.out.println(list2);
}
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
```

2. public static void sort(xxx[] elements):
It is able to sort all the elements which are available in the specified primitive data type array.

EX:

```
import java.util.Arrays;

public class Main {
public static void main(String[] args) {
int[] intArray = {10, 5, 20, 15, 30, 25};
for(int val: intArray){
System.out.print(val+" ");
}
System.out.println();
Arrays.sort(intArray);
for(int val: intArray){
System.out.print(val+" ");
}
}
}
```

```
10 5 20 15 30 25
5 10 15 20 25 30
```

3. public static void sort(Object[] obj):
It is able to sort all the elements which are available in the provided Object[].
EX:

```
import java.util.Arrays;

public class Main {
public static void main(String[] args) {
String[] strArray = {"FFF", "AAA", "EEE", "BBB", "DDD", "CCC"};
for(String str: strArray){
System.out.print(str+" ");
}
System.out.println();
Arrays.sort(strArray);
for(String str: strArray){
System.out.print(str+" ");
}
}
}
```

```
FFF AAA EEE BBB DDD CCC
AAA BBB CCC DDD EEE FFF
```

3. public static void sort(Object[] obj, Comparator c):
It is able to sort all the elements which are available in the provided Object[] as per the provided Comparator sorting logic.

EX:

```
import java.util.Arrays;
import java.util.Comparator;

class MyComparator implements Comparator{

@Override
public int compare(Object o1, Object o2) {
String str1 = (String) o1;
String str2 = (String) o2;

return str1.length() - str2.length();
}
}
public class Main {
public static void main(String[] args) {
String[] strArray = {"F", "AAA", "EE", "BBBB", "DDDDDD", "CCCCC"};
for(String str: strArray){
System.out.print(str+" ");
}
System.out.println();
Arrays.sort(strArray, new MyComparator());
for(String str: strArray){
System.out.print(str+" ");
}
}
}
```

```
F AAA EE BBBB DDDDDD CCCCC
F EE AAA BBBB CCCCC DDDDDD
```

4. public static int binarySearch(xxx[] values, xxx value):
It is able to search the specified element in the specified primitive data types array as per the binary search algorithm if the elements are sorted.
EX

```
public class Main {
public static void main(String[] args) {
int[] intArray = {10, 5, 20, 15, 30, 25};
Arrays.sort(intArray);
```

```
System.out.println(Arrays.binarySearch(intArray, 15));
System.out.println(Arrays.binarySearch(intArray, 100));
}
}
```

5. Public static int binarySearch(Object[] objs, Object obj):
It is able to search the specified element over the specified Object[] as per the Binary Search but all the elements must be provided in Sorting order.

EX:

```
public class Main {
public static void main(String[] args) {
String[] strArray = {"FFF", "AAA", "EEE", "BBB", "DDD", "CCC"};
Arrays.sort(strArray);
System.out.println(Arrays.binarySearch(strArray, "EEE"));
System.out.println(Arrays.binarySearch(strArray, "ZZZ"));
}
}
```

4
-7

6. public static void binarySearch(Object[] objs, Object obj, Comparator c):
It is able to perform binary search for the specified element over the specified Object[] after performing the User defined sorting over the Object[] as per the provided Comparator.
EX:

```
import java.util.Arrays;
import java.util.Comparator;

class MyComparator implements Comparator{

@Override
public int compare(Object o1, Object o2) {
String str1 = (String) o1;
String str2 = (String) o2;
return -str1.compareTo(str2);
}
}
public class Main {
public static void main(String[] args) {
String[] strArray = {"FFF", "AAA", "EEE", "BBB", "DDD", "CCC"};
```

```
Arrays.sort(strArray, new MyComparator());
System.out.println(Arrays.binarySearch(strArray, "EEE", new
MyComparator()));
System.out.println(Arrays.binarySearch(strArray, "ZZZ", new
MyComparator()));
for(String str: strArray){
System.out.print(str+" ");
}
}
}
```

```
1
-1
FFF EEE DDD CCC BBB AAA
```