Key interface of Collection
a. Collection
b. List => ArrayList, LinkedList, Stack,Vector
=> Duplicates are allowed, insertion order is preserved,null is also allowed.
=> Accessing the element is based on index
=> interfaces it implements are "Serializable,Cloneable,RandomAccess".
=> ArrayList,Vector best suitable when we perform read operation,where as linkedlist
best suited when we perform insert/delete operation.

c. Set => HashSet,LinkedHashSet
=> Duplicates are not allowed, insertion order is not preserved due to hashing.
=> interfaces it implements are "Serializable,Cloneable".
=> If we want Elements to be in inserted order then we need to go for "LinkedHashSet".

d. NavigableSet
e. SortedSet => TreeSet
=> Duplicates are not allowed, insertion is not preserved due to hashing
=> null is not allowed it would result in "NullPointerException".
=> The elements are arranged in sorted order.
=> if the elements are of homogenous type, then by default Comparable logic is used for
Sorting the elements
=> if the elements are of heteregenous type, then it would result in "ClassCastException".
=> To sort the hetergenous type of objects we need to use "Comparator" interface.

f. Map => HashMap,WeakHashMap,LinkedHashMap,IdentityHashMap
=> It represents the data in the form of "<K,V>" pair.
=> Keys can't be duplicate whereas the values can be duplicated.
=> WeakHashMap vs HashMap
a. if the key is null and if it is a part of HashMap then GC can't clean the object.
b. if the key is null and if it is a part of WeakHashMap then GC can clean the object.

=> IdentityHashMap vs HashMap
a. To identify duplicate key, JVM will use "==" operator in case of IdentityHashMap.
b. To identify duplicate key, JVM will use equals() in case of HashMap.

g. NavigableMap

h. SortedMap => TreeMap(K,V)
=> Duplicates keys are not allowed, insertion is not preserved due to hashing.
=> null is not allowed as a key, it would result in "NullPointerException".
=> The elements are arranged in sorted order.
=> if the value are of homogenous type, then by default Comparable logic is used for
Sorting the elements

```
             => if the values are of heteregenous type, then it would result in
"ClassCastException".
                 => To sort the hetergenous type of objects we need to use
"Comparator" interface.

 i. Queue        =>
PriorityQueue,BlockingQueue,PriorityBlockingQueue,LinkedBlockingQueue
             => Prior to processing, if we want to represent a group of individual
objects we go for Queue.
             => it follows the order of "FIFO".


Cursors of Collection
+++++++++++++++++++++
1. Enumeration  => Legacy cursor applicable only for Legacy classes.
                  Operations applicable are :: read

2. Iterator     => Universal Cursor applicable for any collection object.
                  Operations applicable are :: read[only forward],remove

3. ListIterator => Cursor applicable only for List objects.
                  Operations applicable are ::
read[forward,backward],remove,update,insert,delete


Sorting based interfaces
++++++++++++++++++++++++
a. Comparable :: Default Natural Sorting Order.
                  public int compareTo(Object obj)

b. Comparator :: Customized Sorting Order.
                  public int compare(Object obj1,Object obj2)


1.5 version enhancement of Queue
===============================
 1. It is a child interface of Collection
 2. If we want to represent a group of individual Objects before Processing then we
should go
    for Queue.
 3. From 1.5 version LinkedList also implements Queue
 4. Usually Queue follows FIFO Order,Based on our requirement we can implement our
own priority
    also.
 5. LinkedList based implementation Queue also follows FIFO order.

eg: Before sending mail, we need to store the mail id in any one of the
datstructure,the best
    suited datastructure is "Queue".

Important methods associated with Queue
=======================================
1. boolean offer (Object obj)
              => to add object into the Queues

2. Object peek()
              => It return the head element of the Queue
             If Queue is empty it returns null.
```

3. Object element()
   => It return the head element of the Queue
   If Queue is empty it throws NoSuchElementException.

4. Object poll()
   => It remove and return the head element of the queue
   If Queue is empty it returns null.

5. Object remove()
   => It remove and return the head element of the Queue
   If Queue is empty it returns NoSuchElementException.

PriorityQueue
=============

1. To process the elements before processing, we need to store the elements based on some priority order
2. Priority Order
   => natural sorting order
   => customized sorting order
3. insertion order => not preseved based on some sorting it will be added.
4. duplicate => not allowed.
5. null insertion => not allowed
6. heterogenous => if we depend on natural sorting order,no objects should homogenous and it should implements Comparable
   if it is customized sorting order,then Object can be heterogenous and it need not implements Comparable.

Constructor
==========

```
  1. PriorityQueue p=new PriorityQueue();
                  //Default Capacity=> 11
                  //Insertion order => based on default natural sorting order.

  2. PriorityQueue p=new PriorityQueue(int initialCapacity);
  3. PriorityQueue p=new PriorityQueue(int initialCapacity,Comparator comparator)
  4. PriorityQueue p=new PriorityQueue(SortedSet s);
  5. PriorityQueue p=new PriorityQueue(Collection c);
```

eg#1.

```
import java.util.PriorityQueue;
public class TestApp{
      public static void main(String... args){
            PriorityQueue p=new PriorityQueue();
            System.out.println(p.poll());//null
            System.out.println(p.element());//NoSuchElementException

            for (int i=0;i<=10 ;i++ ){
                  p.offer(i);
            }
            System.out.println(p);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

            System.out.println(p.poll());//0

            System.out.println(p);//[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      }
}
```

eg#2.

```
import java.util.PriorityQueue;
import java.util.Comparator;

public class TestApp{
      public static void main(String... args){
            PriorityQueue q=new PriorityQueue(15,new MyComparator());
            q.offer("Z");
            q.offer("A");
            q.offer("L");
            q.offer("B");
            System.out.println(q);//[Z,L,B,A]
      }
}
class MyComparator implements Comparator{

      @Override
      public int compare(Object obj1,Object obj2){

            String s1=obj1.toString();
            String s2=obj2.toString();

            return s2.compareTo(s1);

      }
}
```

Note: Some Operating System wont provide support for PriorityQueue.

1.6 V enchancement of Collection
================================

```
1. NavigableSet
      => It is the child interface of SortedSet
      => It defines several methods for Navigation purposes

  floor(e) => it returns the highest element which is <=e
  lower(e) => it returns the highest eleemnt which is <e

  ceiling(e) => it returns the lowest element which is >=e
  higher (e) => it returns the lowest element which is <e

   pollFirst() => remove and return first element
   pollLast()  => remove and return last element

   descendingSet() =>returns NavigableSet in descending order.
```

eg#1.

```
import java.util.*;


//Client Code
public class Test
{
      public static void main(String[] args)
      {
            TreeSet ts = new TreeSet();
            ts.add(1000);
            ts.add(2000);
            ts.add(3000);
            ts.add(4000);
            ts.add(5000);
            System.out.println(ts);//[1000, 2000, 3000, 4000, 5000]
            System.out.println("Ceiling value :: "+ts.ceiling(2000));//2000
            System.out.println("Higher  value :: "+ts.higher(2000));//3000
            System.out.println("Floor   value :: "+ts.floor(3000));//3000
            System.out.println("Lower   value :: "+ts.lower(3000));//2000
            System.out.println("Poll    First :: "+ts.pollFirst());//1000
            System.out.println(ts);//[2000,3000,4000,5000]

            System.out.println("Poll    Last  :: "+ts.pollLast());//5000
            System.out.println(ts);//[2000,3000,4000]

            System.out.println("DescendingSet :: "+ts.descendingSet());//

      }
}
```

2. NavigableMap
    => It defines several methods of Navigation purpose.
    => It is child interface of SortedMap.

NavigableMap defines the following methods

a. floorKey(e)
b. lowerKey(e)
c. ceilingKey(e)
d. higherKey(e)
e. pollFirstEntry()
f. pollLastEntry()
g. descendingMap()

eg#2.

```
import java.util.*;

//Client Code
public class Test
{
      public static void main(String[] args)
      {
            TreeMap tm = new TreeMap();
            tm.put(10,"sachin");
            tm.put(7,"dhoni");
            tm.put(18,"kohli");
            tm.put(19,"dravid");
            tm.put(45,"rohith");

            //{7=dhoni, 10=sachin, 18=kohli, 19=dravid, 45=rohith}
            System.out.println(tm);
            System.out.println("Ceiling Key   :: "+tm.ceilingKey(10));
            System.out.println("Higher  Key   :: "+tm.higherKey(10));
            System.out.println("Floor   Key   :: "+tm.floorKey(10));
            System.out.println("Lower   Key   :: "+tm.lowerKey(10));
            System.out.println("PollFirst     :: "+tm.pollFirstEntry());
            System.out.println("PollLast      :: "+tm.pollLastEntry());
            System.out.println("DescendingMap :: "+tm.descendingMap());
            System.out.println(tm);
      }
}
```

Output

```
{7=dhoni, 10=sachin, 18=kohli, 19=dravid, 45=rohith}
Ceiling Key   :: 10
Higher  Key   :: 18
Floor   Key   :: 10
Lower   Key   :: 7
PollFirst     :: 7=dhoni
PollLast      :: 45=rohith
DescendingMap :: {19=dravid, 18=kohli, 10=sachin}
{10=sachin, 18=kohli, 19=dravid}
```

## Collection vs Collections

### Collections(c)

- => It is a utility class present in java.util package.
- => It defines the method meant for sorting,searching and reversing the elements

Note:

- Collection(I)
  - |=> List
    - a. It wont speak about sorting,so use Collections(c).
  - |=> Set
    - a. It we want sorting then we can opt for TreeSet.
  - |=> Queue
    - a. If we want sorting then we can opt for PriorityQueue.

### To sort the elemets of List

1. public static void sort(List l)
   1.It sorts the element in ascending order/alphabetical order
   2.The elements should be homogenous and it should comparable otherwise it leads to ClassCastException.
   3.If it contains null,it would result in "NullPointerException".

2. public static void sort(List l,Comparator c)
   1. It sorts the elements based on our customization.

eg#1.

```
import java.util.*;

//Client Code
public class Test
{
      public static void main(String[] args)
      {
            ArrayList al = new ArrayList();
            al.add("Z");
            al.add("A");
            al.add("L");
            al.add("B");
            al.add("D");
            //al.add(new Integer(10)); :: ClassCastException
            //al.add(null); :: NullPointerException
            System.out.println("Before Sorting :: "+al);//[Z, A, L, B, D]

            //Collections
            Collections.sort(al);
            System.out.println("After Sorting  :: "+al);//[A, B, D, L, Z]

      }
}
```

eg#2.

```
import java.util.*;

//Client Code
public class Test
{
      public static void main(String[] args)
      {
            ArrayList al = new ArrayList();
            al.add("Z");
            al.add("A");
            al.add("L");
            al.add("B");
            al.add("D");

            System.out.println("Before Sorting :: "+al);//[Z, A, L, B, D]

            //Collections
            Collections.sort(al,new MyComparator());
            System.out.println("After Sorting  :: "+al);//[Z,L,D,B,A]

      }
}
class MyComparator implements Comparator
{
      @Override
      public int compare(Object obj1,Object obj2)
      {
            //logic for sorting
            String s1 = obj1.toString();
            String s2 = obj2.toString();
            return -s1.compareTo(s2);
      }
}
```

binarySearch()
++++++++++++++

Searching Elements of List:

1) public static int binarySearch(List l, Object target);

If we are Sorting List According to Natural Sorting Order then we have to Use this Method.

2) public static int binarySearch(List l, Object target, Comparator c);

If we are Sorting List according to Comparator then we have to Use this Method.

Conclusions:

=> Internally the Above Search Methods will Use Binary Search Algorithm.<br>
=> Before performing Search Operation Compulsory List should be Sorted. Otherwise we will get Unpredictable Results.<br>
=> Successful Search Returns Index.<br>
=> Unsuccessful Search Returns Insertion Point.<br>
=> Insertion Point is the Location where we can Insert the Target Element in the SortedList.<br>
=> If the List is Sorted according to Comparator then at the Time of Search Operation Also we should Pass the Same Comparator Object. Otherwise we will get Unpredictable Results.

```
import java.util.*;
//Client Code
public class Test
{
      public static void main(String[] args)
      {
            ArrayList al = new ArrayList();
            al.add("Z");
            al.add("A");
            al.add("M");
            al.add("K");
            al.add("a");


            //Collections
            Collections.sort(al);
            System.out.println("After Sorting  :: "+al);//[A, K, M, Z, a]


            //BinarySearch :: success case -> index
            //BinarySearch :: failure case -> insertion point
            System.out.println("Index of Z is :: "+Collections.binarySearch(al,"Z"));
            System.out.println("Index of J is :: "+Collections.binarySearch(al,"J"));
            System.out.println("Index of X is :: "+Collections.binarySearch(al,"X"));


      }
}
```

eg#2.

```
import java.util.*;


//Client Code
```

```
public class Test
{
      public static void main(String[] args)
      {
            ArrayList al = new ArrayList();
            al.add(15);
            al.add(0);
            al.add(20);
            al.add(10);
            al.add(5);


            //Collections
            Collections.sort(al,new MyComparator());
            System.out.println("After Sorting  :: "+al);//[20,15,10,5,0]


            //BinarySearch :: success case -> index
            //BinarySearch :: failure case -> insertion point
            System.out.println(Collections.binarySearch(al,10,new 
MyComparator()));//index
            System.out.println(Collections.binarySearch(al,13,new 
MyComparator()));//insertionpoint
            System.out.println(Collections.binarySearch(al,17,new 
MyComparator()));//insertionpoint



      }
}
class MyComparator implements Comparator
{
      @Override
      public int compare(Object obj1,Object obj2)
      {
            //logic for sorting
            Integer i1= (Integer)obj1;
            Integer i2= (Integer)obj2;
            return -i1.compareTo(i2);
      }
}
```

```
Output
After Sorting  :: [20, 15, 10, 5, 0]
2
-3
-2
```

Eg: For the List of 3 Elements
      A B Z
1) Range of Successful Search: 0 To 2
2) Range of Unsuccessful Search: -4 To -1
3) Total Result Range: -4 To 2

Note: For the List of n Elements
1) Successful Result Range: 0 To n-1
2) Unsuccessful Result Range: -(n+1) To -1

3) Total Result Range: -(n+1) To n-1