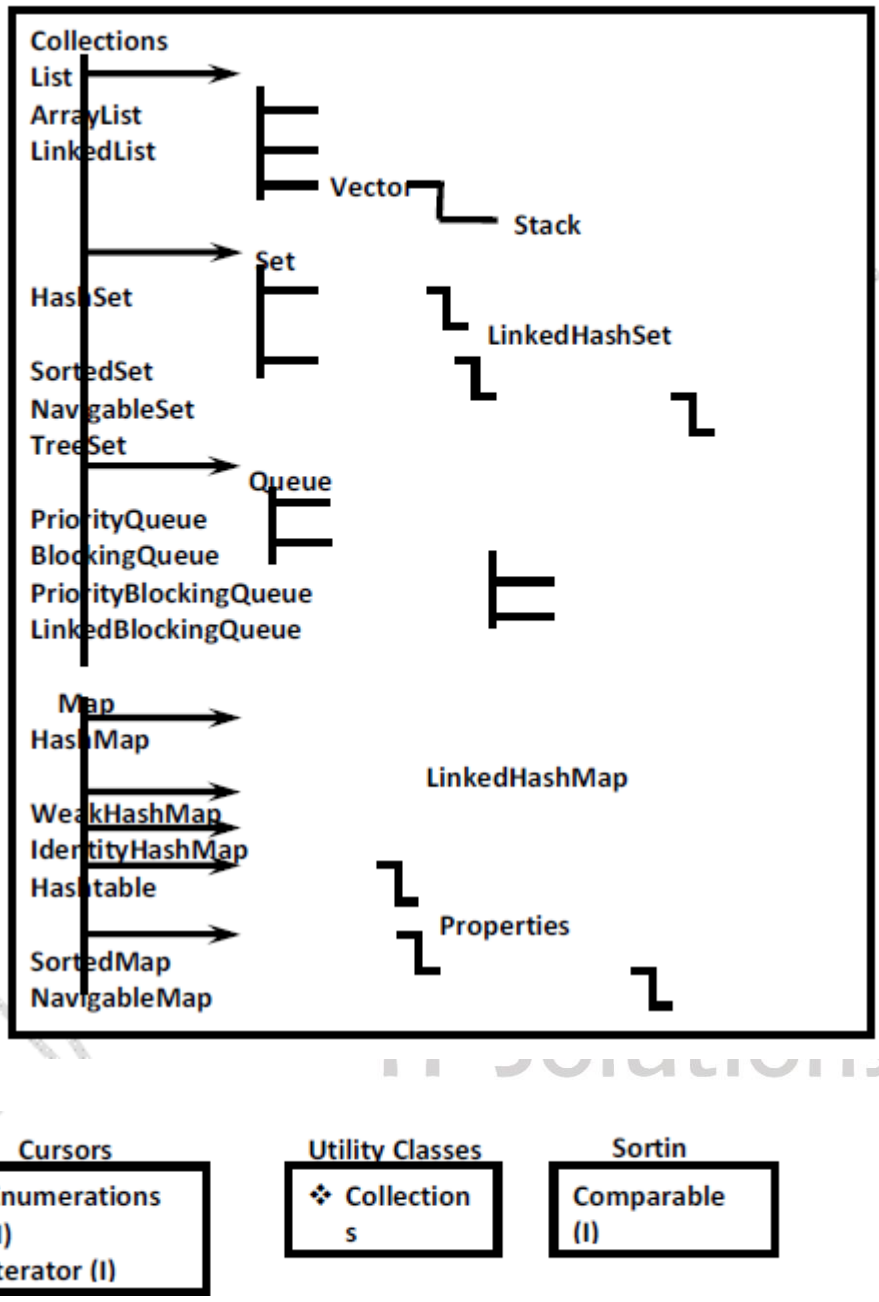
Collections Frame Work



An Array is an Indexed Collection of Fixed Number of Homogeneous Data Elements. The Main Advantage of Arrays is we can Represent Multiple Values by using Single Variable so

that Readability of the Code will be Improved.

Limitations Of Object Type Arrays:

1) Arrays are Fixed in Size that is Once we created an Array there is No Chance of Increasing

OR Decreasing Size based on Our Requirement. Hence to Use Arrays Concept Compulsory

we should Know the Size in Advance which May Not be Possible Always.

2) Arrays can Hold Only Homogeneous Data Type Elements. Eg:

```
Student[] s = new Student[10000];
s[0] = new Student();√

s[1]=new Custpmer();×
CE: incompatible types
found: Costomer
required: Student
```

We can Resolve this Problem by using Object Type Arrays.

Eg:

```
Object[] a = new Object[10000];
a[0] = new Student(); √
a[1] = new Customer(); √
```

3) Arrays Concept is Not implemented based on Some Standard Data Structure Hence Readymade Methods Support is Not Available. Hence for Every Requirement we have to

write the Code Explicitly which Increases Complexity of the Programming.

To Overcome Above Problems of Arrays we should go for Collections.

Advantages Of Collections:

1) Collections are Growable in Nature. That is based on Our Requirement we canIncrease OR

Decrease the Size.

2) Collections can Hold Both Homogeneous and Heterogeneous Elements.

3) Every Collection Class is implemented based on Some Standard Data Structure.Hence for

Every Requirement Readymade Method Support is Available. Being a Programmer we

have to Use those Methods and we are Not Responsible to Provide Implementation

Differences Between Arrays And Collections:

Arrays	Collections
Arrays are Fixed in Size.	Collections are Growable in Nature.
With Respect to Memory Arrays are Not Recommended to Use.	With Respect to Memory Collections are Recommended to Use.
With Respect to Performance Arrays are Recommended to Use.	With Respect to Performance Collections are Not Recommended to Use.
Arrays can Hold Only Homogeneous Data Elements.	Collections can Hold Both *Homogeneous* and *Heterogeneous* Elements.
Arrays can Hold Both Primitives and Objects.	Collections can Hold Only Objects but Not Primitives.
Arrays Concept is Not implemented based on Some Standard Data Structure. Hence Readymade Method Support is Not Available.	For every Collection class underlying Data Structure is Available Hence Readymade Method Support is Available for Every Requirement.

Collection:

If we want to Represent a Group of Individual Objects as a Single Entity then we should go for

Collection.

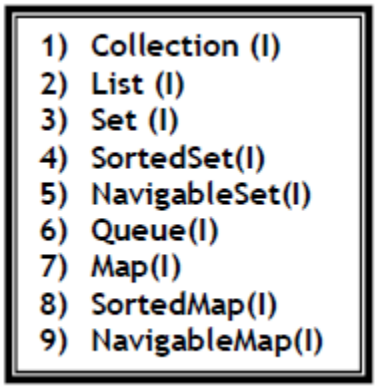
Collection Frame Work

It defines Several Classes and Interfaces which can be used to Represent a Group of Objects as

a Single Entity.

JAVA C++	
Collection Container Collection Frame Work	Standard Template Library (STL)

9 Key Interfaces Of Collection Framework:



1) Collection (I):

 If we want to Represent a Group of Individual Objects as a Single Entity then we should
go for Collections.

 Collection Interface is considered as Root Interface of Collection Framework.

 Collection Interface defines the Most Common Methodswhich are Applicable for any
Collection Object.

Difference Between Collection (I) and Collections (C):

 Collection is an Interface which can be used to Represent a Group of Individual Objects as a
Single Entity.

 Whereas Collections is an Utility Class Present in *java.util* Package to Define Several Utility

Methods for Collection Objects.

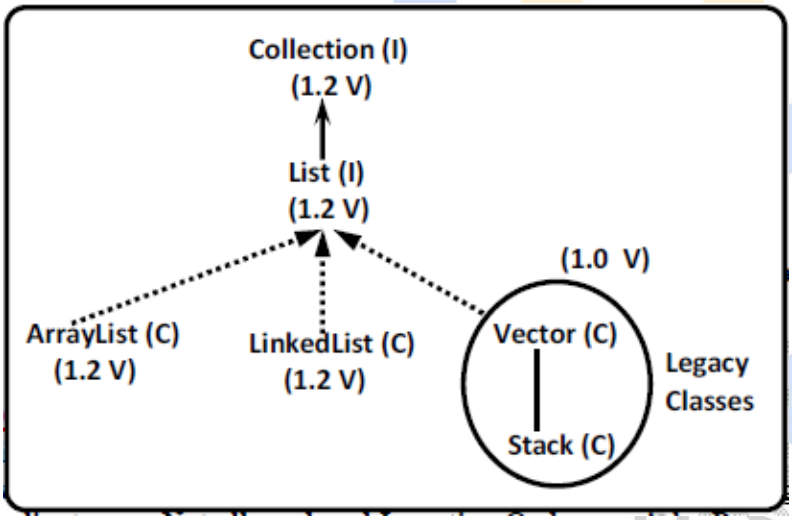
Note: There is No Concrete Class which implements Collection Interface Directly.

2) List (I):

 It is the Child Interface of Collection.

 If we want to Represent a Group of Individual Objects as a Single Entity where Duplicates are allowed and Insertion Order Preserved. Then we should go for List.

Note: In 1.2 Version onwards Vector and Stack Classes are re-engineered to Implement List Interface.

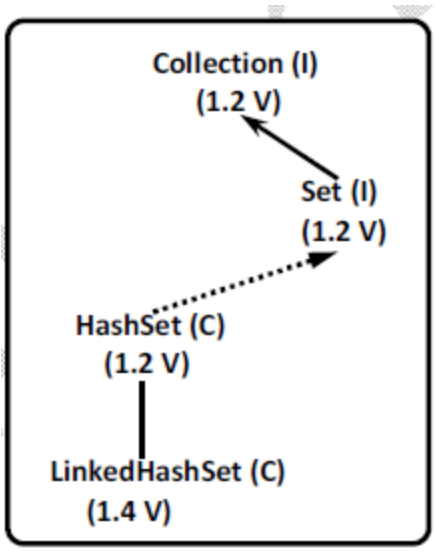


3) Set (I):

It is the Child Interface of the Collection.

 If we want to Represent a Group of Individual Objects as a Single Entity where Duplicates are Not allowed and Insertion Order won't be Preserved.

Then we should go for Set Interface.



4) SortedSet (I):

⍰ It is the Child Interface of Set.

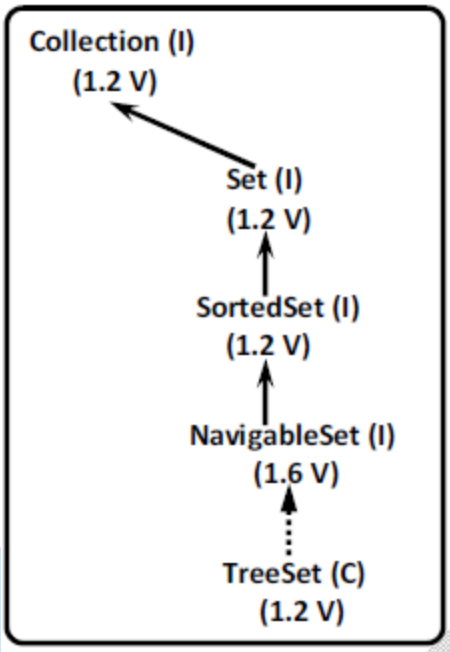
⍰ If we want to Represent a Group of Individual Objects Without Duplicates According to

Some Sorting Order then we should go for SortedSet.

5) NavigableSet (I):

⍰ It is the Child Interface of SortedSet.

⍰ It defines Several Methods for Navigation Purposes.

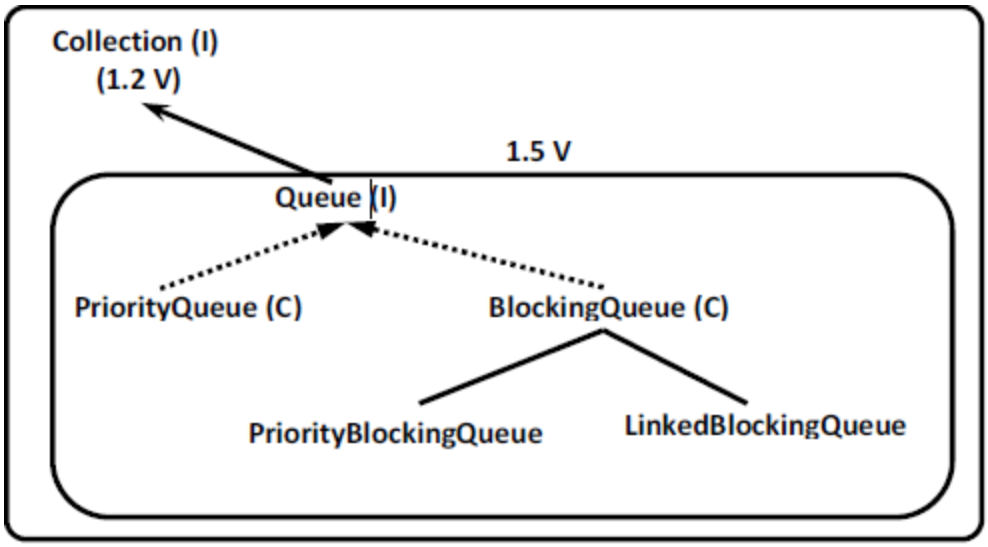


6) Queue (I):

It is the Child Interface of Collection.

If we want to Represent a Group of Individual Objects Prior to Processing then we should go for Queue.

Eg: Before sending a Mail we have to Store All MailID's in Some Data Structure and in which
Order we added MailID's in the Same Order Only Mails should be delivered (FIFO). For this
Requirement Queue is Best Suitable.



Note:

All the Above Interfaces (Collection, List , Set, SortedSet, NavigableSet,and Queue) Meant

for representing a Group of Individual Objects.

If we want to Represent a Group of Key - Value Pairs then we should go for Map.

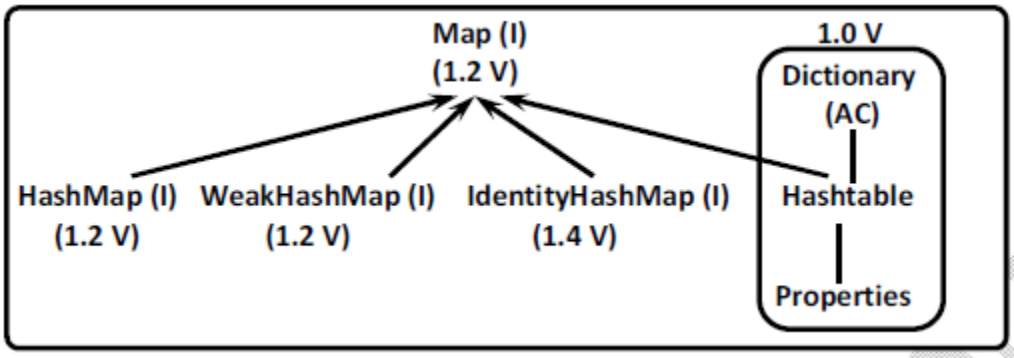
7) Map (I):

Map is Not Child Interface of Collection.

If we want to Represent a Group of Objects as Key- Value Pairs then we should go for

Map Interface.

Duplicate Keys are Not allowed but Values can be Duplicated.



8) SortedMap (I):

It is the Child Interface of Map.

If we want to Represent a Group of Objects as Key- Value Pairs according to Some Sorting Order of Keys then we should go for SortedMap.

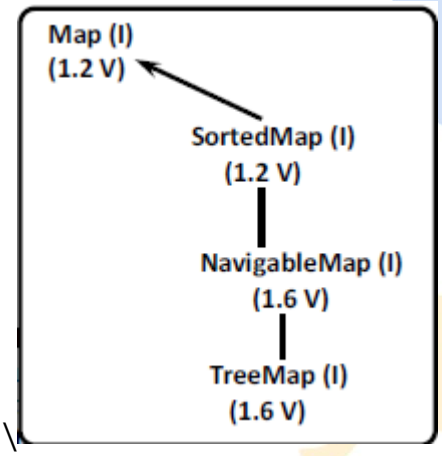
Sorting should be Based on Key but Not Based on Value.

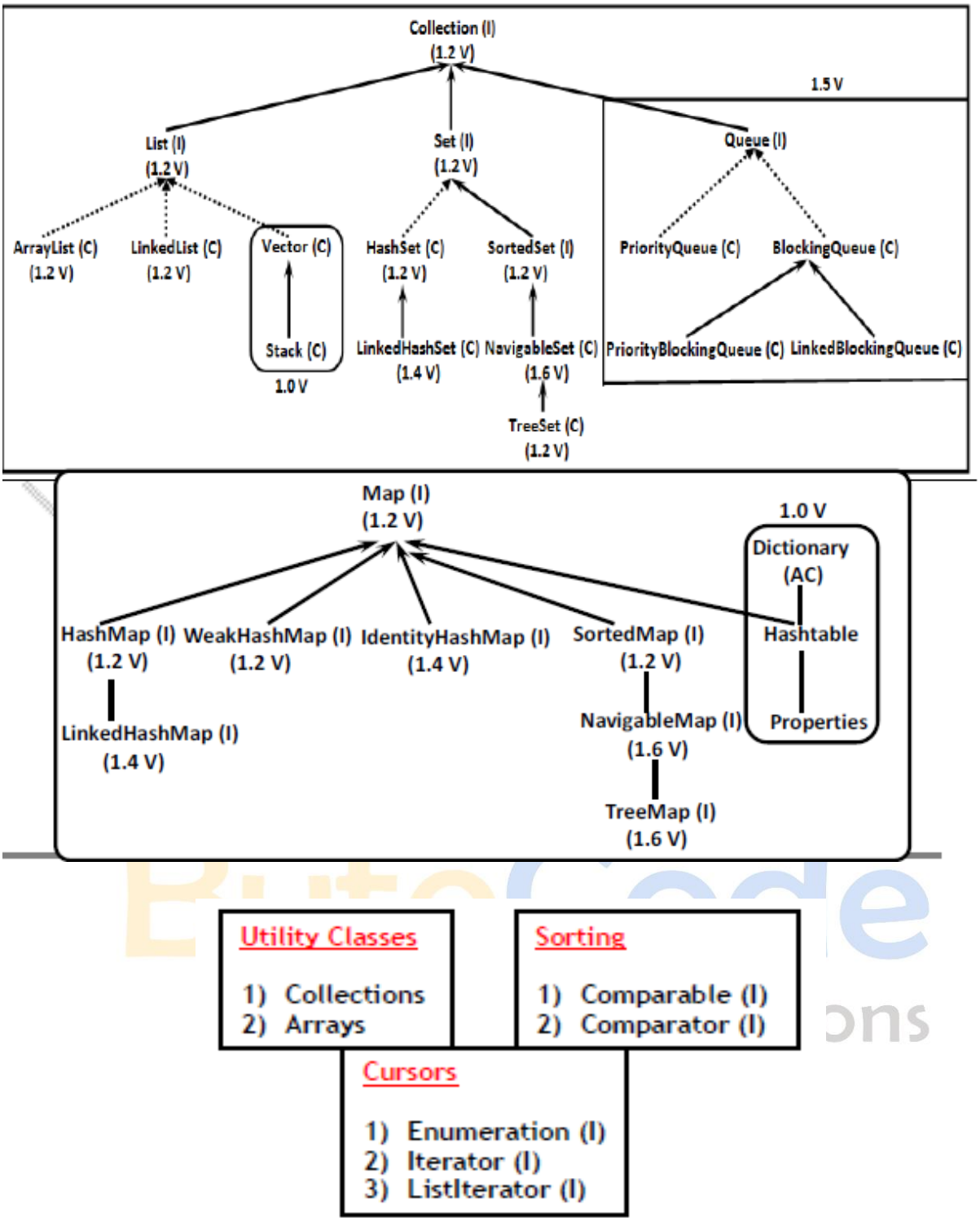
9) NavigableMap (I):

It is the Child Interface of SortedMap.

It Defines Several Methods for Navigation Purposes.

Note:In Collection Framework the following are Legacy Classes. 1) Enumeration (I) 2) Dictionary (Abstract Class) 3) Vector (Concrete Class) 4) Stack (Concrete Class) 5) Hashtable (Concrete Class) 6) Properties (Concrete Class)





1) Collection Interface:

If we want to Represent a Group of Individual Objects as a Single Entity then we should go for Collection Interface.

Methods:

 Collection Interface defines the Most Common Methods which are Applicable for any

Collection Objects.

 The following is the List of the Methods Present Inside Collection Interface.

1) boolean add(Object o)

2) booleanaddAll(Collection c)

3) boolean remove(Object o)

4) booleanremoveAll(Collection c)

5) booleanretainAll(Collection c): To Remove All Objects Except those Present in c.

6) void clear()

7) boolean contains(Object o)

8) booleancontainsAll(Collection c)

9) booleanisEmpty()

10) int size()

11) Object[] toArray()

12) Iterator iterator()

Note:

 There is No Concrete Class which implements Collection Interface Directly.

 There is No Direct Method in Collection Interface to get Objects.

2) List:

 It is the Child Interface of Collection.

 If we want to Represent a Group of Individual Objects where Duplicates are allowed

and Insertion Order Preserved. Then we should go for List.

 We can Preserve Insertion Order and we can Differentiate Duplicate Object by using

Index. Hence Index will Play Very Important Role in List.

Methods:List Interface Defines the following Specific Methods.

1) void add(int index, Object o)

2) booleanaddAll(int index, Collection c)

3) Object get(int index)

4) Object remove(int index)

5) Object set(int index, Object new):To Replace the Element Present at specified Index

with provided Object and Returns Old Object.

6) intindexOf(Object o):Returns Index of 1st Occurrence of 'o'

7) intlastIndexOf(Object o)

8) ListIteratorlistIterator();

2.1) ArrayList:

- **The Underlying Data Structure for ArrayList is Resizable Array ORGrowable Array.**
- **Duplicate Objects are allowed.**
- **Insertion Order is Preserved.**
- **Heterogeneous Objects are allowed (Except *TreeSet* and *TreeMap* Everywhere Heterogeneous Objects are allowed).**
- **null Insertion is Possible.**

Constructors

1) ArrayList l = new ArrayList();

- **Creates an Empty ArrayList Object with Default Initial Capacity 10.**
- **If ArrayList Reaches its Max Capacity then a New ArrayList Object will be Created With**

```
New Capacity = (Current Capacity * 3/2)+1
```

2) ArrayList l = new ArrayList(intinitialCapacity);

Creates an Empty ArrayList Object with specified Initial Capacity.

3 c):) ArrayList l = new ArrayList(Collection

- Creates an EqualentArrayList Object for the given Collection Object.
- This Constructor Meant for Inter Conversion between Collection Objects.

```
importjava.util.ArrayList;
classArrayListDemo {
        public static void main(String[] args){

                ArrayList l = new ArrayList();

                l.add("A");
                l.add(10);
                l.add("A");
                l.add(null);
                System.out.println(l); //[A, 10, A, null]

                l.remove(2);
                System.out.println(l); //[A, 10, null]

                l.add(2,"M");
                l.add("N");
                System.out.println(l); //[A, 10, M, null, N]

        }
}
```

- Usually we can Use Collections to Hold and Transfer Data (Objects) form One Location to
Another Location.

- To Provide Support for this Requirement Every Collection Class Implements *Serializable*
and *Cloneable* Interfaces.
- *ArrayList* and *Vector* Classes Implements *RandomAccess* Interface. So that we can Access
any Random Element with the Same Speed.
- *RandomAccess* Interface Present in *java.util*Package and it doesn't contain any Methods.
Hence it is a *Marker* Interface.
- Hence ArrayList is Best Suitable if Our Frequent Operation is Retrieval Operation.

```
ArrayList l1 = new ArrayList();
LinkedList l2 = new LinkedList();

System.out.println(l1 instanceofSerializable); //true
System.out.println(l2 instanceofCloneable); //true
System.out.println(l1 instanceofRandomAccess); //true
System.out.println(l2 instanceofRandomAccess); //false
```

Differences between *ArrayList* and *Vector*:

ArrayList	Vector
Every Method Present Inside ArrayListis Non – Synchronized.	Every Method Present in Vector is Synchronized.
At a Time Multiple Threads are allow to Operate on ArrayList Simultaneously and Hence ArrayList Object is Not Thread Safe.	At a Time Only One Thread is allow to Operate on Vector Object and Hence Vector Object is Always Thread Safe.
Relatively Performance is High because Threads are Not required to Wait.	Relatively Performance is Low because Threads are required to Wait.
Introduced in 1.2 Version and it is Non – Legacy.	Introduced in 1.0 Version and it is Legacy.

How to get Synchronized Version of ArrayList Object?

By Default ArrayList Object is Non- Synchronized but we can get Synchronized Version

ArrayList Object by using the following Method of Collections Class

```
public static List synchronizedList(List l)
```

Eg:

```
ArrayListal = new ArrayList ();
List l = Collections.synchronizedList(al);
```

Synchronized Version

Non - Synchronized Version

Similarly we can get Synchronized Version of *Set* and *Map* Objects by using the following

Methods of Collection Class.

```
public static Set synchronizedSet(Set s)

public static Map synchronizedMap(Map m)
```

⍰ ArrayList is the Best Choice if we want to Perform Retrieval Operation Frequently.

⍰ But ArrayList is Worst Choice if Our Frequent Operation is Insertion OR Deletion in the

Middle. Because it required Several Shift Operations Internally.

2.2) LinkedList:

⍰ The Underlying Data Structure is Double LinkedList.

⍰ Insertion Order is Preserved.

⍰ Duplicate Objects are allowed.

⍰ Heterogeneous Objects are allowed.

⍰ null Insertion is Possible.

⍰ Implements *Serializable* and *Cloneable* Interfaces but Not *RandomAccess*Interface.

⍰ Best Choice if Our Frequent Operation is *Insertion*OR *Deletion* in the Middle.

⍰ Worst Choice if Our Frequent Operation is Retrieval.

Constructors:

1) LinkedList l = new LinkedList(); Creates an Empty LinkedList Object.

2) LinkedList l = new LinkedList(Collection c);

Creates an Equivalent LinkedList Object for the given Collection.

Methods:

Usually we can Use LinkedList to Implement *Stacks* and *Queues*. To Provide Support for this

Requirement LinkedList Class Defines the following 6 Specific Methods.

1) void addFirst(Object o)

2) void addLast(Object o)

3) Object getFirst()

4) Object getLast()

5) Object removeFirst()

6) Object removeLast()

```
importjava.util.LinkedList;
classLinkedListDemo {
        public static void main(String[] args) {
                LinkedList l = new LinkedList();
                l.add("Durga");
                l.add(30);
                l.add(null);
                l.add("Durga");
                l.set(0, "Software");
                l.add(0,"Venky");
                l.removeLast();
                l.addFirst("CCC");
                System.out.println(l); //[CCC, Venky, Software, 30, null]
        }
}
```

2.3) Vector:

The Underlying Data Structure is Resizable Array ORGrowable Array.

Insertion Order is Preserved.

Duplicate Objects are allowed.

Heterogeneous Objects are allowed.

null Insertion is Possible.

Implements *Serializable, Cloneable* and *RandomAccess* interfaces.

Every Method Present Inside Vector is Synchronized and Hence Vector Object is Thread Safe.

Vector is the Best Choice if Our Frequent Operation is Retrieval.

Worst Choice if Our Frequent Operation is *Insertion* OR *Deletion* in the Middle.

Constructors:

1) Vector v = new Vector();

2) Creates an Empty Vector Object with Default Initial Capacity 10.

3) Once Vector Reaches its Max Capacity then a New Vector Object will be Created with

New Capacity = Current Capacity * 2

2) Vector v = new Vector(intinitialCapacity);

3) Vector v = new Vector(intinitialCapacity, intincrementalCapacity);

4) Vector v = new Vector(Collection c);

Methods:

1) To Add Elements:

- **add(Object o)Collection**
- **add(int index, Object o)List**
- **addElement(Object o) Vector**

2) To Remove Elements:

- **remove(Object o) Collection**
- **removeElement(Object o)Vector**
- **remove(int index) List**
- **removeElementAt(int index)Vector**
- **clear() Collection**
- **removeAllElements()Vector**

3) To Retrive Elements:

- **Object get(int index)List**
- **Object elementAt(int index)Vector**
- **Object firstElement() Vector**
- **Object lastElement()Vector**

4) Some Other Methods:

- **int size()**
- **int capacity()**
- **Enumeration element()**

```
importjava.util.Vector;
classVectorDemo {
        public static void main(String[] args) {
                Vector v = new Vector();
                System.out.println(v.capacity()); //10
                for(int i = 1; i<=10; i++) {
                        v.addElement(i);
                }
                System.out.println(v.capacity()); //10
                v.addElement("A");
                System.out.println(v.capacity()); //20
                System.out.println(v); //[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A]
        }
}
```

2.3.1) Stack:

- **It is the Child Class of Vector.**
- **It is a Specially Designed Class for Last In First Out (LIFO) Order.**

Constructor:Stack s = new Stack();

Methods:

1) Object push(Object o): To Insert an Object into the Stack.

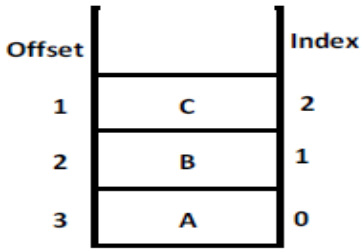
2) Object pop(): To Remove and Return Top of the Stack.

3) Object peek(): Ro Return Top of the Stack without Removal.

4) boolean empty(): Returns true if Stack is Empty

5) int search(Object o):Returns Offset if the Element is Available Otherwise Returns -1.

```
importjava.util.Stack;
classStackDemo {
        public static void main(String[] args) {
                Stack s = new Stack();
                s.push("A");
                s.push("B");
                s.push("C");
                System.out.println(s); //[A, B, C]
                System.out.println(s.search("A")); //3
                System.out.println(s.search("Z")); //-1
        }
}
```



The 3 Cursors of Java:

□ If we want to get Objects One by One from the Collection then we should go for Cursors.

□ There are 3 Types of Cursors Available in Java.

1) Enumeration

2) Iterator

3) ListIterator

1) Enumeration:

□ We can Use Enumeration to get Objects One by One from the Collection.

□ We can Create Enumeration Object by using elements().

public Enumeration elements();

Eg:Enumeration e = v.elements(); //v is Vector Object.

Methods:

1) public booleanhasMoreElements();

2) public Object nextElement();

```
importjava.util.*;
classEnumerationDemo {
        public static void main(String[] args) {
                Vector v = new Vector();
                for(int i=0; i<=10; i++) {
                        v.addElement(i);
                }
                System.out.println(v);
                Enumeration e = v.elements();
                while(e.hasMoreElements()) {
                        Integer I = (Integer)e.nextElement();
                        if(I%2 == 0)
                                System.out.println(I);
                }
                System.out.println(v);
        }
}
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
0
2
4
6
8
10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Limitations of Enumeration:

Enumeration Concept is Applicable Only for Legacy Classes and it is Not a Universal Cursor.

By using Enumeration we can Perform *Read* Operation and we can't Perform *Remove* Operation.

To Overcome Above Limitations we should go for Iterator

2) Iterator:

We can Use Iterator to get Objects One by One from Collection.

We can Apply Iterator Concept for any Collection Object. Hence it is Universal Cursor.

By using Iterator we can Able to Perform Both *Read* and *Remove* Operations.

We can Create Iterator Object by using iterator() of Collection Interface.

public Iterator iterator();

Eg:Iterator itr = c.iterator(); //c Means any Collection Object.

Methods:

1) public booleanhasNext()

2) public Object next()

3) public void remove()

```
importjava.util.*;
classIteratorDemo {
        public static void main(String[] args) {
                ArrayList l = new ArrayList();
                for (int i=0; i<=10; i++) {
                        l.add(i);
                }
                System.out.println(l);
                Iterator itr = l.iterator();
                while(itr.hasNext()) {
                        Integer I = (Integer)itr.next();
                        if(I%2 == 0)
                                System.out.println(I);
                        else
                                itr.remove();
                }
                System.out.println(l);
        }
}
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
0
2
4
6
8
10
[0, 2, 4, 6, 8, 10]
```

3) ListIterator:

▯ ListIterator is the Child Interface of Iterator.

▯ By using ListIterator we can Move Either to the Forward Direction OR to the Backward Direction. That is it is a Bi-Directional Cursor.

▯ By using ListIterator we can Able to Perform Addition of New Objects andReplacing existing Objects. In Addition to Read and Remove Operations.

▯ We can Create ListIterator Object by using listIterator().

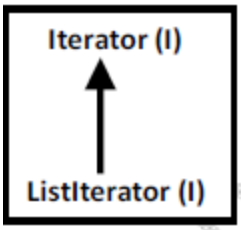
publicListIteratorlistIterator();

Eg:ListIteratorltr = l.listIterator(); //l is Any List Object

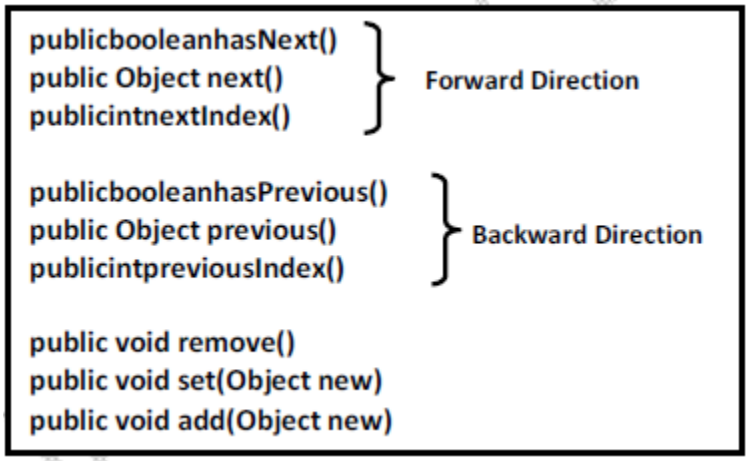
Methods:

▯ ListIteratoris the Child Interface of Iterator and Hence All Iterator Methods by Default

Available to the ListIterator.



ListIteratorDefines the following 9 Methods.



```
importjava.util.*;
classListIteratorDemo {
        public static void main(String[] args) {
                LinkedList l = new LinkedList();
                l.add("Baala");
                l.add("Venki");
                l.add("Chiru");
                l.add("Naag");
                System.out.println(l);
                ListIteratorltr = l.listIterator();
                while(ltr.hasNext()) {
                        String s = (String)ltr.next();
                        if(s.equals("Venki"))
                                ltr.remove();
                        if(s.equals("Naag"))
                                ltr.add("Chaitu");
                        if(s.equals("Chiru"))
                                ltr.add("Charan");
                }
                System.out.println(l);
        }
}
```

```
[Baala, Venki, Chiru, Naag]
[Baala, Chiru, Charan, Naag, Chaitu]
```

Note: The Most Powerful Cursor is ListIterator. But its Limitation is, it is Applicable Only for

List Objects.

Comparison Table of 3 Cursors

Property	Enumeration	Iterator	ListIterator
Applicable For	Only Legacy Classes	Any Collection Objects	Only List Objects
Movement	Single Direction (Only Forward)	Single Direction (Only Forward)	Bi-Direction
How To Get	By using elements()	By using iterator()	By using listIterator() of List (I)
Accessability	Only Read	Read and Remove	Read , Remove, Replace And Addition of New Objects
Methods	hasMoreElements() nextElement()	hasNext() next() remove()	9 Methods
Is it legacy?	Yes (1.0 Version)	No (1.2 Version)	No (1.2 Version)

Internal Implementation of Cursors:

```
importjava.util.*;
classCursorDemo {
        public static void main(String args[]) {
                Vector v = new Vector();
                Enumeration e = v.elements();
                Iterator itr = v.iterator();
                ListIteratorlitr = v.listIterator();
                System.out.println(e.getClass().getName());
                System.out.println(itr.getClass().getName());
                System.out.println(litr.getClass().getName());
        }
}
```

```
java.util.Vector$1
java.util.Vector$Itr
java.util.Vector$ListItr
```

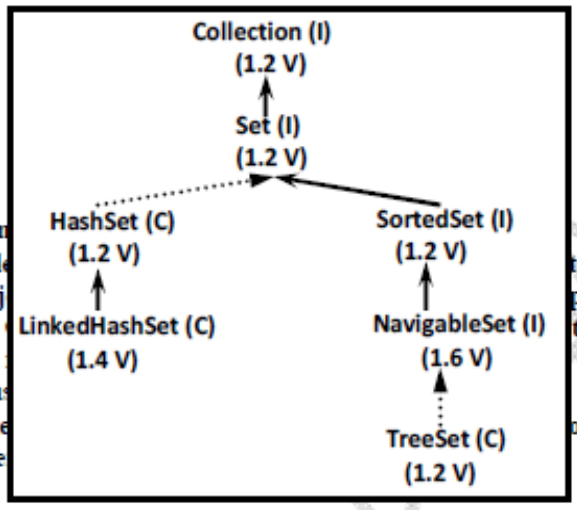
3) Set:

It is the Child Interface of Collection.

If we want to Represent a Group of Individual Objects as a Single Entity where Duplicates are Not allowed and Insertion Order is Not Preserved then we should go for

Set.

Set Interface doesn't contain any New Methods and Hence we have to Use Only Collection Interface Methods



3.1) HashSet:

- The Underlyin
- Insertion Orde
- Duplicate Obj
  won't get any
- null Insertion
- Heterogeneous
- HashSet imple
- If Our Freque

the Objects.
plicate Objects then we
turns false.

ot *RandomAccess*.
the Best Choice.

Constructors:

1) HashSet h = new HashSet();

Creates an Empty HashSet Object with Default Initial Capacity 16 and Default Fill Ratio :

0.75.

2) HashSet h = new HashSet(intinitialCapacity);

Creates an Empty HashSet Object with specified Initial Capacity and Default Fill Ratio :

0.75.

3) HashSet h = new HashSet(intinitialCapacity, float fillRatio);

4) HashSet h = new HashSet(Collection c);

Load Factor:

Fill Ratio 0.75 Means After Filling 75% Automatically a New HashSet Object will be Created.

This Factor is Called *Fill Ratio*OR*Load Factor.*

```
importjava.util.*;
classHashSetDemo {
        public static void main(String[] args) {
                HashSet h = new HashSet();
                h.add("B");
                h.add("C");
                h.add("D");
                h.add("Z");
                h.add(null);
                h.add(10);
                System.out.println(h.add("Z")); //false
                System.out.println(h); //[null, D, B, C, 10, Z]
        }
}
```

3.1.1)

rences.

	Set
The Underlying Data Structure is Hashtable.	The Underlying Data Structure is a Combination of *LinkedList* and *Hashtable*.
Insertion Order is Not Preserved.	Insertion Order will be Preserved.
Introduced in 1.2 Version.	Introduced in 1.4 Version.

In the Above Example if we Replace *HashSet* with *LinkedHashSet* then Output is false

[B, C, D, Z, null, 10]

That is Insertion Order is Preserved.

Note: In General we can Use*LinkedHashSet*and*LinkedHashMap*to Develop Cache Based

Applications where Duplicates are Not Allowed and Insertion Order Must be Preserved.

3.2) SortedSet:

- **It is the Child Interface of Set.**
- **If we want to Represent a Group of Individual Objects without Duplicates and all Objects will be Inserted According to Some Sorting Order, then we should go for SortedSet.**
- **The Sorting can be Either Default Natural Sorting OR Customized Sorting Order.**
- **For String Objects Default Natural Sorting is Alphabetical Order.**
- **For Numbers Default Natural Sorting is Ascending Order.**

Methods:

1) Object first(); Returns 1st Element of the SortedSet.

2) Object last(); Returns Last Element of the SortedSet.

3) SortedSetheadSet(Object obj);

Returns SortedSet whose Elements are < Object.

4) SortedSettailSet(Object obj);

Returns SortedSet whose Elements are >= Object.

5) obj2 SortedSetsubSet(Object obj1, Object);

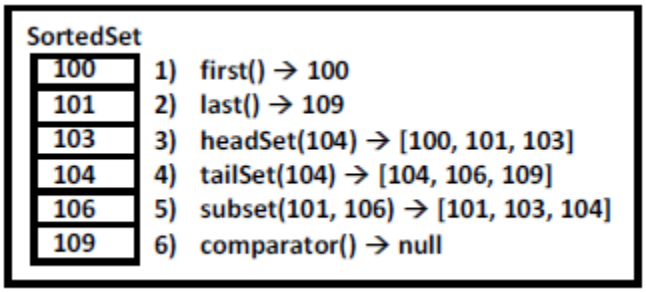
Returns SortedSet whose Elements are >= obj1 and <obj2.

6) Comparator comparator();

- **Returns Comparator Object that Describes Underlying SortingTechnique.**

 If we are using Default Natural Sorting Order then we will get null.

Eg:



3.2.1.1) TreeSet:

 The Underlying Data Structure is Balanced Tree.

 Insertion Order is Not Preserved and it is Based on Some Sorting Order.

 Heterogeneous Objects are Not Allowed. If we are trying to Insert we will get Runtime
Exception Saying ClassCastException.

 Duplicate Objects are Not allowed.

 null Insertion is Possible (Only Once).

 Implements *Serializable* and *Cloneable* Interfaces but Not *RandomAccess* Interface.

Constructors:

1) TreeSet t = new TreeSet();

Creates an Empty TreeSet Object where all Elements will be Inserted According to Default
Natural Sorting Order.

2) TreeSet t = new TreeSet(Comparator c);

Creates an Empty TreeSet Object where all Elements will be Inserted According to Customized Sorting Order which is described by Comparator Object.

3) TreeSet t = new TreeSet(Collection c);

4) TreeSet t = new TreeSet(SortedSet s);

```
importjava.util.TreeSet;
classTreeSetDemo {
        public static void main(String[] args) {
        TreeSet t = new TreeSet();
                t.add("A");
                t.add("a");
                t.add("B");
                t.add("Z");
                t.add("L");
        t.add(new Integer(10));
                t.add(null);//RE: Exception in thread "main" java.lang.NullPointerException
                System.out.println(t); //[A, B, L, Z, a]
        }
}
```

RE: Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer

null Acceptance:

For Empty TreeSet as the 1st Element null Insertion is Possible. But after inserting that null

if we are trying to Insert any Element we will get NullPointerException.

For Non- Empty TreeSet if we are trying to Insert null we will get NullPointerException.

```
importjava.util.TreeSet;
classTreeSetDemo {
        public static void main(String[] args) {
                TreeSet t = new TreeSet();
                t.add(new StringBuffer("A"));
                t.add(new StringBuffer("Z"));
                t.add(new StringBuffer("L"));
                t.add(new StringBuffer("B"));
                System.out.println(t);
        }
}
```

RE: Exception in thread "main" java.lang.ClassCastException: java.lang.StringBuffer cannot be cast to java.lang.Comparable

Note:

If we are Depending on Default Natural Sorting Order Compulsory Objects should be

Homogeneous **and** ***Comparable.*** **Otherwise we will get RE: ClassCastException.**

An object is said to be Comparable if and only if corresponding class implements Comparable interface.

All Wrapper Classes, String Class Already Implements ***Comparable*** **Interface. But StringBuffer Class doesn't Implement** ***Comparable*** **Interface.**

Hence we are ***ClassCastException*** **in the Above Example.**

Comparable (I):

Comparable Interface Present in java.lang Package and it contains Only One Method compareTo().

```
publicintcompareTo(Object o);
```

```
obj1.compareTo(obj2)
Returns -ve if and Only if obj1 has to Come Before obj2
Returns +ve if and Only if obj1 has to Come After obj2
Returns 0 if and Only if obj1 and obj2are Equal
```

Eg:

```
System.out.println("A".compareTo("Z")); //-25
System.out.println("Z".compareTo("K")); //15
System.out.println("Z".compareTo("Z")); //0
System.out.println("Z".compareTo(null)); //RE: java.lang.NullPointerException
```

Whenever we are Depending on Default Natural Sorting Order and if we are trying to Insert

Elements then Internally JVM will Call compareTo() to IdentifySorting Order.

Eg:

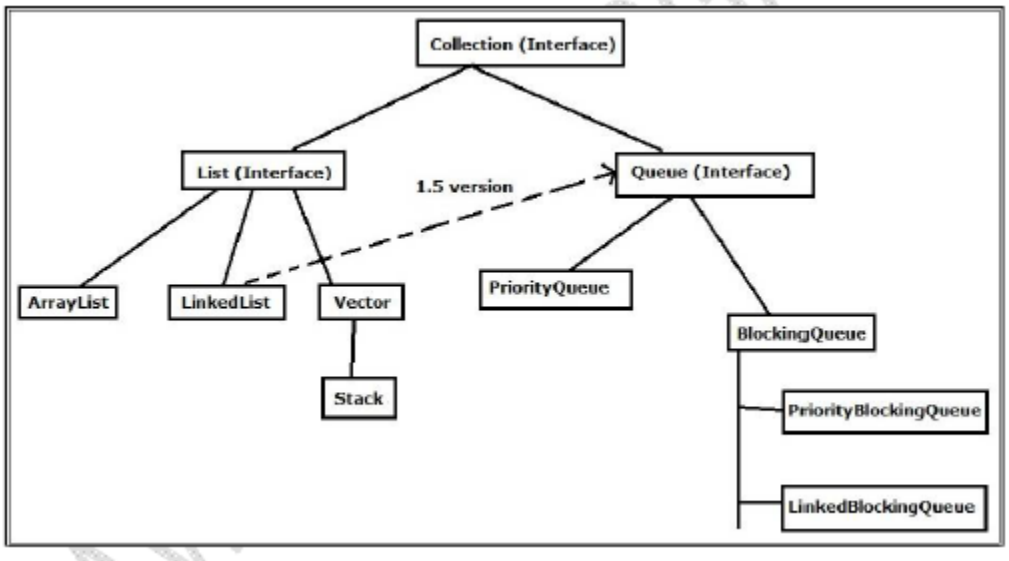
```
TreeSet t = new TreeSet();
t.add("K"); √
t.add("Z");  --(+ve)--> "Z".compateTo("K");
t.add("A");  --(- ve)--> "A".compateTo("K");
t.add("A");  --(0)--> "A".compateTo("A");
t.add(null); --> NullPointerException
System.out.println(t); --> [A, K, Z]
```

Note: If we are Not satisfied with Default Natural Sorting Order OR if Default Natural Sorting Order is Not Already Available then we can Define Our Own Sorting by using

Comparator Object.

Comparable Meant for Default Natural Sorting Order whereas
Comparator Meant for Customized Sorting Order

1.5 Version Enhancements (Queue Interface):



Queue is a Child Interface of Collection.

If we want to Represent a Group of Individual Objects Prior to processing then we should
go for Queue.

From 1.5 Version onwards LinkedListalso implements Queue Interface.

Usually Queue follows FIFO Order. But Based on Our Requirement we can Implement Our
Own Priorities Also (PriorityQueue)

LinkedList based Implementation of Queue always follows FIFO Order.

Eg: Before sending a Mail we have to Store all Mail IDs in Some Data Structure and for the 1st
Inserted Mail ID Mail should be Sent 1st.For this Requirement Queue is the Best Choice.

Methods:

1) boolean offer(Object o); To Add an Object into the Queue.

2) Object peek();

To Return Head Element of the Queue.

If Queue is Empty then this Method Returns null.

3) Object element();

To Return Head Element of the Queue.

If Queue is Empty then this Methodraises RE: NoSuchElementException

4) Object poll();

To Remove and Return Head Element of the Queue.

If Queue is Empty then this Method Returns null.

5) Object remove();

To Remove and Return Head Element of the Queue.

If Queue is Empty then this Method raise RE: NoSuchElementException.

PriorityQueue:

This is a Data Structure which can be used to Represent a Group of Individual Objects
Prior to processing according to Some Priority.

The Priority Order can be Either Default Natural Sorting Order OR Customized Sorting

Order specified by Comparator Object.

If we are Depending on Natural Sorting Order then the Objects should be *Homogeneous*

and *Comparable* otherwise we will get *ClassCastException.*

If we are defining Our Own Sorting by Comparator then the Objects Need Not be*Homogeneous* and *Comparable.*

Duplicate objects are Not Allowed.

Insertion Order is Not Preserved and it is Based on Some Priority.

null Insertion is Not Possible Even as 1st Element Also.

Constructors:

1) PriorityQueue q = new PriorityQueue();

Creates an Empty PriorityQueue with Default Initial Capacity 11 and all Objects will be Inserted according to Default Natural Sorting Order.

2) PriorityQueue q = new PriorityQueue(intinitialcapacity);

3) PriorityQueue q = new PriorityQueue(intinitialcapacity, Comparator c);

4) PriorityQueue q = new PriorityQueue(SortedSet s);

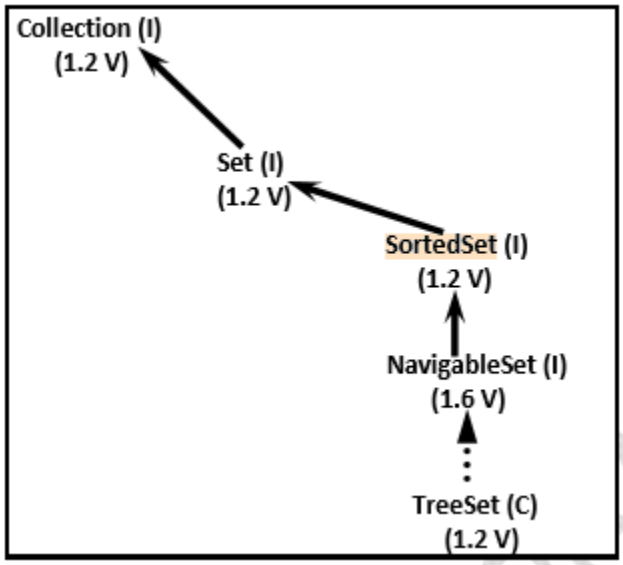
5) PriorityQueue q = new PriorityQueue(Collection c);

```
importjava.util.PriorityQueue;
classPriorityQueueDemo {
        public static void main(String[] args) {
                PriorityQueue q = new PriorityQueue();
                System.out.println(q.peek()); //null
                System.out.println(q.element()); // java.util.NoSuchElementException
                for(int i=0; i<=10; i++) {
                        q.offer(i);
                }
                System.out.println(q); //[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
                System.out.println(q.poll()); //0
                System.out.println(q); //[1, 3, 2, 7, 4, 5, 6, 10, 8, 9]
        }
}
```

Note: Some Operating Systems won't Provide Proper Support for PriorityQueues.

1.6 Version Enhancements:

NavigableSet (I):It is the Child Interface of SortedSet.



Methods:It Defines Several Methods for Navigation Purposes.

1) floor(e); It Returns Highest Element which is <=e.

2) lower(e); It Returns Highest Element which is <e.

3) ceiling(e); It Returns Lowest Element which is >=e.

4) higher(e); It Returns Lowest Element which is >e.

5) pollFirst(); Remove and Return 1st Element.

6) pollLast(); Remove and Return Last Element.

7) descendingSet(); It Returns NavigableSet in Reverse Order.

```
importjava.util.TreeSet;
classNavigableSetDemo {
        public static void main(String[] args) {
                TreeSet<Integer> t = new TreeSet<Integer>();
                t.add(1000);
                t.add(2000);
                t.add(3000);
                t.add(4000);
                t.add(5000);
                System.out.println(t);
                System.out.println(t.ceiling(2000));
                System.out.println(t.higher(2000));
                System.out.println(t.floor(3000));
                System.out.println(t.lower(3000));
                System.out.println(t.pollFirst());
                System.out.println(t.pollLast());
                System.out.println(t.descendingSet());
                System.out.println(t);
        }
}
```

```
[1000, 2000, 3000, 4000, 5000]
2000
3000
3000
2000
1000
5000
[4000, 3000, 2000]
[2000, 3000, 4000]
```