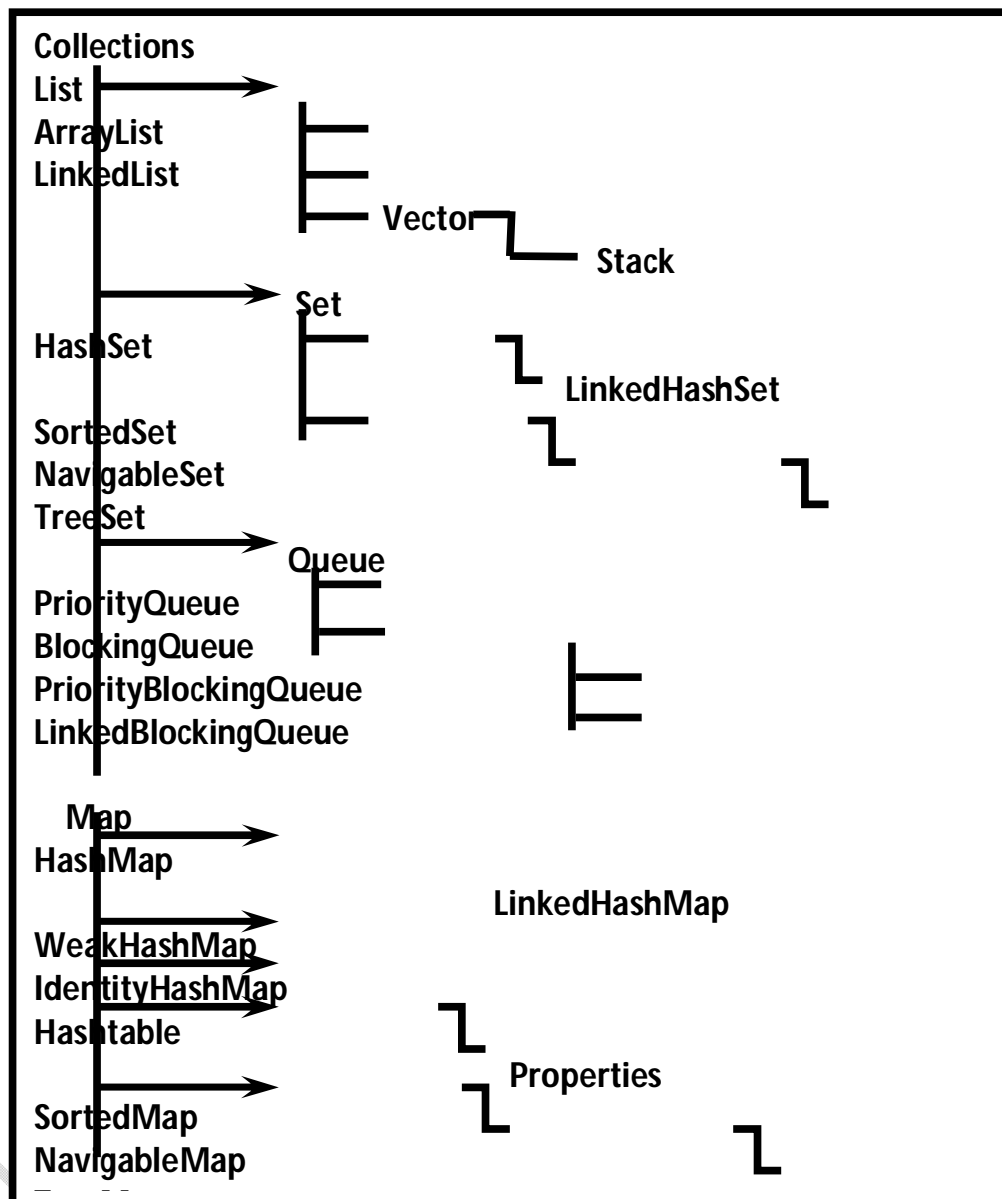


Collections Framework



Cursors

- ❖ Enumerations (I)
- ❖ Iterator (I)

Utility Classes

- ❖ Collections

Sortin

- ❖ Comparable (I)

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 Customer();✗  
CE: incompatible types  
found: Customer  
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 can Increase 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 <i>Homogeneous</i> and <i>Heterogeneous</i> 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)
- 2) List (I)
- 3) Set (I)
- 4) SortedSet(I)
- 5) NavigableSet(I)
- 6) Queue(I)
- 7) Map(I)
- 8) SortedMap(I)
- 9) NavigableMap(I)

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 Methods which 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.

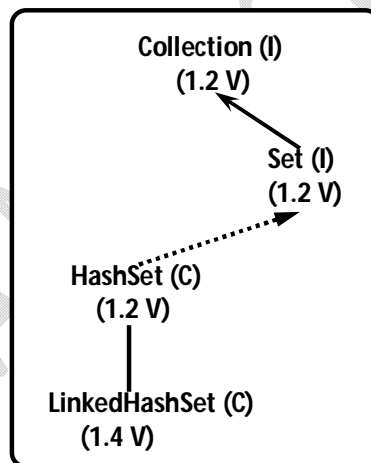
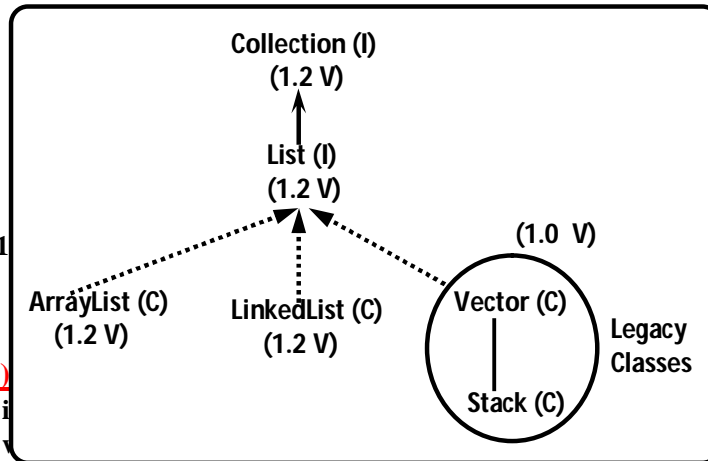
2) List (I):

- 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 V Interface.

3) Set (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 Not allowed and Insertion Order won't be Preserved. Then we should go for Set Interface.

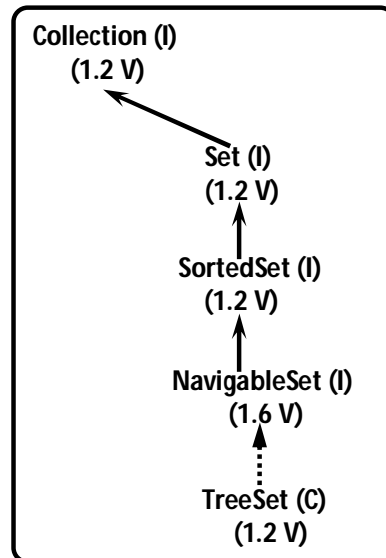


4) SortedSet (I):

- 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):

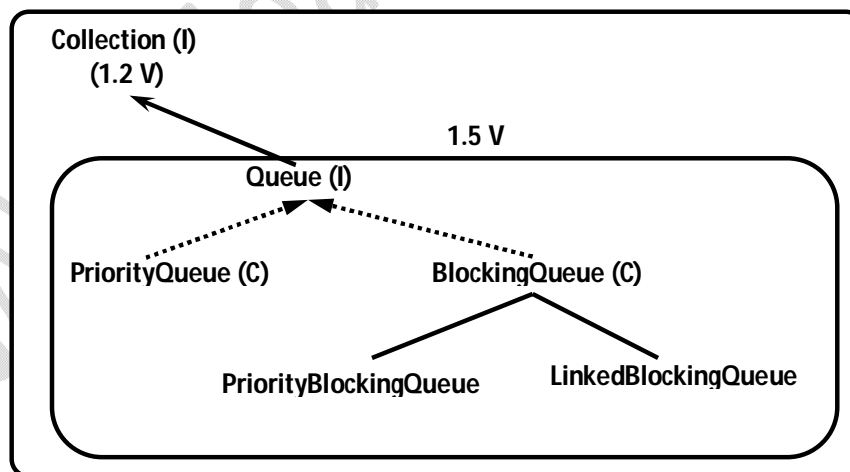
- It is the Child Interface of SortedSet.
- It defines Several Methods for Navigation Purposes.



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.

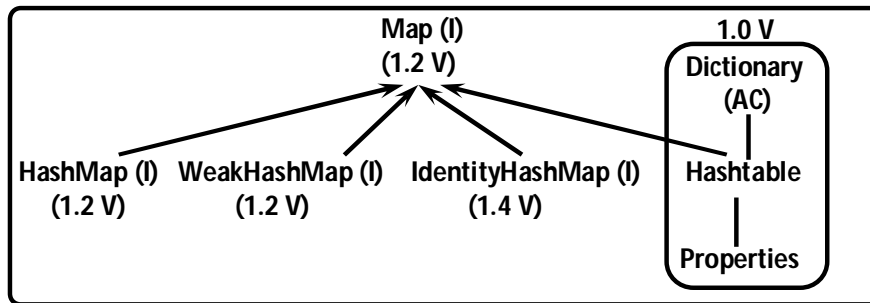


Note:

- 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.

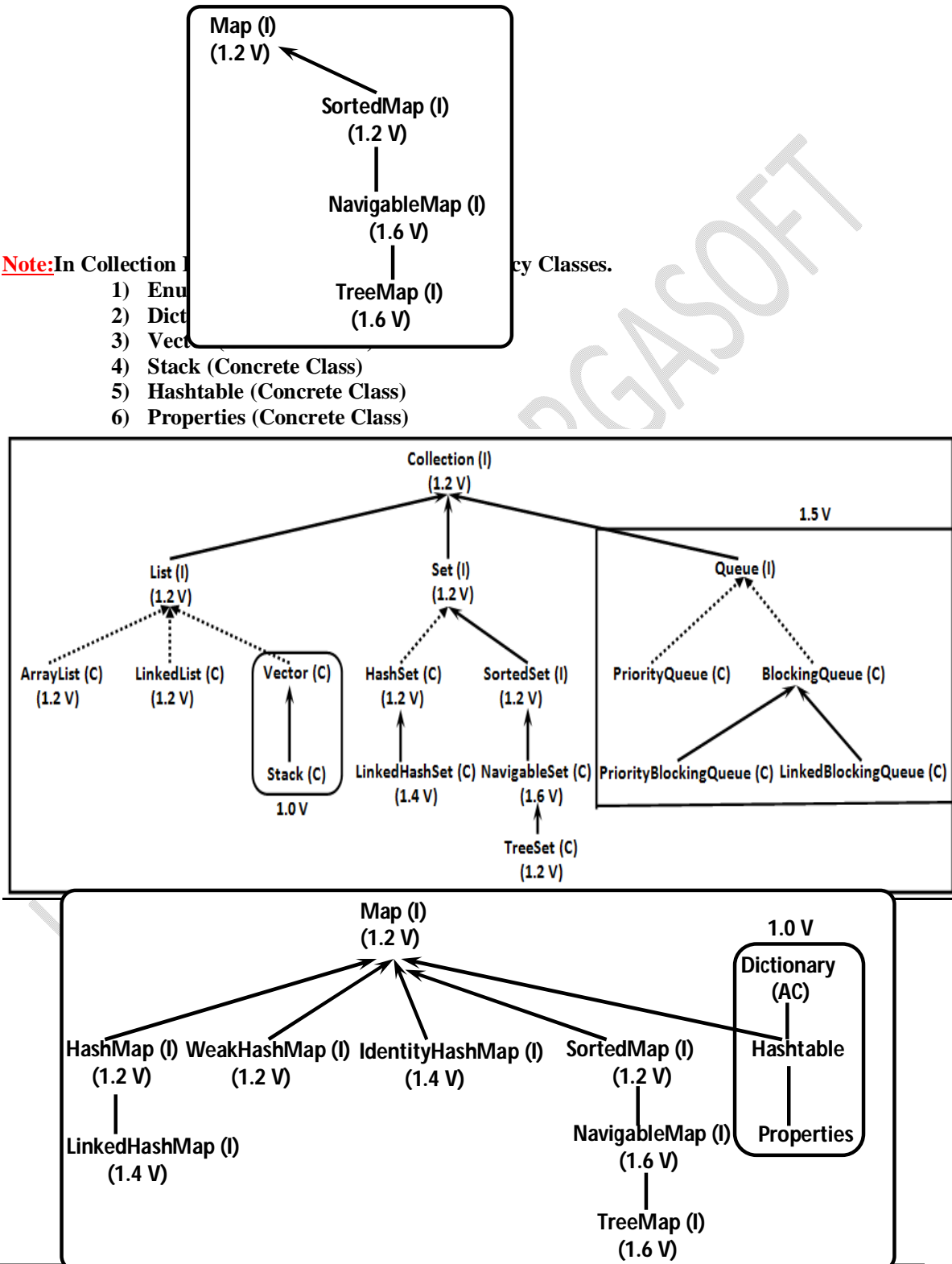
9) NavigableMap (I):

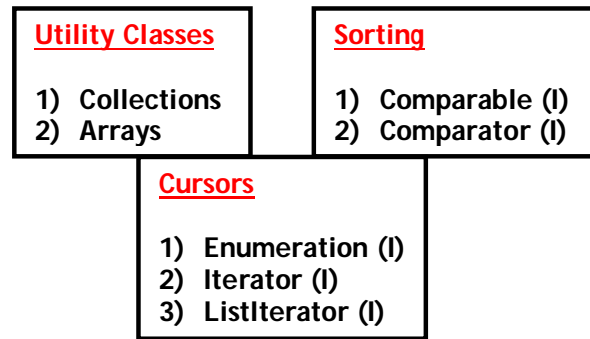
- It is the Child Interface of SortedMap.
- It Defines Several Methods for Navigation Purposes.

Note: In Collection

- 1) Enum
- 2) Dict
- 3) Vect
- 4) Stack (Concrete Class)
- 5) Hashtable (Concrete Class)
- 6) Properties (Concrete Class)

cy Classes.





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) boolean addAll(Collection c)
- 3) boolean remove(Object o)
- 4) boolean removeAll(Collection c)
- 5) **boolean retainAll(Collection c):** To Remove All Objects Except those Present in c.
- 6) void clear()
- 7) boolean contains(Object o)
- 8) boolean containsAll(Collection c)

- 9) boolean isEmpty()
- 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) boolean addAll(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) **int indexOf(Object o):** Returns Index of 1st Occurrence of 'o'
- 7) **int lastIndexOf(Object o)**
- 8) **ListIterator listIterator();**

2.1) ArrayList:

- The Underlying Data Structure for ArrayList is Resizable Array OR Growable 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

$$\text{New Capacity} = (\text{Current Capacity} * 3/2) + 1$$

2) ArrayList l = new ArrayList(intinitialCapacity);

Creates an Empty ArrayList Object with specified Initial Capacity.

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

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

```
import java.util.ArrayList;
class ArrayListDemo {
    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 instanceof Serializable); //true
System.out.println(l2 instanceof Cloneable); //true
System.out.println(l1 instanceof RandomAccess); //true
System.out.println(l2 instanceof RandomAccess); //false
```

Differences between *ArrayList* and *Vector*:

ArrayList	Vector
Every Method Present Inside ArrayList is 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:

<pre>ArrayList al = new ArrayList (); List l = Collections.synchronizedList(al);</pre>
<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> ↓ Synchronized Version </div> <div style="text-align: center;"> ↓ Non - Synchronized Version </div> </div>

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 *RandomAccessInterface*.
- 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()

```

import java.util.LinkedList;
class LinkedListDemo {
    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 OR Growable 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();**

- Creates an Empty Vector Object with Default Initial Capacity 10.
- Once Vector Reaches its Max Capacity then a New Vector Object will be Created with

$$\text{New Capacity} = \text{Current Capacity} * 2$$

- 2) Vector v = new Vector(int initialCapacity);
- 3) Vector v = new Vector(int initialCapacity, int incrementalCapacity);
- 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()

```

import java.util.Vector;
class VectorDemo {
    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():** To 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.

```

import java.util.Stack;
class StackDemo {
    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
    }
}

```

Offset		Index
1	C	2
2	B	1
3	A	0

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 boolean hasMoreElements();`
- 2) `public Object nextElement();`

```
import java.util.*;
class EnumerationDemo {
    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 l = (Integer)e.nextElement();
            if(l%2 == 0)
                System.out.println(l);
        }
        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 boolean hasNext()
- 2) public Object next()
- 3) public void remove()

```
import java.util.*;
class IteratorDemo {
    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 l = (Integer)itr.next();
            if(l%2 == 0)
                System.out.println(l);
            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]
```

Limitations:

- By using *Enumeration* and *Iterator* we can Move Only towards Forward Direction and we can't Move Backward Direction. That is these are Single Direction Cursors but Not Bi-Direction.
- By using Iterator we can Perform Only *Read* and *Remove* Operations and we can't Perform Addition of New Objects and Replacing Existing Objects.

To Overcome these Limitations we should go for ListIterator.

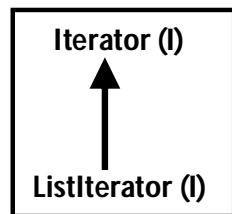
3) ListIterator:

- 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 and Replacing existing Objects. In Addition to Read and Remove Operations.
- We can Create ListIterator Object by using listIterator().
public ListIterator listIterator();

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

Methods:

- ListIterator is the Child Interface of Iterator and Hence All Iterator Methods by Default Available to the ListIterator.



- ListIterator Defines the following 9 Methods.

<pre> public boolean hasNext() public Object next() public int nextIndex() </pre>	<pre> } </pre>	Forward Direction
<pre> public boolean hasPrevious() public Object previous() public int previousIndex() </pre>	<pre> } </pre>	Backward Direction
<pre> public void remove() public void set(Object new) public void add(Object new) </pre>		

```
import java.util.*;
class ListIteratorDemo {
    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);
        ListIterator itr = l.listIterator();
        while(itr.hasNext()) {
            String s = (String)itr.next();
            if(s.equals("Venki"))
                itr.remove();
            if(s.equals("Naag"))
                itr.add("Chaitu");
            if(s.equals("Chiru"))
                itr.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)

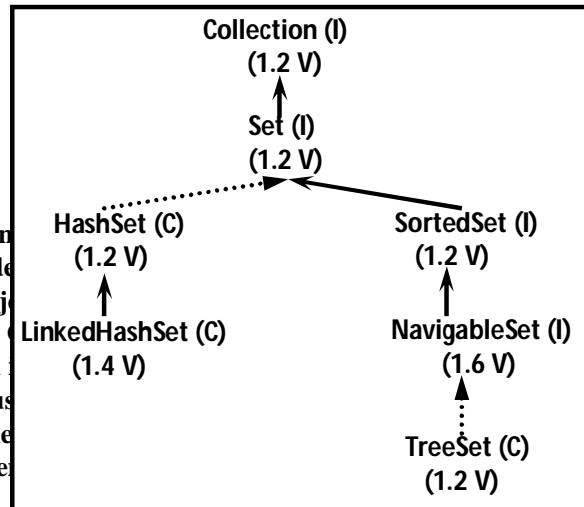
Internal Implementation of Cursors:

```
import java.util.*;
class CursorDemo {
    public static void main(String args[]) {
        Vector v = new Vector();
        Enumeration e = v.elements();
        Iterator itr = v.iterator();
        ListIterator litr = 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 Underlying
- Insertion Order
- Duplicate Objects won't get any
- null Insertion
- Heterogeneous
- HashSet implementation
- If Our Frequency

the Objects.
uplicate 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*.

3.1.1)

```
import java.util.*;
class HashSetDemo {
    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]
    }
}
```

HashSet	LinkedHashSet
The Underlying Data Structure is Hashtable.	The Underlying Data Structure is a Combination of <i>LinkedList</i> and <i>Hashtable</i> .
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) **SortedSet headSet(Object obj);**
Returns SortedSet whose Elements are < Object.
- 4) **SortedSet tailSet(Object obj);**
Returns SortedSet whose Elements are >= Object.

5) SortedSetsubSet(Object obj1, Object obj2);

Returns SortedSet whose Elements are \geq 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:

SortedSet	
100	1) first() \rightarrow 100
101	2) last() \rightarrow 109
103	3) headSet(104) \rightarrow [100, 101, 103]
104	4) tailSet(104) \rightarrow [104, 106, 109]
106	5) subset(101, 106) \rightarrow [101, 103, 104]
109	6) comparator() \rightarrow null

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);`

```
import java.util.TreeSet;
class TreeSetDemo {
    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.

```
import java.util.TreeSet;
class TreeSetDemo {
    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().

`public int compareTo(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 obj2 are 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 Identify Sorting Order.

Eg:

```
TreeSet t = new TreeSet();
t.add("K"); ✓

t.add("Z"); → +ve → "Z".compareTo("K");
t.add("A"); → -ve → "A".compareTo("K");
t.add("A"); → 0 → "A".compareTo("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

Comparator (I):

This Interface Present in java.util Package.

Methods: It contains 2 Methods *compare()* and *equals()*.

```
public int compare(Object obj1, Object 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 obj2 are Equal.

```
public boolean equals(Object o);
```

Whenever we are implementing Comparator Interface
Compulsory we should Provide Implementation for *compare()*.

Implementing *equals()* is Optional because it is Already
Available to Our Class from Object Class through Inheritance.

Write a Program to Insert Integer Objects into the TreeSet where Sorting Order is Descending Order:

```

import java.util.*;
class TreeSetDemo {
    public static void main(String[] args) {
        TreeSet t = new TreeSet(new MyComparator()); → 1
        t.add(10);
        t.add(0); → compare(0,10); +1
        t.add(15); → compare(15,10); -1
        t.add(5); → compare(5,15); +ve
        compare(5,10); +1
        compare(5,0); -1
        t.add(20); → compare(20,15); -1
        t.add(20); → compare(20,20); 0
        System.out.println(t); // [20, 15, 10, 5, 0]
    }
}

class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        Integer i1 = (Integer)obj1;
        Integer i2 = (Integer)obj2;
        if (i1 < i2)
            return +1;
        else if (i1 > i2)
            return -1;
        else
            return 0;
    }
}

```

- At Line 1 if we are Not Passing Comparator Object as an Argument then Internally JVM will Call `compareTo()`. Which is Meant for Default Natural Sorting Order (Ascending Order). In this Case the Output is [0, 5, 10, 15, 20].
- At Line 1 if we are Passing Comparator Object then JVM will Call `compare()` Instead of `compareTo()`. Which is Meant for Customized Sorting (Descending Order). In this Case the Output is [20, 15, 10, 5, 0].

Various Possible Implementations of `compare()`:

```

public int compare(Object obj1, Object obj2) {
    Integer i1 = (Integer)obj1;
    Integer i2 = (Integer)obj2;
    return i1.compareTo(i2); // [0, 5, 10, 15, 20] Ascending Order
    return -i1.compareTo(i2); // [20, 15, 10, 5, 0] Descending Order
    return i2.compareTo(i1); // [20, 15, 10, 5, 0]
    return -i2.compareTo(i1); // [0, 5, 10, 15, 20]
    return +1; // [10, 0, 15, 5, 20, 20] Insertion Order
    return -1; // [20, 20, 5, 15, 0, 10] Reverse of Insertion Order
    return 0; // [10] Only 1st Inserted Element Present And All Remaining Elements Treated as Duplicates
}

```

Write a Program to Insert String Objects into the TreeSet where the Sorting Order is of Reverse of Alphabetical Order:

```
import java.util.*;
class TreeSetDemo {
    public static void main(String[] args) {
        TreeSet t = new TreeSet(new MyComparator());
        t.add("Roja");
        t.add("Sobha Rani");
        t.add("Raja Kumari");
        t.add("Ganga Bhavani");
        t.add("Ramulamma");
        System.out.println(t);
    }
}
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        String s1 = obj1.toString();
        String s2 = (String)obj2;
        return s2.compareTo(s1); // [Sobha Rani, Roja, Ramulamma, Raja Kumari, Ganga Bhavani]
        // return -s1.compareTo(s2); // Valid
    }
}
```

Write a Program to Insert StringBuffer Objects into the TreeSet where Sorting Order is Alphabetical Order:

```
import java.util.*;
class TreeSetDemo {
    public static void main(String[] args) {
        TreeSet t = new TreeSet(new MyComparator1());
        t.add(new StringBuffer("A"));
        t.add(new StringBuffer("Z"));
        t.add(new StringBuffer("K"));
        t.add(new StringBuffer("L"));
        System.out.println(t);
    }
}
class MyComparator1 implements Comparator {
    public int compare(Object obj1, Object obj2) {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return s1.compareTo(s2); //[A, K, L, Z]
    }
}
```

Write a Program to Insert String and StringBuffer Objects into the TreeSet where Sorting Order is Increasing Length Order. If 2 Objects having Same Length then Consider their Alphabetical Order:

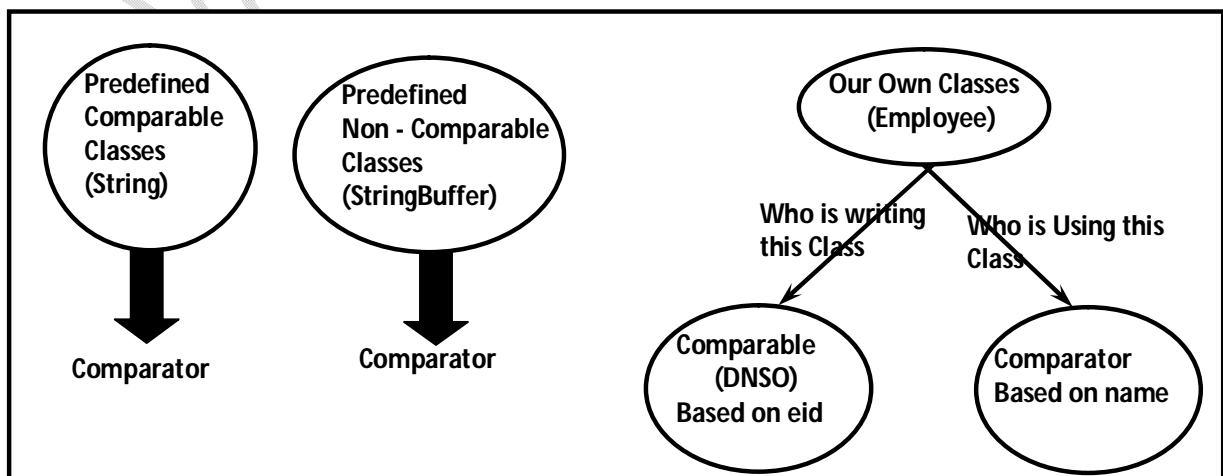
```
import java.util.*;
class TreeSetDemo {
    public static void main(String[] args) {
        TreeSet t = new TreeSet(new MyComparator());
        t.add("A");
        t.add(new StringBuffer("ABC"));
        t.add(new StringBuffer("AA"));
        t.add("XX");
        t.add("ABCE");
        t.add("A");
        System.out.println(t);
    }
}
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        int i1 = s1.length();
        int i2 = s2.length();
        if (i1 < i2) return -1;
        else if (i1 > i2) return 1;
        else return s1.compareTo(s2); //[A, AA, XX, ABC, ABCE]
    }
}
```

Note:

- If we are Depending on Default Natural Sorting Order Compulsory Objects should be *Homogeneous* and *Comparable* Otherwise we will get RE: ClassCastException.
- If we defining Our Own Sorting by Comparator then Objects Need Not be Homogeneous and Comparable. That is we can Add Heterogeneous Non Comparable Objects to the TreeSet.

When we go for Comparable and When we go for Comparator:
Comparable Vs Comparator:

- For Predefined Comparable Classes (Like String) Default Natural Sorting Order is Already Available. If we are Not satisfied with that we can Define Our Own Sorting by Comparator Object.
- For Predefine Non- Comparable Classes (Like StringBuffer) Default Natural Sorting Order is Not Already Available. If we want to Define Our Own Sorting we can Use Comparator Object.
- For Our Own Classes (Like Employee) the Person who is writing Employee Class he is Responsible to Define Default Natural Sorting Order by implementing Comparable Interface.
- The Person who is using Our Own Class if he is Not satisfied with Default Natural Sorting Order he can Define his Own Sorting by using Comparator Object.
- If he is satisfied with Default Natural Sorting Order then he can Use Directly Our Class.



Write a Program to Insert Employee Objects into the TreeSet where DNSO is Based on Ascending Order of EmployeeId and Customized Sorting Order is Based on Alphabetical Order of Names:

```
import java.util.*;
class Employee implements Comparable {
    String name;
    int eid;
    Employee(String name, int eid) {
        this.name = name;
        this.eid = eid;
    }
    public String toString() { return name+"-----"+eid;}
    public int compareTo(Object obj) {
        int eid1 = this.eid;
        Employee e = (Employee) obj;
        int eid2 = e.eid;
        if (eid1 < eid2) return -1;
        else if (eid1 > eid2) return 1;
        else return 0;
    }
}
class CompComp {
    public static void main(String[] args) {
        Employee e1 = new Employee("Nag", 100);
        Employee e2 = new Employee("Bala", 200);
        Employee e3 = new Employee("Chiru", 50);
        Employee e4 = new Employee("Venki", 150);
        Employee e5 = new Employee("Nag", 100);
        TreeSet t = new TreeSet();
        t.add(e1);
        t.add(e2);
        t.add(e3);
        t.add(e4);
        t.add(e5);
        System.out.println(t);
        TreeSet t1 = new TreeSet(new MyComparator());
        t1.add(e1);
        t1.add(e2);
        t1.add(e3);
        t1.add(e4);
        t1.add(e5);
        System.out.println(t1);
    }
}
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        Employee e1 = (Employee) obj1;
        Employee e2 = (Employee) obj2;
        String s1 = e1.name;
        String s2 = e2.name;
        return s1.compareTo(s2);
    }
}
```

Con

Pres

It is

Sorting Order.

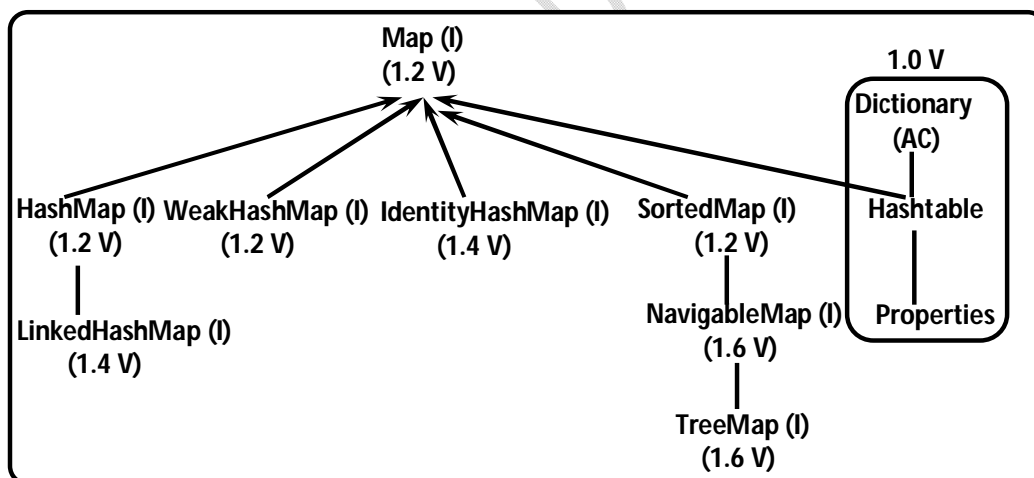
[Chiru-----50, Nag-----100, Venki-----150, Bala-----200]
[Bala-----200, Chiru-----50, Nag-----100, Venki-----150]

Order.	
Defines Only One Method <code>compareTo()</code> .	Defines 2 Methods <code>compare()</code> and <code>equals()</code> .
All Wrapper Classes and String Class implements Comparable Interface.	The Only implemented Classes of Comparator are <i>Collator</i> and <i>RuleBaseCollator</i> .

Comparison Table of Set implemented Classes:

Property	HashSet	LinkedHashSet	TreeSet
Underlying Data Structure	Hashtable	Hashtable and LinkedList	Balanced Tree
Insertion Order	Not Preserved	Preserved	Not Preserved
Sorting Order	Not Applicable	Not Applicable	Applicable
Heterogeneous Objects	Allowed	Allowed	Not Allowed
Duplicate Objects	Not Allowed	Not Allowed	Not Allowed
null Acceptance	Allowed (Only Once)	Allowed (Only Once)	For Empty TreeSet as the 1 st Element null Insertion is Possible. In all Other Cases we will get NullPointerException .

Map



- 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.
- Both Keys and Values are Objects Only.
- Duplicate Keys are Not allowed. But Values can be Duplicated.
- Each Key- Value Pair is Called an Entry.

	Key	Value	
Entry	101	Durga	value
	103	Pawan	
	104	NaNa	

Methods

Map Interface Defines the following Methods

1) **Object put(Object key, Object value);**

To Add One Key- Value Pair.If the specified Key is Already Available then Old Value will be Replaced with New Value and Returns Old Value.

2) void putAll(Map m)

3) Object get(Object key)

4) Object remove(Object key)

5) booleancontainsKey(Object key)

6) booleancontainsValue(Object value)

7) booleanisEmpty()

8) int size()

9) void clear()

10) Set keySet()

11) Collection values()

12) Set entrySet()

} Collection Views of Map

Entry (I):

- Each Key- Value Pair is Called One Entry.
- Without existing Map Object there is No Chance of existing Entry Object.
- Hence Interface Entry is Define Inside Map Interface.

```
interface Map{
interface Entry{
    Object getKey()
    Object getValue()
    Object setValue(Object new)
}
}
```

HashMap:

- The Underlying Data Structure is Hashtable.
- Duplicate Keys are Not Allowed. But Values can be Duplicated.
- Heterogeneous Objects are allowed for Both Keys and Values.
- Insertion Order is not preserved and it is based on hash code of the keys.
- null Insertion is allowed for Key (Only Once) and allowed for Values (Any Number of Times)

Differences between and HashMap and Hashtable:

HashMap	Hashtable
No Method Present in HashMap is Synchronized.	Every Method Present in Hashtable is Synchronized.
At a Time Multiple Threads are allowed to Operate on HashMap Object simultaneously and Hence it is Not Thread Safe.	At a Time Only One Thread is allowed to Operate on the Hashtable Object and Hence it is Thread Safe.
Relatively Performance is High.	Relatively Performance is Low.
null is allowed for Both Keys and Values.	null is Not allowed for Both Keys and Values. Otherwise we will get NPE.
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 HashMap:

By Default HashMap is Non- Synchronized. But we can get Synchronized Version of HashMap by using *synchronizedMap()* of Collections Class.

Constructors:

1) HashMap m = new HashMap();

Creates an Empty HashMap Object with Default Initial Capacity 16 and Default Fill Ratio 0.75

2) HashMap m = new HashMap(intinitialcapacity);

3) HashMap m = new HashMap(intinitialcapacity, float fillRatio);

4) HashMap m = new HashMap(Map m);

```

import java.util.*;
class HashMapDemo {
    public static void main(String[] args) {
        HashMap m = new HashMap();
        m.put("Chiru", 700);
        m.put("Bala", 800);
        m.put("Venki", 200);
        m.put("Nag", 500);
        System.out.println(m);
        System.out.println(m.put("Chiru", 1000));
        Set s = m.keySet();
        System.out.println(s);
        Collection c = m.values();
        System.out.println(c);
        Set s1 = m.entrySet();
        System.out.println(s1);
        Iterator itr = s1.iterator();
        while(itr.hasNext()) {
            Map.Entry m1 = (Map.Entry)itr.next();
            System.out.println(m1.getKey()+"....."+m1.getValue());
            if(m1.getKey().equals("Nag")) {
                m1.setValue(10000);
            }
        }
        System.out.println(m);
    }
}

```

```

{Chiru=700, Venki=200, Nag=500, Bala=800}
700
[Chiru, Venki, Nag, Bala]
[1000, 200, 500, 800]
[Chiru=1000, Venki=200, Nag=500, Bala=800]
Chiru.....1000
Venki.....200
Nag.....500
Bala.....800
{Chiru=1000, Venki=200, Nag=10000, Bala=800}

```

LinkedHashMap:

- It is the Child Class of HashMap.
- It is Exactly Same as HashMap Except the following Differences.

HashMap	LinkedHashMap
The Underlying Data Structure is Hashtable.	The Underlying Data Structure is Combination of Hashtable and LinkedList.
Insertion is Not Preserved.	Insertion Order is Preserved.
Introduced in 1.2 Version.	Introduced in 1.4 Version.

In the Above Example if we Replace HashMap with LinkedHashMap then Output is

```
{Chiru=700, Bala=800, Venki=200, Nag=500}
700
[Chiru, Bala, Venki, Nag]
[1000, 800, 200, 500]
[Chiru=1000, Bala=800, Venki=200, Nag=500]
Chiru.....1000
Bala.....800
Venki.....200
Nag.....500
{Chiru=1000, Bala=800, Venki=200, Nag=10000}
```

That is Insertion Order is Preserved.

Note: In General we can Use *LinkedHashSet* and *LinkedHashMap* for developing Cache Based Applications where Duplicates are Not Allowed. But Insertion Order Must be Preserved.

IdentityHashMap:

It is Exactly Same as HashMap Except the following Difference.

- In *HashMap* JVM will Use `.equals()` to Identify Duplicate Keys, which is Meant for *Content* Comparison.
- In *IdentityHashMap* JVM will Use `==` Operator to Identify Duplicate Keys, which is Meant for *Reference* Comparison.

```
import java.util.HashMap;
class IdentityHashMapDemo {
    public static void main(String[] args) {
        HashMap m = new HashMap();
        Integer l1 = new Integer(10);
        Integer l2 = new Integer(10);
        m.put(l1, "Pawan");
        m.put(l2, "Kalyan");
        System.out.println(m); //{10=Kalyan}
    }
}
```

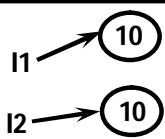
If we Replace *HashMap* with *IdentityHashMap* in the Above Application then Output is {10=Pawan, 10=Kalyan}.

Because I1 and I2 are Not Duplicate as I1 == I2 Returns false.

What is the Difference between == Operator and .equals()?

In General we can Use == Operator for Reference Comparison whereas .equals() for Content Comparison.

```
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);
System.out.println(I1 == I2); //false
System.out.println(I1.equals(I2)); //true
```



WeakHashMap:

It is Exactly Same as HashMap Except the following Difference.

- In Case of HashMap, HashMap Dominates Garbage Collector. That is if Object doesn't have any Reference Still it is Not Eligible for Garbage Collector if it is associated with HashMap.
- But In Case of WeakHashMap if an Object doesn't contain any References then it is Always Eligible for GC Even though it is associated with WeakHashMap. That is Garbage Collector Dominates WeakHashMap.

```
import java.util.HashMap;
class WeakHashMapDemo {
    public static void main(String[] args) throws InterruptedException {
        HashMap m = new HashMap();
        Temp t = new Temp();
        m.put(t, "Durga");
        System.out.println(m);
        t = null;
        System.gc();
        Thread.sleep(5000);
        System.out.println(m);
    }
}
class Temp {
    public String toString() {
        return "temp";
    }
    public void finalize() {
        System.out.println("finalize() Called");
    }
}
```

{temp=Durga}
 {temp=Durga}

If we Replace *HashMap* with *WeakHashMap* then the Output is

```
{temp=Durga}
finalize() Called
{}
```

SortedMap:

- It is the Child Interface of Map.
- If we want to Represent a Group of Key - Value Pairs According Some Sorting Order of Keys then we should go for SortedMap.

Methods:

SortedMap Defines the following Specific Methods.

- 1) Object firstKey();
- 2) Object lastKey();
- 3) SortedMapheadMap(Object key)
- 4) SortedMaptailMap(Object key)
- 5) SortedMapsubMap(Object key1, Object key2)
- 6) Comparator comparator()

TreeMap:

The Underlying Data Structure is Red -Black Tree.

Duplicate Keys are Not Allowed. But Values can be Duplicated.

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

If we are depending on Default Natural Sorting Order then the Keys should be *Homogeneous* and *Comparable*. Otherwise we will get Runtime Exception Saying *ClassCastException*.

If we defining Our Own Sorting by Comparator then Keys can be *Heterogeneous* and *Non-Comparable*.

But there are No Restrictions on Values. They can be *Heterogeneous* and *Non- Comparable*.

null Acceptance:

- For Empty TreeMap as the 1st Entry with null Key is Allowed. But After inserting that Entry if we are trying to Insert any Other Entry we will get RE: *NullPointerException*.
- For Non- Empty TreeMap if we are trying to Insert null Entry then we will get Runtime Exception Saying *NullPointerException*.
- There are No Restrictions on null Values.

Constructors:

- 1) **TreeMap t = new TreeMap();** For Default Natural Sorting Order.
- 2) **TreeMap t = new TreeMap(Comparator c);** For Customized Sorting Order.
- 3) **TreeMap t = new TreeMap(SortedMap m);** Inter Conversion between Map Objects.
- 4) **TreeMap t = new TreeMap(Map m);**

Example on Natural Sorting:

```
import java.util.TreeMap;
class TreeMapDemo {
    public static void main(String[] args) {
        TreeMap m = new TreeMap();
        m.put(100, "ZZZ");
        m.put(103, "YYY");
        m.put(101, "XXX");
        m.put(104, 106);
        m.put(107, null);
        m.put("FFF", "XXX");
        m.put(null, "XXX"); //RE: Exception in thread "main" java.lang.NullPointerException
        System.out.println(m); //{100=ZZZ, 101=XXX, 103=YYY, 104=106, 107=null}
    }
}
```

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

Example on Customized Sorting:

```
import java.util.*;
class TreeMapDemo {
    public static void main(String[] args) {
        TreeMap m = new TreeMap(new MyComparator());
```


Hashtable:

- The Underlying Data Structure for Hashtable is Hashtable Only.
- Duplicate Keys are Not Allowed. But Values can be Duplicated.
- Insertion Order is Not Preserved and it is Based on Hashcode of the Keys.
- Heterogeneous Objects are Allowed for Both Keys and Values.
- null Insertion is Not Possible for Both Key and Values. Otherwise we will get Runtime Exception Saying NullPointerException.
- Every Method Present in Hashtable is Synchronized and Hence Hashtable Object is Thread Safe.

Constructors:**1) Hashtable h = new Hashtable();**

Creates an Empty Hashtable Object with Default Initial Capacity 11 and Default Fill Ratio 0.75.

2) Hashtable h = new Hashtable(intinitialcapacity);**3) Hashtable h = new Hashtable(intinitialcapacity, float fillRatio);****4) Hashtable h = new Hashtable(Map m);**

```
import java.util.Hashtable;
class HashtableDemo {
    public static void main(String[] args) {
        Hashtable h = new Hashtable();
        h.put(new Temp(5), "A");
        h.put(new Temp(2), "B");
        h.put(new Temp(6), "C");
        h.put(new Temp(15), "D");
        h.put(new Temp(23), "E");
        h.put(new Temp(16), "F");
        h.put("Durga", null); //RE: java.lang.NullPointerException
        System.out.println(h); //{6=C, 16=F, 5=A, 15=D, 2=B, 23=E}
    }
}
```

Pro

-
-
-
-
-
-

```
class Temp {
    int i;
    Temp(int i) {
        this.i = i;
    }
    public int hashCode() {
        return i;
    }
    public String toString() {
        return i+"";
    }
}
```

10	
9	
8	
7	
6	6 = C
5	5 = A, 16 = F
4	15 = D
3	
2	2 = B
1	23 = E
0	

From Top To Bottom

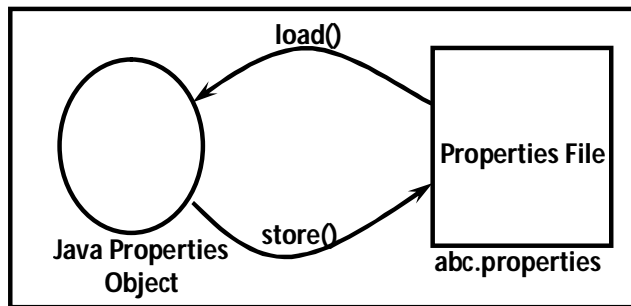
From Right To Left

Value should be String Type.

Constructor: Properties p = new Properties();

Methods:

- 1) **public String getProperty(String pname);**
To Get the Value associated with specified Property.
- 2) **public String setProperty(String pname, String pvalue);**
To Set a New Property.
- 3) **public Enumeration propertyNames();** It Returns All Property Names.
- 4) **public void load(InputStream is);**
To Load Properties from Properties File into Java Properties Object.
- 5) **public void store(OutputStream os, String comment);**
To Store Properties from Java Properties Object into Properties File.



```

import java.util.Properties;
import java.io.*;
class PropertiesDemo {
    public static void main(String[] args) throws Exception {
        Properties p = new Properties();
        FileInputStream fis = new FileInputStream("abc.properties");
        p.load(fis);
        System.out.println(p);
        String s = p.getProperty("Venki");
        System.out.println(s);
        p.setProperty("Nag", "88888");
        FileOutputStream fos = new FileOutputStream("abc.properties");
        p.store(fos, "Updated by Durga for SCJP Class");
    }
}
  
```

abc.properties

User Name: Scott
Password: Tiger
Venki = 9999;

#Updated by Durga for SCJP Class
#Wed May 20 08:23:37 IST 2015
Venki=9999;
Password=Tiger
Nag=88888
User=Name\.: Scott

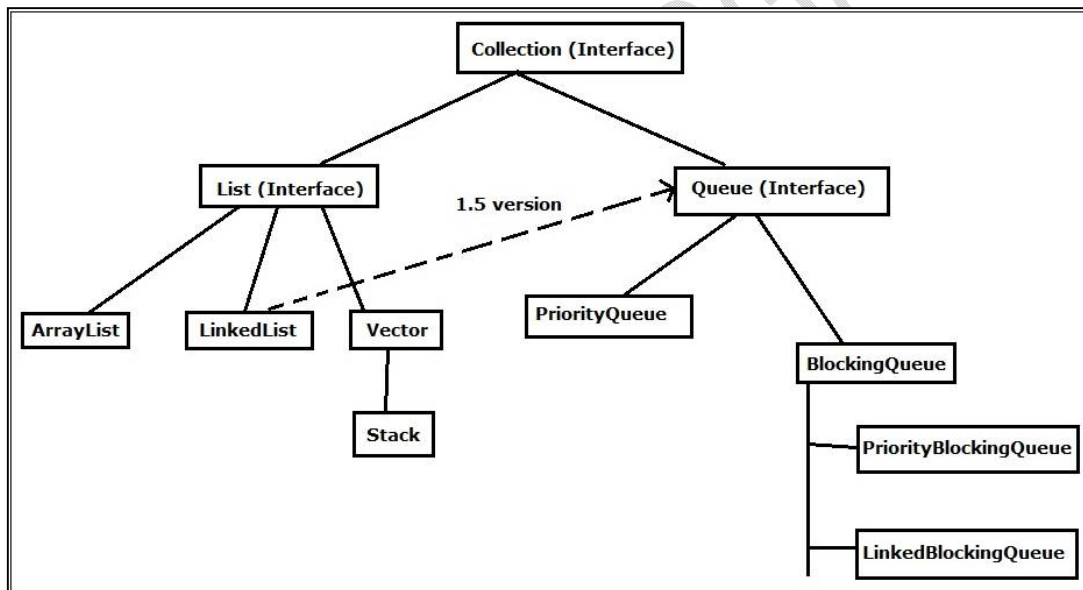
ki=9999;, Password=Tiger, User=Name: Scott}

Eg: Pseudo Code

```

import java.util.*;
import java.io.*;
class PropertiesDemo {
    public static void main(String[] args) throws Exception {
        Properties p = new Properties();
        FileInputStream fis = new FileInputStream("db.properties");
        p.load(fis);
        String url = p.getProperty("url");
        String user = p.getProperty("user");
        String pwd = p.getProperty("pwd");
        Connection con = DriverManager.getConnection(url,user,pwd);
        //.....
    }
}

```

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 LinkedList also 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 Method raises 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);`

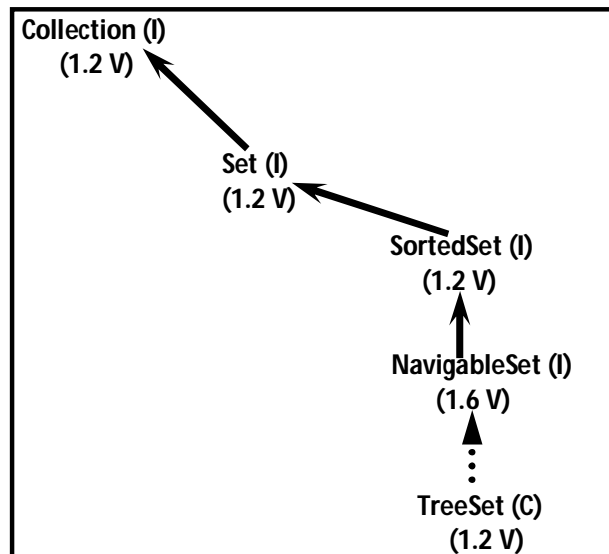
```
import java.util.PriorityQueue;
class PriorityQueueDemo {
    public static void main(String[] args) {
        PriorityQueue q = new PriorityQueue();
        System.out.println(q.peek()); //null
        System.out.println(q.element()); // java.util.NoSuchElementException
        for(int i=0; i<=10; 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.

```
import java.util.*;
class PriorityQueueDemo {
    public static void main(String[] args) {
        PriorityQueue q = new PriorityQueue(15, new MyComparator());
        q.offer("A");
        q.offer("Z");
        q.offer("L");
        q.offer("B");
        System.out.println(q); //[Z, B, L, A]
    }
}
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        String s1 = (String)obj1;
        String s2 = obj2.toString();
        return s2.compareTo(s1);
    }
}
```

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 $\leq e$.
- 2) **lower(e);** It Returns Highest Element which is $< e$.
- 3) **ceiling(e);** It Returns Lowest Element which is $\geq e$.
- 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.

```

import java.util.TreeSet;
class NavigableSetDemo {
    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]

```

NavigableMap: It is the Child Interface of SortedMap.

Methods:

- 1) floorKey(e)
- 2) lowerKey(e)
- 3) ceilingKey(e)
- 4) higherKey(e)
- 5) pollFirstEntry()
- 6) pollLastEntry()
- 7) descendingMap()


```

import java.util.TreeMap;
class NavigableMapDemo {
    public static void main(String[] args) {
        TreeMap<String, String> t = new TreeMap<String, String>();

        t.put("b", "Banana");
        t.put("c", "Cat");
        t.put("a", "Apple");
        t.put("d", "Dog");
        t.put("g", "Gun");

        System.out.println(t);
        System.out.println(t.ceilingKey("c"));
        System.out.println(t.higherKey("e"));
        System.out.println(t.floorKey("e"));
        System.out.println(t.lowerKey("e"));
        System.out.println(t.pollFirstEntry());
        System.out.println(t.pollLastEntry());
        System.out.println(t.descendingMap());
        System.out.println(t);
    }
}

```

```

{a=Apple, b=Banana, c=Cat, d=Dog, g=Gun}
c
g
d
d
a=Apple
g=Gun
{d=Dog, c=Cat, b=Banana}
{b=Banana, c=Cat, d=Dog}

```

Utility Classes

- ☀ Collections
- ☀ Arrays

Collections (C):

Collections Class is an Utility Class Present in *java.util* Package to Define Several Utility Methods for Collection Objects.

☀ To Sort Elements of List:

Collections Class Defines the following Methods for this Purpose.

1) public static void sort(List l);

- To Sort Based on Default Natural Sorting Order.
- In this Case Compulsory List should contain Only *Homogeneous* and *Comparable* Objects. Otherwise we will get Runtime Exception Saying *ClassCastException*.
- List should Not contain null Otherwise we will get *NullPointerException*.

2) public static void sort(List l, Comparator c);

To Sort Based on Customized Sorting Order.

Program: To Sort Elements of List According to Natural Sorting Order

```
import java.util.*;
class CollectionsSortDemo {
    public static void main(String args[]) {
        ArrayList al = new ArrayList();
        al.add("Z");
        al.add("A");
        al.add("K");
        al.add("N");
        al.add(new Integer(10));
        al.add(null); //RE: Exception in thread "main" java.lang.NullPointerException
        System.out.println("Before Sorting:" + al); //Before Sorting:[Z, A, K, N]
        Collections.sort(al);
        System.out.println("After Sorting:" + al); //After Sorting:[A, K, N, Z]
    }
}
```

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

Program: To Sort Elements of List According to Customized Sorting Order

```
import java.util.*;
class CollectionsSortDemo {
    public static void main(String args[]) {
        ArrayList al = new ArrayList();
        al.add("Z");
        al.add("A");
        al.add("K");
        al.add("N");
        System.out.println("Before Sorting:" + al); //Before Sorting:[Z, A, K, N]
        Collections.sort(al, new MyComparator());
        System.out.println("After Sorting:" + al); //After Sorting: [Z, N, K, A]
    }
}
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        String s1 = (String)obj1;
        String s2 = obj2.toString();
        return s2.compareTo(s1);
    }
}
```

☀ 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);

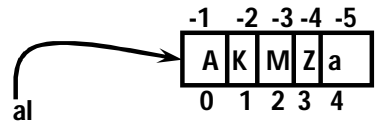
If we are Sorting List according to Comparator then we have to Use this Method.

Conclusions:

- ❖ Internally the Above Search Methods will Use Binary Search Algorithm.
- ❖ Before performing Search Operation Compulsory List should be Sorted. Otherwise we will get Unpredictable Results.
- ❖ Successful Search Returns Index.
- ❖ Unsuccessful Search Returns Insertion Point.
- ❖ Insertion Point is the Location where we can Insert the Target Element in the SortedList.
- ❖ 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.

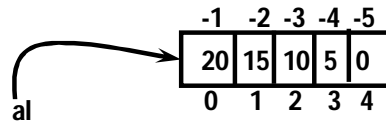
Program: To Search Elements of List According to Natural Sorting Order

```
import java.util.*;
class CollectionsSearchDemo {
    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");
        System.out.println(al); //[Z, A, M, K, a]
        Collections.sort(al);
        System.out.println(al); //[A, K, M, Z, a]
        System.out.println(Collections.binarySearch(al, "Z")); //3
        System.out.println(Collections.binarySearch(al, "J")); //-2
    }
}
```



Program: To Search Elements of List According to Customized Sorting Order

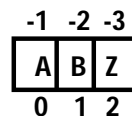
```
import java.util.*;
class CollectionsSearchDemo {
    public static void main(String args[]) {
        ArrayList al = new ArrayList();
        al.add(15);
        al.add(0);
        al.add(20);
        al.add(10);
        al.add(5);
        System.out.println(al); //[15, 0, 20, 10, 5]
        Collections.sort(al, new MyComparator());
        System.out.println(al); //[20, 15, 10, 5, 0]
        System.out.println(Collections.binarySearch(al, 10, new MyComparator())); //2
        System.out.println(Collections.binarySearch(al, 13, new MyComparator())); //-3
        System.out.println(Collections.binarySearch(al, 17)); //-6
    }
}
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        Integer I1 = (Integer)obj1;
        Integer I2 = (Integer)obj2;
        return I2.compareTo(I1);
    }
}
```

**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

Eg: For the List of 3 Elements

- 1) Range of Successful Search: 0 To 2
- 2) Range of Unsuccessful Search: -4 To -1
- 3) Total Result Range: -4 To 2

**☀ Reversing the Elements of List:** public static void reverse(List l);

Program: To Reverse Elements of List

```

import java.util.*;
class CollectionsReverseDemo {
    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);
        System.out.println(al); //[15, 0, 20, 10, 5]
        Collections.sort(al);
        System.out.println(al); //[0, 5, 10, 15, 20]
    }
}

```

reverse() Vs reverseOrder():

- We can Use reverse() to Reverse Order of Elements of List.
- We can Use reverseOrder() to get Reversed Comparator.

Eg:

Comparator c1 = Collections.reverseOrder(Comparator c);	
↓	↓
Descending Order	Ascending Order

Arrays

Arrays Class is an Utility Class to Define Several Utility Methods for Array Objects.

☀ Sorting Elements of Array:

- 1) **public static void sort(primitive[] p);** To Sort According to Natural Sorting Order.
- 2) **public static void sort(Object[] o);** To Sort According to Natural Sorting Order.

3) **public static void sort(Object[] o, Comparator c);**To Sort According to Customized Sorting Order.

Note:

- For Object Type Arrays we can Sort According to *Natural Sorting Order* OR *Customized Sorting Order*.
- But we can Sort primitive[] Only Based on Natural Sorting.

Program: To Sort Elements of Array

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

        int[] a = {10, 5, 20, 11, 6};
        System.out.println("Primitive Array Before Sorting:");
        for (int a1 : a) {
            System.out.println(a1);
        }

        Arrays.sort(a);
        System.out.println("Primitive Array After Sorting:");
        for (int a1 : a) {
            System.out.println(a1);
        }

        String[] s = {"A", "Z", "B"};
        System.out.println("Object Array Before Sorting:");
        for (String s1 : s) {
            System.out.println(s1);
        }

        Arrays.sort(s);
        System.out.println("Object Array After Sorting:");
        for (String s1 : s) {
            System.out.println(s1);
        }
    }
}
```

Primitive Array Before Sorting:

10
5
20
11
6

Primitive Array After Sorting:

5
6
10
11
20

Object Array Before Sorting:

A
Z
B

Object Array After Sorting:

A
B
Z

Object Array After Sorting By Comparator:

Z
R

☀ Searching the Elements of Array:

- 1) **public static int binarySearch(primitive[] p, primitive target);**
If the Primitive Array Sorted According to Natural Sorting Order then we have to Use this Method.
- 2) **public static int binarySearch(Object[] a, Object target);**
If the Object Array Sorted According to Natural Sorting Order then we have to Use this Method.
- 3) **public static int binarySearch(Object[] a, Object target, Comparator c);**
If the Object Array Sorted According to Comparator then we have to Use this Method.

Note: All Rules of Array Class binarySearch() are Exactly Same as Collections Class binarySearch().

Program: To Search Elements of Array

```
import java.util.Arrays;
import java.util.Comparator;
import static java.util.Arrays.*;

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

        int[] a = {10, 5, 20, 11, 6};
        Arrays.sort(a); //Sort By Natural Order
        System.out.println(Arrays.binarySearch(a, 6)); //1
        System.out.println(Arrays.binarySearch(a, 14)); //-5

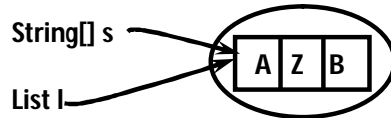
        String[] s = {"A", "Z", "B"};
        Arrays.sort(s);
        System.out.println(binarySearch(s, "Z")); //2
        System.out.println(binarySearch(s, "S")); //-3

        Arrays.sort(s, new MyComparator());
        System.out.println(binarySearch(s, "Z", new MyComparator())); //0
        System.out.println(binarySearch(s, "S", new MyComparator())); //-2
        System.out.println(binarySearch(s, "N")); //-4
    }
}

class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return s2.compareTo(s1);
    }
}
```

Conversion of Array to List:

- Arrays Class contains asList() for this → public static List asList(Object[] a);
- Strictly Speaking this Method won't Create an Independent List Object, Just we are Viewing existing Array in List Form.



- By using Array Reference if we Perform any Change Automatically that Change will be reflected to List Reference.
- Similarly by using List Reference if we Perform any Change Automatically that Change will be reflected to Array.
- By using List Reference if we are trying to Perform any Operation which Varies the Size then we will get Runtime Exception Saying *UnsupportedOperationException*.

Eg:

```
l.add("K");//RE: UnsupportedOperationException
l.remove(1);//RE: UnsupportedOperationException
l.set(1, "K");//✓
```

- By using List Reference if we are trying to Replace with Heterogeneous Objects then we will get Runtime Exception Saying *ArrayStoreException*.

Program: To View Array in List Form

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

        String[] s = {"A", "Z", "B"};
        List l = Arrays.asList(s);
        System.out.println(l); //[A, Z, B]

        s[0] = "K";
        System.out.println(l); //[K, Z, B]

        l.set(1, "L");

        for (String s1 : s)
            System.out.println(s1); //K L B

        l.add("Durga"); //RE: java.lang.UnsupportedOperationException
        l.remove(2); //RE: java.lang.UnsupportedOperationException
        l.set(1, new Integer[10]); //RE: java.lang.ArrayStoreException:[Ljava.lang.Integer;
    }
}
```


Concurrent Collections (1.5)

Need fo Concurrent Collections

The Important Concurrent Classes

ConcurrentHashMap

CopyOnWriteArrayList

CopyOnWriteArraySet

ConcurrentMap (I)

ConcurrentHashMap

Difference between HashMap and ConcurrentHashMap

Difference between ConcurrentHashMap, synchronizedMap() and

Hashtable

CopyOnWriteArrayList (C)

Differences between ArrayList and CopyOnWriteArrayList

Differences between CopyOnWriteArrayList, synchronizedList() and vector()

CopyOnWriteArraySet

Differences between CopyOnWriteArraySet() and synchronizedSet()

Fail Fast Vs Fail Safe Iterators

Differences between Fail Fast and Fail Safe Iterators

Enum with Collections

EnumSet

EnumMap

Queue

PriorityQueue

BlockingQueue

TransferQueue

Deque

BlockingDeque (I)

Need for Concurrent Collections

- Traditional Collection Object (Like ArrayList, HashMap etc) can be accessed by Multiple Threads simultaneously and there may be a chance of Data Inconsistency Problems and Hence these are Not Thread Safe.
- Already existing Thread Safe Collections (Vector, Hashtable, synchronizedList(), synchronizedSet(), synchronizedMap()) Performance wise Not Up to the Mark.
- Because for Every Operation Even for Read Operation Also Total Collection will be loaded by Only One Thread at a Time and it Increases waiting Time of Threads.

```
import java.util.ArrayList;
import java.util.Iterator;
class Test {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add("A");
        al.add("B");
        al.add("C");
        Iterator itr = al.iterator();
        while (itr.hasNext()){
            String s = (String)itr.next();
            System.out.println(s);
            //al.add("D");
        }
    }
}
```

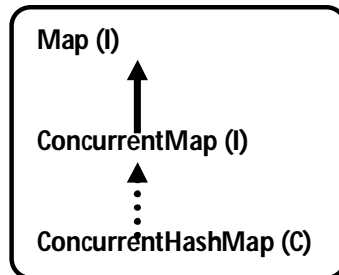
- Another Big Problem with Traditional Collections is while One Thread iterating Collection, the Other Threads are Not allowed to Modify Collection Object simultaneously if we are trying to Modify then we will get *ConcurrentModificationException*.
 - Hence these Traditional Collection Objects are Not Suitable for *Scalable Multi Threaded Applications*.
 - To Overcome these Problems SUN People introduced *Concurrent Collections* in 1.5 Version.
- 1) Concurrent Collections are Always Thread Safe.
 - 2) When compared with Traditional Thread Safe Collections Performance is More because of different Locking Mechanism.
 - 3) While One Thread interacting Collection the Other Threads are allowed to Modify Collection in Safe Manner.

Hence Concurrent Collections Never throw *ConcurrentModificationException*.

The Important Concurrent Classes are

- ❖ ConcurrentHashMap
- ❖ CopyOnWriteArrayList
- ❖ CopyOnWriteArraySet

ConcurrentMap (I):



Methods: It Defines the following 3 Specific Methods.

1) Object putIfAbsent(Object Key, Object Value)

To Add Entry to the Map if the specified Key is Not Already Available.

```

Object putIfAbsent(Object key, Object value)
if (!map.containsKey(key)) {
    map.put(key, value);
}
else {
    return map.get(key);
}
  
```

put()	putIfAbsent()
If the Key is Already Available, Old Value will be replaced with New Value and Returns Old Value.	If the Key is Already Present then Entry won't be added and Returns Old associated Value. If the Key is Not Available then Only Entry will be added.

```

import java.util.concurrent.ConcurrentHashMap;
class Test {
    public static void main(String[] args) {
        ConcurrentHashMap m = new ConcurrentHashMap();
        m.put(101, "Durga");
        m.put(101, "Ravi");
        System.out.println(m); //{101=Ravi}
        m.putIfAbsent(101, "Siva");
        System.out.println(m); //{101=Ravi}
    }
}
  
```

2) boolean remove(Object key, Object value)

Removes the Entry if the Key associated with specified Value Only.

```
if ( map.containsKey (key) &&map.get(key).equals(value) ) {  
    map.remove(key);  
    return true;  
}  
else {  
    return false;  
}
```

```
import java.util.concurrent.ConcurrentHashMap;  
class Test {  
    public static void main(String[] args) {  
        ConcurrentHashMap m = new ConcurrentHashMap();  
        m.put(101, "Durga");  
        m.remove(101, "Ravi"); //Value Not Matched with Key So Nor Removed  
        System.out.println(m); //{101=Durga}  
        m.remove(101, "Durga");  
        System.out.println(m); //{}  
    }  
}
```

3) boolean replace(Object key, Object oldValue, Object newValue)

If the Key Value
Matched then
Replace with

```
if ( map.containsKey (key) &&map.get(key).equals(oldvalue) ) {
    map.put(key, newValue);
    return true;
}
else {
    return false;
}
```

```
import java.util.concurrent.ConcurrentHashMap;
class Test {
    public static void main(String[] args) {
        ConcurrentHashMap m = new ConcurrentHashMap();
        m.put(101, "Durga");
        m.replace(101, "Ravi", "Siva");
        System.out.println(m); //{101=Durga}
        m.replace(101, "Durga", "Ravi");
        System.out.println(m); //{101=Ravi}
    }
}
```

ConcurrentHashMap

- Underlying Data Structure is Hashtable.
- ConcurrentHashMap allows Concurrent Read and Thread Safe Update Operations.
- To Perform Read Operation Thread won't require any Lock. But to Perform Update Operation Thread requires Lock but it is the Lock of Only a Particular Part of Map (Bucket Level Lock).

- Instead of Whole Map Concurrent Update achieved by Internally dividing Map into Smaller Portion which is defined by *Concurrency Level*.
- The Default Concurrency Level is 16.
- That is ConcurrentHashMap Allows simultaneous Read Operation and simultaneously 16 Write (Update) Operations.
- null is Not Allowed for Both Keys and Values.
- While One Thread iterating the Other Thread can Perform Update Operation and ConcurrentHashMap Never throw *ConcurrentModificationException*.

Constructors:

- 1) **ConcurrentHashMap m = new ConcurrentHashMap();**
Creates an Empty ConcurrentHashMap with Default Initial Capacity 16 and Default Fill Ratio 0.75 and Default Concurrency Level 16.
- 2) **ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity);**
- 3) **ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity, float fillRatio);**
- 4) **ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity, float fillRatio, int concurrencyLevel);**
- 5) **ConcurrentHashMap m = new ConcurrentHashMap(Map m);**

```
import java.util.concurrent.ConcurrentHashMap;
class Test {
    public static void main(String[] args) {
        ConcurrentHashMap m = new ConcurrentHashMap();
        m.put(101, "A");
        m.put(102, "B");
        m.putIfAbsent(103, "C");
        m.putIfAbsent(101, "D");
        m.remove(101, "D");
        m.replace(102, "B", "E");
        System.out.println(m); //{103=C, 102=E, 101=A}
    }
}
```

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.*;
class MyThread extends Thread {
//static HashMap m = new HashMap(); // java.util.ConcurrentModificationException
static ConcurrentHashMap m = new ConcurrentHashMap();
    public void run() {
        try {
            Thread.sleep(2000);
        }
        catch (InterruptedException e) {}
        System.out.println("Child Thread updating Map");
        m.put(103, "C");
    }
    public static void main(String[] args) throws InterruptedException {
        m.put(101, "A");
        m.put(102, "B");
        MyThread t = new MyThread();
        t.start();
        Set s = m.keySet();
        Iterator itr = s.iterator();
        while (itr.hasNext()) {
            Integer l1 = (Integer) itr.next();
            SOP("Main Thread iterating and Current Entry is:"+l1+"....."+m.get(l1));
            Thread.sleep(3000);
        }
        System.out.println(m);
    }
}
```

```
Main Thread iterating and Current Entry is:102.....B
Child Thread updating Map
Main Thread iterating and Current Entry is:101.....A
{103=C, 102=B, 101=A}
```

Update and we won't get any *ConcurrentModificationException*.

- If we Replace *ConcurrentHashMap* with *HashMap* then we will get *ConcurrentModificationException*.

```
import java.util.Iterator;
class Test {
    public static void main(String[] args) throws InterruptedException {
        ConcurrentHashMap m = new ConcurrentHashMap();
        m.put(101, "A");
        m.put(102, "B");
        Iterator itr = m.keySet().iterator();
        m.put(103, "C");
        while (itr.hasNext()) {
            Integer l1 = (Integer) itr.next();
            System.out.println(l1+"....."+m.get(l1));
            Thread.sleep(3000);
        }
        System.out.println(m);
    }
}
```

```
102.....B
101.....A
{103=C, 102=B, 101=A}
```

Reason:

- In the Case of ConcurrentHashMap iterator creates a Read Only Copy of Map Object and iterates over that Copy if any Changes to the Map after getting iterator it won't be affected/ reflected.
- In the Above Program if we Replace ConcurrentHashMap with HashMap then we will get ConcurrentModificationException.

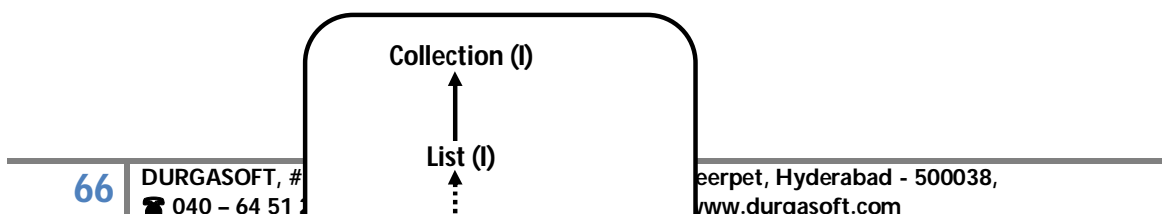
Difference between HashMap and ConcurrentHashMap

HashMap	ConcurrentHashMap
It is Not Thread Safe.	It is Thread Safe.
Relatively Performance is High because Threads are Not required to wait to Operate on HashMap.	Relatively Performance is Low because Some Times Threads are required to wait to Operate on ConcurrentHashMap.
While One Thread iterating HashMap the Other Threads are Not allowed to Modify Map Objects Otherwise we will get Runtime Exception Saying ConcurrentModificationException.	While One Thread iterating ConcurrentHashMap the Other Threads are allowed to Modify Map Objects in Safe Manner and it won't throw ConcurrentModificationException.
Iterator of HashMap is Fail-Fast and it throws ConcurrentModificationException.	Iterator of ConcurrentHashMap is Fail-Safe and it won't throwsConcurrentModificationException.
null is allowed for Both Keys and Values.	null is Not allowed for Both Keys and Values. Otherwise we will get NullPointerException.
Introduced in 1.2 Version.	Introduced in 1.5 Version.

Difference between ConcurrentHashMap, synchronizedMap() and Hashtable

ConcurrentHashMap	synchronizedMap()	Hashtable
We will get Thread Safety without locking Total Map Object Just with Bucket Level Lock.	We will get Thread Safety by locking Whole Map Object.	We will get Thread Safety by locking Whole Map Object.
At a Time Multiple Threads are allowed to Operate on Map Object in Safe Manner.	At a Time Only One Thread is allowed to Perform any Operation on Map Object.	At a Time Only One Thread is allowed to Operate on Map Object.
Read Operation can be performed without Lock but write Operation can be performed with Bucket Level Lock.	Every Read and Write Operations require Total Map Object Lock.	Every Read and Write Operations require Total Map Object Lock.
While One Thread iterating Map Object, the Other Threads are allowed to Modify Map and we won't get ConcurrentModificationException.	While One Thread iterating Map Object, the Other Threads are Not allowed to Modify Map. Otherwise we will get ConcurrentModificationException	While One Thread iterating Map Object, the Other Threads are Not allowed to Modify Map. Otherwise we will get ConcurrentModificationException
Iterator of ConcurrentHashMap is Fail-Safe and won't raise ConcurrentModificationException.	Iterator of synchronizedMap is Fail-Fast and it will raise ConcurrentModificationException.	Iterator of synchronizedMap is Fail-Fast and it will raise ConcurrentModificationException.
null is Not allowed for Both Keys and Values.	null is allowed for Both Keys and Values.	null is Not allowed for Both Keys and Values.
Introduced in 1.5 Version.	Introduced in 1.2 Version.	Introduced in 1.0 Version.

CopyOnWriteArrayList (C):



- It is a Thread Safe Version of ArrayList as the Name indicates CopyOnWriteArrayList Creates a Cloned Copy of Underlying ArrayList for Every Update Operation at Certain Point Both will Synchronized Automatically Which is taken Care by JVM Internally.
- As Update Operation will be performed on cloned Copy there is No Effect for the Threads which performs Read Operation.
- It is Costly to Use because for every Update Operation a cloned Copy will be Created. Hence CopyOnWriteArrayList is the Best Choice if Several Read Operations and Less Number of Write Operations are required to Perform.
- Insertion Order is Preserved.
- Duplicate Objects are allowed.
- Heterogeneous Objects are allowed.
- null Insertion is Possible.
- It implements Serializable, Clonable and RandomAccess Interfaces.
- While One Thread iterating CopyOnWriteArrayList, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException. That is iterator is Fail Safe.
- Iterator of ArrayList can Perform Remove Operation but Iterator of CopyOnWriteArrayList can't Perform Remove Operation. Otherwise we will get RuntimeException Saying UnsupportedOperationException.

Constructors:

- 1) CopyOnWriteArrayList l = new CopyOnWriteArrayList();
- 2) CopyOnWriteArrayList l = new CopyOnWriteArrayList(Collection c);
- 3) CopyOnWriteArrayList l = new CopyOnWriteArrayList(Object[] a);

Methods:

1. boolean addIfAbsent(Object o): The Element will be Added if and Only if List doesn't contain this Element.

```
CopyOnWriteArrayList l = new CopyOnWriteArrayList();
l.add("A");
l.add("A");
l.addIfAbsent("B");
l.addIfAbsent("B");
System.out.println(l); //[A, A, B]
```

2. **addAllAbsent(Collection c):** The Elements of Collection will be Added to the List if Elements are Absent and Returns Number of Elements Added.

```
ArrayList l = new ArrayList();
l.add("A");
l.add("B");

CopyOnWriteArrayList l1 = new CopyOnWriteArrayList();
l1.add("A");
l1.add("C");
System.out.println(l1); //[A, C]
l1.addAll(l);
System.out.println(l1); //[A, C, A, B]

ArrayList l2 = new ArrayList();
l2.add("A");
l2.add("D");
l1.addAllAbsent(l2);
System.out.println(l1); //[A, C, A, B, D]
```

```
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.ArrayList;
class Test {
    public static void main(String[] args) {
        ArrayList l = new ArrayList();
        l.add("A");
        l.add("B");

        CopyOnWriteArrayList l1 = new CopyOnWriteArrayList();
        l1.addIfAbsent("A");
        l1.addIfAbsent("C");
        l1.addAll(l);

        ArrayList l2 = new ArrayList();
        l2.add("A");
        l2.add("E");
        l1.addAllAbsent(l2);

        System.out.println(l1); //[A, C, A, B, E]
    }
}
```

```
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.*;
class MyThread extends Thread {
    static CopyOnWriteArrayList l = new CopyOnWriteArrayList();
    public void run() {
        try { Thread.sleep(2000); }
        catch (InterruptedException e) {}
        System.out.println("Child Thread Updating List");
        l.add("C");
    }
}
```

- In the Above Example while Main Thread iterating List Child Thread is allowed to Modify and we won't get any ConcurrentModificationException.
- If we Replace CopyOnWriteArrayList with ArrayList then we will get ConcurrentModificationException.
- Iterator of CopyOnWriteArrayList can't Perform Remove Operation. Otherwise we will get RuntimeException: UnsupportedOperationException.

```
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.Iterator;
class Test {
    public static void main(String[] args){
        CopyOnWriteArrayList l = new CopyOnWriteArrayList();
        l.add("A");
        l.add("B");
        l.add("C");
        l.add("D");
        System.out.println(l); //[A, B, C, D]
        Iterator itr = l.iterator();
        while (itr.hasNext()) {
            String s = (String)itr.next();
            if (s.equals("D"))
                itr.remove();
        }
        System.out.println(l); //RE: java.lang.UnsupportedOperationException
    }
}
```

- If we Replace CopyOnWriteArrayList with ArrayList we won't get any UnsupportedOperationException.
- In this Case the Output is
 - [A, B, C, D]
 - [A, B, C]

```
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.Iterator;
class Test {
    public static void main(String[] args) {
        CopyOnWriteArrayList l = new CopyOnWriteArrayList();
        l.add("A");
        l.add("B");
        l.add("C");
        Iterator itr = l.iterator();
        l.add("D");
        while (itr.hasNext()) {
            String s = (String)itr.next();
            System.out.println(s);
        }
    }
}
```

A
B
C

Reason:

- Every Update Operation will be performed on Separate Copy Hence After getting iterator if we are trying to Perform any Modification to the List it won't be reflected to the iterator.
- In the Above Program if we ReplaceCopyOnWriteArrayList with ArrayList then we will get RuntimeException: java.util.ConcurrentModificationException.

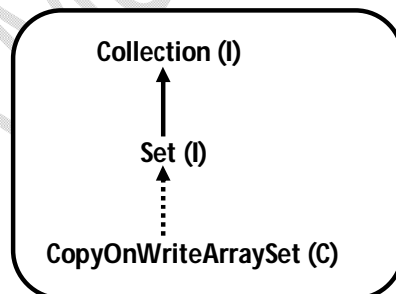
Differences between ArrayList and CopyOnWriteArrayList

ArrayList	CopyOnWriteArrayList
It is Not Thread Safe.	It is Not Thread Safe because Every Update Operation will be performed on Separate cloned Coy.
While One Thread iterating List Object, the Other Threads are Not allowed to Modify List Otherwise we will get ConcurrentModificationException.	While One Thread iterating List Object, the Other Threads are allowed to Modify List in Safe Manner and we won't get ConcurrentModificationException.
Iterator is Fail-Fsat.	Iterator is Fail-Safe.
Iterator of ArrayList can Perform Remove Operation.	Iterator of CopyOnWriteArrayList can't Perform Remove Operation Otherwise we will get RuntimeException: UnsupportedOperationException.
Introduced in 1.2 Version.	Introduced in 1.5 Version.

Differences between CopyOnWriteArrayList, synchronizedList() and vector()

CopyOnWriteArrayList	synchronizedList()	vector()
We will get Thread Safety because Every Update Operation will be performed on Separate cloned Copy.	We will get Thread Safety because at a Time List can be accessed by Only One Thread at a Time.	We will get Thread Safety because at a Time Only One Thread is allowed to Access Vector Object.
At a Time Multiple Threads are allowed to Access/ Operate on CopyOnWriteArrayList.	At a Time Only One Thread is allowed to Perform any Operation on List Object.	At a Time Only One Thread is allowed to Operate on Vector Object.
While One Thread iterating List Object, the Other Threads are allowed to Modify Map and we won't get ConcurrentModificationException.	While One Thread iterating, the Other Threads are Not allowed to Modify List. Otherwise we will get ConcurrentModificationException	While One Thread iterating, the Other Threads are Not allowed to Modify Vector. Otherwise we will get ConcurrentModificationException
Iterator is Fail-Safe and won't raise ConcurrentModificationException.	Iterator is Fail-Fast and it will raise ConcurrentModificationException.	Iterator is Fail-Fast and it will raise ConcurrentModificationException.
Iterator can't Perform Remove Operation Otherwise we will get UnsupportedOperationException.	Iterator can Perform Remove Operation.	Iterator can Perform Remove Operation.
Introduced in 1.5 Version.	Introduced in 1.2 Version.	Introduced in 1.0 Version.

CopyOnWriteArraySet :



- It is a Thread Safe Version of Set.
- Internally Implement by CopyOnWriteArrayList.
- Insertion Order is Preserved.
- Duplicate Objects are Notallowed.
- Multiple Threads can Able to Perform Read Operation simultaneously but for Every Update Operation a Separate cloned Copy will be Created.
- As for Every Update Operation a Separate cloned Copy will be Created which is Costly Hence if Multiple Update Operation are required then it is Not recommended to Use CopyOnWriteArraySet.

- While One Thread iterating Set the Other Threads are allowed to Modify Set and we won't get ConcurrentModificationException.
- Iterator of CopyOnWriteArraySet can Perform Only Read Operation and won't Perform Remove Operation. Otherwise we will get RuntimeException: UnsupportedOperationException.

Constructors:

1) CopyOnWriteArraySets = new CopyOnWriteArraySet();

Creates an Empty CopyOnWriteArraySet Object.

2) CopyOnWriteArraySet s = new CopyOnWriteArraySet(Collection c);

Creates CopyOnWriteArraySet Object which is Equivalent to given Collection Object.

Methods: Whatever Methods Present in Collection and Set Interfaces are the Only Methods Applicable for CopyOnWriteArraySet and there are No Special Methods.

```
import java.util.concurrent.CopyOnWriteArraySet;
class Test {
    public static void main(String[] args) {
        CopyOnWriteArraySet s = new CopyOnWriteArraySet();
        s.add("A");
        s.add("B");
        s.add("C");
        s.add("A");
        s.add(null);
        s.add(10);
        s.add("D");
    }
}
```

Differences between CopyOnWriteArraySet() and synchronizedSet()

CopyOnWriteArraySet()	synchronizedSet()
It is Thread Safe because Every Update Operation will be performed on Separate Cloned Copy.	It is Thread Safe because at a Time Only One Thread can Perform Operation.
While One Thread iterating Set, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException.	While One Thread iterating, the Other Threads are Not allowed to Modify Seta Otherwise we will get ConcurrentModificationException.
Iterator is Fail Safe.	Iterator is Fail Fast.

Iterator can Perform Only Read Operation and can't Perform Remove Operation Otherwise we will get RuntimeException Saying UnsupportedOperationException.	Iterator can Perform Both Read and Remove Operations.
Introduced in 1.5 Version.	Introduced in 1.7 Version.

Fail Fast Iterator

Fail Fast Vs Fail Safe Iterators:

Fail Fast Iterator: While One Thread iterating Collection if Other Thread trying to Perform any Structural Modification to the underlying Collection then immediately Iterator Fails by raising ConcurrentModificationException. Such Type of Iterators are Called Fail Fast Iterators.

```
import java.util.ArrayList;
import java.util.Iterator;
class Test {
    public static void main(String[] args) {
        ArrayList l = new ArrayList();
        l.add("A");
        l.add("B");
        Iterator itr = l.iterator();
        while(itr.hasNext()) {
            String s = (String)itr.next();
            System.out.println(s); //A
            l.add("C"); // java.util.ConcurrentModificationException
        }
    }
}
```

Fail Fast Iterator

Note: Internally Fail Fast Iterator will Use Some Flag named with MOD to Check underlying Collection is Modified OR Not while iterating.

Fail Safe Iterator:

- While One Thread iterating if the Other Threads are allowed to Perform any Structural Changes to the underlying Collection, Such Type of Iterators are Called Fail Safe Iterators.
- Fail Safe Iterators won't raise ConcurrentModificationException because Every Update Operation will be performed on Separate cloned Copy.

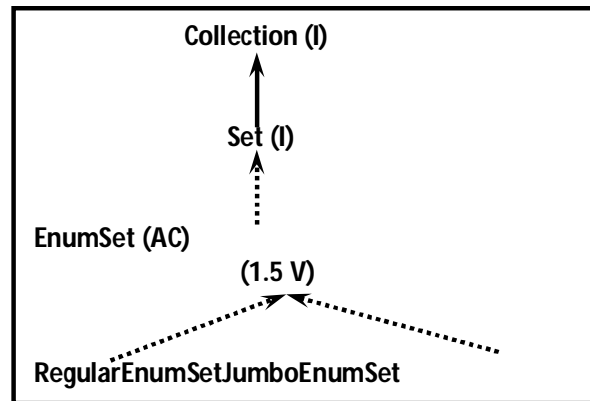
```
import java.util.concurrent.CopyOnWriteArraySet;
import java.util.Iterator;
class Test {
    public static void main(String[] args) {
        CopyOnWriteArraySet l = new CopyOnWriteArraySet();
        l.add("A");
        l.add("B");
        Iterator itr = l.iterator();
        while(itr.hasNext()) {
            String s = (String)itr.next();
            l.add("C");
        }
    }
}
```

Fail Safe Iterator

Differences between Fail Fast and Fail Safe Iterators:

Property	Fail Fast	Fail Safe
Does it through ConcurrentModificationException?	Yes	No
Is the Cloned Copy will be Created?	No	Yes
Memory Problems	No	Yes
Examples	ArrayList, Vector, HashMap, HashSet	ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet

Enum with Collections



EnumSet:

- It is a specially designed Set implemented Collection Applicable Only for Enum.
- Introduced in 1.5 Version.
- EnumSet is Internally implemented as Bit Vectors which Improves Performance Internally.
- The Performance of EnumSet is Very High if we want to Store Enum Constants than Traditional Collections (Like HashSet, LinkedHashSetEtc).
- All Elements of the EnumSet should be from Same Enum Type Only if we are trying to Add Elements from different enums then we will get Compile Time Error (i.e. EnumSet is Type Safe Collection).
- Iterator Returned by EnumSet Traverse, Iterate Elements in their Natural Order i.e. the Order in which the Enum Constants are declared i.e. the Order Returned by ordinal().
- Enum Iterator Never throw ConcurrentModificationException.
- Inside EnumSet we can't Add null Otherwise we will get NullPointerException.
- EnumSet is an Abstract Class and Hence we can't Create Object directly by using new Key Word.
- EnumSet defined Several Factory Methods to Create EnumSet Object.
- EnumSet defines 2 Child Classes.
 - RegularEnumSet
 - JumboEnumSet
- The Factory Methods will Return this Class Objects Internally Based on Size if the Size is < 64 then RegularEnumSet will be choosed Otherwise if Size > 64 then JumboEnumSet will be choosed.

EnumMap:

- It is a specially designed Map to Use Enum Type Objects as Keys.
- Introduced in 1.5 Version.
- It implements *Serializable* and *Cloneable* Interfaces.
- EnumMap is Internally implemented by using Bit Vectors (Arrays), which Improves Performance when compared with Traditional Map Object Like HashMap Etc.
- All Keys to the EnumMap should be from a Single Enum if we are trying to Use from different Enum then we will get Compile Time Error. Hence EnumMap is Type Safe.
- Iterator Never throw ConcurrentModificationException.

- Iterators of EnumMap iterate Elements according to Ordinal Value of Enum Keys i.e. in which Order Enum Constants are declared in the Same Order Only Iterator will be iterated.
- null Key is Not allowed Otherwise we will get NullPointerException.

Constructors

1) EnumMap m = new EnumMap(Class KeyType)

Creates an Empty EnumMap with specified Key Type.

2) EnumMap m = new EnumMap(EnumMap m1)

Creates an EnumMap with the Same Key Type and the specified EnumMap. Internally containing Same Mappings.

3) EnumMap m = new EnumMap(Map m1)

To Create and Equivalent EnumMap for given Map.

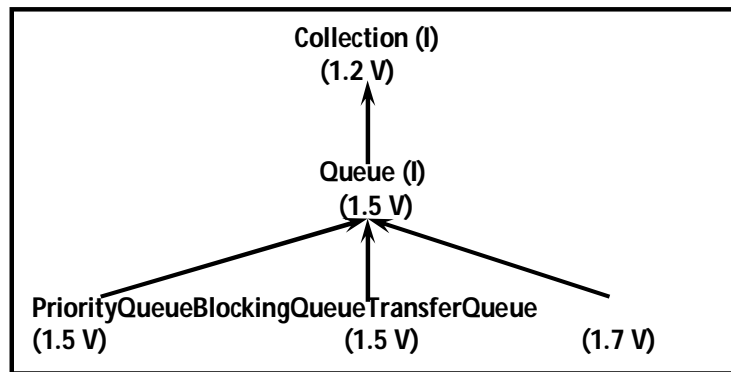
Methods:

EnumMap doesn't contain any New Methods. We have to Use General Map Methods Only.

```
import java.util.*;
enum Priority {
    LOW, MEDIUM, HIGH
}
class EnumMapDemo {
    public static void main(String[] args) {
        EnumMap<Priority, String> m = new EnumMap<Priority, String> (Priority.class);
        m.put(Priority.LOW, "24 Hours Response Time");
        m.put(Priority.MEDIUM, "3 Hours Response Time");
        m.put(Priority.HIGH, "1 Hour Response Time");
        System.out.println(m);
        Set s = m.keySet();
        Iterator<Priority> itr = s.iterator();
        while(itr.hasNext()) {
            Priority p = itr.next();
            System.out.println(p+"....."+m.get(p));
        }
    }
}
```

```
{LOW=24 Hours Response Time, MEDIUM=3 Hours Response Time, HIGH=1 Hour Response Time}
LOW.....24 Hours Response Time
MEDIUM.....3 Hours Response Time
HIGH.....1 Hour Response Time
```

Overview of java Queues



Queue:

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

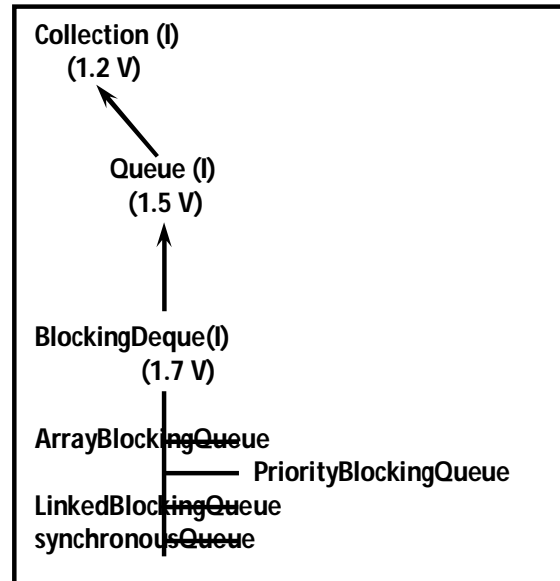
- Queue is Child Interface of Collection.

PriorityQueue:

- It is the Implementation Class of Queue.
- If we want to Represent a Group of Individual Objects Prior to processing according to Priority then we should go for PriorityQueue.

BlockingQueue:

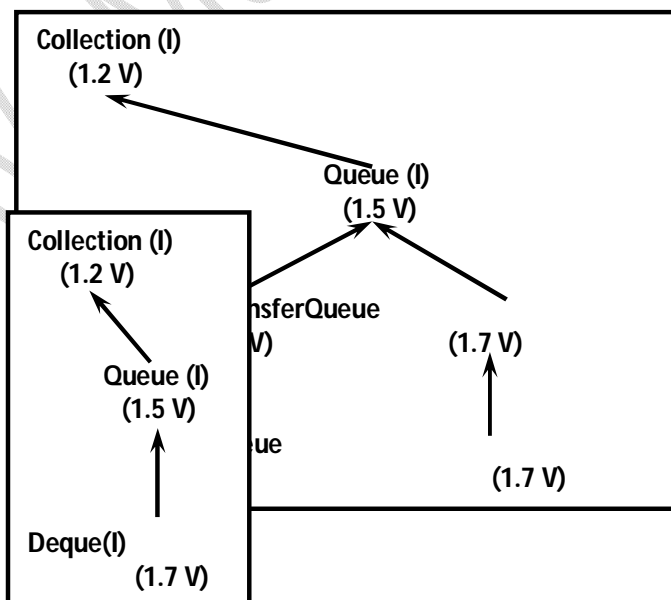
- It is the Child Interface of Queue. Present in java.util.Concurrent Package.
- It is a Thread Safe Collection.
- It is a specially designed Collection Not Only to Store Elements but also Supports Flow Control by Blocking Mechanism.
- If Queue is Empty take() (Retrieval Operation) will be Blocked until Queue will be Updated with Items.
- put() will be blocked if Queue is Full until Space Availability.
- This Property Makes BlockingQueue Best Choice for Producer Consumer Problem. When One Thread producing Items to the Queue and the Other Thread consuming Items from the Queue.



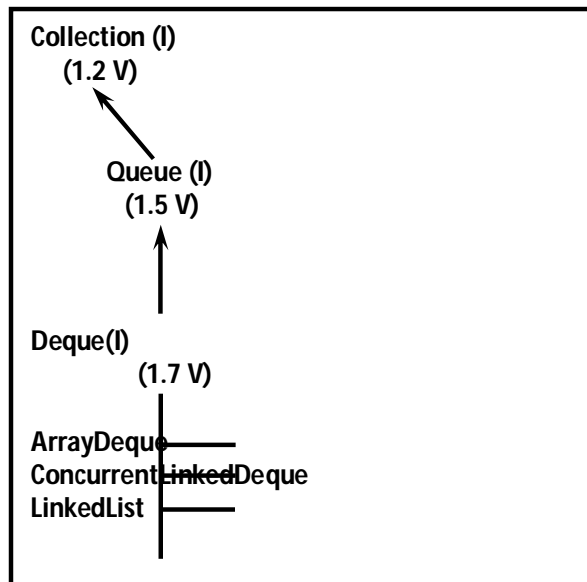
TransferQueue:

- In `BlockingQueue` we can Only Put Elements into the Queue and if Queue is Full then Our `put()` will be blocked until Space is Available.
- But in `TransferQueue` we can also Block until Other Thread receiving Our Element. Hence this is the Behavior of `transfer()`.
- In `BlockingQueue` we are Not required to wait until Other Threads Receive Our Element but in `TransferQueue` we have to wait until Some Other Thread Receive Our Element.
- `TransferQueue` is the Best Choice for Message Passing Application where Guarantee for the Delivery.

Deque (I)



- It Represents a Queue where we can Insert and Remove Elements from Deque, Both Ends of Queue i.e. Deque Means Double Ended Queue.
- It is Also pronounced as Deck Like Deck of Cards.



BlockingDeque (I) 1.6 V

- It is the Child Interface of BlockingQueue and Deque.
- It is a Simple Deque with Blocking Operations but wait for the Deque to become Non Empty for Retrieval Operation and wait for Space to Store Element.

