13. Collection Framework

→ An Array is an indexed collection of fixed no. of homogeneous data elements.

→ The main advantage of arrays is 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 i.e., once we created an array there is no chance of increasing/ decreasing the size based on our requirement.

→ Hence to use arrays concept compulsory we should know the size in advance, which may not possible always.

2. Arrays can hold only homogeneous data type elements.

Ex: Student[] s = new Student[10000];

s[0] = new Student(); DEMO

s[1] = new Customer(); → CE: incompatible types
found: Customer
required: Student

→ We can resolve this problem by using Object type arrays.

Ex: Object[] a = new Object[10000];

a[0] = new Student(); ✓

a[1] = new Customer(); ✓

3. There is no underlying data structure for arrays. Hence readymade data support we can't expect.

For every requirement we have to write code explicitly, which increases complexity of the programming.

→ To overcome the above problems of arrays we should go for Collections.

1. Collections are growable in nature i.e., based on our requirement we can increase / decrease the size.
2. Collections can hold both homogeneous & heterogeneous elements.
3. Every Collection class is implemented based some standard data structure. Hence for every requirement readymade data support is available.

Being a programmer we have to use those methods & we aren't responsible to implement.

*** Differences blw Arrays and Collections :—

Arrays	Collections
1. Arrays are fixed in size.	1. Collections are growable in nature.
2. W.r.t memory arrays are not recommended to use.	2. W.r.t memory Collections are recommended to use.
3. W.r.t performance arrays are recommended to use.	3. W.r.t performance Collections are not recommended to use.
4. Arrays can hold only homogeneous data elements.	4. Collections can hold both homogeneous and heterogeneous elements.
5. Arrays can hold both primitives and objects.	5. Collections can hold only objects not primitives.
6. There is no underlying data structure for arrays. Hence we can't expect readymade data support in arrays.	6. For every Collection class underlying data structure is available. Hence we can expect readymade data support for every requirement.

Collection :—

→ If we want to represent a group of objects as a single entity then we should go for Collection.

Collection Framework :—

→ It defines several classes & interfaces which can be used to represent a group of objects as a single entity.

Java	C++
Collection	Container
Collection.FW	STL (Standard Template Library)

*** 9 key interfaces of Collection Framework :—

1. Collection
2. List
3. Set
4. SortedSet
5. NavigableSet
6. Queue
7. Map
8. SortedMap
9. NavigableMap.

DEMO

1. Collection :—

→ If we want to represent a group of individual objects as a single entity then we should go for Collection.

→ In general Collection interface is considered as root interface of Collection Framework.

→ Collection interface defines the most general methods which are applicable for any Collection object.

*** Difference blw Collection and Collections :—

Collection is an interface which can be used to represent a group of individual objects as a single entity.

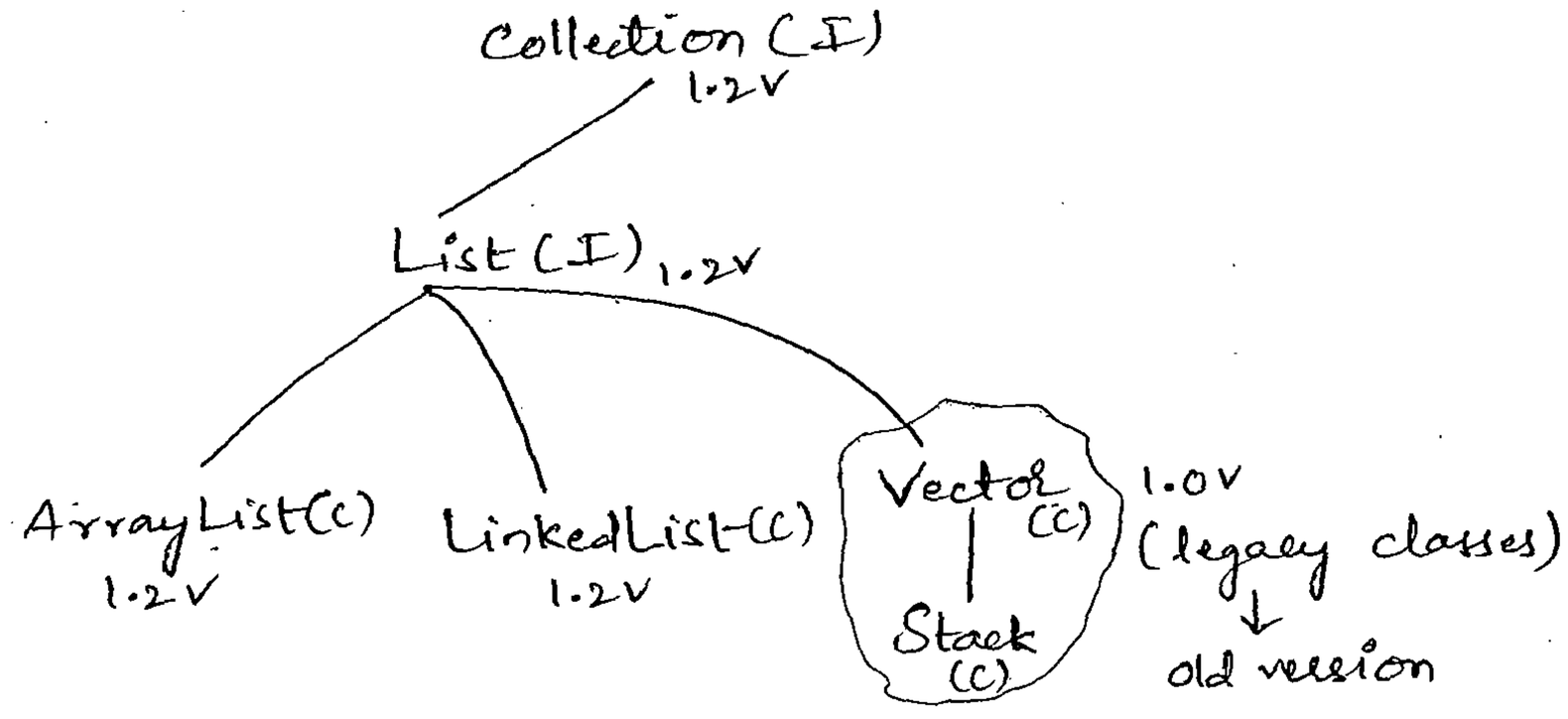
Collections is an utility class present in java.util package to define several utility methods for Collection objects.

*** Note:— There is no concrete class which implements Collection interface directly.

2. List(I) :—

→ It is the child interface of Collection.

→ If we want to represent a group of individual objects as a single entity where duplicates are allowed and insertion order preserved then we should go for List.

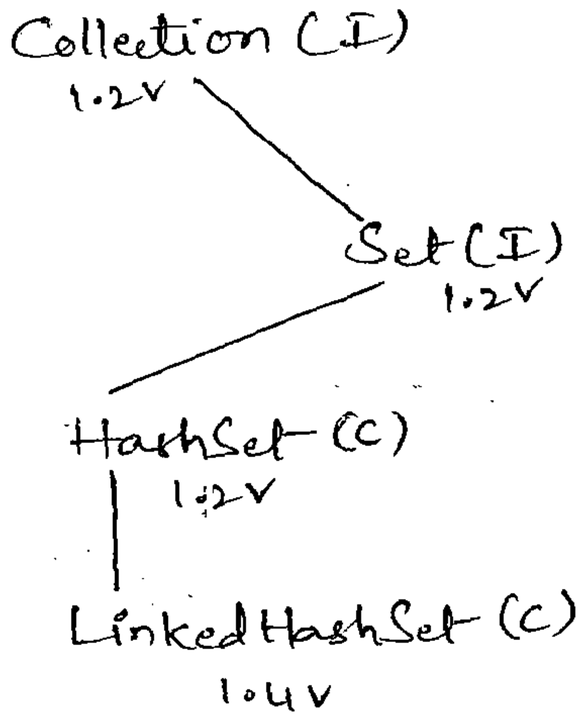


Note:— In 1.2 version, Vector & Stack classes are re-engineered to implement List interface.

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 & insertion order won't be preserved then we should go for Set.



4. SortedSet(I):—

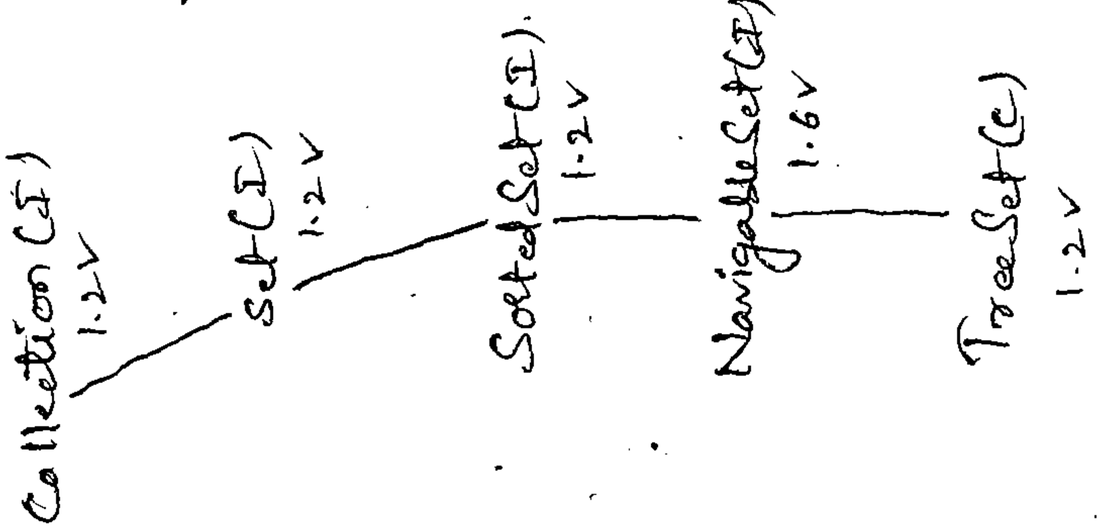
→ It is the child interface of Set.

→ 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.

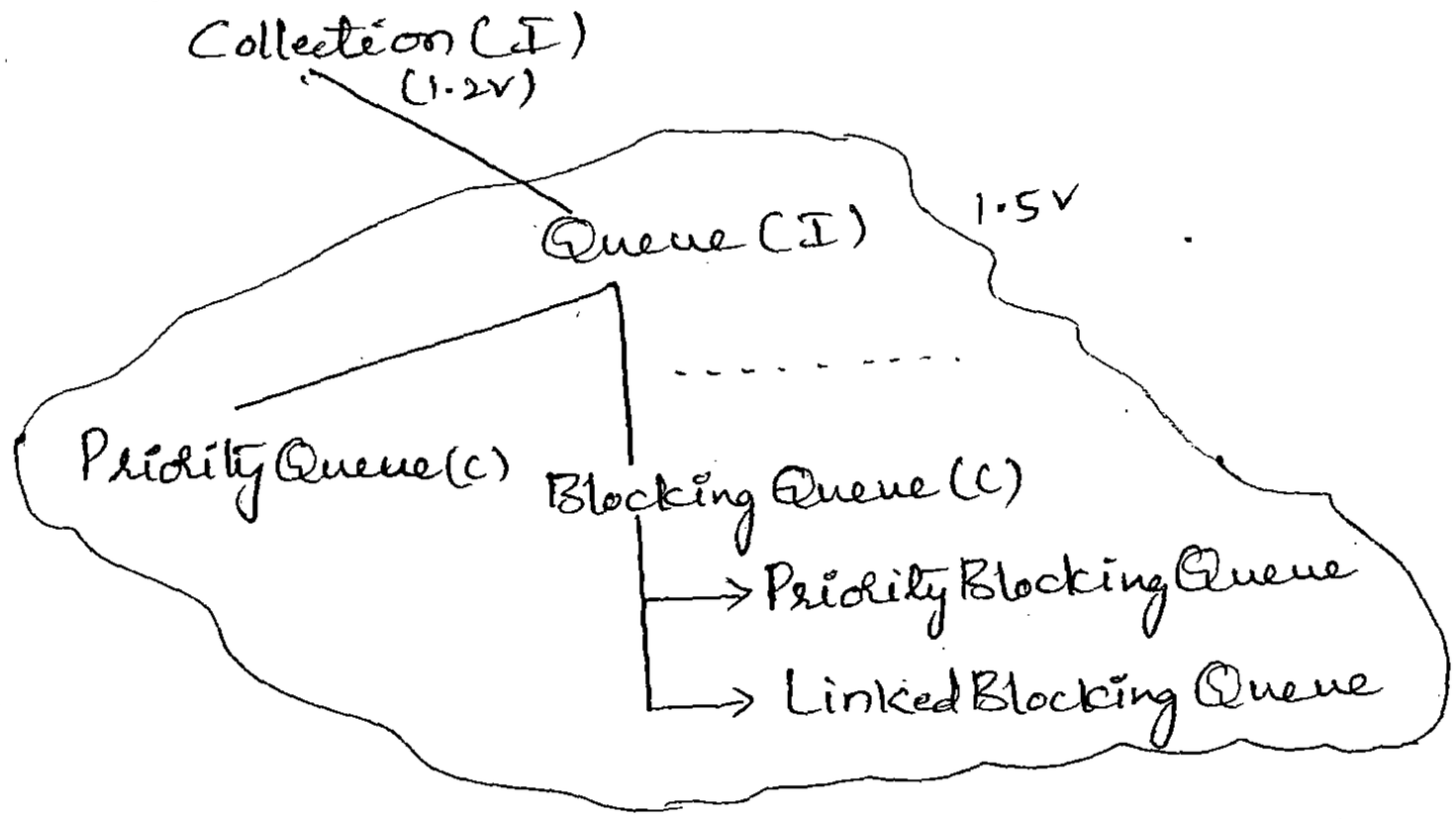
→ It defines several methods for navigation purposes.



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.



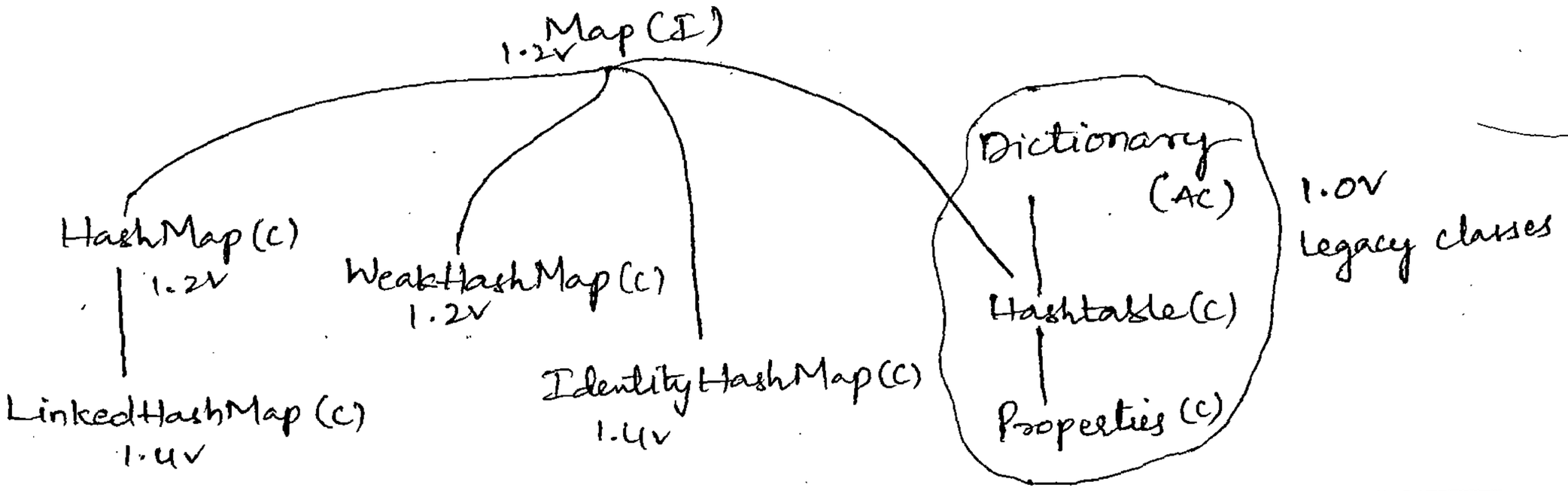
***

Note!- All above interfaces (Collection, List, Set, SortedSet, NavigableSet and Queue) meant for representing a group of individual objects and we can't use for representing key-value pairs.

7. Map (I):—

→ If we want to represent a group of objects as key-value pairs then we should go for Map.

→ It is not child Interface of Collection.



8. SortedMap(I):—

→ It is the child interface of Map.

→ If we want to represent a group of key-value pairs according to some sorting order of keys then we should go for SortedMap.

9. NavigableMap(I):—

→ It is the child interface of SortedMap.

→ It defines several methods for navigation purposes.

Map(I)
1.2V

SortedMap(I)
1.2V

NavigableMap(I)
1.6V

DEMO

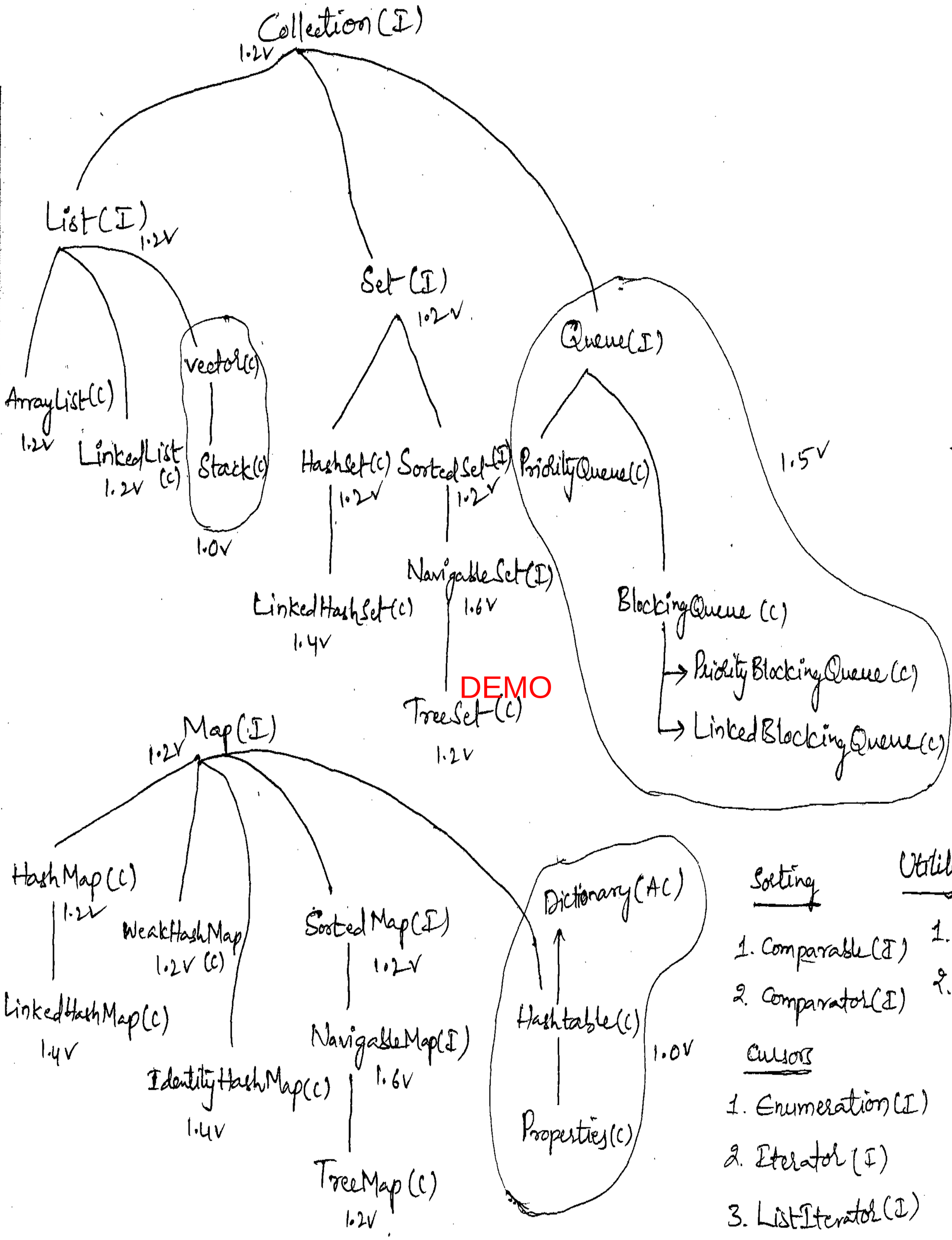
TreeMap(C)
1.2V

*** Note:— In Collection Framework, the following are legacy Characters

1. Enumeration(I)
2. Dictionary(AC)
3. Vector
4. Stack
5. Hashtable
6. Properties

} concrete classes

PLOT NO:56, HARSHA CHAMBERS, SAI NAGAR,NEAR BIG C MOBILES, MADHAPUR-500034, MOBILE:9505905786



Sorting

1. Comparable(I)
2. Comparator(I)

Utility classes

1. Collections
2. Arrays

Cursors

1. Enumeration(I)
2. Iterator(I)
3. ListIterator(I)

1. Collection:—

→ If we want to represent a group of individual objects as a single entity then we should go for Collection interface.

→ Collection interface defines the most common methods which are applicable for any Collection object.

→ The following is the list of 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) // To remove
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.

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 & differentiate duplicate objects by using index. Hence index will play very important role in List.

→ 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 first occurence of 'o'
7. int lastIndexOf(Object o)
8. ListIterator listIterator();

a. ArrayList :—

1. The underlying data structure for ArrayList is Resizable or Growable Array.
2. Duplicate objects are allowed.
3. Insertion order will be preserved.
4. Heterogeneous objects are allowed. (Except TreeSet & TreeMap, everywhere heterogeneous objects are allowed).
5. Null insertion is possible.

Constructors :—

① `ArrayList l = new ArrayList();`

→ creates an empty ArrayList object with default initial capacity 10.

→ If ArrayList reaches max. capacity a new ArrayList object will be created with

$$\text{new capacity} = (\text{current capacity} * \tfrac{3}{2}) + 1$$

② `ArrayList l=new ArrayList(int initialcapacity);`

→ Creates an empty ArrayList object with specified initial capacity.

③. `ArrayList l=new ArrayList(Collection c);` (equivalent)

→ Creates an ~~empty~~ ArrayList object for the given Collection object.

→ This constructor meant for inter-conversion b/w Collection objects.

Ex:

```
import java.util.*;
class ArrayListDemo
{
   p s v m(-)
   {
      ArrayList l=new ArrayList();
      l.add("A");
      l.add(10);
      l.add("A");
      l.add(null);
      S.o.p(l); ⇒ o/p : [A, 10, A, null]
      l.remove(2);
      S.o.p(l); ⇒ o/p: [A, 10, null]
      l.add(2, "M");
      l.add("N");
      S.o.p(l); ⇒ o/p: [A,10,M, null,N]
   }
}
```

DEMO

→ Usually we can use Collection to hold & transfer data (objects) from one location to another location.

→ To provide support for this requirement every Collection class implements Serializable & Cloneable interfaces.

*** → ArrayList & Vector classes implements RandomAccess interface. So that we can any random element with the same Speed.

*** → Hence ArrayList is best suitable if our frequent operation is retrieval operation.

→ RandomAccess interface present in java.util package and docsn't contain any methods. Hence it is Marker interface.

Ex: ArrayList l1=new ArrayList();
LinkedList l2=new LinkedList();
S.o.p (l1 instanceof Serializable); ⇒ o/p : true
S.o.p (l2 instanceof Cloneable); ⇒ o/p : true
S.o.p (l2 instanceof RandomAccess); ⇒ o/p : false
S.o.p (l1 instanceof RandomAccess); ⇒ o/p : true.

DEMO

*** Differences blw ArrayList and Vector :-

ArrayList	Vector
1. No method present inside ArrayList is Synchronized.	1. Every method present inside Vector is Synchronized.
2. At a time multiple threads are allowed to operate on ArrayList object simultaneously. Hence AL object is not Thread Safe.	2. At a time only one thread is allowed to operate on Vector object. Hence Vector object is always Thread Safe.
3. Relatively performance is high	3. Realively performance is low.

Q: How to get synchronized version of ArrayList object?

Ans:- By default ArrayList object is non-synchronized. But we can get synchronized version of ArrayList object by using the following method of Collections class.

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

Ex: ArrayList l = new ArrayList();

List l1 = Collections.synchronizedList(l);

l1 → Synchronized version

l → non-synchronized version.

→ Ily we can get synchronized version of Set and Map objects by using the following methods.

```
public static Set synchronizedSet(Set s)
public static Map synchronizedMap(Map m)
```

DEMO

→ ArrayList is the best choice if we want to perform retrieval operation frequently.

*** → ArrayList is the worst choice if our frequent operation is insertion or deletion in the middle beeoz it is required several shift operations internally.

b. LinkedList :-

1. The underlying data structure is double LinkedList.
2. Insertion order is preserved.
3. Duplicate objects are allowed.
4. Heterogeneous objects are allowed.

5. Null insertion is possible.
6. Implement Serializable and Cloneable interfaces, but not RandomAccess.
7. Best choice if our frequent operation is insertion or deletion in the middle.
8. 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.

→ 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()

Ex:-
```
import java.util.*;
class LinkedListDemo
{
    p s v m(-)
    {
```

```
LinkedList l=new LinkedList();
l.add("durga");
l.add(30);
l.add(null);
l.add("durga");
l.set(0,"Software");
l.add(0,"venkey");
l.removeLast();
l.addFirst("ccc");
S.O.p(l); ⇒ O/P : [ccc, venky, software, 30, null]
}
}
```

ccc, venkey, ~~durga~~ software, 30, null, ~~durga~~

c. Vector :—

1. The underlying data structure is Resizable array or Growable array.
2. Insertion order is preserved. DEMO
3. Duplicate objects are allowed.
4. Heterogeneous objects are allowed.
5. Null insertion is possible.
6. Implements Serializable, Cloneable & RandomAccess interfaces.
7. Every method present inside vector is synchronized and hence vector object is Thread safe.

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

new capacity = 2 * current capacity

⑨. Vector v=new Vector(int initialcapacity);

***③. Vector v=new Vector(int initialcapacity, int incremental capacity);

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

DEMO

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 retrieve elements from the Vector.

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 elements()
```

Ex:

```
import java.util.*;
class VectorDemo
{
    P S V m(-)
    {
      Vector v=new Vector();
       S.o.p(v.capacity()); ⇒ o/p : 10
       for(int i=1; i<=10; i++)
       {
          v.addElement(i);
       }
       S.o.p (v.capacity()); ⇒ o/p : 10
       v.addElement("A");
       S.o.p(v.capacity()); ⇒ o/p : 20
       S.o.p(v); ⇒ o/p : [1, 2, 3, ..... , 10, A]
    }
}
```

DEMO

d. Stack :—

→ It is the child class of Vector.

→ It is a specially designed class for Last In First Out (LIFO) order.

Constructor :—

① . `Stack s=new Stack();`

Methods :—

1. Object push(Object o)
   to insert an object into the stack.

2. Object pop()

   to remove & return top of the stack.

3. Object peek()

   to returns 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 o.w

   returns -1.

Ex:

```
import java.util.*;
class StackDemo
{
    p s v m(-)
    {
        Stack s=new Stack();
        S.push("A");
        S.push("B");
        S.push("C");
        S.o.p(s); ⇒ O/P : [A, B, C]
        S.o.p(s.search("A")); ⇒ O/P: 3
        S.o.p(s.search("Z")); ⇒ O/P: -1.
    }
}
```

DEMO

The 3 cursors of Java:—

→ We can use cursor to get objects one by one from the Collection.

→ There are 3 types of cursors available in Java.

1. Enumeration (I)
2. Iterator (I)
3. ListIterator (I)

1. Enumeration :—

→ We can use Enumeration object to get objects one by one from the Collection.

→ We can create Enumeration object by using elements() method.

```
public Enumeration elements()
```

Ex: `Enumeration e = v.elements();`
(v: Vector object)

Methods :—

```
1. public boolean hasMoreElements()
2. public Object nextElement()
```

DEMO

Ex:
```
import java.util.*;
class EnumerationDemo
{
  P S V m(-)
  {
    Vector v=new Vector();
    for(int i=0; i<=10; i++)
    {
      v.addElement(i);
    }
    S.o.p(v); ⇒ o/p : [0,1,2,------10]
    Enumeration e=v.elements();
```

```
while(e.hasMoreElements())
{
  Integer i = (Integer)e.nextElement();
  if(i%2==0)
      S.O.p(i); ⇒ o/p: 0 2 4 6 8 10
 }
 S.O.p(v); ⇒ o/p: [0, 1, 2, ------10]
 }
}
```

Limitations of Enumeration :—

1. Enumeration concept is applicable only for legacy classes and it is not a universal cursor.
2. By using Enumeration we can perform only read operation and we can't perform remove operation.

→ To overcome the above limitations of Enumeration we should go for Iterator. DEMO

2. Iterator :—

1. We can use Iterator to get objects one by one from Collection.
2. We can apply Iterator concept for any Collection object. Hence it is universal cursor.
3. By using Iterator we can able to perform both read and remove operations.

→ We can create Iterator object by using iterator() method of Collection interface.

```
public Iterator iterator();
```

Ex: `Iterator itr = c.iterator();`
(c: any Collection object.)

Methods :—

1. public boolean hasNext();
2. public Object next();
3. public void remove();

Ex:
```
import java.util.*;
class IteratorDemo
{
  p s v m(-)
  {
    ArrayList l=new ArrayList();
    for(int i=0; i<=10; i++)
    {
      l.add(i);
    }
    S.o.p(l); olp : [0,1, 2,- - - - -10]
    Iterator itr=l.iterator();
    while(itr.hasNext())
    {
      Integer I=(Integer) itr.next();
      if(I%2==0)
          S.o.p(I); ⇒ olp : 0,2,4, 6, 8, 10
      else
          itr.remove();
    }
    S.o.p(l); ⇒ olp : [0, 2, 4, 6, 8, 10]
  }
}
```

DEMO

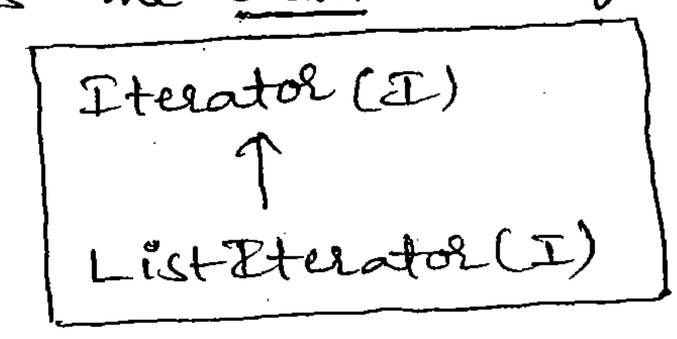
Limitations of Iterator :—

1. By using Enumeration & Iterator we can always move only towards forward direction i.e., these are 1-direction cursors (forward direction).
2. By using Iterator we can perform only read & remove operations and we can't perform replacement and addition of new objects.

→ To overcome these limitations we should go for ListIterator.

3. ListIterator :—

1. By using ListIterator we can move either to the forward direction or backward direction i.e., it is a bi-directional cursor.
2. By using ListIterator we can able to perform replacement and addition of new objects in addition to read & remove operations.
3. ListIterator is the child interface of Iterator.



→ We can create ListIterator object by using listIterator() method of List interface.

```
public ListIterator listIterator();
```

Ex: `ListIterator ltr = l.listIterator();`
↓
Any List object

→ ListIterator interface contains the following 9 methods.

1. public boolean hasNext()
2. public Object next()
3. public int nextIndex()

} Forward operation

4. public boolean hasPrevious()
5. public Object previous()
6. public int previousIndex()

} Backward operation

7. public void remove()
8. public void set(Object new)
9. public void add(Object new)

Ex:

```
import java.util.*;
class ListIteratorDemo
{
    p s v m(-)
    {
      LinkedList l=new LinkedListc);
      l.add("balakrishna");
      l.add("venki");
      l.add("chiru");
      l.add("nag");
      S.o.p(l); ⇒ O|P : [balakrishna, venki, chiru, nag]
      ListIterator ltr=l.listIterator()
      while(ltr.hasNext())
      {
        String s=(String)ltr.next();
        if(s.equals("venki"))
        {
          ltr.remove();
        }
        if(s.equals("chiru"))
        {
          ltr.set("charan");
        }
        if(s.equals("nag"))
        {
          ltr.add("chaitu");
        }
      }
```

DEMO

```
        S.o.p(l); ⇒ o/p : [balakrishna, chiru, nag]
    }
}
```

→ The most powerful cursor is ListIterator. But its limitation is applicable only for List objects.

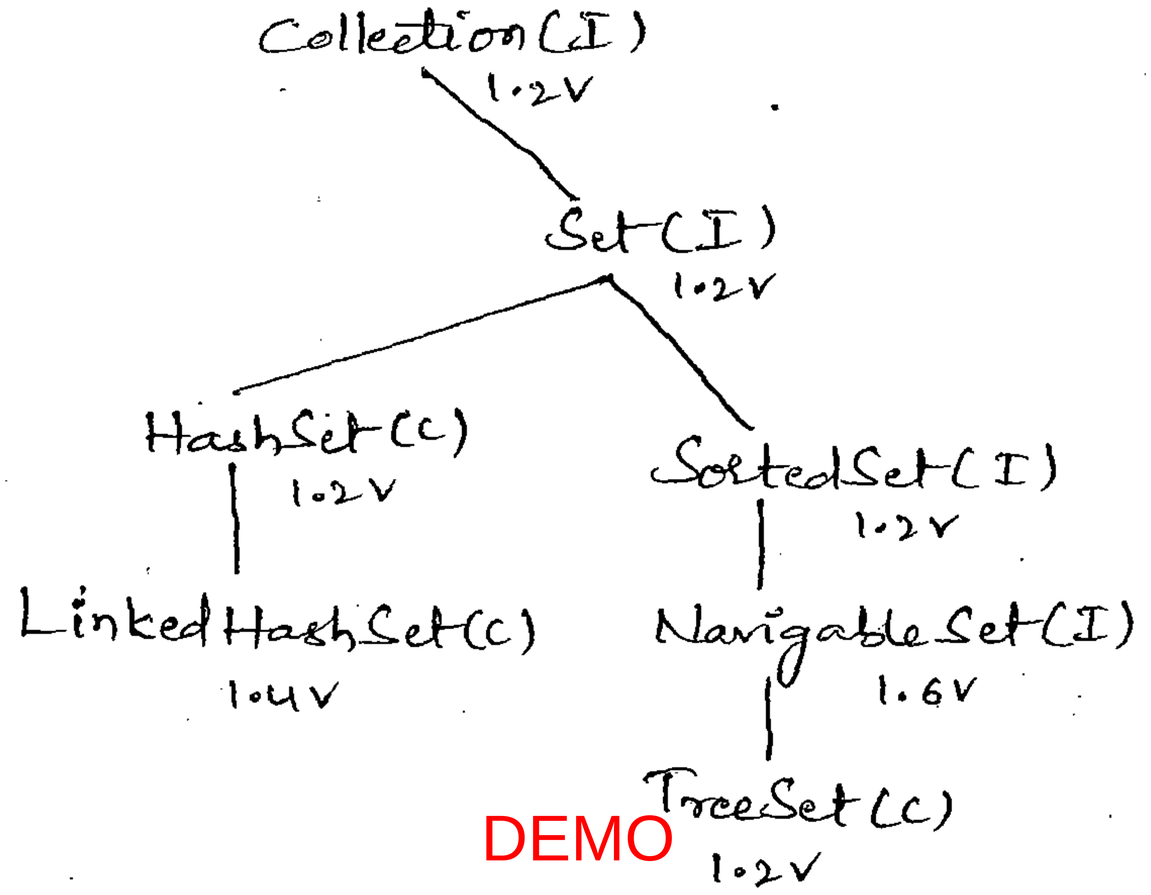
** Comparison of Enumeration, Iterator and ListIterator :—

Property	Enumeration	Iterator	ListIterator
1. Applicable	Only for Legacy classes	for any Collection object	Only for List objects
2. Movement	Single direction (Forward)	Single direction (Forward)	Bi-directional (Forward & Backward)
3. Getting	By using elements() method	By using iterator() method	By using listIterator() method
4. Access permissions	only read	Read and Remove	Read / Remove / Replace / add
5. Methods	2 methods hasMoreElements() nextElement()	3 methods hasNext() next() remove()	9 methods
6. Is it Legacy?	Yes	No	No

3. Set :—

→ It is the child interface of Collection.

→ If we want to represent a group of individual objects where duplicates are not allowed and insertion order is not preserved then we should go for Set.



DEMO

→ Set interface doesn't contain any new methods and hence we have to use only Collection interface methods.

a. HashSet :—

1. The underlying data structure is Hash table.
2. Insertion order is not preserved & it is based on hashcode of objects.
3. Duplicate objects are not allowed. and if we are trying to insert duplicate objects then we won't any CE and RE, simply add() method returns false.
4. Null insertion is possible.
5. Heterogeneous objects are allowed.
6. HashSet implements Serializable & Cloneable interfaces, but

not RandomAccess interface.

7. If our frequent operation is search operation then HashSet is the best choice.

Constructors :—

① `HashSet h=new HashSet();`

creates an empty HashSet object with default initial capacity 16 & default fill ratio/ load factor 0.75.

② `HashSet h=new HashSet(int initialCapacity);`

creates an empty HashSet object with specified initial capacity and default fill ratio 0.75.

③ HashSet h=new HashSet(int initialCapacity, float fillRatio);

④ HashSet h=new HashSet(Collection c);

DEMO

Load Factor :—

→ After loading this much factor a new HashSet object will be created, this factor is called Load Factor / Fill Ratio.

Ex:
```
import java.util.*;
class HashSetDemo
{
    P s v m(-)
    {
        HashSet h=new HashSet();
        h.add("B");
        h.add("C");
        h.add("D");
        h.add("Z");
        h.add(null);
        h.add(10);
```

```
        S.o.p(h.add("Z")); ⇒ o/p: false
        S.o.p(h); ⇒ o/p: [null, D, B, C, 10, Z]
    }
}
```

b. LinkedHashSet :—

→ It is the child class of HashSet.

→ It is exactly same as HashSet except the following differences.

***

HashSet	LinkedHashSet
1. The underlying data structure is Hashtable.	1. The underlying data structure is a combination of LinkedList & Hashtable.
2. Insertion order won't be preserved.	2. Insertion order will be preserved.
3. Introduced in 1.2 version	3. Introduced in 1.4 version.

→ In the above example, if we replace HashSet with LinkedHashSet then o/p is [B, C, D, Z, null, 10] i.e., insertion order is preserved.

Note :— Very common application area of LinkedHashSet is developing Cache based applications, where duplicates are not allowed and insertion order must be preserved.

4. SortedSet :—

→ 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.

→ SortedSet defines the following the specific methods.

1. Object first();

   returns first element of the SortedSet.

2. Object last();

   returns last element of the SortedSet.

3. SortedSet headSet(Object obj)

   returns the SortedSet whose elements are less than obj.

4. SortedSet tailSet(Object obj)

   returns SortedSet whose elements are >= obj.

5. SortedSet subSet(Object obj1, Object obj2)

   returns SortedSet whose elements are >= obj1 and < obj2.

6. Comparator comparator()

   returns Comparator object describes underlying sorting technique. If we are using default natural sorting order then we will get null.

***

Note:- For numbers, default natural sorting order is Ascending order where as for String objects it is Alphabetical order.

ex:

100
101
102
104
106
107
108
110

first() ⟶ 100

last() ⟶ 110

headSet(104) ⟶ [100, 101, 102]

tailSet(104) ⟶ [104, 106, 107, 108, 110]

subSet(102, 107) ⟶ [102, 104, 106]

comparator() ⟶ null.

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 then we will get runtime exception saying ClassCastException.

→ Duplicate objects are not allowed.

→ Null insertion is possible (only once).

→ Implements Serializable & Cloneable interfaces but not RandomAccess.

Constructors :—

① `TreeSet t=new TreeSet();` DEMO

Creates an empty TreeSet object, where the elements will be inserted according to default natural sorting order.

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

Creates an empty Treeset object, where elements will be inserted according to customised sorting order which is described by Comparator object.

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

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

Ex: ①
```
import java.util.*;
class TreeSetDemo
{
  p s v m(_)
  {
```

```
TreeSet t=new TreeSet();
t.add("A");
t.add("a");
t.add("B");
t.add("Z");
t.add("L");
//t.add(new Integer(10)); → RE: ClassCastException
//t.add(null); → RE: NullPointerException
S.o.p(t); ⇒ o/p: [A, B, L, Z, a]
}
}
```

Null Acceptance :—

→ For empty TreeSet as the first element null insertion is possible. But after inserting that null if we are trying to insert any other element then we will get NullPointerException.

→ For non-empty TreeSet if we are trying to insert null then we will get NullPointerException.

Ex②:

```
import java.util.*;
class TreeSetDemo1
{
psvm(-)
{
TreeSet t=new TreeSet();
t.add(new StringBuffer("A"));
t.add(new StringBuffer("Z"));
t.add(new StringBuffer("L"));
t.add(new StringBuffer("B"));
S.o.p(t); → RE: ClassCastException.
}
}
```

→ If we are depending on default natural sorting order compulsory object should be homogeneous & Comparable O.W. we will get runtime exception saying ClassCastException.

→ An object is said to be Comparable iff corresponding class implements Comparable interface.

→ All Wrapper classes and String class implement Comparable interface, but StringBuffer class doesn't implement Comparable interface.

→ Hence we are getting ClassCastException in the above example.

Comparable interface :-

→ Comparable interface present in java.lang package and it contains only one method i.e., DEMO

```
public int compareTo(Object obj);
```

obj1.compareTo(obj2);

- → returns -ve, iff obj1 has to come before obj2
- → returns +ve, iff obj1 has to come after obj2
- → returns 0, iff obj1 & obj2 are equal.

Ex: S.o.p("A".compareTo("Z")); ⇒ O/P : -ve value

S.o.p("K".compareTo("A")); ⇒ O/P : +ve value

S.o.p("A".compareTo("A")); ⇒ O/P : 0

S.o.p("A".compareTo(null)); → RE : NullPointerException.

→ If we are depending on default natural sorting order internally Jvm will compareTo() method to place objects in proper sorting order.

Ex: TreeSet t=new TreeSet();

t.add("A"); → (A)

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

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

t.add("A"); 0 → "A".compareTo("A"); → (A) (B) (Z)

S.o.p(t); ⇒ o/p : [A, B, 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 sorting order using Comparator.

***
→ Comparable meant for Default Natural Sorting order where as Comparator meant for Customized sorting order.

Comparator(I):—

DEMO

→ This interface present in java.util package.

→ It contains 2 methods.

① public int compare(Object obj1, Object obj2)

→ returns +ve, iff obj1 has to come after obj2
→ returns -ve, iff obj1 has to come before obj2
→ returns 0, iff obj1 & obj2 are equal.

② public boolean equals(Object obj)

→ Whenever we are implementing Comparator interface we have to provide implementation only for compare() method and implementing equals() method is optional becoz it is already available for every 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 TreeSetDemo3
{
   p s v m(-)
   {
      TreeSet t=new TreeSet(new MyComparator()); ⟶ ①

       t.add(10);
      t.add(0); +1⟶ compare(0,10);
      t.add(15); -1⟶ compare(15,10);
      t.add(5); +1⟶ compare(5,15);
                +1⟶ compare(5,10);
                -1⟶ compare(5,0);
      t.add(20); -1⟶ compare(20,15);
      t.add(20); 0⟶ compare(20,20);

      S.o.p(t); ⇒ o/p : [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;
    }
}
```

DEMO

→ At line ①, if we are not passing Comparator object then JVM will call compareTo(—) method, which is meant for Default natural sorting order (Ascending Order). In this case, the o/p is [0, 5, 10, 15, 20].

→ At line ①, if we are passing Comparator object then JVM will call compare(—,—) method instead of compareTo(—) method. In this case, the o/p is [20, 15, 10, 5, 0].

Various possible implementations of compare(—,—) method :—

```
class MyComparator implements Comparator
{
   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] Descending order
         return -I2.compareTo(I1); ⟶ [0, 5, 10, 15, 20] Ascending order
         return +1; ⟶ [10, 0, 15, 5, 20, 20] Insertion order
         return -1; ⟶ [20, 20, 5, 15, 0, 10] Reverse of insertion order
         return 0; ⟶ [0] only first inserted element present and all remaining elements are treated as duplicates.
   }
}
```

DEMO

// Ex②: Write a program to insert String objects into the TreeSet where sorting order is reverse of Alphabetical order.

```
import java.util.*;
class TreeSetDemo4
{
   p s v m(-)
   {
     TreeSet t=new TreeSet(new MyComparator());
     t.add("Roja");
     t.add("ShobhaRani");
     t.add("RajaKumari");
     t.add("GangaBhavani");
     t.add("Ramulamma");
     S.o.p(t); ⇒ o/p : [ShobhaRani, Roja, Ramulamma, RajaKumari,
                                                GangaBhavani]
   }
}
```

DEMO

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

//③Write a program to insert StringBuffer objects into the TreeSet where sorting order is Alphabetical order.

```
import java.util.*;
class TreeSetDemo5
{
```

```
p s v m(-)
{
  TreeSet t=new TreeSet(new MyComparator());
   t.add(new StringBuffer("A"));
   t.add(new StringBuffer("Z"));
   t.add(new StringBuffer("K"));
   t.add(new StringBuffer("L"));
   S.o.p(t); ⇒ o/p: [A, K, L, Z]
}
}
class MyComparator implements Comparator
{
  public int compare(Object obj1, Object obj2)
  {
      String s1 = obj1.toString();
      String s2=obj2.toString();
      return s1.compareTo(s2);
  }
}
```

DEMO

// ④ Write a program to insert String & StringBuffer objects into the TreeSet, where sorting order is increasing length order. If two objects having the same length then consider their Alphabetical order.

```
import java.util.*;
class TreeSetDemo6
{
  p s v m(-)
  {
    TreeSet t=new TreeSet(new MyComparator());
     t.add("A");
     t.add(new StringBuffer("ABC"));
     t.add(new StringBuffer("AA"));
```

```
        t.add("xx");
        t.add("ABCD");
        t.add("A");
        S.o.p(t); => o/p : [A, AA, xx, ABC, ABCD]
    }
}
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        int l1 = s1.length();
        int l2 = s2.length();
        if (l1 < l2)
            return -1;
        else if (l1 > l2)
            return +1;
        else
            return s1.compareTo(s2);
    }
}
```

DEMO

*** (Note:— If we are depending on default natural sorting order compulsory objects should be homogeneous & Comparable. o.w. we will get runtime Exception saying ClassCastException.

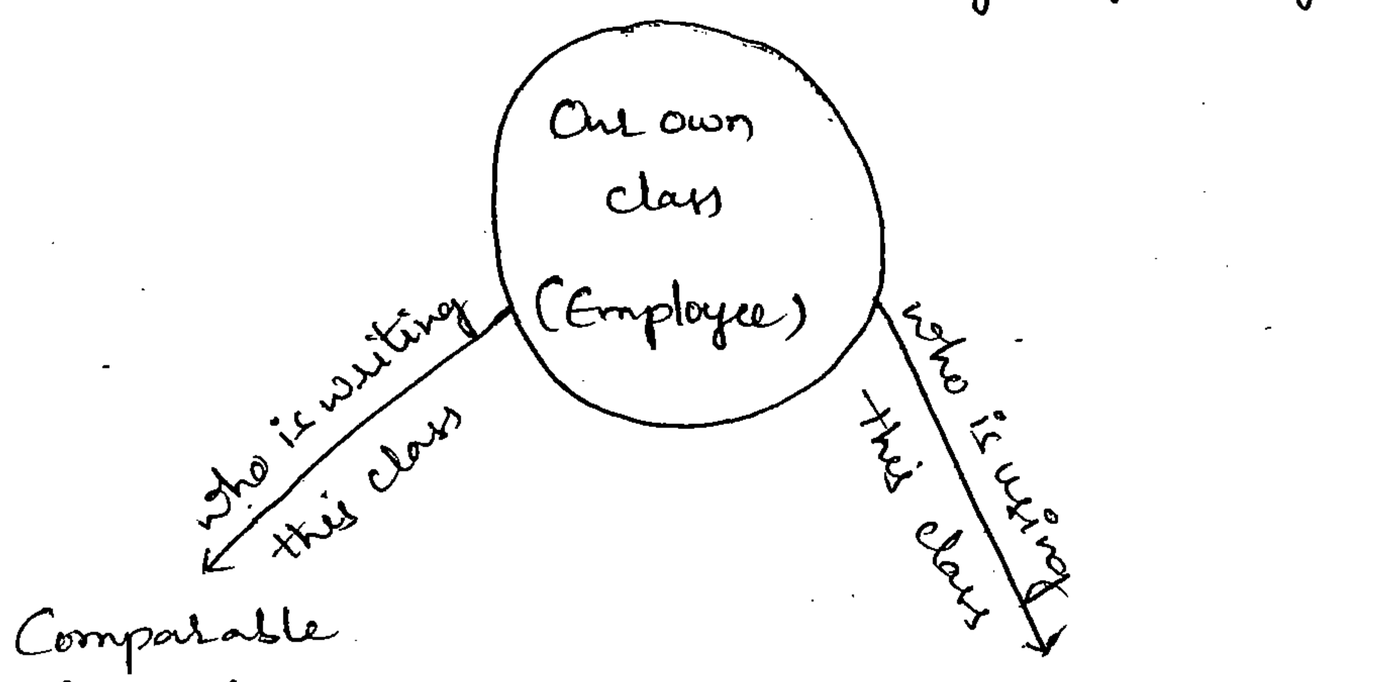
If we are defining our own sorting by Comparator then objects need not be homogeneous and Comparable i.e., we can add heterogeneous non-Comparable objects also.

Comparable Vs Comparator :—

1. 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.
2. For predefined non Comparable classes (like StringBuffer), Default natural sorting order is not already available. If we want to define our own sorting then we can use Comparator object.

→ In the case of our own classes, the person who is writing our class he can define default natural sorting order by implementing Comparable interface.

→ The person who is using our classes, if he is satisfied with default natural sorting order he can use directly our class. If he is not satisfied with DNSO then he can define our own sorting by using Comparator object.



Ex:

```
import java.util.*;
class Employee implements Comparable
{
    String name;
    int eid;
    Employee(String name, int eid)
    {
      this.name=name;
      this.eid=eid;
    }
public Sting 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
{
    p s v m(-)
    {
      Employee e1=new Employee("nag",100);
      Employee e2=new Employee("balaiah",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);
S.o.p(t); ⇒ o/p : [chiru--50, nag--100, venki--150, balaiah--200]
TreeSet t1=new TreeSet(new MyComparator());
t1.add(e1);
t1.add(e2);
t1.add(e3);
t1.add(e4);
t1.add(e5);
S.o.p(t1); ⇒o/p : [balaiah--200, chiru--50, nag--100, venki--150]
}
}
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);
}
}
```

DEMO

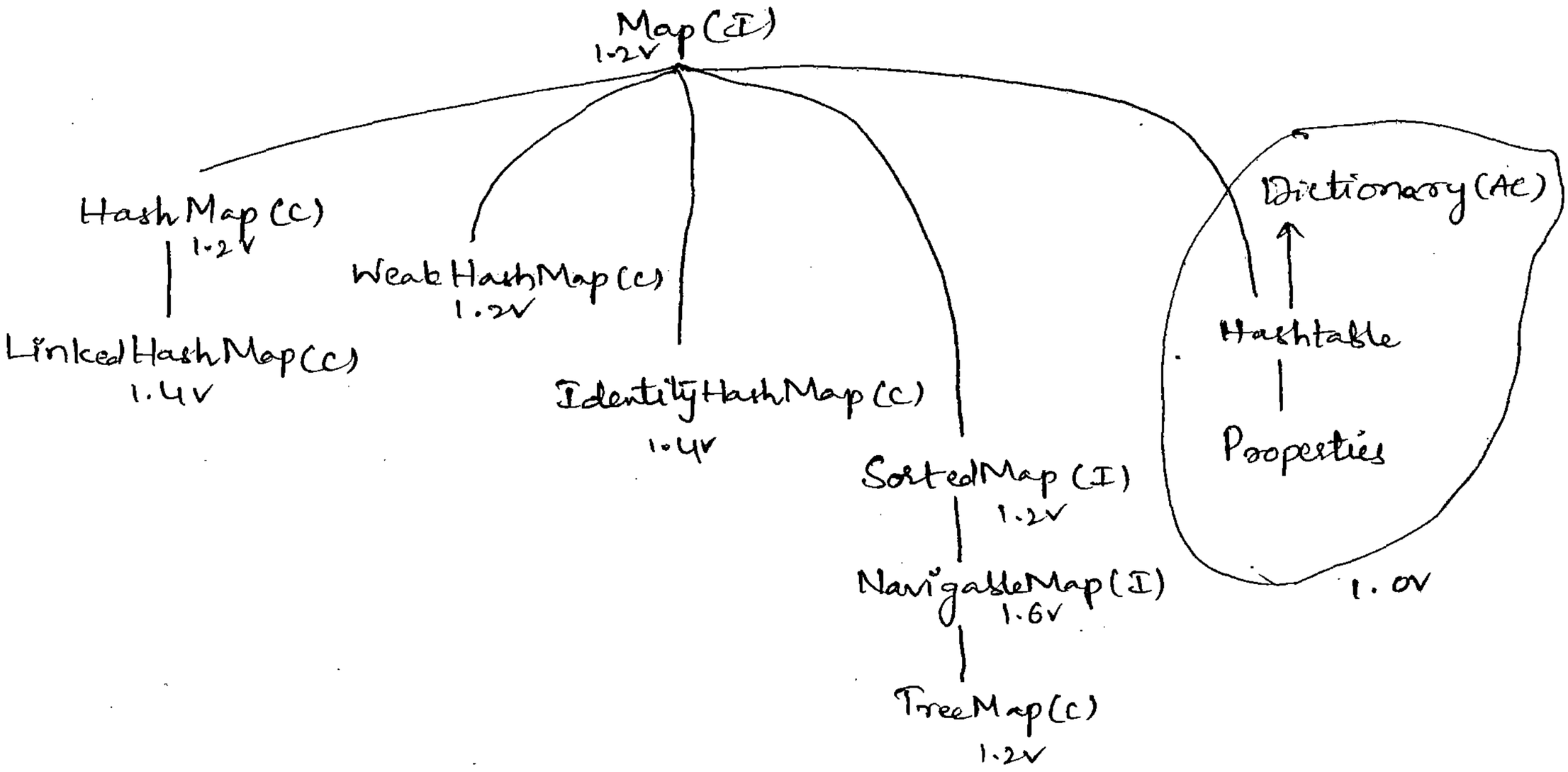
*** Comparison blw Comparable and Comparator :—

Comparable	Comparator
1. present in java.lang package.	1. present in java.util package.
2. It is meant for Default natural sorting order.	2. It is meant Customized sorting order.
3. Define only one method i.e., compareTo(—)	3. Define 2 methods 1. compare(—,—) 2. equals(—)
4. All wrapper classes and String classes implements Comparable interface.	4. The only implemented classes of Comparator are Collator and RuleBasedCollator.

*** Comparison blw HashSet, LinkedHashSet and TreeSet :—

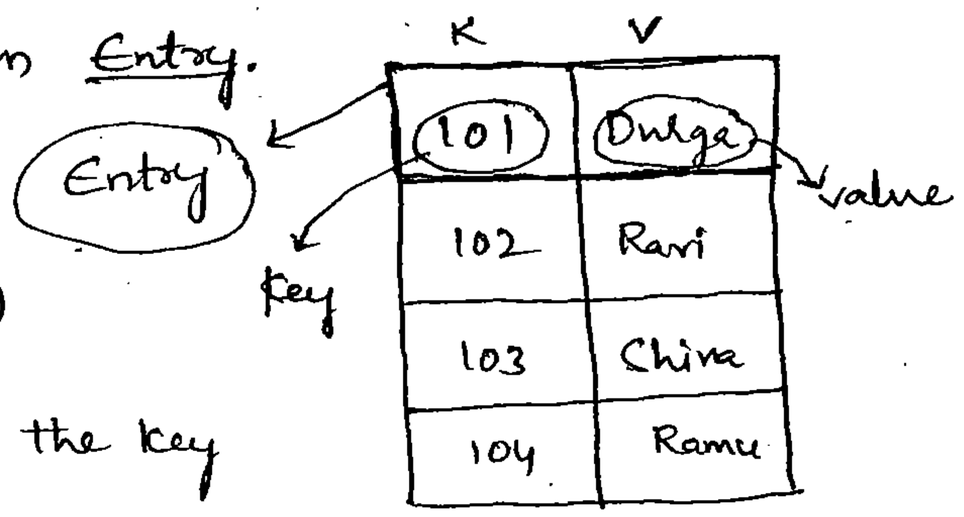
Property	HashSet	LinkedHashSet	TreeSet
1. Underlying data Structure	Hashtable	Hashtable+LinkedList	BalancedTree
2. Insertion order	not preserved	preserved	not preserved
3. Sorting order	Not Applicable	Not Applicable	Applicable
4. Heterogeneous objects	Allowed	Allowed	Not Allowed
5. Duplicate objects	Not Allowed	Not Allowed	Not Allowed
6. Null Acceptance	Allowed (Only once)	Allowed (Only once)	For empty TreeSet as the first element is possible, o.w. NullPointerException.

7. Map :—



1. Map is not child interface of Collection.
2. If we want to represent a group of objects as key-value pairs then we should go for Map.
3. Both keys and values are objects only.
4. Duplicate keys are not allowed. But values can be duplicated.
5. Each key-value pair is called an Entry.

DEMO



Map interface Methods:—

1. Object put (Object key, Object value)

   To add one key-value pair, if the 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. boolean containsKey(Object key)
6. boolean containsValue(Object value)
7. boolean isEmpty()
8. int size() → to return no. of entries (key-value pairs)
9. void clear()

1. Set keySet()
2. Collection values()
3. Set entrySet()

} Collection views of Map

Entry (I) :—

→ Each key-value pair is called an Entry.
→ Without existing Map object there is no chance of existing Entry object. Hence Entry interface is defined inside Map interface.

DEMO

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

} Inner interface

a. HashMap :—

1. The underlying data structure is Hashtable.
2. Duplicate keys are not allowed. But values can be duplicated.
3. Heterogeneous objects are allowed for both keys & values.
4. Insertion order is not preserved & it is based on HashCode

of the keys.

5. Null is allowed for key (only once) and allowed for values (any no. of times).

*** Differences blw HashMap and Hashtable :—

HashMap	Hashtable
1. No method present inside HashMap is synchronized.	1. Every method present inside Hashtable is synchronized.
2. At a time multiple threads are allowed to operate on HashMap object simultaneously and hence it is not Thread safe.	2. At a time only one thread is allowed to operate on Hashtable object and hence it is Thread safe.
3. Relatively performance is high.	3. Relatively performance is low.
4. Null is allowed for both keys and values.	4. Null is not allowed for both keys & values, o.w, we will get NullPointerException.

*** Q: How to get synchronized version of HashMap?

Ans: By default HashMap is non-synchronized. But we can get synchronized version of HashMap by using synchronizedMap() method 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 (or) 75%.

② HashMap m=new HashMap(int initialcapacity);

③ HashMap m=new HashMap(int initialcapacity, float fillRatio);

④ HashMap m=new HashMap(Map m);

Ex:
```
import java.util.*;
class HashMapDemo
{
  p s v m(-)
  {
    HashMap m=new HashMap();
    m.put("chiru",700);
    m.put("balaiaeh",800);
    m.put("venki",200);
    m.put("nag",500);
    S.o.p(m); ⇒ o/p : {nag=500, venki=200, balaiah=800, chiru=700}
    S.o.p(m.put("chiru",1000));
    Set s=m.keySet(); ⇒ o/p : 700
    S.o.p(s); ⇒ o/p : [nag, venki, balaiah, chiru]
    Collection c=m.values();
    S.o.p(c); ⇒ o/p : [500, 200, 800, 1000]
    Set s1=m.entrySet();
    S.o.p(s1); ⇒ o/p : [nag=500, venki=200, balaiah=800, chiru=1000]
    Iterator itr=s1.iterator();
    while(itr.hasNext())
    {
      Map.Entry m1=(Map.Entry)itr.next();
      S.o.p(m1.getKey()+"---"+m1.getValue()); ⇒ o/p : nag---500
                                                      venki---200
                                                      balaiah---800
                                                      chiru---1000
      if(m1.getKeye().equals("nag"))
        m1.setValue(10000);
    }
    S.o.p(m); ⇒ o/p : {nag=10000, venki=200, balaiah=800, chiru=1000]
  }
}
```

DEMO

b. LinkedHashMap :—

→ *** It is exactly same as HashMap except the following differences.

HashMap	LinkedHashMap
1. The underlying data structure is Hashtable.	1. The underlying data structure is combination of Hashtable & LinkedList.
2. Insertion order is not preserved.	2. Insertion order is preserved.
3. Introduced in 1.2 version.	3. Introduced in 1.4 version.

→ In the above example, if we replace HashMap with LinkedHashMap then o/p is {chiru=700, balaiah=800, venki=200, nag=500}
i.e., insertion order is preserved.

Note :— In general we can use LinkedHashSet & LinkedHashMap for developing Cache based applications, where duplicates are not allowed but, insertion order must be preserved.

DEMO

c. IdentityHashMap :—

→ It is exactly same as HashMap except the following difference.

In case of Normal HashMap, JVM will use .equals(-) method to identify duplicate keys, which is meant for content comparison.

But in case of IdentityHashMap, JVM will use == operator to identify duplicate keys, which is meant for reference comparison.

Ex:
```
HashMap m=new HashMap();
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);
m.put(I1, "pawan");
m.put(I2, "kalyan");
```

I1.equals(I2) => true
I1 == I2 => false

S.o.p(m); ⇒ o/p: {10=Kalyan}

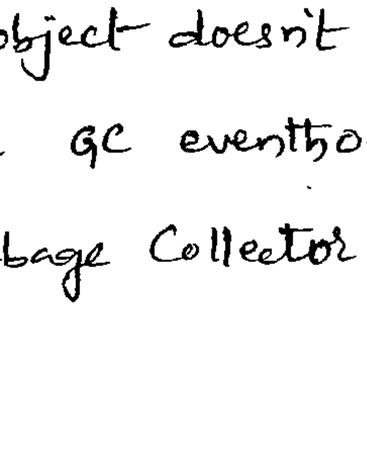
→ If we replace HashMap with IdentityHashMap then o/p is

*** {10=pawan, 10=kalyan}.

Q: What is the difference b/w == operator & .equals() method?

Ans:- In general we can use == operator for reference comparison where as .equals() method for content comparison.

Ex: Integer I1 = new Integer(10);
Integer I2 = new Integer(10);
S.o.p(I1==I2); ⇒ false
S.o.p(I1.equals(I2)); ⇒ o/p: true



d. WeakHashMap :—

→ It is exactly same as HashMap except the following difference.

In case of Normal HashMap if an object associated with HashMap then it is not eligible for Garbage Collection, eventhough it doesn't contain any external references i.e., HashMap dominates Garbage Collector.

But in case of WeakHashMap, if an object doesn't contain any references then it is always eligible for GC eventhough it is associated with WeakHashMap i.e., Garbage Collector dominates WeakHashMap.

DEMO

Ex:
```
impost java.util.*;
clas  WeakHashMapDemo
{
    p  s  v  m(_) throws Exception
    {
      HashMap m=new HashMap();
      Temp t=new Temp();
```

```
m.put(t, "durga");
S.o.p(m); ⇒ o/p : {temp=durga}
t=null;
System.gc();
Thread.sleep(5000);
S.o.p(m); ⇒ o/p : {temp=durga}
}
}
class Temp
{
  public String toString()
  {
    return "temp";
  }
  public void finalize()
  {
    S.o.p("finalize method called");
  }
}
```

DEMO

→ If we replace HashMap with WeakHashMap then o/p is

```
{temp=durga}
Finalize method called
{}
```

8. SortedMap(I):-

→ It is the child interface of Map.

→ If we want to represent a group of key-value pairs according to some sorting order of keys then we should go for SortedMap.

→ Sorting should be based on keys, but not based on values.

→ SortedMap defines the following specific methods.

1. Object firstKey();
2. Object lastKey();
3. SortedMap headMap(Object key);
4. SortedMap tailMap(Object key);
5. SortedMap subMap(Object key1, Object key2);
6. Comparator comparator()

TreeMap(c):—

1. The underlying data structure is Red-Black Tree.
2. Duplicate keys are not allowed. But values can be duplicated.
3. Insertion order is not preserved and it is based on some sorting order of keys.

→ If we are depending on DNSO then keys should be homogeneous & Comparable, o.w we will get runtime exception ClassCastException.

→ If we are defining our own sorting by Comparator then keys can be heterogeneous & non Comparable.

→ But there are no restrictions on values, they can be heterogeneous & non Comparable.

Null Acceptance:—

1. For empty TreeMap as the first Entry with null key is allowed, but after inserting that Entry if we are trying to insert any other Entry we will get runtime exception saying NullPointerException.
2. For non-empty TreeMap if we are trying to insert entry with

null key then we will get runtime exception saying NullPointerException.

3. 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);

④ TreeMap t=new TreeMap(Map m);

Ex: ① impost java.util.*;

DEMO

```
class TreeMapDemo
{
  P S V m(-)
  {
    TreeMap t=new TreeMap();
    m.put(100, "ZZZ");
    m.put(103, "YYY");
    m.put(101, "XXX");
    m.put(104, 106);
    m.put(107, null);
    // m.put("FFFF", "xxx"); => RE: ClassCastException
    // m.put(null, "xxx"); => RE: NullPointerException.
  } S.O.P(m); => O/P: {100=ZZZ, 101=XXX, 103=YYY, 104=106,
}                                                107=null}
```

Ex②:

```
import java.util.*;
class TreeMapDemo
{
  p s v m(-)
  {
    TreeMap t=new TreeMap(new MyComparator());
     t.put("xxx", 10);
     t.put("AAA", 20);
     t.put("zzz", 30);
     t.put("LLL", 40);
     S.o.p(t); ⇒ o/p : {zzz=30, xxx=10, LLL=40, AAA=20}
  } }
class MyComparator implements Comparator
{
  public int compare(Object obj1, Object obj2)
  {
     String s1=obj1.toString();
     String s2=obj2.toString();
     return s2.compareTo(s1);
  } }
```


Hashtable :-

1. The underlying data structure for Hashtable is Hashtable.
2. Duplicate keys are not allowed, but values can be duplicated.
3. Insertion order is not preserved & it is based on Hashcode of the keys.
4. Heterogeneous objects are allowed for both keys & values.
5. Null insertion is not possible for both key & values, o.w we will get runtime exception saying NullPointerException.

6'. Every method present in Hashtable is synchronized & 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(int initialcapacity);

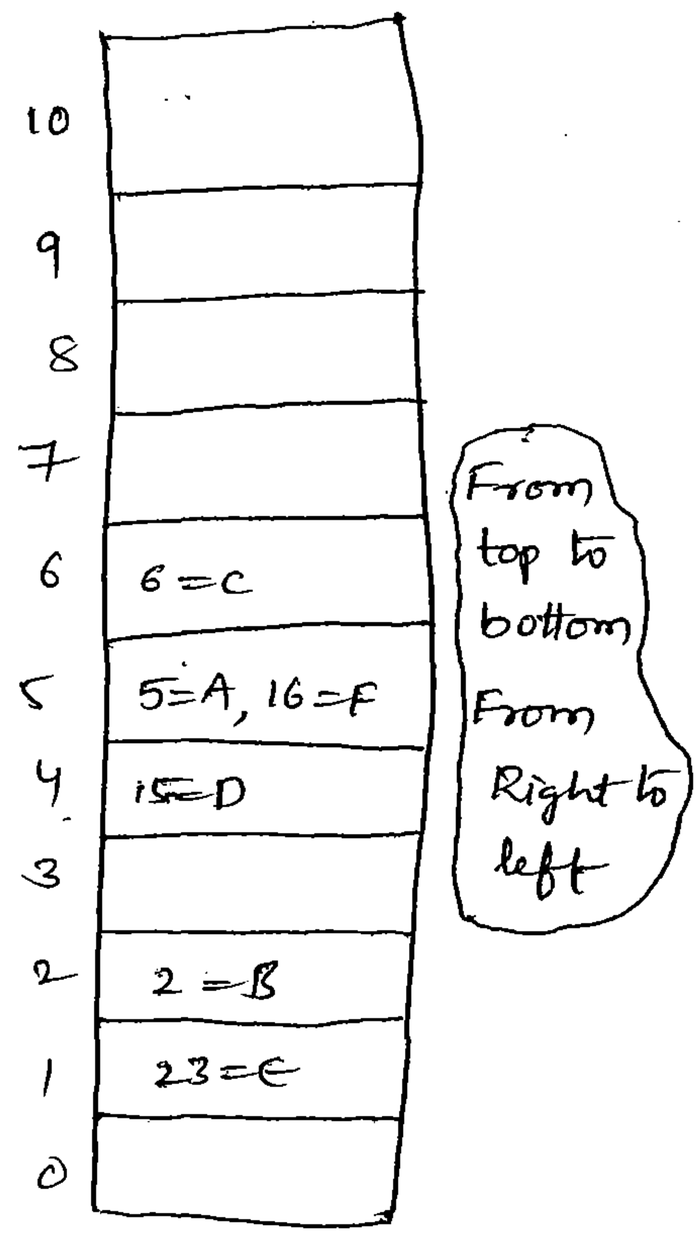
③ . Hashtable h=new Hashtable(int initiakapacity, float floatRatio);

④ . Hashtable h=new Hashtable(Map m);

Ex:

```
import java.util.*;
class HashtableDemo
{
   P s v m(-)
   {
     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"); →15%.11 =4
     h.put(new Temp(23), "E"); →23%.11 =1
     h.put(new Temp(16), "F"); → 16%.11=5
     //h.put("durga", null); → RE : NPE
     S.o.p(h); ⇒ o/p : {6=C, 5=A, 16=F,
                        15=D, 2=D, 23=E}
   }
}
class Temp
{
   int i;
```

DEMO



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

Properties :—

→ In our program, anything which changes frequently never recommended to hardcode in Java program becoz for every change in Java source file we have to recompile, rebuild and redeploy the application and sometimes server restart also required, which creates big business impact to the client.

DEMO

→ Such type of variables we have to configure in Properties file and we have to read those properties from Properties file into Java application.

→ The main advantage in this approach is if there is a change in Properties file, to reflect that change just redeployment is enough, which won't create any business impact.

→ We can use Properties object to hold properties which are coming from properties file.

→ Properties class is the child class of Hashtable.

→ In Properties, both key and value should be String type.

Constructor :—

① `Properties p = new Properties();`

Methods :—

①. public String getProperty(String pname);

To get the value associated with specified property name.

②. public String setProperty(String pname, String pvalue);

To set a new property.

③. public Enumeration propertyNames();

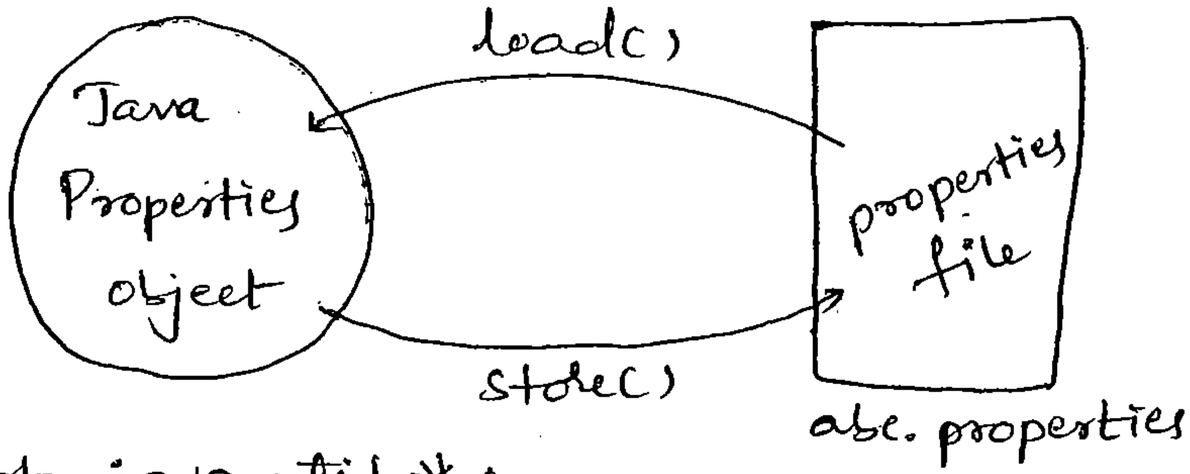
It returns all property names.

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.



Ex: ① import java.util.*;
import java.io.*;
class PropertiesDemo1
{
p s v m(-)
{

```
Properties p=new Properties();
FileInputStream fis=new FileInputStream("abc.properties");
p.load(fis);
S.o.p(p); ⇒ {user=scott, pwd=Tiger, venki=9999}
String s = p.getProperty("venki");
S.o.p(s); ⇒ o/p : 9999
p.setProperty("nag","88888");
FileOutputStream fos=new FileOutputStream("abc.properties");
p.store(fos, "Updated by Durga for SCJP Demo class");
    }
}
```

abc.properties :—

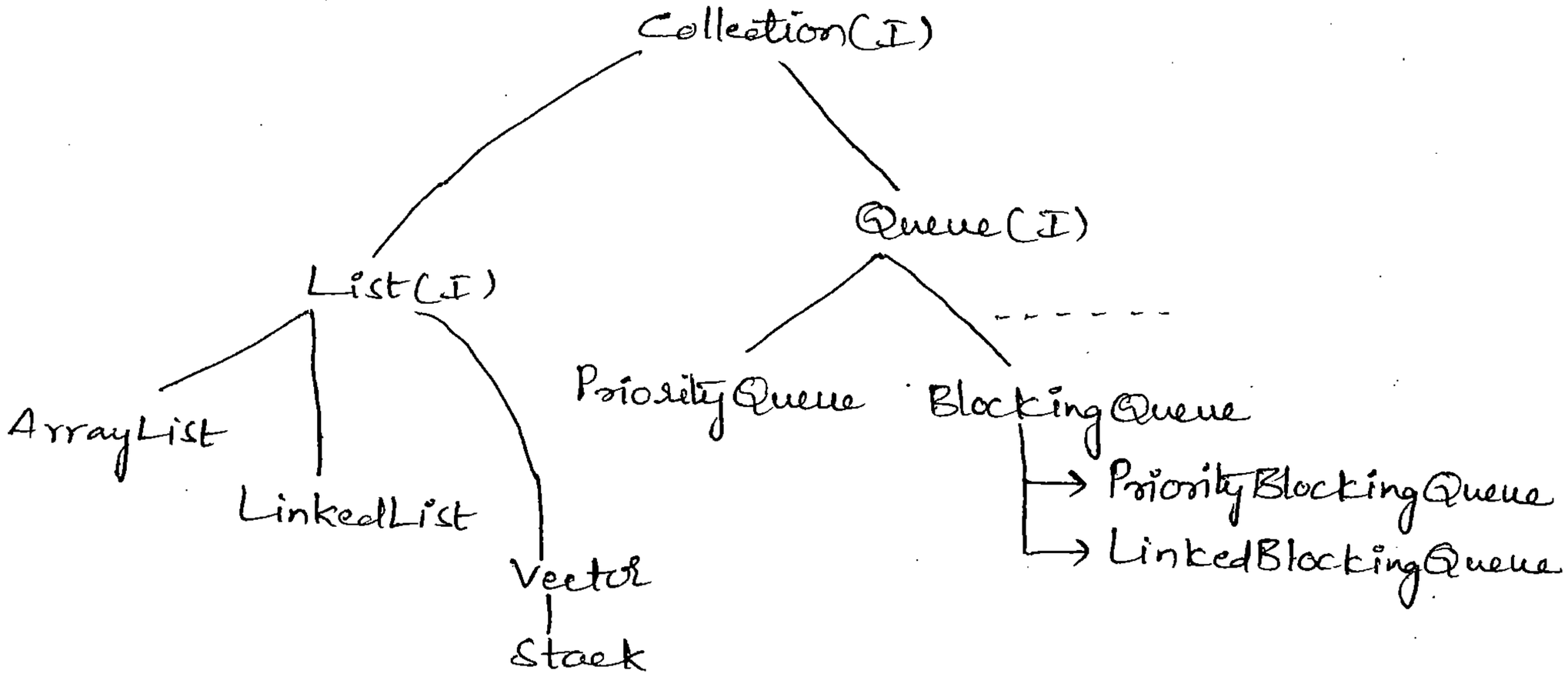
```
user = scott
pwd = Tiger
venki = 9999
```

DEMO

Ex②:

```
class PropertiesDemo2
{
  p s v m(_)
  {
    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 the 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.

→ Usually Queue follows First In First Out (FIFO) order. But based on our requirement we can implement our own priorities also (PriorityQueue).

→ LinkedList based implementation of Queue always follows FIFO order.

Ex: Before sending a mail we have to store all mail id's in some data structure and for the first inserted mail id mail should be send first.

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 queue. If queue is empty then this method returns null.

③ . Object element()

To return head element of queue. If queue is empty then this method raises RE: NoSuchElementException.

④ . Object poll()

To remove & return head element of the queue. If queue is empty then this method returns null.

⑤ . Object remove()

To remove & return head element of the queue. If queue is empty then this method raises RE: NoSuchElementException.

a. Priority Queue :—

→ This is a data structure to store a group of individual objects prior to processing according to some priority.

→ The priority order can be either DNSO or Customized sorting order.

→ If we are depending on DNSO then the objects should be homogeneous & Comparable, O.W we will get ClassCastException.

→ If we are defining our own sorting by Comparator then the objects need not be homogeneous & Comparable.

→ Duplicate objects are not allowed.

→ Insertion order is not preserved.

→ Null insertion is not possible even as first element also.

Constructors :-

① . PriorityQueue q=new PriorityQueue();

Creates an empty PriorityQueue with default initial capacity 11 & all objects will be inserted according to DNSO.

② . PriorityQueue q=new PriorityQueue(int initialCapacity);

③ . PriorityQueue q=new PriorityQueue(int initialCapacity, Comparator c);

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

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

DEMO

Ex: ① import java.util.*;

```
class PriorityQueueDemo1
{
  P s v m(-)
  {
   PriorityQueue q=new PriorityQueue();
   || S.o.p(q.peek()); ⇒ o/p : null
   || S.o.p(q.element()); → RE : NSEE
   for(int i=0; i<=10; i++)
    {
      q.offer(i);
    }
    S.o.p(q); ⇒ o/p : [0,1,2,-----10]
    S.o.p(q.poll()); ⇒ o/p : 0
    S.o.p(q); ⇒ o/p : [1,2,3,-----10]
  }
}
```

Note:- Some operating systems won't provide proper support for PriorityQueue's.

Ex②:

```
import java.util.*;
class PriorityQueueDemo2
{
  P S V m(-)
  {
    PriorityQueue q=new PriorityQueue(15, new
    q.offer("A");                    MyComparator());
    q.offer("Z");
    q.offer("L");
    q.offer("B");
  } S.O.p(q); ⇒ O/P:[Z,L,B,A]
}
class MyComparator implements Comparator
{
  public int compare(Object obj1, Object obj2)
  {
    String s1=(String) obj1;
    String s2=obj2.toString();
    return s2.compareTo(s1);
  } }
```

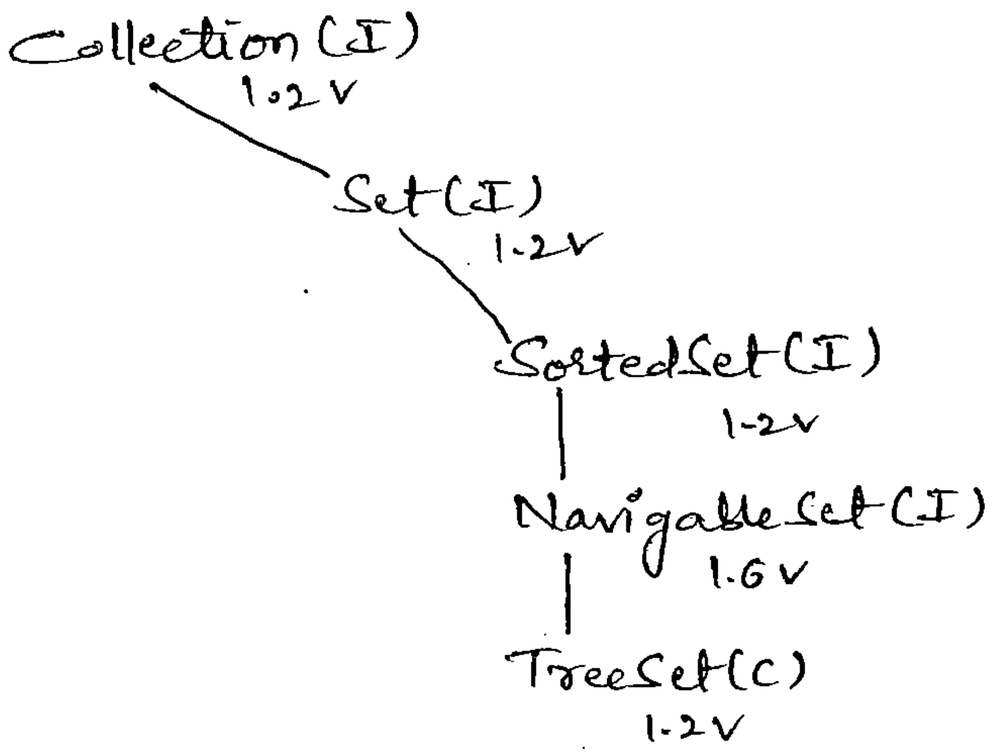
DEMO

1.6 version Enhancements :-

1. NavigableSet (I):-

→ It is the child interface of SortedSet.

→ It defines several methods for navigation purposes.



Methods :—

① . floor(e)

It returns highest element which is <=e.

② . lower (e)

It returns highest element which is <e.

③ . ceiling (e)

It returns lowest element which is >=e.

④ higher (e)

It returns lowest element which is >e.

⑤ . pollFirst()

Remove & return first element.

⑥ . pollLast()

Remove & return last element.

⑦ . descendingSet()

It returns NavigableSet in reverse order.

Ex:-

```
import java.util.*;
class NavigableSetDemo
{
   P s v m(--)
   {
```

```
TreeSet<Integer> t=new TreeSet<Integer>();
t.add(1000);
t.add(2000);
t.add(3000);
t.add(4000);
t.add(5000);
S.o.p(t); ⇒ o/p : [1000, 2000, 3000, 4000, 5000]
S.o.p(t.ceiling(2000)); ⇒ o/p : 2000
S.o.p(t.higher(2000)); ⇒ o/p : 3000
S.o.p(t.floor(3000)); ⇒ o/p : 3000
S.o.p(t.lower(3000)); ⇒ o/p : 2000
S.o.p(t.pollFirst()); ⇒ o/p : 1000
S.o.p(t.pollLast()); ⇒ o/p : 5000
S.o.p(t.descendingSet()); ⇒ o/p : [4000, 3000, 2000]
S.o.p(t); ⇒ o/p : [2000, 3000, 4000]
    }
}
```

DNSO

~~1000~~
2000
3000
4000
~~5000~~

DEMO

2. NavigableMap(I):—

→ It is the child interface of SortedMap.

→ It defines several methods for navigation purposes.

Methods:—

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

Map(I) 1.2v
↓
SortedMap(I) 1.2v
↓
NavigableMap(I) 1.6v
↓
TreeMap(C) 1.2v

Ex: import java.util.*;

```
class NavigableMapDemo
{
  P S V m(-)
  {
    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");
    S.o.p(t); ⇒ O/P: {a=apple, b=banana, c=cat, d=dog,
                                          g=gun}
    S.o.p(t.ceilingKey("c")); ⇒ O/P : c
    S.o.p(t.higherKey("d")); ⇒ O/P : g
    S.o.p(t.floorKey("e")); ⇒ O/P : d
    S.o.p(t.lowerKey("e")); ⇒ O/P : d
    S.o.p(t.pollFirstEntry()); ⇒ O/P : a=apple
    S.o.p[t.pollLastEntry()); ⇒ O/P : g=gun
    S.o.p(t.descendingMap()); ⇒ O/P : {d=dog, c=cat,
    S.o.p(t); ⇒ O/P : {b=banana, c=cat, d=dog}   b=banana}
  }
}
```

DNSO

~~a=apple~~
b=banana
c=cat
d=dog
~~g=gun~~

DEMO

Utility classes (Collections & Arrays):—

1. Collections:—

→ 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 & Comparable objects, o.w. we will get RE saying ClassCastException.

→ List should not contain null, o.w. we will get NPE. DEMO

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

To sort based on Customized Sorting order.

Ex: To sort elements of List according to natural sorting order:—

```
import java.util.*;
class CollectionsSortDemo
{
  P S V M(-)
  {
    ArrayList l=new ArrayList();
    l.add("Z");
    l.add("A");
    l.add("K");
    l.add("N");
    // l.add(new Integer(10)); → RE: CCE
```

```
    // l.add(null); ⟶ RE : NPE
    S.o.p("Before Sorting :" + l); ⇒ o/p : [Z, A, K, N]
    Collections.sort(l);
    S.o.p("After Sorting :" + l); ⇒ o/p : [A, K, N, Z]
  }
}
```

Ex②: To sort elements of List according to Customized sorting order:-

```
import java.util.*;
class CollectionsSortDemo1
{
  P S V m(-)
  {
    Arraylist l=new ArrayList();
    l.add("Z");
    l.add("A");
    l.add("K");                    DEMO
    l.add("L");
    S.o.p("Before Sorting :" + l); ⇒ o/p : [Z, A, K, L]
    Collections.sort(l, new Comparator());
    S.o.p("After Sorting :" + l); ⇒ o/p : [Z, L, 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 DNSO then we have to use this method.

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

→ If we sort List according to Comparator then we use this method.

Conclusions :—

1. Internally the above search methods will use Binary Search algorithm.
2. Before performing search operation compulsory List should be sorted, o.w. we will get unpredictable results.

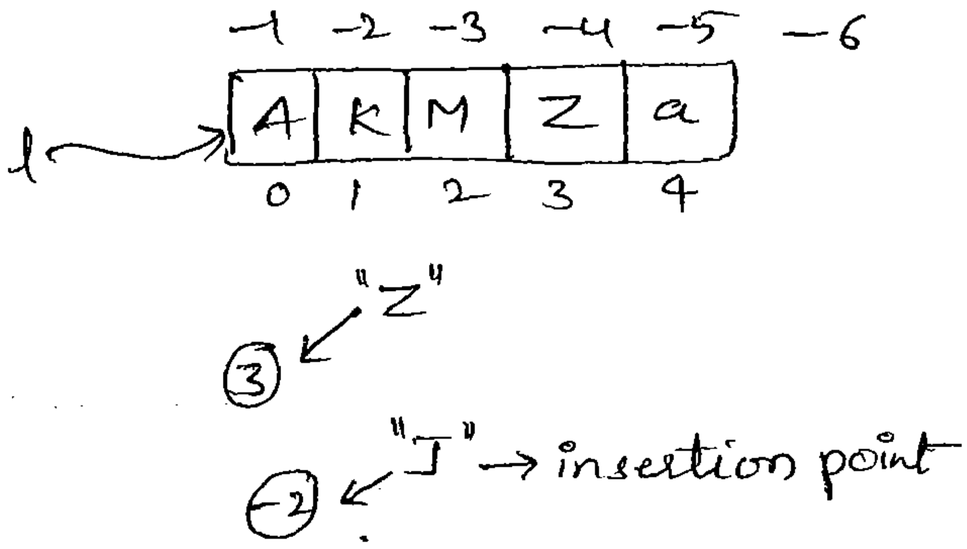
DEMO

3. Successful search returns index.
4. Unsuccessful search returns insertion point.
5. Insertion point is the place where we can place target element in sorted List.
6. If the List is sorted according to Comparator then at the time of search operation also then we should pass the same Comparator object, o.w. we will get unpredictable results.

Ex①: List is sorted according to natural sorting order :—

```
import java.util.*;
class CollectionsSearchDemo
{
  P s v m(—)
  {
```

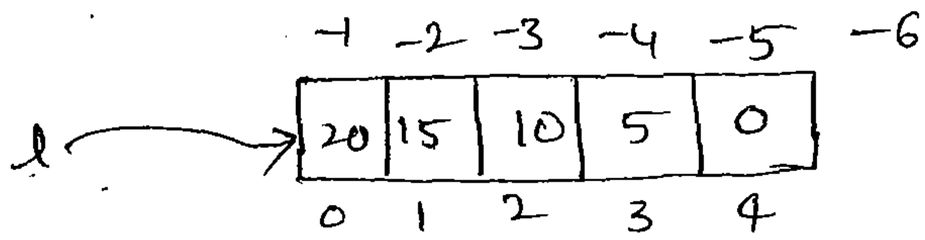
```
ArrayList l=new ArrayList();
l.add("Z");
l.add("A");
l.add("M");
l.add("K");
l.add("a");
S.o.p(l); ⇒ O/P : [Z, A, M, K, a]
Collections.sort(l);
S.o.p(l); ⇒ O/P : [A, K, M, Z, a]
S.o.p(Collections.binarySearch(l,"Z")); ⇒ O/P : 3
S.o.p(Collections.binarySearch(l,"J")); ⇒ O/P : -2
}
}
```



Ex②: List is sorted according to Customized sorting order:-

DEMO

```
import java.util.*;
class CollectionsSearchDemo1
{
  P s v m(-)
  {
    ArrayList l=new ArrayList();
    l.add(15);
    l.add(0);
    l.add(20);
    l.add(10);
    l.add(5);
    S.o.p(l); ⇒ O/P : [15, 0, 20, 10, 5]
    Collections.sort(l, new MyComparator());
    S.o.p(l); ⇒ O/P : [20, 15, 10, 5, 0]
    S.o.p(Collections.binarySearch(l,10, new MyComparator())); //2
```



```
S.o.p(Collections.binarySearch(l,13, new MyComparator2()));  ⇒ o/p :-3
S.o.p(Collections.binarySearch(l,17));  ⇒ o/p : -6 (unpredictable)
}
}
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        Integer i1=(Integer)obj1;
        Integer i2=(Integer)obj2;
        return i2.compareTo(i1);
    }
}
```

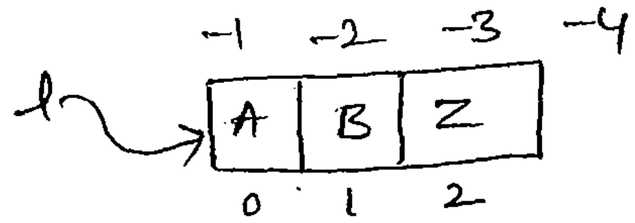
Note :- For the list of n elements,

Range of successful search : 0 to n-1

Range of unsuccessful search : -(n+1) to -1

Total Result Range : -(n+1) to n-1.

Ex: 3 elements



Range of successful search : 0 to 2

Range of unsuccessful search : -4 to -1

Total Result Range : -4 to 2

Reversing the elements of List :-

```
public static void reverse(List l);
```

Ex:
```
import java.util.*;
class CollectionsReverseDemo
{
  p s v m(-)
  {
    ArrayList l=new ArrayList();
    l.add(15);
    l.add(0);
    l.add(20)
    l.add(10);
    l.add(5);
    S.o.p(l); ⇒ o/p : [15, 0, 20, 10, 5]
    Collections.reverse(l);
    S.o.p(l); ⇒ o/p : [5, 10, 20, 0, 15].
  }
}
```

reverse() Vs reverseOrder() :— DEMO

→ We can use reverse() method to reverse order of elements of List.

→ We can use reverseOrder() method to get reversed Comparator.

Ex: Comparator c = Collections.reverseOrder(Comparator c);

↓ Descending order

↓ Ascending order.

2. Arrays :—

→ Arrays class is an utility class to define several utility methods for Array objects.

1. Sorting elements of Array :—

1. public static void sort(primitive[] p)

To sort according to DNSO.

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 NSO or CSO. But we can sort primitive array only based on NSO, but not based on CSO.

Ex: To sort elements of Array:-

```
import java.util.Arrays;
import java.util.Comparator;
class ArraysSortDemo
{
    p s v m(-)
    {
      int[] a={10,5,20,11,6};
      S.o.p("Primitive Array before sorting");
      for(int a1:a)
      {
        S.o.p(a1); ⇒ o/p: [10,5,20,11,6]
      }
      Arrays.sort(a);
      S.o.p("Primitive Array After sorting");
      for(int a1:a)
      {
        S.o.p(a1); ⇒ o/p: [5,6,10,11,20]
      }
      String[] s={"A","Z","B"};
      S.o.p("Object Array before sorting");
```

DEMO

```
for(String s1 : s)
{
  S.o.p(s1); ⇒ O/P : [A, Z, B]
}
Arrays.sort(s);
S.o.p("Object Array After sorting");
for(String s1 : s)
{
  S.o.p(s1); ⇒ O/P : [A, B, Z]
}
Arrays.sort(s, new MyComparator());
S.o.p("Object Array After sorting by Comparator");
for(String s1 : s)
{
  S.o.p(s1); ⇒ O/P : [Z, B, A]
}
}
}
class MyComparator implements Comparator
{
  public int compare(Object o1, Object o2)
  {
    String s1=o1.toString();
    String s2=o2.toString();
    return s2.compareTo(s1);
  }
}
```

DEMO

2. Searching Elements of Array :—

→ Arrays class defines the following methods.

1. public static int binarySearch(primitive[] p, primitive target)

→ If the primitive Array sorted according to NSO then we have to use this method.

2. public static int binarySearch(Object[] o, Object target)

If the Object Array sorted according to NSO then we have to use this method.

3. public static int binarySearch(Object[] o, Object target, Comparator c)

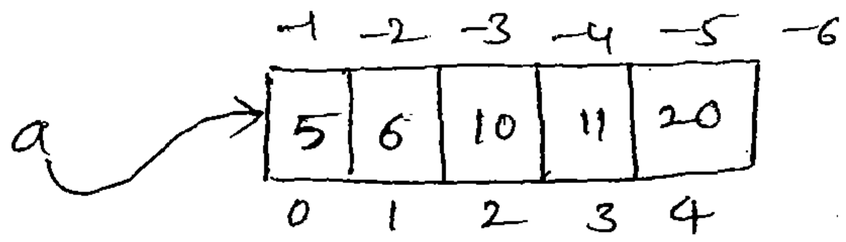
If the Object Array sorted according to Comparator then we have to use this method.

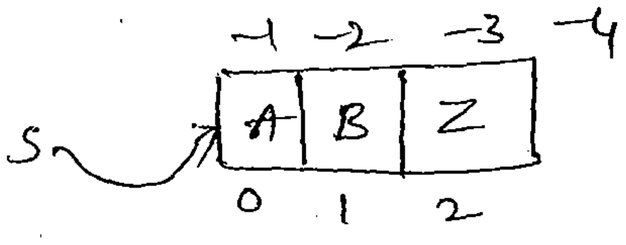
Note:- All rules of Arrays class binarySearch() method are same as Collections class binarySearch() method.

Ex: To Search Elements of Array:-

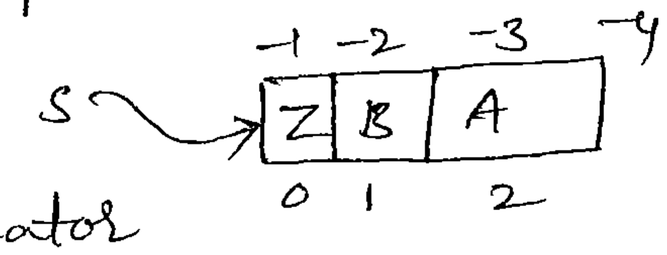
```
import java.util.*;
class ArraysSearchDemo
{
    P S V m(-)
    {
      int[] a={10,5,20,11,6};
      Arrays.sort(a); // sort by NSO
      S.o.p(Arrays.binarySearch(a,6)); ⇒ o/p: 1
      S.o.p(Arrays.binarySearch(a,14)); ⇒ o/p: -5

      String[] s={"A","Z","B"};
      Arrays.sort(s);
      S.o.p(Arrays.binarySearch(s,"Z")); ⇒ o/p: 2
      S.o.p("Arrays.binarySearch(s,"S")); ⇒ o/p: -3
```

DEMO





```
        Arrays.sort(s, new MyComparator());
        S.o.p(Arrays.binarySearch(s, "Z", new MyComparator())); //0
        S.o.p(Arrays.binarySearch(s, "S", new MyComparator())); //-2
        S.o.p(Arrays.binarySearch(s, "N")); //unpredictable result.
    }
}
class MyComparator implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        String s1 = o1.toString();
        String s2 = o2.toString();
        return s2.compareTo(s1);
    }
}
```



DEMO

3. Conversion of Array to List:—

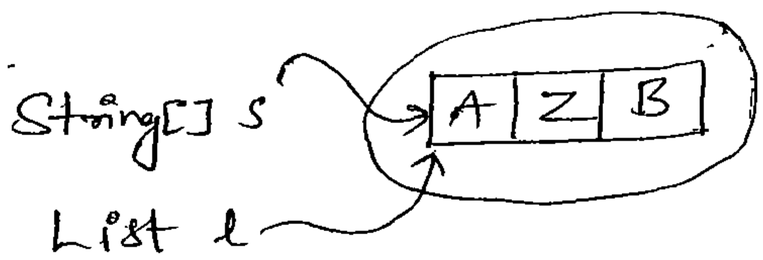
→ Arrays class contains asList() method for this.

```
public static List asList(Object[] a)
```

→ This method won't create an independent List object, just we are viewing existing array in List form.

Ex: `String[] s = {"A", "Z", "B"};`

`List l = Arrays.asList(s);`



Conclusions:—

1. By using array reference we can perform any change automatically that change will be reflected to List reference.
   lly using List reference if we perform any change

automatically that change will be reflected to array.

2. By using List reference we can't perform any operation which varies the size, o.w. we will get RE saying UnsupportedOperationException.

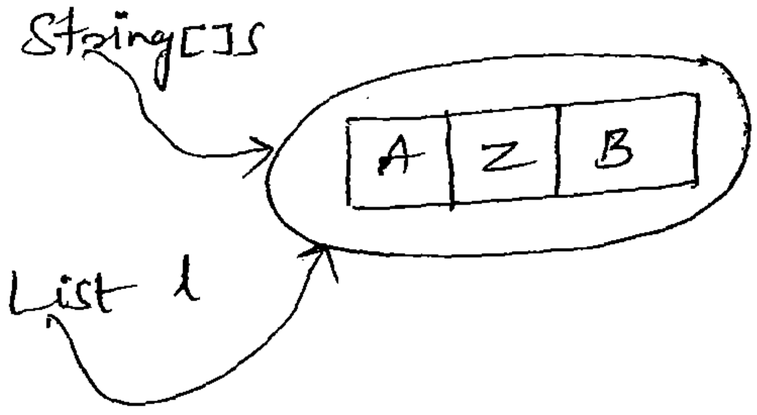
Ex: l.add("K");
l.remove(1); } → RE: UnsupportedOperationException.

l.set(1,"K"); ✓

3. By using List reference we can't replace heterogeneous objects, o.w. we will get RE saying ArrayStoreException.

Ex:
```
import java.util.*;
class ArraysAsListDemo
{
  p s v m(-)
  {
    String[] s={"A","Z","B"};
    List l=Arrays.asList(s);
    S.o.p(l); ⇒ o/p : [A, Z, B]
    s[0]="K";
    S.o.p(l); ⇒ o/p : [K, Z, B]
    l.set(1,"L");
    for(String s1 : s)
      S.o.p(s1); ⇒ o/p : [K, L, B]
    // l.add("durga"); → RE: UnsupportedOperationException
    // l.remove(2); → RE: UnsupportedOperationException
    // l.set(1, new Integer(10)); → RE: ArrayStoreException
  }
}
```

DEMO



DEMO