Array

- An Array is an Indexed collection of fixed number of homogeneous data Element.
- The main advantage of array is we can represent multiple value by using single variable. so that readability of the code will be improved.

Limitations of Arrays:-

- Arrays are fixed in size that is once we create an array there is no chance of increasing of decreasing the size based on our requirement due to this to use arrays concept compulsory we should know the size in advance which may not posible always.
- Array can hold only homogeneous data type elements

Ex:- `Student[] s = new Student[10000];`
`s[0] = new Student();` ✓
`s[1] = new Customer();` ✗
CE : incompatible types
found : Customer
required : Student

We can solve this problem by using object type arrays.

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

- Arrays concept is not implemented based on some standard data structure and hence readymate method support is not available for every requirement we have to write the code explicitly which increases the complexity of programing

* To overcome above problem of Arrays we should go for Collections concept.

- Collections are growable in nature that is based on our requirement we can increase or decrease the size.
- Collection can hold both homogeneous & heterogeneous element.
- Every collection class is implemented based of some standard data structure hence for every requirement ready-made method support is available.
- being a programmer we are responsible to use those method and we are not responsible to implement those method

Differences between Arrays & Collections

Arrays	Collections
• Arrays are fixed in size that is once we creates an array we can't increase and decrease the size based on our requirement.	• Collections are growable in nature that is based on our required we can increase or decrease the size.
• 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 type elements.	• Collections can hold both homogeneous & heterogeneous element
• There is no underlying data structure for array and hence readymate method support is not available. for every requirement we have to write the code explicitly which increase complexity of programming.	• Every Collections class is implemented based on standard data structure and hence for every requirment readymate method support is available, being a programmer we can use these method directly and we are not responsible to implement these methods.
• Arrays can hold both Primitive and object	• Collection can hold only object type but not primitives.

Collection:- If you want to represent A group of individual object as a single entity then we should go for Collection.

Collections framework:- It Contains several classes and interfaces which can be used to represent A group of individual object as a single entity.

Java	C++
• Collection	• Container
• Collection framework	• 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

1. Collection (I):

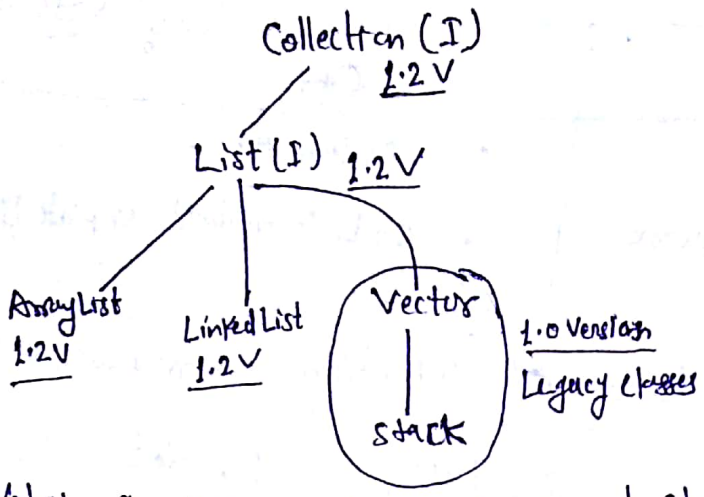
- If we want to represent a group of individual object as a single Entity then we should go for Collection.
- Collection Interface defines the most common methods which are aplicale for any Collection Object.
- In general Collection Interface Consider as root interface of Collection framework.
- There is no Concreate class which implement Collection interface directly.

Difference between Collection & Collections

- Collection is an interface. If we want to represent a group of individual object as a single Entity then we should go for Collection.
- Collections is an utility class present in Java.util Package to define several utility methods for Collection Object (like, sorting, searching etc).

2. List (I)

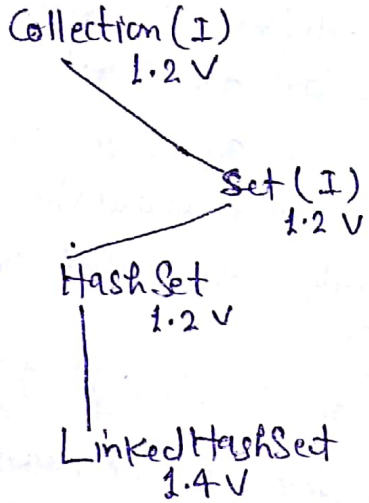
- It is the child interface of Collection.
- If we want to represent a group of individual object as a single Entity where duplicates are allowed and insertion order must be preserved then we should for List.



Note:- In 1.2 version Vector and stack classes are Re-engineered (modified) for implenting the List interface

3. Set :-

- It is the child interface of Collection.
- If we want to represent a group of individual object as a single entity where duplicates are not allowed and insertion order not required then we should go for Set.

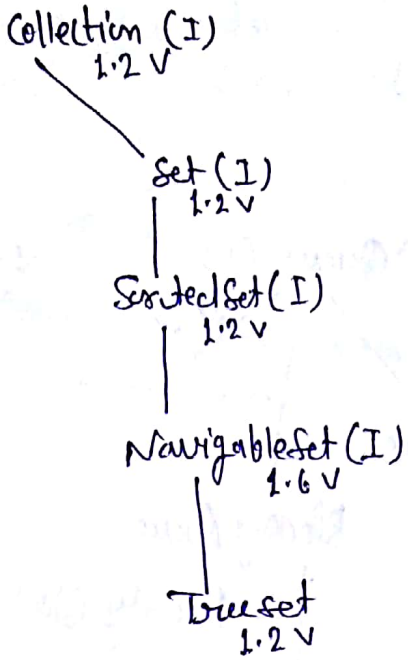


4. SortedSet(I)

- It is the child interface of Set.
- If we want to represent a group of individual object as a single entity where duplicates are not allowed and all object should be inserted according to some sorting order. then we should go for sorted Set.

5. NavigableSet (I)

It is the child interface of Sorted Set. It contains several methods for navigation ~~purpose~~ purpose



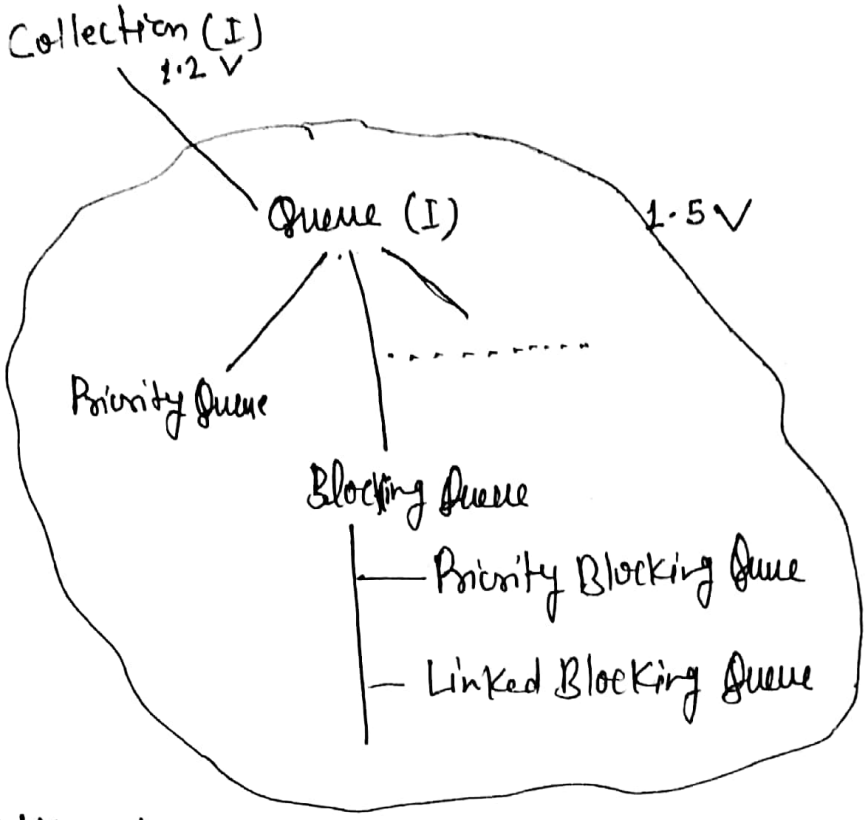
Differences between List and Set

List	Set
① Duplicate are allowed	① Duplicate are not allowed
② Insertion order Preserved	② Insertion order not Preserved

6. Queue (I) :-

- It is the child interface of Collection.
- If we want to represent a group of individual object prior to processing then we should go for Queue.
- Usally Queue follows FIFO order but based on our requirement we can our own priority order also.

  Ex:- Before sending the mail all mail-Id we have to store in some data structure
- In which order we added mail Id in the same order only mail should be delevered. for this requirement Queue is best choice.



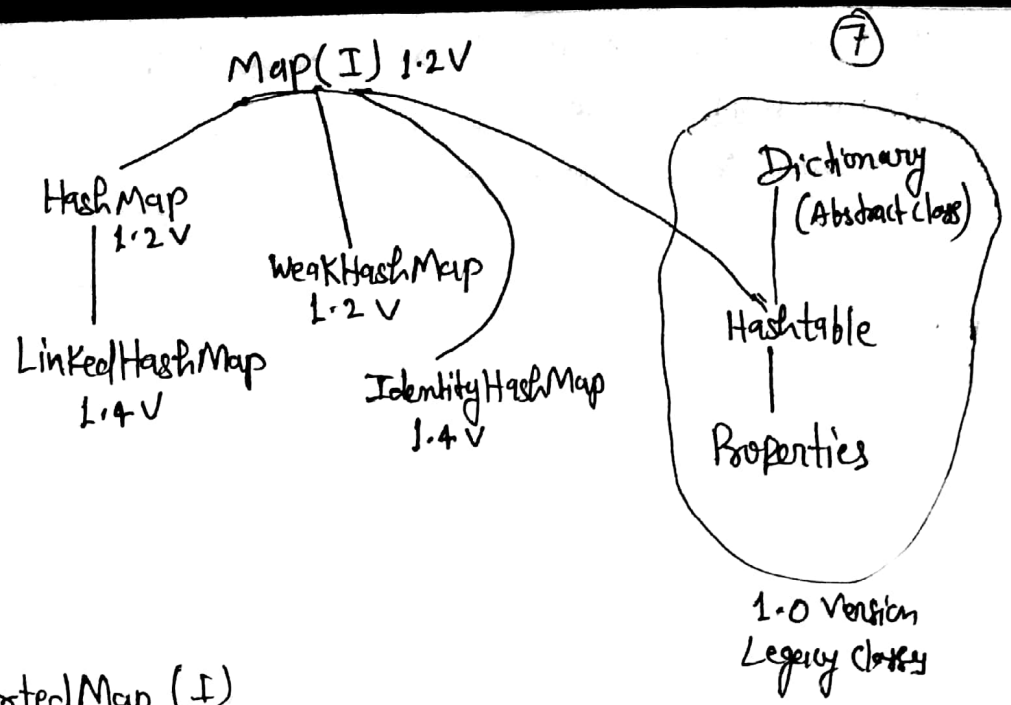
Note:- All the above Interfaces (Collection, List, Set - Sorted Set, Navigable Set, Queue) meant for representing a group of individual object. If we want to represent a group of object as key-value pairs then we should go for Map.

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.

Key	Value
S.NO	Name
101	Durga
102	Ravi
103	Shiva

- Both key and value are objects only.
- Duplicate keys are not allowed but values can be duplicated.

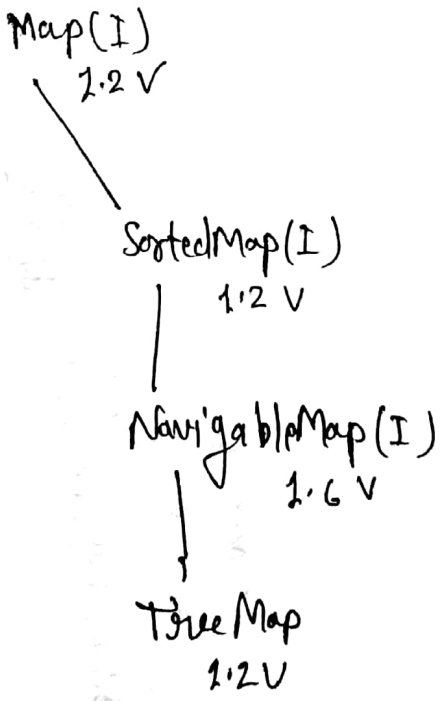


8. SortedMap (I)

- It is the child interface of map.
- If we want to represent a group of Key-Value pairs according to some sorting order of keys then we should go for Sorted Map
- In Sorted map the sorting should be based on key but not based on value.

9. Navigable Map (I)

- It is the child interface of SortedMap.
- It define Several method for navigation purposes.



The following are legacy characters present in Collection framework ②

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

Collection (I) :-

① If we want represent a group of individual object as a single entity then we should go for collection.

② Collection interface defines the most common method which are applicable for any collection object.

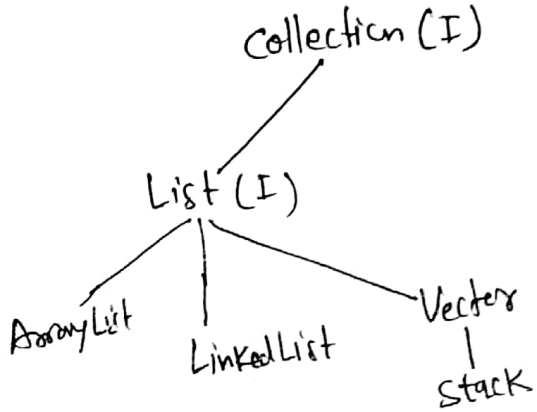
- → boolean add (Object O)
- → boolean addAll (Collection C)
- → boolean remove (Object O)
- → boolean removeAll (Collection C)
- → boolean retainAll (Collection C)
  - To remove all objects except those present in C.
- → Void clear()
- → boolean contains (Object O)
- → boolean containsAll (Collection c)
- → boolean isEmpty().
- → int size()
- → Object[] toArray();
- → Iterator iterator()

Note:- there is no concreate class which implements Collection interface directly.

→ List (I) :-

- List is child interface of Collection.
- If we want to represent a group of individual object as a single entity where duplicates are allowed and insertion order must be preserved then we should go for List.
- we can preserve insertion order via index and we can differentiate duplicate objects by using Index. Hence index will play very important role in List.
- List Interface define the following specifices Method.
  - → void add (int index, Object O)
  - → boolean addAll (int index, Collection C)

→ Object get (int index)
→ Object remove (int index)
→ Object set (int index, Object new)
To replace the element present at specified Index with provided object and return old object.
→ int indexOf (Object o)
return index of first occurrence of 'o'
~~→ int lastIndexOf (o) st occurrence~~
→ int lastIndexOf (Object o)
→ ListIterator listIterator ();



ArrayList :-

① the underlying data structure is resizable array or Growable array.
② Duplicate obj are allowed
③ Insertion order is preserved
④ Heterogeneous objects are allowed. (except TreeSet and TreeMap, every where Heterogeneous objects are allowed.
⑤ null insertion is possible.

Constructors :-

① ArrayList l = new ArrayList ();
• Create the Empty arrayList object with default initial Capacity 10.

• once Arraylist reaches its max capacity then new Arraylist object will be created with new capacity is generate by following formula.

$$\text{New Capacity} = (\text{current capacity} * \frac{3}{2}) + 1$$

② Arraylist l = new Arraylist(int initialCapacity);
creates empty arraylist object with specified initial capacity.

③ Arraylist l = new Arraylist(Collection c);
creates an equivalent Arraylist Object for given Collection.

Example:-

```
Import Java.util.*;
Class ArraylistDemo {
Public static void main(String[] args) {
Arraylist l = new Arraylist();
    l.add("A");
    l.add(10);
    l.add("A");
    l.add(null);
    Sout(l); // [A, 10, A, null]
     l.remove(2);
    Sout(l); // [A, 10, null]
     l.add(2,"M");
     l.add("N");
     Sout(l); // [A, 10, M, null, N]
```

* Usally we can use collection to hold and transfer object from one location to another location (to provide support for this requirement every Collection class by default implements Serializable and clonable interfaces.

* ArrayList and Vector classes implement RandomAccess interface so that any Random Element we can access with the same speed.

Random Access :- RandomAccess interface present in java.util package and it doesn't contain any Method and it is the Marker interface. Where Required ability will be provided automatically by the JVM.

```
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
```

* ArrayList is the best choice if our frequent operation is retrival operation (because arraylist implements RandomAccess interface).

* ArrayList is the worst choice if our frequent operation is insertion and Deletion in the middle.

:- Differences between ArrayList & Vector :-

(1) every method present in the ArrayList is Non-synchronized	(1) every method present in the Vector is synchronized.
(2) At a time multiple threads are allowed to operate on ArrayList object and hence it is not thread safe.	(2) At a time only one thread is allowed to operate on Vector object and hence it is thread safe

③ relatively performance is high because threads are not required to wait to operate on arrayList object.	③ relatively Performance is low because threads are required to wait to operate on vector object.
④ introduced in 1.2 v and it is non legacy.	④ introduced in 1.0 version and it is legacy.

* How to get Synchronized version of ArrayList Object.

* By Default ArrayList is Non-Synchronized but we can get synchronized version of ArrayList object by using synchronizedList() method of Collections class.

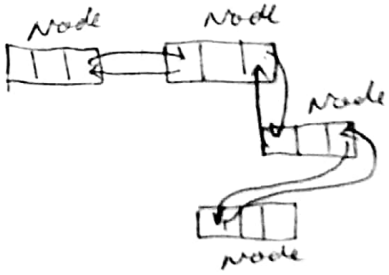
```
Public static List synchronizedList(List l);
```

Ex:-
```
ArrayList l = new ArrayList();
List l1 = Collections.synchronizedList(l);
```
(l1 ← Synchronized; l ← Non-synchronized)

* Similarly we can get synchronized version of Set and Map objects by using the following methods of Collections class.

```
Public static Set synchronizedSet(Set s);
Public static Map synchronizedMap(Map m)
```

LinkedList ;- The underlying data structure is double linked list.



* insertion ordered is preserved.
* duplicate objects are allowed
* heterogeneous objects are allowed
* Null insertion is possible.
* Linklist Implements Serializable and Clonable Interface but not RandomAccess Interface.
* LinkedList is best choice if our frequent operation is insertion or deletion in the middle.

* LinkedList is the worst choice if our frequent operation is retrivale operation.

Constructers :-

① LinkedList l = new LinkedList();
creates an empty LinkedList object

② LinkedList l = new LinkedList(Collection c);
creates an equivalent LinkedList object for given Collection.

LinkedList class specifics methods:

usally we can use LinkedList to develop Stacks & Queue to provide support for this requirement LinkedList class define the following specifice methods.

```
Void addFirst(Object o)
void addLast(Object o)
Object getFirst()
Object getLast()
Object removeFirst()
Object removeLast()
```

Ex:-

```
Import java.util.*;
Class LinkedListDemo
{
  public static void main(String[] arg)
  {
    LinkedList l = new LinkedList();
    l.add("durga");
    l.add(30);
    l.add(null);
    l.add("durga"); // [durga, 30, null, durga]
    l.set(0, Software); //[Software, 30, null, durga]
    l.add(0, venky); //[venky, Software, 30, null, du
    l.removeLast();
    l.addFirst("ccc"); // [venky, software, 30, null]
    System.out.println(l); [ccc, venky, software, 30, null]
                           [ccc, venky, Software, 30, nu
  }
}
```

Differences Between ArrayList & LinkedList

ArrayList	LinkedList
① ArrayList is the best choice if our frequent operation is retrivale operation.	① LinkedList is the best choice if our frequent operation is insertion or deletion in the middle
② ArrayList is the worst choice if our frequent operation is insertion or deletion in the middle. because internally several shift operations are performed.	② LinkedList is the worst choice if oure frequent operation is retrivale operation
③ In ArrayList the Elements will be stored in consugative memory location and hence the retrivale operation will become easy.	③ In LinkedList the Elements will not be stored in cosucative memory Location and hence retrivale operation will become complex.

Vector *:-

① The underline Data Structure is Resizable Array or growable array.
② Insertion order is preserved.
③ Duplicates are allowed.
④ Heterogeneous objects are allowed
⑤ Null insertion is possible.
⑥ It implements Serilizable, Clonable & RandomAccess interfaces
⑦ Every method present in the Vector is ~~Thread~~ Synchronized and hence Vector object is Thread Safe.

Constructers:-

① `Vector v = new Vector();`

Creates an empty Vector object with default initial capicity 10. once vector reaches its Max capacity then a new vector object will be created with

New Capacity = Current Capacity *2

② Vector v = new Vector(int initialCapacity)

Creates an empty Vector Object with Specified Initial capacity.

③ Vector v = new Vector(int initialCapacity, int Increment Capacity)

Vector v

④ Vector v = new Vector(Collection c);

Creates an Equivalent Vector object for given Collection this Constructor mint for Inter Conversion between Collection Object.

Vector Specific methods:-

- add(Object o) ---C
- add(int index, Object o) ----L
- addElement(Object o) ---- V

- remove(Object o) ----C
- removeElement(Object o) ---- V
- remove(int index) ----L
- removeElementAt(int index) ---V
- clear() ----C
- removeAllElements() ---- V

- Object get(int index) ----L
- Object elementAt(int index) ---- V
- Object firstElement() ----V
- Object lastElement() --- V

- int size() ---- V
- int capacity() ---V
- Enumeration elements() ----V

Ex:-

```
import java.util.*;
class VectorDemo1
{
  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); // here primitive value we are adding. here Autoboxing will be done and primitive value will be convert in object.
    }
    System.out.println(v.capacity()); //10
    v.addElement("A");
    System.out.println(v.capacity()); //20
    System.out.println(v); //[1,2,....,10,A]
  }
}
```

Stack :-

- It is the child class of vector
- It is specially designed class Last in first out order.

Constructor :-

```
Stack s = new Stack();
```

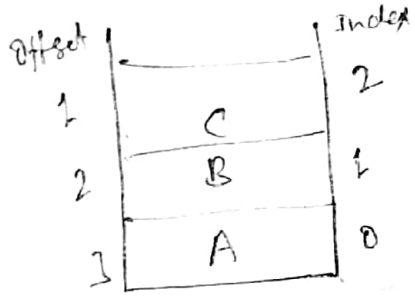
Various method in stack :-

- Object push(Object O)
  to insert and object in to the stack
- Object pop()
  to remove and return top of the stack.
- Object peek()
  to return top of the stack without removal.
- boolean empty()
  returns true if the stack is empty.

- int search(Object o)
  return offset if the element is available otherwise returns -1.

Example:-

```
Import java.util.*;
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
  }
}
```



The Three Cursors of Java

If we want to get the object one by one from the collection then we should go for cursor.

There are Three (3) types of cursor Available in Java

1) Enumeration
2) Iterator
3) ListIterator

(1) Enumeration:- We can use Enumeration to get objects one by one from legacy collection object. We can create Enumeration object by using elements methods of vector.

Public Enumeration elements();

eg:- Enumeration e = v.elements();
→ vector

Methods in Enumeration

- Public boolean hasMoreElements();
- Public Object nextElement();

Ex:-

```
Import java.util.*;
class EnumerationDemo {
public static void main(String[] args){
  Vector v = new Vector();
    for(int i = 0; i<=20; i++){
        v.addElement(i);
    }
  System.out.println(v); //[0,1,2,3,....20]
   Enumeration e = v.elements();
   while(e.hasMoreElements()){
        Integer I = (Integer)e.nextElement();
        if(I%2==0)
         System.out.println(I); // 0 2 4 6 8 10
         else{
          System.out.println(I+"will be removed");
            v.remove(I);
          System.out.println(v);
        }
   }
    // System.out.println(v); // [0,1,2,3,....10]
}
}
```

Limitations of Enumeration

- We can apply Enumeration concept only for legacy classes and it is not an universal cursor.
- By using Enumeration we can get only read access and we can't perform remove operation.
- To overcome above limitations we should go for Iterator

Iterator(I) :-

- We can apply Iterator concept for any Collection object and hence it is universal cursor.
- By using Iterator we can perform both read and remove operation.
- We can create Iterator object by using iterator method of Collection Interface.

* Public Iterator iterator():

Example :- `Iterator itr = c.iterator();` (Any collection object.)

Methods :-

- Public boolean hasNext();
- Public Object next();
- Public void remove();

Example :-

```
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); // [0, 1, 2, .... 10]
  Iterator itr = l.iterator();
    While(itr.hasNext()) {
    Integer I = (Integer) itr.next();
      if (I%2 == 0)
       System.out.println(I); //0 2 4 6 8 10
     else
       itr.remove();
     }
    System.out.println(l); //[0, 2, 4, 6, 8, 10]
  }
}
```

Limitations of Iterator :-

- By using Enumeration and Iterator we can allways move only towards forward direction and we can't move towards Backward direction. There are single direction cursors but not Bi-Directional cursor.
- By using Iterator we can perform only read and remove operation and we can't perform replace and addition of new object.
- To overcome above limitation we should go for list iterator.

ListIterator (I)

- By using list iterator we can move either to the forward direction or to the Backward direction. and hence it is Bi-Directional Cursor.
- By using ListIterator we can perform replacement and addition of new objects in addition to read and remove operations.
- We can create ListIterator object by using listIterator method of List interface.

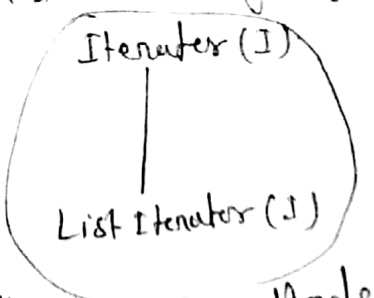
```
Public ListIterator listIterator()
```

Ex:- `ListIterator itr = l.listIterator()`

any List objects.

Methods :-

- ListIterator is the child Interface of Iterator and hence all methods present in Iterator by Default available in the ListIterator



- ListIterator define following 9 methods

forward moment
- Public boolean hasNext();
- Public Object next();
- Public int nextIndex()

Backward moment:
- Public boolean hasPrevious()
- Public Object Previous()
- Public int PreviousIndex()

Extra operations:
- Public void remove()
- Public void add(Object o)
- Pubic void set(Object o)

Example:-

```
Import java.util.*;
Class ListIteraterDemo {
Public static void main(String args[]) {
   LinkedList l = new LinkedList();
      l.add("balakrishna");
      l.add("venki");
      l.add("chiru");
      l.add("nag");
   System.out.println(l); //[balakrishna, venki, chiru, nag]
    ListIterator itr = l.listIterater();
      while(itr.hasNext(){
       String s = (String) itr.next();
        if(s.equals("venki")){
           itr.remove(); // [balakrishna, chiru, nag]
        }else if(s.equals("nag")){
           itr.add("charity"); // [balakrishna, chiru, nag, charity]
        }
        else if ( s.equals("chiru"))
        {
          itr.set("charan"); //[balakrishna, charan, nag, ch
        }
    }
    system.out.println(l); // [balakrishna, charan, nag, charity
}
}
```

- The most powerful cursor is ListIterator but its limitation is, it applicable only for List object.

Comparision table of Three Cursors

Property	Enumeration	Iterator	ListIterator
1) Where we can apply?	only for legacy classes	for any collection object	only for List objects
2) Is it legacy?	yes (1.0V)	No (1.2V)	No (1.2V)
3) Movement	Single direction (only forward direction)	Single direction (only forward direction)	Bidirectional
4) Allowed operations	only Read	Read / Remove	Read / Remove / Replace / Add
5) How we can get?	By using elements() method of vector class	By using iterator() method of Collection (I)	By using listIterator() of List (I)
6) Methods	2 Methods • hasMoreElements() • nextElement()	3 Methods • hasNext() • next() • remove()	9 methods

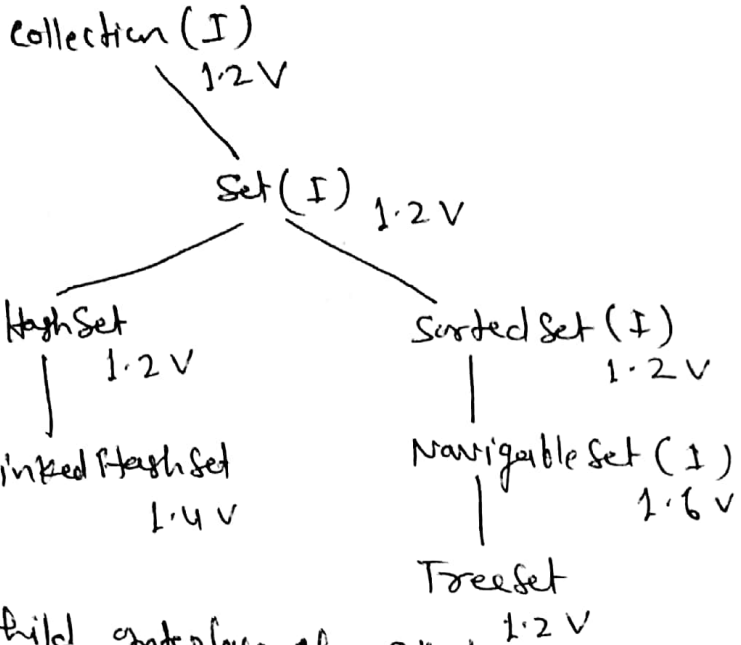
Examples:

Internal implementations of Cursors :-

```
Imports java.util.*;
Class CursorsDemo{
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());
}
}
```

output:-
```
Java.util.Vector$1
Java.util.Vector$Itr
Java.util.Vector$ListItr
```

SET



- Set is child interface of Collection
- If we want to represent a group of individual object as a single entity where Duplicates are not allowed and insertion ordered not preserved.
- Set Interface doesn't contains any new method and we have to use only collection interface methods.

HashSet :-

- The underlying Data structure is HashTable.
- Duplicate objects are not allowed.
- Insertion order is not preserved and it is based of HashCode of object
- Null insertion is possible (only once)
- Hetrogeneous object all allowed
- Implements Serializable & Cloneable but not RandomAccess interface.
- HashSet is the best choice if our frequent operation is search operation.

* **Note→** In HashSet Duplicate are not allowed if we are trying to insert Duplicate then we will not get any Compile or Runtime Error and add method simply return false.

Ex:- HashSet h = new HashSet();
sopl(h.add("A")); // true
sopl(h.add("A")); // false

Constructors :-

① HashSet h = new HashSet();

→ Creates an Empty hashSet object with Default initially capacity 16 and default fill ratio 0.75.

② HashSet h = new HashSet(int initalCapacity);

→ Create an Empty HashSet object with specified initially Capacity and default fill ratio 0.75

③ HashSet h = new HashSet(int initialCapacity, float fillRatio);
(0.9)

→ Create an Empty HashSet object with specified initial capacity and default fill ratio will be also specified.

④ HashSet h = new HashSet(Collection C);

- Creats an Equivalent HashSet for the given Collection
- This Constructor ment for interConversion between Collection objects.

Fill Ratio / Load factor :-

- After filling How much ratio A new HashSet will be Created, this ratio is called fill ratio OR Load Factor.

Ex:- fill ratio 0.75 means, after filling 75% ratio a new HashSet object will be created.

Ex:-
```
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]
}
}
```

Insertion order is not preserved

LinkedHashSet;-

① It is the child class of HashSet

② It is Exactly Same as HashSet (including Constructors an methods) except the following differences.

HashSet	LinkedHashSet
• the underlying Data Structure is HashTable	underlying data structure is combination of LinkedList and HashTable
• insertion order not preserved	• insertion order preserved
• introduced in 1.2 V.	• introduced in 1.4 V

Note:- In the above program if we Replace HashSet with LinkedHashSet then the output is:-

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

• That is insertion order Preserved

Note:- In General we can use LinkedHashSet to developed Cache Based application where Duplicates are not allowed and insertion order preserved.

SortedSet (I);

- SortedSet is the Child interface of Set.
- If we want to represent a group of individual object according to some Sorting order without Duplicates then we should go for Sorted Set.

- SortedSet interface defines the following 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 sorted set whose elements are less than object.

④ SortedSet tailSet(Object obj);
returns sorted set whose elements are >= obj

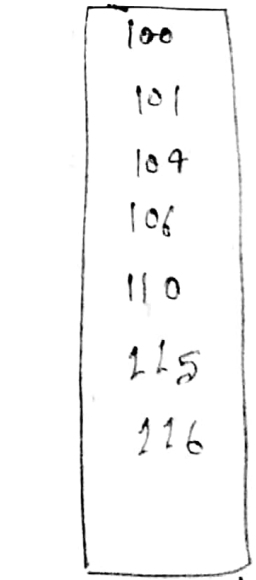
⑤ SortedSet subSet(Object obj1, Object obj2);
returns SortedSet whose elements are >= obj1 and < obj2

⑥ Comparator comparator();
returns Comparator object that describes underlying sorting technique. If we are using default natural sorting order then we will get null.

Note:- The default Natural sorting order for number Ascending order and for String object Alphabetical order.

Example:-

① first() ⇒ 100
② last() ⇒ 120
③ headSet(106) ⇒ [100, 101, 104]
④ tailSet(106) → [106, 110, 115, 120]
⑤ subSet(101, 115) ⇒ [101, 104, 106, 110]
⑥ comparator() [null]



Treeset :-

- The underlying Data Structure is Balanced tree.
- Duplicate objects are not allowed
- Insertion order not preserved.
- Heterogeneous objects are not allowed otherwise we will get runtime Exception saying ClassCastException.
- Insertion Null is allowed but once
- Implements Serializable and Cloneable but not RandomAccess interface.
- All objects will be inserted based on some sorting order it may be Default Natural Sorting order or Customizing Sorting order.

Constructors :-

① TreeSet t = new TreeSet();

Creates an empty treeset object where the element will be inserted according to default natural sorting order.

② TreeSet t = new TreeSet(Comparator C);

Creates an empty Treeset object where the elements will be inserted according to Customized sorting order specified by Comparator object.

③ TreeSet t = new TreeSet(Collection C);

④ TreeSet t = new TreeSet(SortedSet s);

Ex:-

```
Import java.util.*;
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)); // CCE */
```

```
        // t.add(null); // ---NPE
        System.out.println(t); // [A, B, L, Z, 9]
    }
}
```

* null acceptance :-

* for non empty TreeSet if we are trying to insert null then we will get null pointer Exception.
* for Empty TreeSet as the first Element null is allowed but after inserting that null if we are trying to insert any other Element then will get RuntimeException saying Null pointer Exception.

*Note: until 1.7 Version null is allowed as the first Element to the Empty TreeSet. but from 1.7 version onwards null is not allowed even as the first Element that is "null" such type of story not applicable for TreeSet 1.7 version onwards.

Ex:-

```
Impost java.util.*;
Class TreeSetDemo1{
    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 : ClassCastException

* If we are depending on default Natural sorting order compulsory should be Homogeneous & Comparable otherwise we will get RuntimeException saying ClassCastException.
* An Object is said to be Comparable if and only if corresponding class implement Comparable interface.
* String class and all Wrapper classes already implements Comparable interface. but StringBuffer class Doesn't implement Comparable interface.

Hence we got ClassCastException on the above Example.

Comparable (I) :-

- It is present in java.lang package and it contains only one method "CompareTo()".

```
Public int CompareTo(Object obj)
```

obj1.compareTo(obj2)

- return -ve if obj1 has to come before obj2.
- return +ve if obj1 has to come after obj2.
- return 0 if obj1 and obj2 are equal.

Example :-

```
Class Test {
    Public static void main(String[] args) {
        System.out.println("A".compareTo("Z")); // -ve
        System.out.println("Z".compareTo("K")); // +ve
        System.out.println("A".compareTo("A")); // 0
        System.out.println("A".compareTo(null)); // RE: NPE
    }
}
```

* If we are depending on default Natural sorting order then while adding object into the TreeSet JVM will call compareTo() method.

```
TreeSet t = new TreeSet();
t.add("K");
t.add("Z");  → "Z".compareTo("K");
t.add("A");  → "A".compareTo("K");
t.add("A");  → "A".compareTo("A");
S.out(t); // [A, K, Z]
```

obj1.compareTo(obj2)

- The object which is to be inserted
- The object which already Inserted

Note:- If default Natural Sorting order not available or if we are not satisfy with default Natural Sorting order then we can go for customizing sorting by using Comparator.

> *Comparable meant for Default Natural sorting order
> Where as
> * Comparator meant for Customized sorting order

Comparator(I):-

- Comparator present in java.util Package and it defines Two methods Compare() & Equals():

① `Public int Compare(Object obj1, Object obj2);`

→ returns -ve if obj1 has to come before obj2.
→ returns +ve if obj1 has to come after obj2.
→ returns 0 if obj1 & obj2 are equals.

② `public boolean equals(Object obj);`

* Whenever we are implementing Comparator interface. Compulsory we should provide implementations only for Compare method and we are not required to provide implementation for equals method because it is already available to our class from object class through inheritance.

* Write a program to insert the Integer Object in to the Treeset where the sorting order is decending order.

Ex:=
```
import java.util.*;
class TreeSetDemo3 {
    Public static void main(String[] args) {
        TreeSet t = new TreeSet(new MyComparator()); —①
        t.add(10);
        t.add(0);
        t.add(15);
```

```
        t.add(5);
        t.add(20);
        t.add(20);
    System.out.println(t);
}}
```

```
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;
    }
}
```

```
TreeSet t = new TreeSet(new MyComparator); ——— (1)
t.add(10);
t.add(0);  +ve ⇒ Compare(0,10)
t.add(15);  -ve ⇒ Compare(15,10)
t.add(5);  +ve ⇒ Compare(5,10)
           -ve ⇒ Compare(5,0)
t.add(20);  -ve ⇒ Compare(20,10);
            -ve ⇒ Compare(20,15);
t.add(20);  -ve ⇒ Compare(20,10).
            -ve ⇒ Compare(20,15)
             0  ⇒ Compare(20,20)
Sopln(t); [20,15,10,5,0]
```

```
Class MyComparator implements
                        Comparable
Public int Compare(Object obj1,
                        Object
{
    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 then internally JVM will call CompareTo() method which is meant for default natural Sorting order. In this case output is [0, 5, 10, 15, 20]
* at line 1 if we are passing Comparator Object then internally JVM will call Compare method which is ment for customized sorting. in this case output is [20, 15, 10, 5, 0]:

Various possible Implementation of Compare Method :-

```
Public int compare(object obj1 , Object obj2){
        Integer I1 = (Integer) obj1;
        Integer I2 = (Integer) obj2;
```

① return I1.compareTo(I2); Default natural sorting order [Ascending order] [0,5,10,15,20]

② return -I1.compareTo(I2); [Descending order] [20, 15, 10, 5, 0]

③ return I2.compareTo(I1); [Descending order] [20,15,10,5,0]

④ return -I2.compareTo(I1); [Ascending order] [0,5,10,15,20]

⑤ return +1; [Insertion order] [10,0,15,5,20,20]

⑥ return -1; [Reverse of Insertion order] [20,20,5,15,0,10]

⑦ return 0; [only first element will be Inserted & all remaining Considered as duplicate] [10]

```
}
```

Example :- Write a program to insert string object into the treeSet where all elements should be inserted according to reverse of alphabatical order.

Solution :-

```
import java.util.*;
class TreeSetDemo2 {
  Public static void main(String[] args) {
  TreeSet t = new TreeSet(new MyComparator());
  t.add("Roja");
  t.add("ShobhaRani");
  t.add("RajaKumari");
  t.add("GangaBhavani");
  t.add("Ramukumma");
  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);
        // return -s1.compareTo(s2);
    }
}
```

Output:- [ShobhaRani, Roja, Ramulamma, RajaKumari, GangaBhavani]

Example:→ Write a program to insert StringBuffer objects into the TreeSet where sorting order is alphabetical order.

Solution:→

```
import java.util.*;
class TreeSetDemo20 {
    public static void main(String[] args) {
        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"));
        System.out.println(t);
    }
}
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return s1.compareTo(s2);
    }
}
```

Output:- [A, K, L, Z]

Note* → If we are depending on default Natural sorting order then compulsory Object should be Homogeneous & Comparable. Otherwise we will get Runtime Exception saying CCE.

* If we are defining our own Sorting by Comparator then object need not be Comparable and Homogeneous that is we can add Heterogeneous non Comparable object also.

Example:- Write a program to insert String & StringBuffer objects into TreeSet where sorting order is increasing length order.

Solution:-

```
import java.util.*;
Class TreeSetDemo$2 {
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("ABCD");
   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 l1 = s1.length();
         int l2 = s2.length();
      if(l1<l2)
         return -1;
      else if(l1>l2)
         return +1;
      else
         return s1.compareTo(s2);
   }
}
```

output:-

[A, AA, XX, ABC, ABCD]

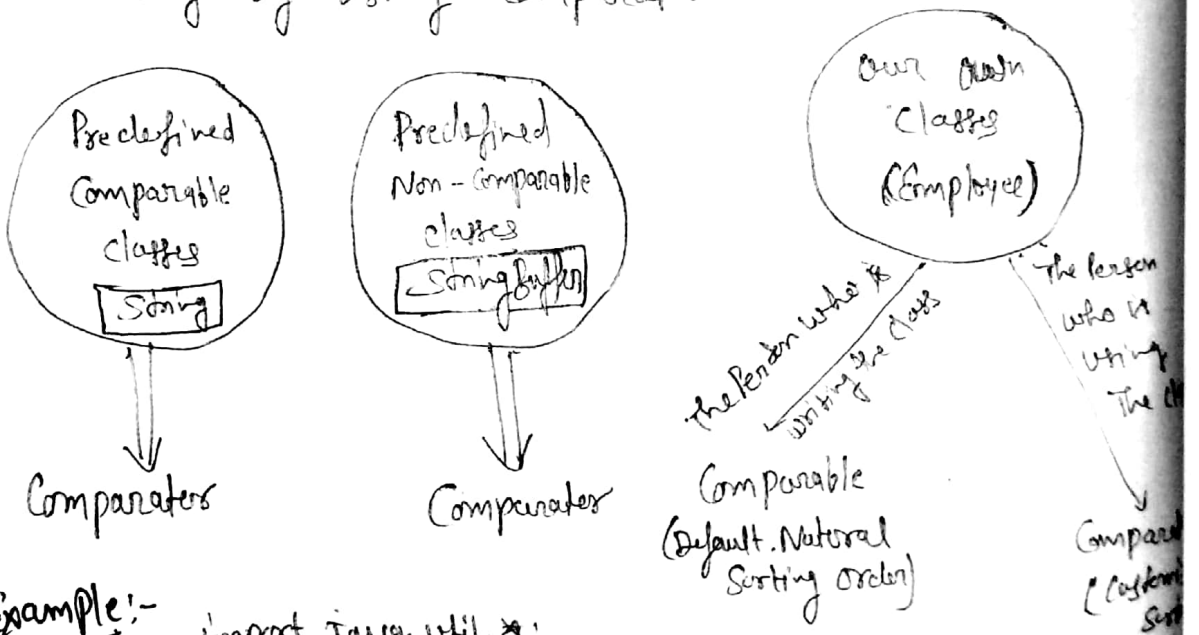
Comparable vs Comparator

(1) for Predefine Comparable classes default natural sorting order already available. If we are not satisfy with that default natural sorting order then we can define our own sorting by using Comparater.

(2) for Predife non Comparable classes (like StringBuffer) default natural sorting order not already available, we can define our own Sorting by using Comparater.

(3) for our own classes like "Employee", the Person who is writing the class is responsible to define default natural sorting order by implementing Comparable interface,

- The Person who is using our class, if he is not satisfy with default sorting order then he can define their own Sorting by using Comparater.



Example:-

```
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("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);
    System.out.println(t);

  TreeSet t1 = new TreeSet(new MyComparator());
        t1.add(e1)
        t1.add(e2)
        t1.add(e3)
```

```
        tl.add(e4);
        tl.add(e3);
        System.out.println(tl);
    }
}

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);
    }
}
```

Output:- [chiru--50, nag--10, venki--150, balaiah--200]
[balaiah--200, chiru--50, nag-100, venki--150]

* Comparision of Comparable & Comparator *

Comparable	Comparator
① It is meant for default natural sorting order	① It is meant for customized sorting order
② present in Java.lang Package	② Present in Java.util Package
③ It define only one method compareTO()	③ It define 2 methods compare(); equals();
④ String & all wrapper classes implement Comparable interface.	④ the only implemented class of Comparator or Collator. RuleBasedCollator (works in Gui Based Applications)

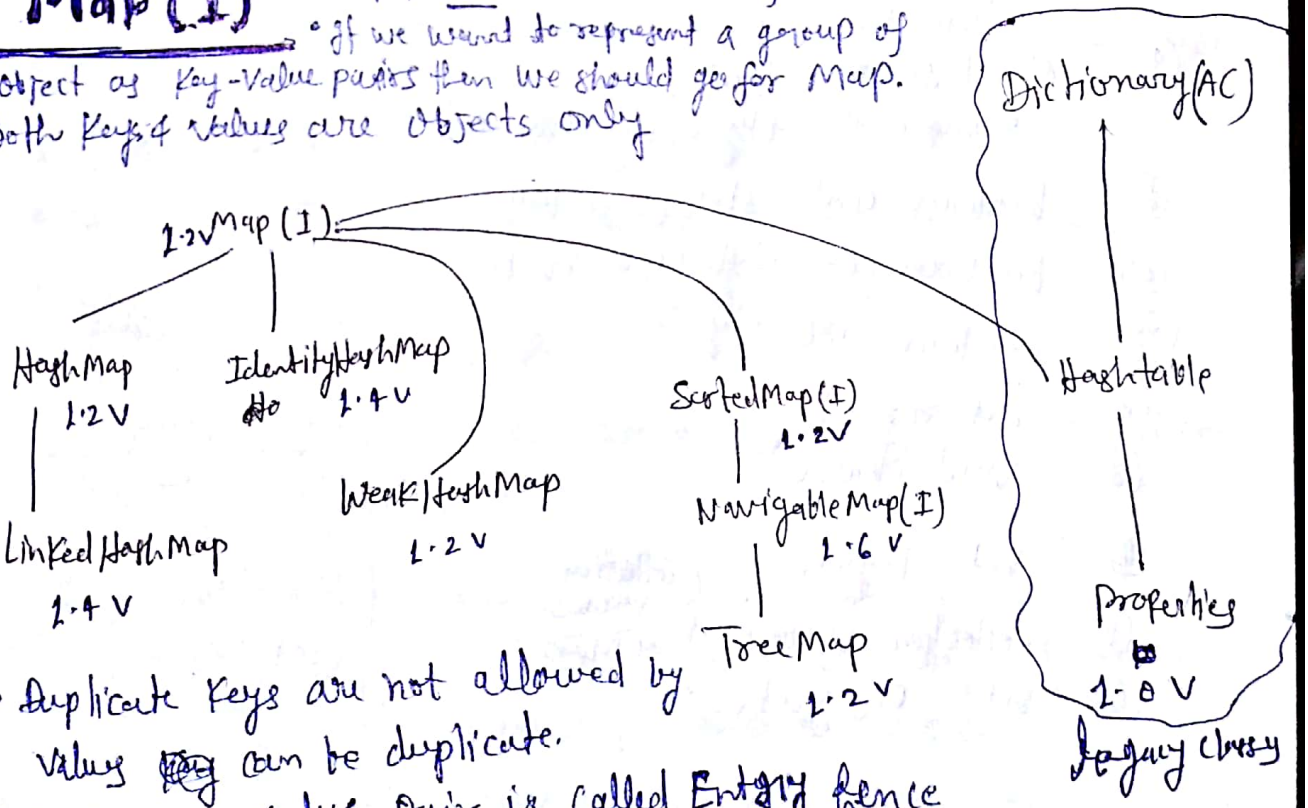
Comparision Table of Set Implemented Classes

Properties	HashSet	LinkedHashSet	TreeSet
1) Underlying Data Structure	HashTable	LinkedList + HashTable	Balanced Tree
2) Duplicate objects	Not allowed	Not allowed	Not allowed
3) Insertion order	Not preserved	Preserved	Not preserved
4) Sorting order	N A	N A	Applicable
5) Heterogeneous objects	Allowed	Allowed	Not Allowed
6) null acceptance	Allowed	Allowed	For Empty TreeSet as First Element null is allowed.

*Note:- For Empty tree as the first element null is allowed but this rule is applicable until 1.6 Version only. from 1.7 V Onwards null is not allowed Even as the first Element.

* Map (I) *

- Map is not child Interface of Collection
- If we want to represent a group of object as key-value pairs then we should go for Map.
- Both Keys & values are Objects only



- Duplicate Keys are not allowed by Values can be duplicate.
- Each key-value pair is called Entry hence Map is considered as A Collection of Entry Objects.

Key	Value
101	durga
102	Ravi
103	Shiva
104	Pavan

→ Entry

Map Interface Methods

① Object put(Object Key, Object Value)

- To add one key-value pair to the Map
- If the key is already present then old value will be replaced with new value and returns old value

Example:-

null — m.put(101, "durga");
null — m.put(102, "Shiva");
durga — m.put(101, "Ravi");

101 – durga / Ravi
102 – Shiva

② void putAll(Map m)
A group of key-value pair will be added

③ Object get(Object key)
returns the value associated with specified key.

④ Object remove(Object key)
remove the entry associated with specified key

⑤ boolean containsKey(Object key)

⑥ boolean containsValue(Object key)

⑦ boolean isEmpty()

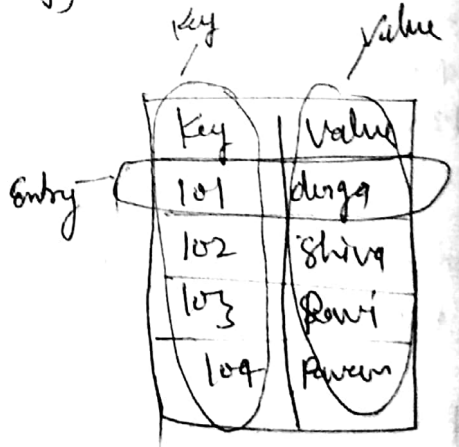
⑧ int size()

⑨ void clear();

⑩ Set keySet() } Collection views of Map

⑪ Collection values() } Collection views of Map

⑫ Set entrySet() } Collection views of Map



Entry (I) :- A Map is a group of key-value pairs and each key-value pair is called an Entry hence Map is considered as a collection of Entry Object. without existing Map object there is no chance of existing entry object hence Entry Interface is define inside Map interface.

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

Entry Specific methods and we can apply only on Entry Object

* HashMap *:-

- The underlying DataStructure is Hashtable
- Insertion order is not preserved and it is based on Hashcode of keys.
- Duplicate keys are not allowed but values can be Duplicated.
- Hetrogeneous objects are allowed for both key and value
- Null is allowed for key (only once)
- Null is allowed for values (any number of times)
- HashMap Implements Serializable & Cloneable Interface but not Randomm Access.
- HashMap is the best choice if our frequent operation is Search operation.

Constructors:-

- HashMap m = new HashMap();
  Creates an Empty HashMap object with Default initial Capacity 16 and Default fill ratio 0.75
- HashMap m = new HashMap(int initialCapacity)
  Creates an Empty HashMap object with specified initial capacity. and default fill ratio 0.75.

- HashMap m = new HashMap(int initialCapacity, float fillRatio);
- HashMap m = new HashMap(Map m);

Example:-

```
Import java.util.*;
  Class HashMapDemo{
    HashMap m = new HashMap()
      m.put("chiranjeevi", 700);
      m.put("balaiah", 800);
      m.put("venkatesh", 200);
       m.put("nagarjuna", 500);
     System.out.println(m) // {K=V, K=V, ---- }
    System.out.println(m.put("chiranjeevi", 1000)); // 700
     Set s = m.keySet();
     System.out.println(s); // [K, K, ---]
     Collection c = m.values();
     System.out.println(c);
      Set s1 = m.entrySet();
       System.out.println(s1); // [K=V, K=V, ---]
       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("nagarjuna"))
              {
               m1.setValue(10000);
              }
          }
           System.out.println(m);
    }
}
```

output :-

{nagarjuna=500, venkatesh=200, balaiah=800, chiranjeevi=700}

700

[nagarjuna, venkatesh, balaiah, chiranjeevi]

[500, 200, 800, 1000]

[nagarjuna=500, venkatesh=200, balaiah=800, chiranjeevi=1000]

nagarjuna 500

venkatesh 200

balaiah 800

chiranjeevi 1000

{nagarjuna=10000, venkatesh=200, balaiah=800, chiranjeevi=1000}

* Differences between HashMap & Hashtable *

HashMap	Hashtable
• Every Method present in HashMap is not synchronized	• Every method present in Hashtable is synchronized
• At a time multiple threads are allowed to operate on HashMap object and hence it is not thread safe	• At a time only one thread is allowed to operate on Hashtable and hence it is thread safe.
• Relatively Performence is high because threads are not required to wait to operate on HashMap object	• Relatively Performence is low because threads are required to wait to operate on Hashtable object
• null is allowed for both Key & value	• Null is not allowed for Keys & values otherwise we will get NullPointerException
• Introduced in 1.2 version and it is not legacy	• Introduced in 1.0 version and it is legacy

* How to get Synchronized version of HashMap object ?

```
HashMap m = new HashMap();
Map m1 = Collections.synchronizedMap(m);
```

m1 ← Synchronized; m → non-synchronized

* By default HashMap is non synchronized but we can get synchronized version of HashMap by using synchronizedMap() Method of Collections class.

LinkedHashMap :-

- It is the child class of HashMap
- It is Exactly Same as HashMap (Including Method & Constructor Excepts the following Differences.

HashMap	LinkedHashMap
• Underlying DataStructure is Hashtable.	• Underlying DataStructure is a Combination of LinkedList + Hashtable (Hybbrid Datastructure).
• Insertion Order is not Preserved & it is based on HashCode of Keys	• Insertion order is preserved.
• Intro-duced in 1.2 V	• Introduced in 1.4V

* in the above HashMap Program if we Replace HashMap with LinkedHashMap then output is:

```
{Chiranjeevi = 700, balaiah = 800, venkatesh = 200,
 nagarjuna = 500}
```

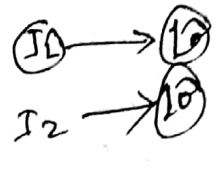
that is insertion order is preserved.

Note:- LinkedHashSet & LinkedHashMap are Commonly use for Developing Cache based applications.

Difference Between == operator & .equals() Method.

- In general == operator meant for reference comparision (Address comparision) where .equals() method meant for content comparision.

```
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);

Sopln(I1 == I2);  // false
Sopln(I1.equals(I2); // true
```



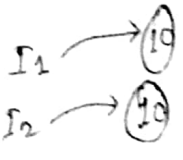
Identity HashMap:-

It is exactly same as HashMap (including method & constructor) except the following Difference:-

- In the case of Normal HashMap JVM will use .equals() method to identify duplicate keys, which is meant for content comparision.
- But in the case of IdentityHashMap JVM will use == operator to identify Duplicate keys which is meant for reference comparision (Address comparision.

Example:→

```
HashMap m = new HashMap();
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);

m.put(I1, "Pawan");
m.put(I2, "Kalyan");

Sopln(m); // {10 = Kalyan}
```



- I1 & I2 are Duplicate keys because I1.equals(I2) returns 'true'.
- If we replace HashMap with IdentityHashMap then I1 & I2 are not Duplicate keys because I1 == I2 returns 'false'. in this case output is

output:- {10 = Pawan, 10 = Kalyan}

WeakHashMap:-

- It is exactly same as HashMap except the following difference:-
  - → In the case of HashMap even though object doesn't not have any references it is not eligible for GC. if it is associated with HashMap. that is HashMap Dominate Garbage Collector.
  - → But in the case of WeakHashMap, if Object Does' not Contain any refferences it is eligible for GC. Eventhough object associated with WeakHashMap. that is Garbage Collecter Dominate WeakHashMap.

Example.

```
import java.util.*;
class WeakHashMapDemo {
Public static void main(String [] args) throws Except
{
  HashMap m = new HashMap();
     Temp t = new Temp();
       m.put(t, "durga");
     System.out.println(m);
     t = null;
     System.gc();
     Thread.sleep(3000);
     System.out.println(m);
} }

Class Temp {
   Public String toString() {
        return "temp";
   }
  Public void finalize()
   {
   System.out.println("finalize method Called");
   }
}
```

- In the above Example Temp object not eligible for GC. because it is associated with HashMap. In this case output is:-

{temp = durga}

{tem = durga}

- In the above program if we replace HashMap with WeakHashMap then Temp eligible for G.C. In this case output is:-

{temp = durga}

finalize method called

{}

SORTEDMap (I):-

- It is the child interface of Map.
- If we want to represent A Group of key-value pair according to some sorting order of keys then we should go for Sorted Map.
- Sorting is based on the key but not based on value.
- SortedMap Defines the following specifices Methods:-
  - → Object firstKey();
  - → Object LastKey();
  - → SortedMap headMap(Object key);
  - → SortedMap tailMap(Object key);
  - → SortedMap subMap(Object key1, Object key2);
  - → Comparator comparator();

Example:-

firstKey() ⟶ 101

lastKey() ⟶ 136

headMap() → {101=A, 103=B, 104=C}

tailMap() → {107=D, 125=E, 136=F}

subMap() ⟶ {103=B, 104=C, 107=D}

Comparator() ⟶ null.

101 → A
103 → B
104 → C
107 → D
125 → E
136 → F

TreeMap :→

- The underlying DataStructure is Red-Black Tree.
- Insertion order is not preserved and it is based on Same sorting order of keys.
- Duplicate keys are not allowed but values can be duplicated.
- If we are depending on default Natural sorting order then keys should be Homogeneous & Comparable otherwise we will get RuntimeException saying ClassCastException.
- If we are defining our own Sorting by Comparator then keys need not be Homogeneous & Comparable, we can take heterogeneous non comparable object also.
- Whether we are depending on default natural sorting order or Customize sorting order there are no restriction for value, we can take heterogenous non comparable object also.

~~null accep~~

Null Acceptence :-

- For Non Empty treemap if we are trying to insert an Entry with Null key then we will get RuntimeException saying NullPointerEx.
- For Empty Treemap as the first Entry with Null key is allow. but after inserting that Entry if we are trying to insert other Entry then we will get RuntimeException saying NullPointerException.

Note:- The above null acceptence rule applicable until 1.6 version only from 1.7 V. onwards Null is not allowed for key.

- But for values we can use null any number of times there is no restriction whether it is 1.6V or 1.7 V.

Constructors :-

- TreeMap t = new TreeMap();
  for default Natural Sorting order
- TreeMap t = new TreeMap(Comparator c);
  for Customized Sorting order

- TreeMap t = new TreeMap(SortedMap m);
- TreeMap t = new TreeMap(SortedMap m);

Demo Program for Default Natural Sorting order :-

```
import java.util.*;
Class TreeMapDemo3 {
    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("FFF", "XXX"); // CCE
       // m.put(null, "XXX") // NPE
      System.out.println(m);
            // {100=ZZZ, 101=XXX, 103=YYY, 104=106}
} }
```

Demo program for Customized Sorting :-

```
import Java.util.*;
Class TreeMapDemo{
public static void main(String args[]){
TreeMap t = new TreeMap(new MyComparator());
      t.put("XXX", 10);
      t.put("AAA", 20);
       t.put("ZZZ", 30);
       t.put("LLL", 40);
     System.out.println(t);
      // {ZZZ=30, XXX=10, LLL=40, AAA=20}
   }
}
Class MyComparator implements Comparator{
      Public int Compare(Object o1, Object o2) {
```

```
        String s1 = o1.toString();
        String s2 = o2.toString();
        return s2.compareTo(s1);
    }
}
```

HashTable :-

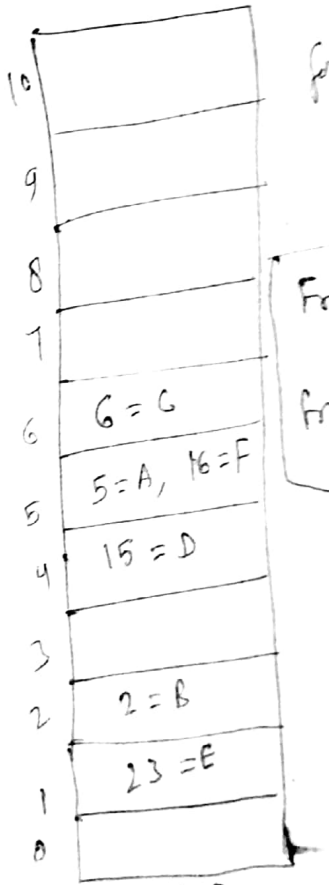
- Underlying DataStructure for Hashtable is Hashtable.
- Insertion order is not preserved and it is based on hashCode of Keys.
- duplicate Keys are not allowed and Values can be duplicated.
- Heterogeneous objects are allow for both Keys & Values
- Null is not allow for both Keys & Values otherwise we will get RuntimeException saying NullPointerException
- It implements Serializable & Cloneable interfaces but not RandomAccess.
- Every Method Present in Hashtable is Synchronized and hence Hashtable object is ThreadSafe.
- Hashtable is the best choice if our frequent operation is Search operation.

Constructors :-

- Hashtable h = new Hashtable();
  Create an Empty Hashtable object with default initial capacity 11 and default fill ratio 0.75.
- Hashtable h = new Hashtable(int initialCapacity);
- Hashtable h = new Hashtable(int initialCapacity, float fillRatio);
- Hashtable h = new Hashtable(Map m);

Example :→

```
import java.util.*;
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); // NPE
    System.out.println(h);
  }
}

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



formate to print output

From Top to Bottom From Right to Left

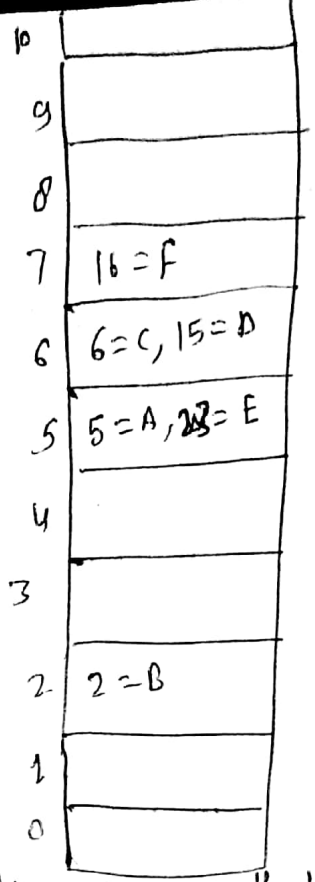
output → { 6 = C , 16 = F, 5 = A, 15 = D, 2 = B, 23 = E }

→ If we change hashCode method of temp class as

```
public int hashCode(){
    return i%9;
}
```

output:-

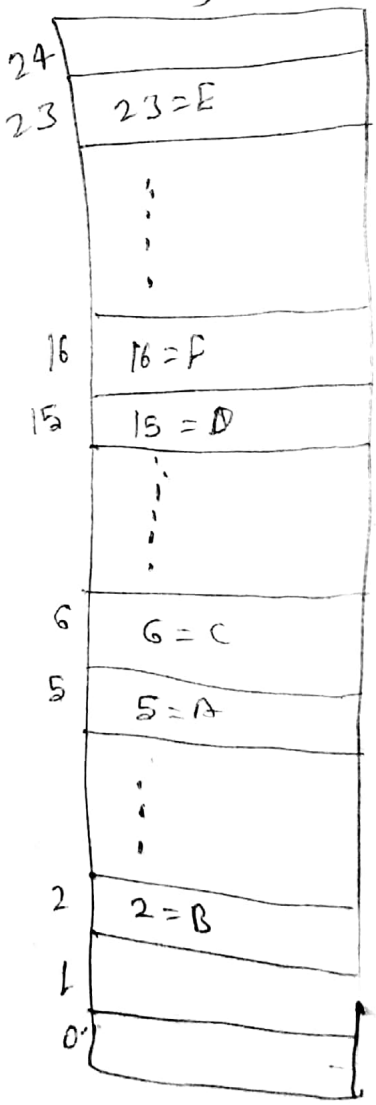
{16=F, 15=D, 6=C, 23=E,
5=A, 2=B}



→ If we configure initial capacity as 25 that is :
Hashtable h = new Hashtable(25);

output:-

{23=E, 16=F, 15=D, 6=C,
5=A, 2=B}



Properties :-

- In our program if anything which changes frequently like (UserName, Password, mailId, mobileNo. etc) or not recommended to hardcode in java program because if there is any change to reflect that change, recompilation, rebuild and redeployee application are required even sometime Server restart also required which create is big business impact to the client.
- We can overcome this problem by using Properties file. Such type of variable things we have to configure in the Properties file.
- From that Properties file we have to read into java program and we can use those properties. the Main advantage of this approach is if there is a change in Properties file to reflect that change just redeployement is Enough which will not create any business impact to the client.
- We can use java Properties Object to hold Properties which are coming from Properties file.

Imp
- In Normal Map (like HashMap, Hashtable, TreeMap) Key and Value can be any type but in the case of Properties Key and Value should be String type.

Constructor :-

```
Properties p = new Properties();
```

Methods :-

① String setProperty(String pname, String pvalue);
- to set the new property
- if the specified Property already available then old value will be replaced with new value and returns old value.

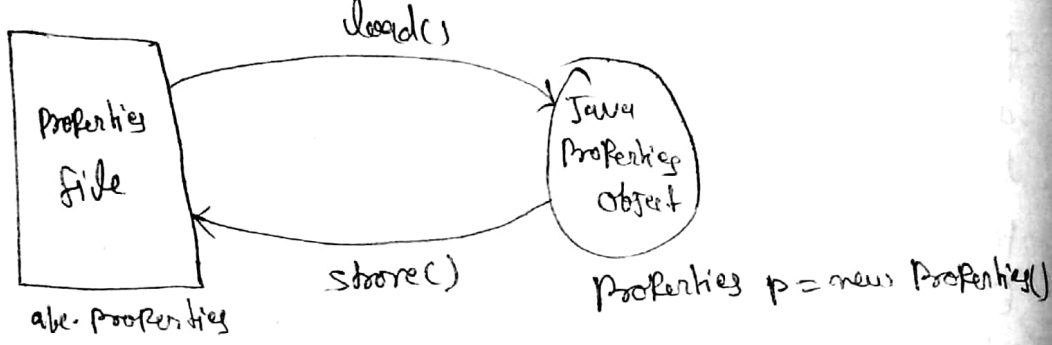
② String getProperty(String pname);
- to get value associated with the specified Property

- If the specified property not available then this method return null.

③ Enumeration propertyNames();
- returns all property Names present in property object.

④ Void load(InputStream is)
- to load properties from properties file into Java property object.

⑤ Void store(OutputStream os, String comment)
- to store properties from Java properties object into properties file.



Example:-

abc.properties

```
user = Scott
Pwd = tiger
venki = 9999
```

```
import java.util.*;
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); // {PN=PV, ....}
String S = P.getProperty("venki"); 9999
System.out.println(S); // 9999
P.setProperty("nag", "8888");
FileOutputStream fos = new FileOutputStream("abc.prop
```

p.store(fos, "updated by Durga for SCJP Demo");

}

}

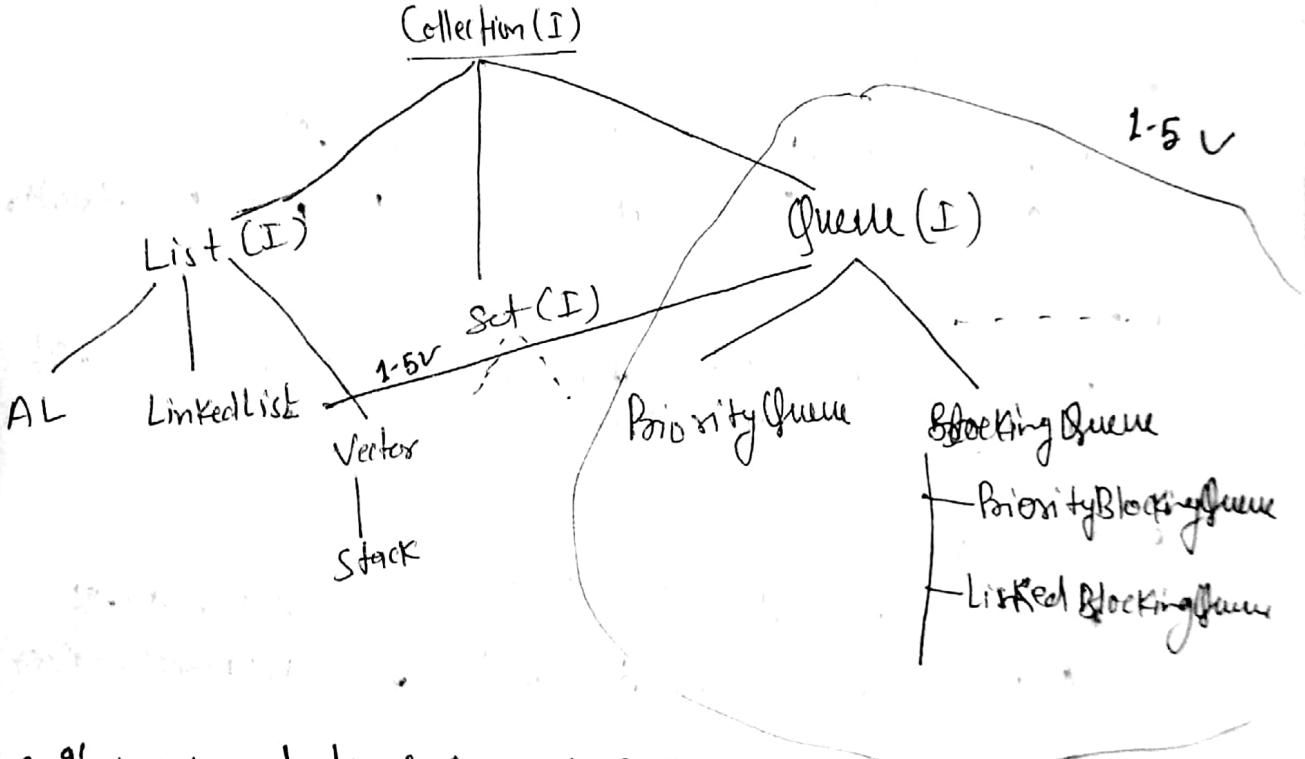
output:-

```
# updated by Durga for SCJP Demo Class
# Sun Sep 21 19:04:39 IST 2014
nag = 888888
user = Scott
venkat = 9999
pwd = tiger
```

abc.properties

*1.5 Version Enhancement (Queue Interface)

- 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.
- for example before sending SMS message all mobile numbers we have to store in some data structure.
- In which order we added mobile number in the same order only msg should be deliver. for this first in first out requirent Queue is the best choice.
- usually Queue follows first in first out order but based on our requirent we can implement our own priority order also.

(Priority Queue).

- from 1.5 v onwards LinkedList class also implements Queue interface.
- LinkedList based implementation of queue always follows First in first out order.

Queue interface specifics methods

- boolean offer(Object o)
  to add an object into the queue.
- Object peek()
  to return head element of the queue. If the queue is empty then this method returns null.
- Object element()
  to return head element of the Queue. If Queue is Empty then this method raises RE: NoSuchElementException
- Object poll()
  to remove and return head element of the queue. If Queue is empty then this's method returns null.
- Object remove()
  to remove and return head element of the queue. If queue is empty then this method raises RE: NoSuchElementExcep

Priority Queue :-

- If we want to represent a group of individual object prior to processing according to some priority then we should go for priority queue.
- The priority can be either default ~~Default~~ natural sorting order or customized sorting order defined by Comparator.
- Insertion order is not preserved and it is based on some priority

- duplicate objects are not allowed
- if we are depending on default natural sorting order compulsory Object should be homogeneous & comparable otherwise we will get RE: saying ClassCastException.
- If we are defing our own sorting by Comparator then objects need not be Homogeneous & comparable.
- Null is not allowed even as the first element also.

Constructors :-

- PriorityQueue q = new PriorityQueue();
  → creates and Empty Priority Queue with default initial capacity 11 & all objects will be inserted according to default Natural sorting order.
- PriorityQueue q = new PriorityQueue(int initialCapacity);
- PriorityQueue q = new PriorityQueue(int initialCapacity, Comparator c);
- PriorityQueue q = new PriorityQueue(SortedSet s);
- PriorityQueue q = new PriorityQueue(Collection c);

Example:-

```
import java.util.*;
Class PriorityQueueDemo
{
  Public Static Void main(String[] args)
  {
     PriorityQueue q = new PriorityQueue();
    System.out.println(q.peek()); // null
    // System.out.println(q.element()); //RE: NSEE
    for(int i=0; i<=10; i++)
    { q.offer(i);
    }
    System.out.println(q); // [0, 1, 2 .... 10]
    System.out.println(q.poll); // 0
    System.out.println(q); // [1, 2, 3 ..... 10]
  }
}
```

Note:- Some platform will not provide proper suppor for thread priority, Priority Queue.

Example:- Demo program for Customize Priority:-

```
import java.util.*;
Class PriorityQueueDemo2
{
 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,L,B,A]
  }
}

  Class MyComparator implements Comparator
   {
      Public int Compare(Object obj1, Object obj2)
      {
        String s1 = (String) obj1;
        String s2 = (String) obj2;
         return s2.compareTo(s1);
      }
  }
```

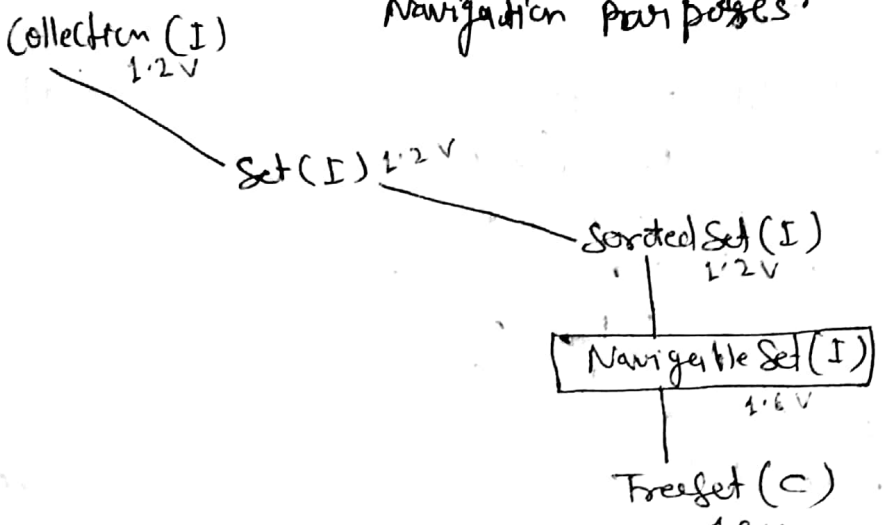
1.6 version Enhancement in Collection Framework

As the Part of 1.6 version the following 2 concepts introduced in ~~1.5 version~~ Collection framework.

1. NavigableSet (I)
2. NavigableMap (I)

NavigableSet (I) :-

It is the child interface of SortedSet and it define several method for Navigation purposes.



NavigableSet defines the following Methods :-

- floor (e)
  → It returns highest element which is <= e
- lower (e)
  → It returns highest element which is < e
- ceiling (e)
  → It returns highest element which is >= e
- higher (e)
  → It returns lowest element which is > e
- PollFirst()
  → remove & return first element
- PollLast()
  → remove & return last element
- descendingSet()
  → It returns NavigableSet in reverse order.

Example:

```
import java.util.*;
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); [1000, 2000, 3000, 4000, 5000]
    System.out.println(t.ceiling(2000)); // 2000
    System.out.println(t.higher(2000)); // 3000
    System.out.println(t.floor(3000)); // 3000
    System.out.println(t.lower(3000)); // 2000
    System.out.println(t.pollFirst()); // 1000
    System.out.println(t.pollLast()); // 5000
    System.out.println(t.descendingSet()); // [4000, 3000, 2000]
    System.out.println(t); // [2000, 3000, 4000]
  }
}
```

NavigableMap (I):-

- NavigableMap 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

↓

TreeMap 1.2V

Navigable Map Defines following Methods :-

- floorKey(e)
- lowerKey(e)
- CeilingKey(e)

- higherKey(e)
- pollFirstEntry()
- pollLastEntry()
- descendingMap()

Example :-

```
import java.util.*;
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); // {a = apple, b = banana, c = cat, d = dog
                                                  g, = gun}
System.out.println(t.ceilingKey("c")); //c
System.out.println(t.higherKey("e")); // g
System.out.println(t.lowerKey("e"); // d
System.out.println(t.pollFirstEntry()); // a = apple
System.out.println(t.pollLastEntry()); // g = gun
System.out.println(t.descendingMap());
                // {d = dog, c = cat, b = banana}

System.out.println(t);
    // {b = banana, c = eat, d = dog}
}
}
```

Collections:-

* Collections class define several utility methods for collection Object like Sorting, Searching, reversing etc.

Sorting Elements Of List:-

Collections class define the following two 'sort' methods

(1) Public Static Void sort(List l);

- To sort based on Default Natural sorting order.
- In this case List should compulsory contains Homogeneous & Comparable Objects. otherwise we will get RE 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.

Demo Program for sorting Elements of List Accordig DNSO

(Default Natural Sorting order)

Example:-

```
import java.util.*;
Class CollectionSortDemo {
    Public Static Void main(String[] args) {
        ArrayList l = new ArrayList();
        l.add("Z");
        l.add("A");
        l.add("K");
        l.add("N");
        // l.add(new Integer(10)); // ---.CCE
        // l.add(null); // NPE
        System.out.println("Before Sorting:"+l);
                                    // [Z, A, K, N]
        Collections.sort(l);
        System.out.println("After Sorting:"+l);
                                    //[A, K, N, Z]
    }
}
```

Example:- Demo program to Sort Elements of list according to Customized sorting:-

```
import java.util.*;
class CollectionSortDemo2{
    public static void main(String[] args){
        ArrayList l = new ArrayList();
        l.add("Z");
        l.add("A");
        l.add("K");
        l.add("L");
        System.out.println("Before Sorting:"+l); // [Z, A, K, L]
        Collections.sort(l, new MyComparator());
        System.out.println("After Sorting:"+l); // [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);
    }
}
```

Output:

Before Sorting: [Z, A, K, L]

After Sorting: [Z, L, K, A]

Searching elements of List:-

Collections Class define the following Binary search methods:

① Public static int binarySearch(List l, Object target);
- If the list is sorted according to default natural sorting order then we have to use this method.

② Public static int binarySearch(List l, Object target, Comparator c);
- If the list is sorted according to customized sorting order then we have to use this method.

Conclusions:-

① The above search methods internally will use Binary Search Algorithm.

② Successfull Search returns index;

③ Unsuccessfull Search returns insertion Point.

④ insertion point is the location where we can place target element in sorted list.

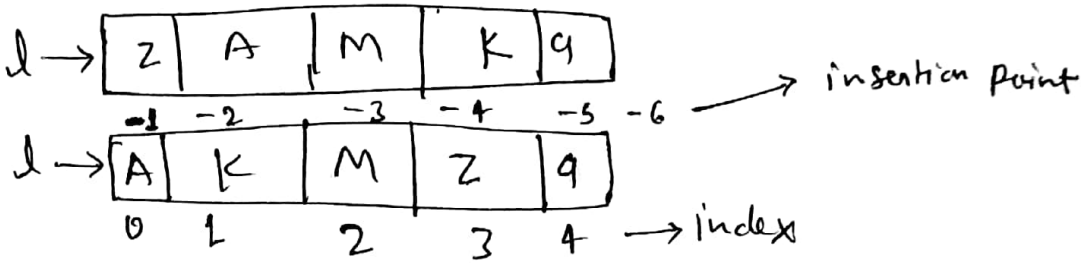
⑤ before calling BinarySearch method compulsory list should be sorted otherwise we will get Unpredictible Results.

⑥ if the list is sorted according to Comparator then at the time of search operation also we have to pass same Comparator object other wise we will get unpredictable results.

Example:-

```
import java.util.*;
Class CollectionSearchDemo{
  Public static void main(String[] args)
  {
    ArrayList l= new ArrayList();
```

```
l.add("Z");
l.add("A");
l.add("M");
l.add("K");
l.add("a");
System.out.println(l); // [Z, A, M, K, a]
Collections.sort(l)
System.out.println(l); // [A, K, M, Z, a]
System.out.println(Collections.binarySearch(l, "Z")); // 3
System.out.println(Collections.binarySearch(l, "J")); // -2
}
}
```

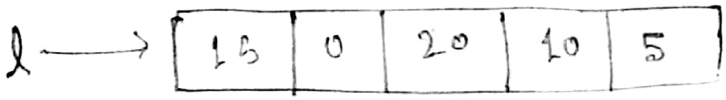


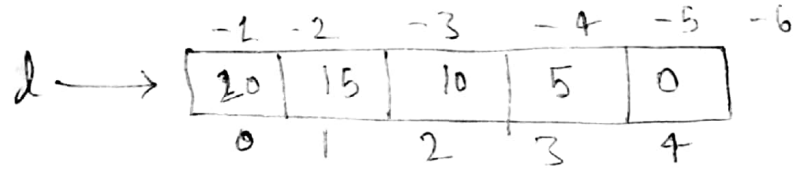

```
Collections.binarySearch(l, "Z"); // 3
Collections.binarySearch(l, "J"); // -2
```

Example :-

```
import java.util.*;
class CollectionSearchDemo1 {
  Public Static void main(String[] args)
  {
   ArrayList l = new ArrayList();
    l.add(15);
    l.add(0);
    l.add(20);
    l.add(10);
    l.add(5);
  System.out.println(l); // [15, 0, 20, 10, 5]
   Collections.sort(l, new MyComparator());
    System.out.println(l); // [20, 15, 10, 5, 0]
  System.out.println(Collections.binarySearch(l, 10, new
                          MyComparator())); // 2
```

```
System.out.println(Collections.binarySearch(l, 13, new
                                    MyComparator()); // -3
System.out.println(Collections.binarySearch(l, 17)); //
                                    unpredictable
    }
}

Class MyComprater implements Comparator
 {
   Public int compare( Object obj1, Object obj2)
     {
        Integer i1 = (Integer) obj1;
        Integer i2 = (Integer) obj2;
      return i2.compareTo( i1);
     }
 }
```



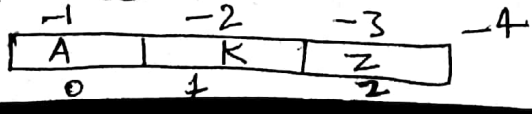



```
Collections.binarySearch(l, 10, new MyComparator()); //2
Collections.binarySearch(l, 13, new MyComparator()); //-3
Collections.binarySearch(l, 17); // unpredictable
Collection.binarySearch(l, 17, new MyComparator()); // -2
```

Note:- for the list of N Element in the case of Binary Search Method:-

1. Successful search Result Range 0 to n-1
2. Unsuccessful search Result Range: -(n+1) to -1
3. Total Result Range :- -(n+1) to (n-1)

Example:- For 3 Elements



Successful Result Range :- 0 to 2
Unsuccessful Result Range :- -4 to -1
Total Result Range :- -4 to 2

Reversing Elements of List :-

Collections class define following reverse method to reverse elements of list.

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

Example :-

```
import java.util.*;
class CollectionsReverseDemo
{
  Public static void main (String [] args)
  {
    ArrayList l = new ArrayList();
        l.add (15);
        l.add (0);
        l.add (20);
        l.add (10);
        l.add (5);
    System.out.println(l); // [15, 0, 20, 10, 5]
    Collections.reverse(l);
    System.out.println(l); // [5, 10, 20, 0, 15]
  }
}
```

reverse() vs reverseOrder() :-

- We can use reverse() method to reverse order of elements of list.
- Where we can use reverseOrder() method to get reversed comparator.

Comparator c = Collections.reverseOrder(Comparator c)

Descending order — Ascending order

Arrays :→

Arrays class to define several utility methods for array object.

① Sorting Elements of Array:

Arrays class define the following Sort methods to Sort Elements of Primitive & object type Arrays.

- Public static void Sort(primitive[] P)
  → To sort according to natural sorting order
- Public static void sort(Object[] O)
  ⇒ To sort according to Natural sorting order
- Public static void sort(Object[] O, Comparator C)
  → To sort according to Customized sorting order.

Example:-

```
import java.util.Arrays;
import java.util.Comparator;
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); // 10, 5, 20, 11, 6

    Arrays.sort(a);
    System.out.println("Primitive Array After sorting");
    for(int a1: a){
  } System.out.println(a1); // 5, 6, 10, 11, 20
```

```
String[] s = {"A","Z","B"};
System.out.println("Object Array Before Sorting:");
for (String q2 : s){
    System.out.println(q2); // A,Z,B
}
Arrays.sort(s);
System.out.println("Object Array After Sorting:");
for (String q2 : s){
    System.out.println(q2); // A,B,Z
}
Arrays.sort(s, new MyComparator());
System.out.println("Object Array After Sorting by
                    Comparator:");
for (String q2 : s){
    System.out.println(q2); Z,B,A
}
}
}
class MyComparator implements Comparator{
    Public int Compare(Object obj1, Object obj2)
    {
        String s1 = obj1.toString()
        String s2 = obj2.toString()
        s2.compareTo(s1);
    }
}
```

Note:- We can Sort primitive Arrays only based on Default Natural Sorting order where as we can sort Object Array Either based on Default Natural Sorting order or based on Customized Sorting order.

Searching elements of Arrays:-

Arrays class defines the following BinarySearch methods:

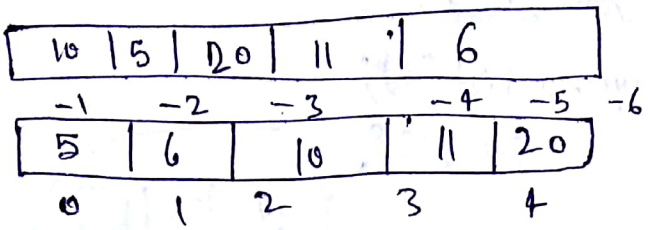
- Public static int binarySearch(Primitive[] p, Primitive target);
- Public static int binarySearch(Object[] a, Object target);
- Public static int binarySearch(Object[] a, Object target, Comparator c);

Note:- All flavors of Arrays class BinarySearch methods are exactly same as Collections class Binary search methods.

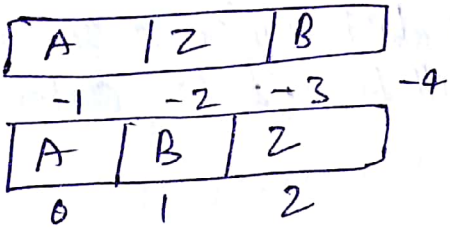
Example:-

```
import java.util.*;
import static java.util.Arrays.*;
class ArraySearchDemo
{
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()));
System.out.println(binarySearch(s,"S", new MyComparator())); // 0
System.out.println(binarySearch(s,"A" [illegible] MyComparator())); // -2
                                    // Unpredictable result
}}
```

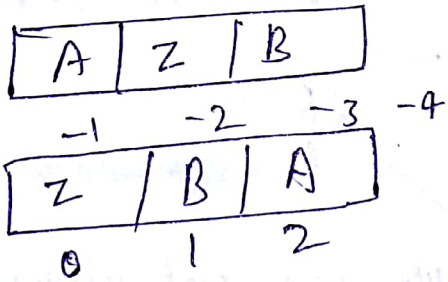
```
class MyComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        String s1 = o1.toString();
        String s2 = o2.toString();
        return s2.compareTo(s1);
    }
}
```




```
Arrays.binarySearch(a, 6); //1
Arrays.binarySearch(a, 14); //-5
```




```
Arrays.binarySearch(s, "Z"); 2
Arrays.binarySearch(s, "S"); -3
```



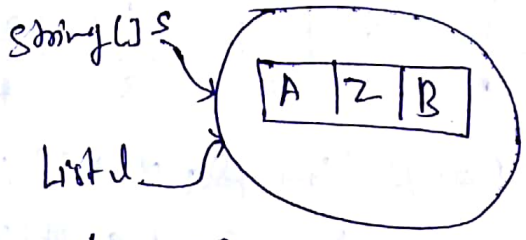

```
binarySearch(s, "Z", new MyComparator()); //0
binarySearch(s, "S", new MyComparator()); //-2
binarySearch(s, "N"); //unpredictable result
```

Conversion of Array to List :-

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

- Strictly speaking this will not create an independent List object for the existing Array we are getting List view.

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



- By using Array reference if we perform any change automatically that change will be reflected to the ~~array~~ List. similarly by using List reference if we perform any change that change will be reflected automatically to the array.

- By using List reference we can't perform any operation which alters its size. otherwise we will get RuntimeException saying UnsupportedOperationException

```
l.add("M");      }
l.remove(1);     }  → RE; UnsupportedOperationException
l.set(1,"N");    ✓
```

- By using List reference we are not allowed to replace with heterogenous objects otherwise we will get RuntimeException saying ArrayStoreException.

```
l.set(1, new Integer(10));  RE: ArrayStoreException
```

Example:-

```
import java.util.*;
class ArrayAsListDemo
{
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 sl : s)
     {
      System.out.println(sl); // K,L,B
     }
   // l.add("durga"); // UsOE (UnsupportedOperationException)
   // l.remove(2); // UsOE
   // l.set(1, new Integer(10)); // ArrayStoreException
}
}
```