2/02/11

Collection frame work

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

* Limitations of object arrays :-

→(1) Arrays are fixed in Size. i.e, once we Created an array there is no chance of increasing or decreasing Size based on our requirement. Hence, to use arrays Concept Compulsory we should know the Size in advance, which may not possible always.

(2) Arrays Can hold only Homogeneous data elements. i.e, (Same type)

ex:-

```
Student[] S = new Student[1000];
        S[0] = new Student[]; ✓
        S[1] = new Student[]; ✓
        S[2] = new Customer[]; ✗  c.e:- Incompatiable types
                                      found : Customer
                                      required : Student.
```

↳ But we Can resolve this problem by using object-type arrays.

ex!-

```
Object[] a = new Object[1000];
        a[0] = new Student[]; ✓
        a[1] = new Customer[]; ✓
```

(3) Arrays Concept not built based on Some datastructure. Hence readymed method support is not available for every requirement. Compulsory programmer is responsible to write the logic.

→ To resolve the above problems SUN people introduced Collections Concept.

→ Advantages of Collections over arrays :-

(1) Collections are growable in nature. Hence based on our requirement we can increase or decrease the size.

(2) Collections Can hold both Homogeneous & Heterogeneous objects.

(3) Every Collection class is implemented based on some datastructures. Hence readymed method support is available for Every requirement.

dis. of Collections:-

→ Performance point of view Collections are not recommended to use. This is the Limitation of Collections.

difference b/w arrays & Collections:-

Array	Collections(AL, VL, LL - - - ..)
1) Arrays are fixed in size	1) Collections are growable in nature
2) Memory point of view arrays Concept is not recommended to use	2) memory point of view Collections Concept is highly recommended to use.
3) performance point of view arrays Concept is highly recommended to use.	3) performance point of view Collections is not recommended to use.
4) Arrays Can hold only homogeneous data elements	4) Collections Can hold both Homogeneous & Heterogeneous objects.
5) There is no underlying d.s for arrays. Hence readymed method support is not available	5) underlying D.S is available for every Collection Class. Hence readymed method Support is available.

→ Arrays Can be used to hold both premitives & objects.	→ Collections Can be used to hold only objects but not for premitives.

Collection :-

→ A group of individual objects as a Single entity is called Collection

Collection framework :-

→ It defines Several classes & Interfaces, which Can be used to represent a group of objects as a Single Entity.

Terminology :-

Java	C++
Collection →	Container
Collection framework →	STL (Standard Template Library)

9-Key interfaces of Collection frame work :-

① Collection (Interface) :

→ 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 frame work.

→ Collection Interface defines the most Common methods which Can be applicable for any Collection object.

30 Collection vs Collections :-

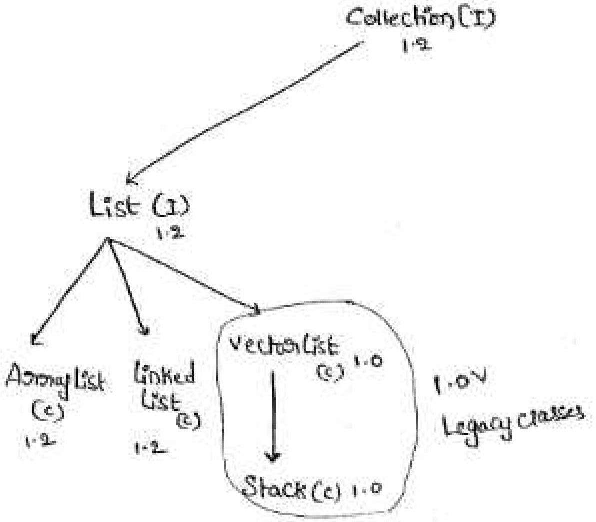
→ Collection is an interface, Can be used to represent a group of individual object as a Single Entity. where as

→ Collections is an Utilityclass, present in java.util package, to define Several Utility methods for Collections.

31 List (Interface) :-

→ It is the child interface of Collection.

→ If we want to represent a group of individual objects where insertion order is preserved & duplicates are allowed. Then we should go for List.

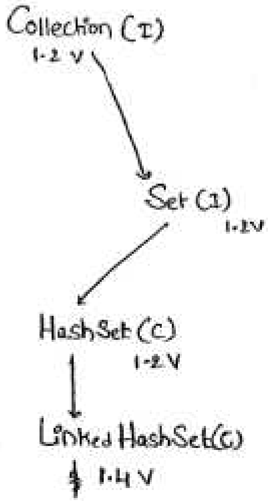


→ Vector & Stack Classes are re Engineered in 1.2 version to fit into Collection framework.

③ Set (Interface):-

→ It is the child interface of Collection.

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



④ SortedSet (I):-

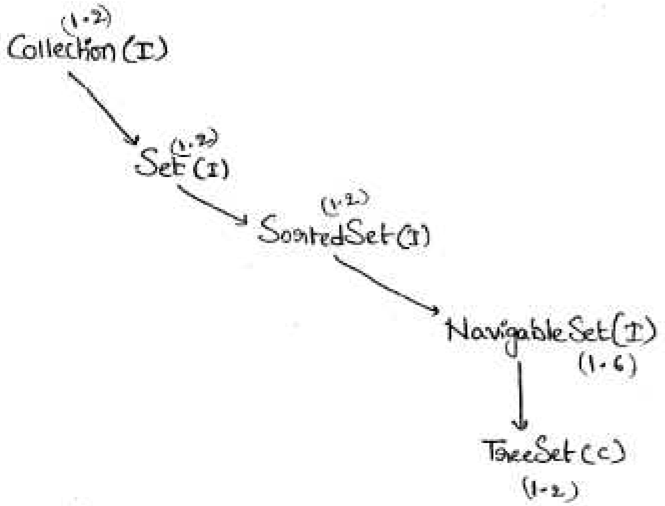
→ It is the child interface of Set.

→ If we want to represent a group of ~~unique~~ individual objects, according to some sorting order then we should go for SortedSet.

⑤ NavigableSet (I):-

→ It is the child interface of SortedSet, to provide several methods for Navigation purposes.

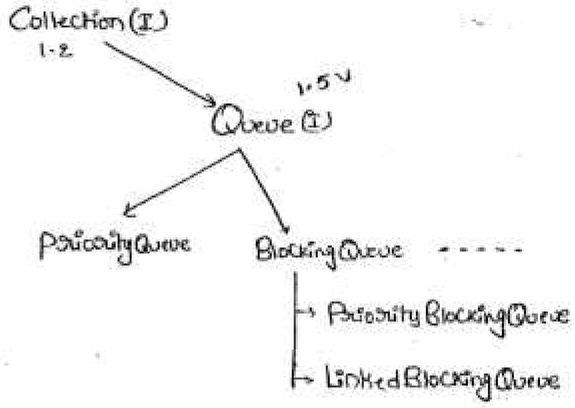
→ It is introduced in 1.6Version.



⑥ Queue (I) :- (1.5V)

→ 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 the above Interfaces (Collection, List, Set, SortedSet, NavigableSet, Queue) ment for representing a group of individual objects.

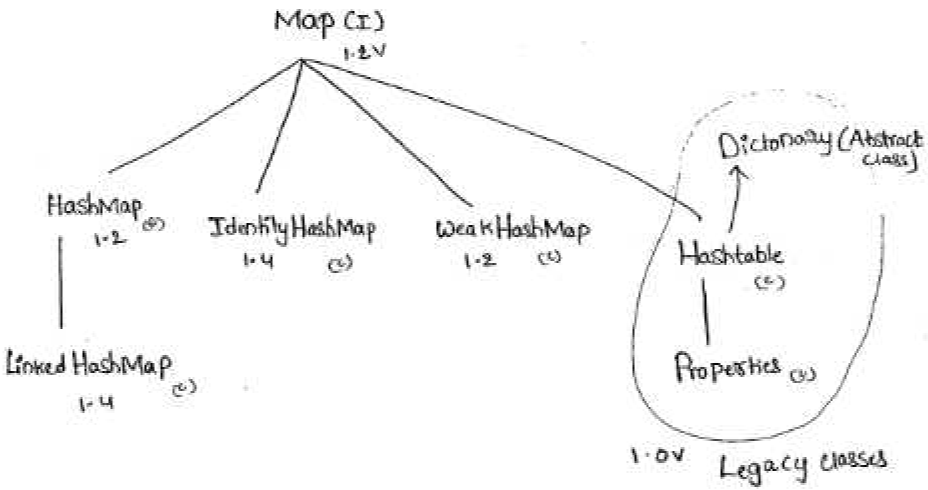
→ If we want to represent a group of objects as key-value pairs then we should go for 'Map'.

(7) Map(I):-

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

→ Both Key & value are objects only.

→ duplicate Keys are not allowed. But values Can be duplicated.



Note:-

→ **Map is not child Interface of Collection.

(8) SortedMap (I):-

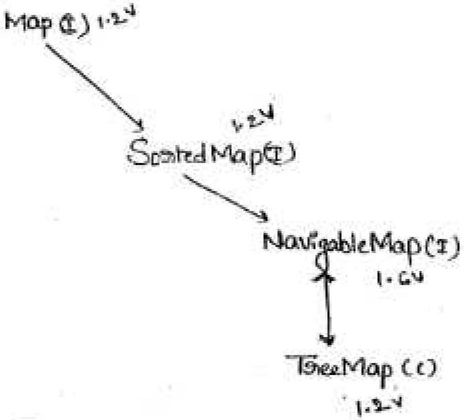
→ If we want to represent a group of objects as Key-value pairs according to Some Sorting Order. then we Should go for SortedMap.

→ Sorting should be done only based on Keys. but not based-on values.

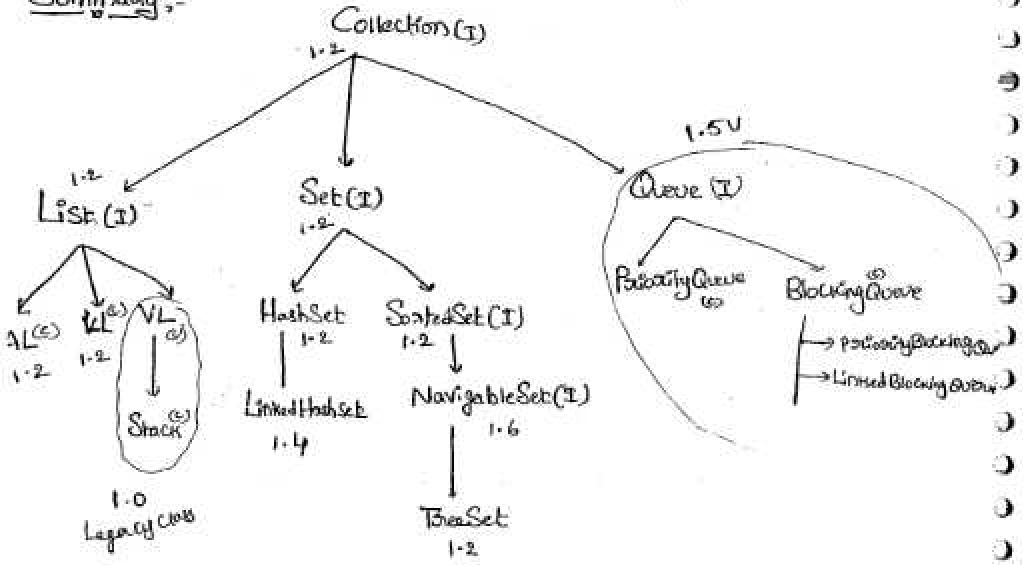
→ SortedMap is child Interface of Map.

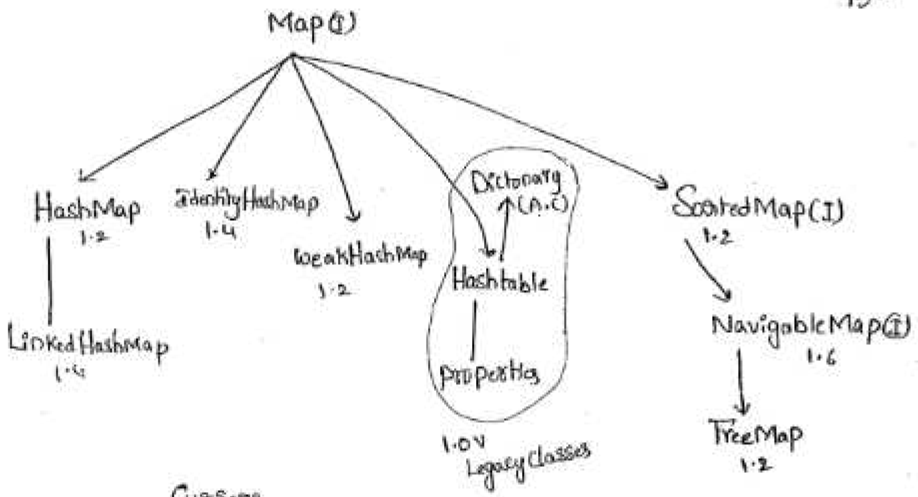
(9) NavigableMap (I):-

→ It is The child Interface of SortedMap & define Several methods for Navigation purposes.



Summary:-





Arrays
Collections

Cursors
Enumeration
Iterator
ListIterator

Comparable
Comparator

→ In the Collection frame work the following are Legacy characters

(1) Enumeration (I)

(2) Dictionary (Ab c)

(3) Vector

(4) Stack

(5) Hashtable } classes } 1.0V

(6) properties

Collection framework:

Collection(I):-

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

→ Collection Interface defines the most Common methods which Can be applied for any Collection object.

→ The following is the list of methods present in Collection Interface.

① boolean add(Object o)

② boolean addAll(Collection c)

③ boolean remove(Object o)

④ boolean removeAll(Collection c)

⑤ boolean retainAll (Collection c)

→ To remove all objects Except those present in c.

⑥ void clear()

⑦ boolean isEmpty()

⑧ int size()

⑨ boolean contains(Object o)

⑩ boolean ContainsAll(Collection c)

⑪ Object[] toArray()

⑫ Iterator iterator()

(2) List (I):-

→ List is the child Interface of Collection.

→ If we want to represent a group of individual objects where duplicate Objects are allowed & insertion Order is preserved. then we Should go for List.

→ Insertion Order will be preserved by means of Index.

→ we Can differenciate duplicate objects by using Index. Hence Index place a Very important role in List.

→ List Interface defines the following methods

① boolean add(int index, Object o)

② boolean addAll(int index, Collection c)

③ Object remove(int index)

④ Object get(int index)

⑤ Object set(int index, Object new)
old

⑥ int indexOf(Object o)

⑦ int lastIndexOf(Object o)

⑧ ListIterator listIterator()

It Contains 4 Classes:-

(i) ArrayList (C):- 1.2

(ii) Linked List (C):- 1.2

(iii) VectorList (C):- 1.0

(iv) Stack (C):- 1.0

(i) ArrayList (c):-

→ The underlying datastructure for Arraylist is Resizable Array (or) Growable Array.

→ Insertion order is preserved.

→ duplicate objects are allowed.

→ Heterogeneous objects are allowed.

→ Null insertion is possible.

Constructors :-

① ArrayList Al = new ArrayList();

→ Creates an Empty Arraylist object, with default initial Capacity 10.

→ Once AL reaches it's max. capacity then a new AL object will be Created with.

$$\text{New Capacity} = \text{CurrentCapacity} * \frac{3}{2} + 1$$

② ArrayList l = new ArrayList(int initialCapacity);

→ Creates an Empty Arraylist object with the Specified initial Capacity.

③ Arraylist l = new Arraylist(Collection c);

→ Creates an Equivalent Arraylist object for the Given Collection objects
ie, this Constructor is for dancing b/w Collection objects

Ep!.
```
import java.util.*;
     Class ArrayListDemo
     {
       P.S.v.m(String[] args)
      {
       Arraylist a = new Arraylist().
       a.add("A").
       a.add(10);
       a.add('A');
       a.add(null);
       S.o.pln(a);    [A, 10, A, null]
       a.remove(2);
       S.o.pln(a);    [A, 10, null]
       a.add(2,"M");  [A, 10, M, null]
       a.add("N");    [A, 10, M, null, N]
        S.o.pln(a);   [A, 10, M, null, N]
       }
     }}               S.o.pln(a.size()); //5
                       a.clear(); // [ ]
                       a.addAll(a); //[A,10,M, null, N, A,10,M,null,N]
```

Note:-

- In Every Collection class toString() is overridden to return its Content directly in the following formatt.

    [obj1, obj2, obj3 .....]

→ Usually we Can Use Collection to Store & transfer Objects. to provide Support for this requirement Every Collection class implements Serializable & Clonable Interfaces.

→ Arraylist & Vector classes implements RandomAccess Interface, So that any random element we can access with Same Speed. Hence, if our frequent operation is retrievable operation then best Suitable data structure is Arraylist. (Advantage)

→ If our frequent operation is Insertion or deletion operation in the middle then Arraylist is the worest choice, because it required Serveral shift operations. (disadvantage)

Q: difference b/w ArrayList & Vector?

Arraylist	Vector
① No method is Synchronized	① Every method is Synchronized
② multiple threads can access Arraylist Simultaneously, hence Arraylist Object is not threadsafe	② At any point only one Thread is allowed to operate on Vector Object at a time. Hence vector Object is Thread Safe.
③ Threads are not required to wait, & Hence performance is high.	③ It increases waiting time of Threads & Hence performance is Low.
④ Introduced in 1.2 version & Hence it is non-legacy	④ Introduced in 1.0 version & Hence it is Legacy

Q) How to get Synchronized Version of Arraylist ?

A) → By using Collections class SynchronizedList() ~~go~~ we can get synchronized version of Arraylist.

```
public static List SynchronizedList(List l)
```

eg:-

```
Arraylist l = new Arraylist();
List l1 = Collections.synchronizedList(l)
```

↙ ~~Syncho~~ Synchronized

↘ NON-Synchronized.

→ Similarly we can get Synchronized Version of Set & Map objects by using the following methods respectively.

① public static Set synchronizedSet(Set s)

② public static Map SynchronizedMap (Map m)

Note:-

→ If our frequent Operation is Insertion or deletion in the middle then ArrayList is not recommended. to handle this requirement we should go for LinkedList.

3) LinkedList (C) :-

→ The underlying datastructure is double LinkedList.

→ Insertion order is preserved.

→ duplicate objects are allowed.

→ Heterogeneous " "

→ Null insertion is possible.

→ Implements Serializable & Clonable interface but not RandomAccess-interface.

→ Best Suitable if our frequent operation insertion or deletion in the middle.

→ Worrest choice if our frequent operation is retrival.

Constructors:-

① LinkedList l = new LinkedList( );

→ Creates an Empty LinkedList object.

② LinkedList l = new LinkedList(Collection c)

→ for interConversion b/w Collection objects.

LinkedList Specific methods:-

→ Usually we can use LinkedList to implements Stacks & Queues to support this requirements LinkedList class define the following Six Specific methods.

① Void addFirst (Object o);

② void addLast (Object o);

③ Object removeFirst();

④ Object removeLast();

⑤ Object getFirst();

⑥ Object getLast();

Ex:-

```
import java.util.*;
class LinkedListDemo
{
 p.s.v.m(String[] args)
 {
   LinkedList l = new LinkedList();
   l.add("durga");
   l.add(30);
   l.add(null);
[durga, 30, null, durga]. l.add("durga");                    Software
                                               ccc venky durga  30  null  durga
[s/w, 30, null, durga] l.set(0, "Software");
[venky, s/w, 30, null, durga] l.add(0, "venky");
[venky, s/w, 30, null] l.removeLast();
[ccc, venky, s/w, 30, null] l.addFirst("ccc");
   S.o.pln(l);          [ccc, venkey, Software, 30, null]
 }
}
```

ii) Vector(c) :-

→ The underlying datastructure is Resizeable array or growable array.

→ Insertion order is Preserved.

→ duplicate objects are allowed.

→ null insertion is possible.

→ Heterogeneous objects are allowed.

→ implements Serializable, Clonable & RandomAccess interfaces.

→ Best Suitable if our frequent operation is Retrival & worrest choice if our frequent operation is insertion or deletion in the middle.

→ Every method in vector is Synchronized. Hence vector object is ThreadSafe.

Constructors:-

(i) Vector v = new Vector();

→ Creates an Empty Vector object with default initial Capacity 10.

→ Once vector reaches it's max. Capacity a new Vector object will be Created with double Capacity.

new Capacity = 2* Current Capacity.

② Vector v = new Vector(int initialCapacity);

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

④ Vector v = new Vector(Collection c);

⑩

Vector Specific methods :- ⑩

→ To add objects

① add(Object o) ⟶ C

② add(int index, Object o) ⟶ L

③ addElement(Object obj) ⟶ V

→ To remove Elements or objects

① remove(object o) ⟶ C

② removeElement (Object o) ⟶ V

③ remove(int index) ⟶ L

④ removeElementAt(int index) ⟶ V

⑤ clear() ⟶ C

⑥ removeAllElements() ⟶ V

→ To retrive elements

① get(int index) ⟶ L

② elementAt(int index) ⟶ V

③ firstElement(); ⟶ V

④ lastElement(); ⟶ V

→ Other methods

① int Size();

② int Capacity();

* ③ Enumeration elements();

Eg:-

```
import java.util.*;
  class VDemo1
  {
   p.s.v.m(String[] args)
   {
    Vector v = new Vector();
    S.o.pln(v.capacity());
    for(int i=1 ; i<=10 ; i++)
    {
      v.addElement(i);
    }
    S.o.pln(v.capacity());
    V.addElement("A");
     S.o.pln(v.capacity());
     S.o.pln(v);
   }
  }
```

o/p:-

```
10
10
20
[1, 2, 3, 4, 5, 6, ----10, A]
```

```
            ┌ no of objects
v.size()// 11
                        ┌ object
v.removeElement(9)// [1,2,3,4,5,6,7,8,10, A]
v.removeElementAt(3)// [1,2,3,5,6,7,8,10,A]
v.removeAllElements() // []
```

(4) Stack (c) :- (LIFO)

→ It is the child class of vector contains only one Constructor

(1) Stack s = new Stack();

Methods :-

(i) object push(Object o)

To insert an object into the Stack

(ii) Object pop();

To remove and returns top of Stack

(iii) Object peek();

To return top of the Stack.

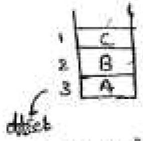
(iv) boolean empty();

returns true when Stack is Empty.

(v) int Search(Object o)

returns the offset from top of the Stack if the Object is available, Otherwise returns -1.

Ex:-



S.Search('A'); 3
S.Search('C'); 1
S.Search('z'); -1

Ex.

```
import java.util.*;
class StackDemo
{
  p.s.v.m(String[] args)
  {
    Stack s = new Stack();
     S.push('A');
     S.push('B');
     S.push('c');
     S.o.pln(s); / [A B C]
     S.o.pln(S.Search('A'));    3
 }}  S.o.pln(S.Search('z'));    -1
```

S.pop(); [A B

Cursors :-

Types of Cursors :-

→ If we want to get objects one by one from the Collection we should go for Cursor.

→ There are 3 types of Cursors available in java.

(i) Enumeration (1.0v)

(ii) Iterator (1.2v)

(iii) ListIterator (1.2v)

(i) Enumeration (in 1.0 Ver)

→ It is a Cursor to retrieve Objects one by one from the Collection.

→ It is applicable for legacy classes.

→ We Can Create Enumeration object by using elements()

```
public Enumeration elements();
```

Eg:-

```
Enumeration e = v.elements();
```

Vector object

→ Enumeration Interface defines the following 2 methods.

(i) public boolean hasMoreElements();

(ii) public Object nextElement();

Ex:-

```
import java.util.*;
Class EnumerationDemo
{
 p.s.v.m(String[] args)
 {
  Vector V =new Vector();
   for(int i=0 ; i<=10 ; i++)
   {
    V.addElement(i);
   }
  S.o.pln(v);    [0, 1, 2, 3 - - - - 10]
  Enumeration e = V.elements();
  While(e.hasMoreElements())
  {
   Integer I = (Integer)e.nextElement();
   if(I%2 ==0)
   S.o.pln(I);   2 4 6 8 10
   }
   S.o.pln(v);    [0, 1, 2,3, 4, - - - 10]
 }
}
```

O/P:

```
[0, 1, 2, 3 - - - - - 10]
0
2
4
6
8
10
[0 , 1, 2 , 3 - - - - 10]
```

Limitations of Enumeration :-

→ Enumeration Concept is applicable only for Legacy classes & hence it is not a universal Cursor.

→ By using Enumeration we can get only ReadAccess & we can't perform any remove operations.

→ To over Come these Limitations SUN people introduced Iterator in 1.2 version.

Iterator :-

→ We can apply Iterator Concept for any Collection object. It is a universal Cursor.

→ While Iterating we can perform remove operation also in addition to read operation.

→ We can get Iterator object by Iterator() of Collection interface.

Iterator itr = c.iterator()

↓

Any collection object

→ Iterator interface defines the following 3 methods.

(i) public boolean hasNext();

(ii) public Object next();

(iii) public void remove();

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

O/P: false
[null , D , B , C, 10, Z]

Note:- Insertion order is not preserved

(ii) LinkedHashSet (c):-

→ LinkedHashSet is the child class of HashSet.

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

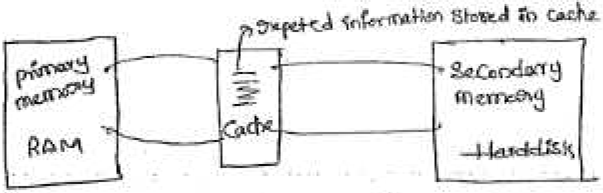
HashSet	LinkedHashSet
(i) The underlying D.S is Hashtable	i) The underlying D.S is a Combination of Hashtable & Linked List
(ii) Insertion order is not preserved	ii) Insertion order is preserved.
(iii) Introduced in 1.2 V	iii) Introduced in 1.4 V

→ In the above program if we are replacing HashSet with LinkedHashSet the following is the o/p.

o/p:- [B, C, D, Z, null, 10] ie, insertion order is preserved.

Note:-

→ The main important application area of LinkedHashSet & LinkedHashMap is implementing Cache applications. where duplicates are not allowed & insertion order must be preserved.



3) SortedSet (I):-

→ It is the child Interface of Set.

→ If we want to represent a group of individual objects according to some sorting order. Then we should go for SortedSet

→ SortedSet Interface defines the following 6 specific methods

(i) Object first()

→ returns the first element of SortedSet.

(ii) Object last()

→ returns last element of Sorted Set

(iii) SortedSet headSet(Object obj)

→ returns the SortedSet whose elements are lessthan obj.

(iv) SortedSet tailSet(object obj)

→ returns the SortedSet whose elements are greater than or equal to obj ≥

(v) SortedSet subSet(Object obj1, object obj2).

→ returns the SortedSet whose elements are ≥ obj1 but < obj2

(vi) Comparator comparator( )

→ returns Comparator object describes underlying sorting technique

→ if we used default natural sorting order then we will get null.

Ex.-

100
101
103
104
107
109

① first( ) ⟶ 100
② last( ) ⟶ 109
③ headSet(104) ⟶ [100, 101, 103]
④ ~~tail~~ tailSet (104) → [104, 107, 109]
⑤ SubSet (101, 107) → [101, 103, 104]
⑥ Comparator( ) ⟶ null

Note:-

→ The default natural sorting order for the no's is assending order

→ the default natural sorting order for characters & Strings are is alphabetical order (dictionary based order).

TreeSet (c):

→ The underlying datastructure is Balanced Tree.

→ duplicate objects are not allowed.

→ Insertion order is not preserved. because objects will be inserted according to some sorting order.

→ Heterogeneous objects are not allowed. otherwise we will get "ClassCastException". & Null insertion is not possible for non-empty Set.

Constructors:-

(i) TreeSet t = new TreeSet();

→ Creates an Empty TreeSet object where the sorting order is default natural sorting order.

(ii) TreeSet t = new TreeSet(Comparator c)

→ Creates an Empty treeset object where the sorting order is customized sorting order specified by Comparator object.

(iii) TreeSet t = new TreeSet (Collection c)

(iv) TreeSet t = new TreeSet(SortedSet c)

Ex:-

```
import java.util.*;
class TreeSetDemo
{
 p.s.v.m(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  ClassCastException
    //t.add(null); // → NPE
    S.o.pln(t);  [A, B , z , L , a]
  }
}
```

* null acceptance :-

(i) for the Non-Empty Tree Set if we are trying to insert null we will get NullPointer Exception(NPE).

(ii) for the Empty TreeSet add the first element null insertion is always possible.

(iii) But after inserting that null, if we are trying to insert any-other, we will get NullPointerException (NPE).

eg:
```
import java.util.*;
class TreeSetDemo1
{
  p.s.v.m(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"));        o/p:- CCE
  } }  S.o.pln(t);
```

↳ If we are depending on default natural sorting order Compolsary objects should be homogeneous & Comparable otherwise we will get ClassCastException (CCE)

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

↳ String class & all wrapper classes already implements Comparable Interface where as StringBuffer doesn't implements Comparable Interface. Hence, In the above Example we got ClassCastException.

Comparable Interface :-

→ This Interface present in java.lang package & Contains only one method i.e., "compareTo()".

```
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 (duplicate)

eg:-
```
import java.util.*;
class Test
{
 p.s.v.m(String[] args)
 {
  S.o.pln("A".compareTo("Z")); // -ve   -25
  S.o.pln("Z".compareTo("K")); // +ve   15
 } S.o.pln("A".compareTo("A")); // 0     0
}
```

13/01/11

→ When we're are depending on default natural sorting order internally JVM calls ~~Com~~ compareTo().

→ Based on the return-type JVM identifies the location of the element in sorting order.

obj1 . compareTo (obj2)

which object we're trying to add ↙ ↘ already existing object in TreeSet.

→
- → 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

Eg:-

```
TreeSet t = new TreeSet();
 t.add("z");
 t.add("K");  ⟹ "K".compareTo(z); -ve
 t.add("D");  ⟹ "D".CompareTo("K"); -ve
 t.add("M");  → "M".CompareTo("D") ⟹ +ve
 t.add("D");    "M".CompareTo("K") ⟹ +ve
                "M".compareTo("z"); → -ve
//t.add(null);  → "D".CompareTo("D") ⟹ 0
S.o.pln(t);
     [D ,K, M, Z]   → ~~ClassCastException, NPE~~
                      null.CompareTo("D") ⟹ RE ⟹ NPE
```

→ If we are not Satisfied with default natural Sorting order ~~Or~~ if the default natural Sorting order is not already Available. Then we Can define Our own Customized Sorting by Using Comparator

- Comparable ment for default natural Sorting order.
- Comparator ment for Customized Sorting order.

Comparator (I):

→ Comparator Interface present in java.util package & defines the following 2 methods.

① public int compare(Object obj1, object 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 (duplicates).

obj1 ⇒ which object we are trying to add
obj2 ⇒ already existing object

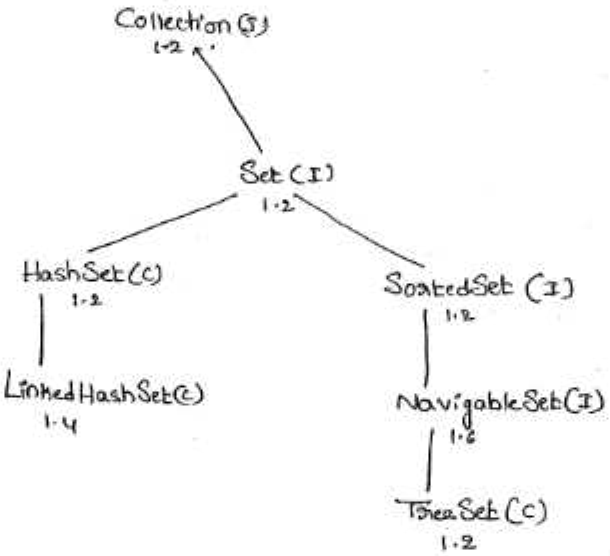
② public boolean equals(Object obj)

→ when ever we are implementing Comparator Interface Compulsary we should provide implementation for Compare(), 2nd method equals() implentation is optional, because it is already available for Our class from Object Class through Inheritance.

(2) Set (I) :-

→ Set is child Interface of Collection.

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



→ Set Interface doesnot Contain any method we have to use only Collection Interface method.

(i) HashSet (c) :-

→ The underlying datastructure is Hashtable.

→ Duplicate objects are not allowed.

→ If we are trying to add duplicate objects we won't to gets any C.E or R.Error add() simply returns false,

→ Insertion order is not preserved & all objects are inserted according to hash Code of the objects

→ Heterogeneous objects are allowed.

→ null insertion is possible (only one) becaese duplicates are not allowed.

→ HashSet Implements Serializable & Clonable Interfaces.

* Constructors:-

1) HashSet h = new HashSet();

→ Creates an Empty HashSet object with diffault default initial Capacity 16 & default fillRatio 0.75 (75%).

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

→ Creates an Empty HashSet object with the specified initial Capacity & default fill Ratio is 0.75.

3) HashSet h = new HashSet(int initialCapacity, $\underbrace{\text{float fillratio}}_{0 \text{ to } 1}$);

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

fillratio:-

→ After Completeing (filling). The Specified ratio then only a new HashSet object will be Created that particular ratio is Called fillratio & load factor.

→ The default fillratio is 0.75 but we Can Customized this value.

eg:-
```
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);
      }
      S.o.pln(l);   [0,1,2,3,....10]
      Iterator itr = l.iterator();
      while(itr.hasNext()
      {
        Integer I =(Integer)itr.next();
        if(I%2 ==0)
        {
         S.o.pln(I);          0
        }                     2
                              4
                              6
       else                   8
                              10
        {
         itr.remove();
        }
      }
      S.o.pln(l);  [0, 2, 4, 6, 8, 10]
```

Limitations of Iterator:-

(i) In the Case of Iterator & Enumeration we can always move towards the forward direction & we can't move backward direction. ie These Cursors are Single directional Cursors but not Bidirectional.

(ii) while performing Iteration we can perform only read & remove operations &

We can't perform replacement & Addition of New objects.

→ To resolve these problem SUN people Introduced ListIterator in 1.2 version.

3) ListIterator :-

→ ListIterator is the child Interface of Iterator.

→ While Iterating Objects by ListInterator we can move either to the forward or to the Backward direction. i.e ListIterator is a Bidirectional Cursor.

→ While Iterating By ListIterator we can perform replacement & addition of new objects also in addition to Read & Remove operations.

→ We can Create ListIterator object by using listIterator() ~~list~~ of List Interface.

→ any List object
→ small letter

```
ListIterator litr = l.listIterator();
```

→ ListIterator Interface defines the following 9 methods.

forward
(i) public boolean hasNext();
(ii) public Object next();
(iii) public int nextIndex();

Backward
(iv) public boolean hasPrevious();
(v) public Object previous();
(vi) public int previousIndex();

⑦ public void remove();

⑧ public void set(Object new); → replace an object with new Object

⑨ public void add(Object new); → add new obj

Eg:-

```
import java.util.*;
class ListIteratorDemo
{
 public static void main(String[] args)
 {
    LinkedList l = new LinkedList();
     l.add("balakrishna");
     l.add("venky");
     l.add("chiru");
     l.add("nag");
     S.o.pln(l);      [balakrishna , venky , chiru , nag]
    ~~LinkedList~~
    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")
      }}
      S.o.pln(l);              [Balakrisha , charan , nag, Chaitu]
 }}
```

Note:-

→ among 3 Cursors ListIterator is the most powerfull Cursor. But it is applicable only for List objects.

Comparision table of 3-Cursors :-

Property	Enumeration (1.0V)	Iterator (1.2V)	ListIterator (1.2V)
① Is it legacy	yes	No	No
② It is applicable only for	only for Legacyclasses	for any Collection objects	Only for List objects
③ movement	Single direction (only forward)	Single direction (forward)	bi-directional (forward & backward)
④ How to get it?	By using elements() method	By using iterator()	By using listIterator()
⑤ Accessibility	only read	read & remove	read/remove/replace/add

⑥ method	hasMoreElements()
	nextElement()
		hasNext()
		next()
		remove()
			4 methods

Eg:-
```
import java.util.*;
class TreeSetDemo3
{
 Public Static void main(String[] args)
 {
  Tree  Integer I1 = (Integer)obj1;
           Integer I2 = (Integer
  TreeSet t = new TreeSet(new myComparator()); → ①
  t.add(20);
  t.add(0); → Compare(0, 20) → +ve
  t.add(15); → Compare (15, 20) → +ve
             → Compare (15, 0) → -ve
  t.add(5); → Compare(5,20) → +ve
            → Compare(5, 0) → +ve
            → Compare(5,15) → -ve
  t.add(10);
            → Compare (10, 20) +ve
              Compare (10, 0) +ve
              Compare (10, 15) +ve
              Compare (10, 5) -ve
 S.o.pln(t);
  }
}      ↘
   [20, 15, 10, 5, 0]

class MyComparator implements Comparator
{
  public int compare(Object obj1, Object obj2)
  {
    Integer I1 = (Integer)obj1;
    Integer I2 = (Integer)obj2;
    if(I1<I2)
      return +100;         } return((I1<I2)?+1:(I1>I2 ?-1 :0));
    else if(I1>I2)
      return -100;
    else
      return0;
}}
```

→ If we are not passing Comparator object at line 1 ① then JVM internally calls CompareTo() which is ment for default natural sorting order. In this case the o/p is [0, 5, 10, 15, 20].

→ If we are passing comparator object at ① then our own Compare method will be execuftaded which is ment for Customized Sorting order. these case the o/p is [20, 15, 10, 5, 0]

Various alternatives of implementing compare() :-

```
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]
 //return -I1.CompareTo(I2);  ⟹ [20, 15, 10, 5, 0]
 //return I2.CompareTo(I1);  → [20, 15, 10, 5, 0]
 //return -I2.CompareTo(I1);  → [0, 5, 10, 15, 20]
 //return -1; ⇒ [10, 5, 15, 0, 20] ⇒ Reverse of insertion order
 //return +1; ⇒ [20, 0, 15, 5, 10] ⇒ insertion order
 / return 0; ⇒ [20]
 }
}
```

✱ W.a.p To insert String objects into the TreeSet where the Sorting order is reverse of alphabetical order.

A:
```
import java.util.*;
class TreeSetDemo2
{
  Public static v. m(String[] args)
  {
    TreeSet t = new TreeSet(new myComparator());
      t.add("A");
      t.add("Z");
      t.add("K");
      t.add("B");
      t.add("a");
    S.o.pln(t);
  }
}
Class MyComparator implements Comparator
{
  Public int compare(Object obj1 , object obj2)
  {
    String s1 = (String)obj1;
    String s2 = obj2.toString(); ✓
    return -s1.compareTo(s2);
  }
}
```

Note:-

~~In object class Comparre method don't Contain Strings only Contain object type so objects Can be Convert into Strings by using typecasting~~

Note:-

→ In objects & StringBuffer there is no Comparators, So we Can Convert into Strings.

* W.a.p to insert String & StringBuffer objects into the TreeSet where the Sorting order increasing length order. If two objects having the Same length then Consider their alphabetical order

Q.

```
import java.util.*;

Class TreeSetDemo12
{
  p s v m(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");
     S.o.pln(t);    [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 (l2 > l2)
       return +1;
                               else
                                 return s1.compareTo(s2);
                               }
                              }
```

* W.a.p To insert StringBuffer objects into The TreeSet where The Sorting order alphabetical order?

A

```
import java.util.*;
Class TreeSetDemo10
 {
  p.s.v.m(String[] args)
  {
    TreeSet t = new TreeSet(new MyComparator()).
     t.add(new StringBuffer("A"));
     t.add(new StringBuffer("Z"));
     t.add(new StringBuffer("K"));
                              ("L"));
     S.o.pln(t);   [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);
   }
}
```

O/P :- [A, K, L, Z]

So SB Canbe Convert into String

Note:-

→ In StringBuffer there is no CompareTo method

→ If we are depending on default natural Sorting order Compulsary objects should be Homogeneous & Comparable. Otherwise we will get CCE

→ If we are depending on our own Sorting by Comparator The Objects need not be Comparable & Homogeneous.

Comparable Vs Comparator :-

① for predefined Comparable classes default natural Sorting order is already available if we are not Satisfied with that we Can define our own Customized Sorting By using Comparator

Ex:- String.

② for predefined NON-Comparable classes default natural Sorting order is not available Compulsary we should define Sorting by using Comparator object only.

Ex:- StringBuffer.

③ for our own Customized classes to define default natural Sorting order we Can go for Comparable & to define Customized Sorting we should go for Comparator.

Ex:- Employee, Student, Customer.

```
import java.util.*;
Class Employee implements Comparable
{
  int eid;
  Employee(int eid)
  {
   this.eid = eid;
  }
  public String toString()
  {
   returns "E-" +eid;
  }
  public int CompareTo(Object obj)
  {
   int eid1 = this.eid;
   Employee e2 = (Employee)obj;
   int eid2 = ~~this~~ e2.eid;
   if (eid1 <eid2)
     return -1;
   else if (eid1 > eid2)
     return +1;
   else
     return 0;
   }
  }
 Class CompCompDemo
 {
  p.s.v.m(String[] args)
 {
```

```
Employee  e1 = new Employee(200);
Employee  e2 = new Employee(100);
Employee  e3 = new Employee(500);
Employee  e4 = new Employee(700);
Employee  e5 = new Employee(700);
TreeSet  t1 = new TreeSet();
  t1.add(e1);
  t1.add(e2);
  t1.add(e3);
  t1.add(e4);
  t1.add(e5);
S.o.pln(t1);      [E-100 , E-200 , E-500 , E-700]
TreeSet  t2 = new  TreeSet(new  MyComparator());
  t2.add(e1);
  t2.add(e2);
  t2.add(e3);
  t2.add(e4);
  t2.add(e5);
S.o.pln(t2);      [ E-700 , E-500 , E200 , E-100]
}
}
Class  MyComparator  implements  Comparator
{
  Public  int  Compare(Object obj1,  Object  obj2)
  {
      Employee  e1 = (Employee) obj1;
      Employee  e2 = (Employee) obj2;
} }   return  e2.CompareTo(e1);  // return -e1.CompareTo(e2);
```

1), W.a.p to insert Employee objects into the TreeSet where default natural Sorting order is asending order of Salary. If Two Emp having the Same Salary then Consider alphabetical orders of their names. &

W.a. Comparator class to define Customized Sorting which is alphabetical order of Employee names. If two Employees having the Same name then Consider desending order of their age.

* Comparision b/w Comparable & Comparator :-

Comparable	Comparator
1) We can use Comparable to define default natural Sorting order.	1) We can use Comparator to define Customized Sorting order.
2) This interface present in Java.lang package.	2) This interface present in java.util package.
3) defines only one method i.e compareTo()	3) defines Two methods (i) compare() (ii) equals()
4) All wrapper classes & String class implements Comparable interface	4) No predefined class implements Comparator interface.

Comparision table for Set Implemented classes ?

Property	HashSet	LinkedHashSet	TreeSet
Underlying D.S	Hash table	~~H~~Hashtable + Linked List	Balanced Tree
1) insertion order	not preserved	preserved	not preserved
2) Sorting order	N.A	N.A	preserved
4) Heterogeneous Objects	allowed	allowed	Not allowed
5) Duplicate objects	not allowed	not allowed	not allowed
3) null acceptance	allowed (1)	allowed (1)	for the empty TreeSet add the first element null insertion is possible. in all other cases we will get NPE

15625

Map (I):-

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

→ Both key & values are objects.

→ Duplicate keys are not allowed. But values can be duplicated.

→ Each key-value pair is called Entry.

Ex:

Rollno	name
101	durga
102	Sainu
103	Ravi
104	Sambu
105	Sundar

entry

key

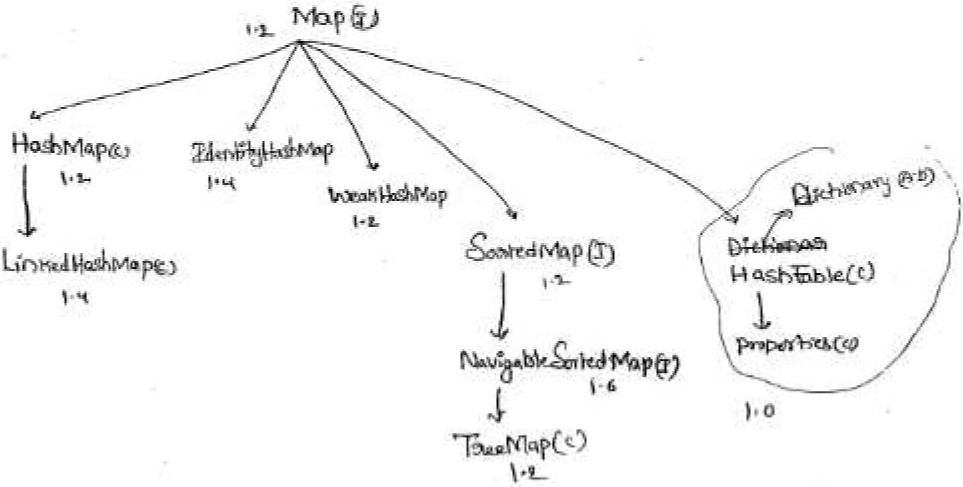
value

→ There is no relationship b/w Collection & Map.

→ • Collection ment for a group of individual objects where as

• Map ment for a group of key-value pairs.

→ Map is not child interface of Collection.



Methods of Map Interface :-

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

→ To add ~~d~~key-value pair to the map

→ If the Specified key is already available then old Value will be replaced with new Value & old value will be returned.

② Void putAll(Map m)

→ To add a group of Key-value Pairs. ~~to~~

*③ Object get(Object key)

→ returns the value associated with Specified Key

→ If the key is not available then we will get Null

④ object remove(Object Key);

⑤ boolean ContainsKey(Object key)

⑥ boolean ContainsValue(Object value)

⑦ int Size();

⑧ boolean isEmpty();

⑨ Void Clear()

① Set KeySet(); }
② Collection values(); } → Collection Views, of the Map.
③ Set entrySet(); }

Entry (Interface) :

→ Each key-value pair is Called One 'Entry'

→ Without existing Map Object There is no chance of Entry object hence, Interface Entry is define inside Map Interface.

Code :

```
interface Map
{
  interface Entry
  {
   ①, Object getKey();
   ②, Object getValue();
   ③, Object setValue();
  }
}
```

① HashMap :

→ The underlying dataStructure is HashTable

→ Heterogeneous Objects are allowed for both Keys & values

→ duplicate Keys are not allowed ~~for~~ but the values Can be duplicated.

→ Insertion order is not preserved because it is based on HashCode of Keys.

→ null Key is allowed (only once)

→ null values are allowed (any number of times).

- differences b/w HashMap & HashTable :-

HashMap	HashTable
① No method is Synchronized	① Every method is Synchronized
2) multiple Threads Can operates Simultaneously & Hence HashMap object is not Thread Safe	② At a time only one Thread is allowed to operate on HashTable object. Hence it is Thread Safe.
3) Threads are not required to wait & hence relatively performance is High.	③ It increases waiting time of the Thread & Hence performance is low.
4) null is allowed for both key & value	④ null is not allowed for Both key & Values. otherwise we will get NPE-An
5) Introduced in 1.2 version & it is non-Legacy	⑤ Introduced in 1.0 Version & it is legacy

Q) How To get Synchronized Version of HashMap?

A) → Bydefault HashMap object is not Synchronized, but we Can get Synchronized Version by using SynchronizedMap() of Collections Class.

```
Map M = Collections.synchronizedMap(HashMap hm);
```

Constructor :-

(i) HashMap m = new HashMap( );

→ Creates an Empty HashMap object with default initial Capacity level is "16" & default fillRatio 0.75 (75%).

(ii) HashMap m = new HashMap(int initialCapacity)

(iii) HashMap m = new HashMap(int initialCapacity, float fillRatio)

(iv) HashMap m = new HashMap(Map m)

Ex :-
```
import java.util.*;
    Class HashMapDemo
    {
     P.S.V.m(String[] args)
     {
       HashMap m = new HashMap();
         m.put("chiranjeevi", 700);
         m.put(" balaiah", 800);
         m.put("venkatesh", 1000);
         m.put(" nagarjuna", 500);
         S.o.pln(m);  {venkatesh =1000, balaiah =800, chiranjeevi=700,
                                                   nagarjuna = 500}
         S.o.pln(m.put("chiranjeevi", 1000));  700
         Set s = m.keySet();
         S.o.pln(s); [venkatesh, balaiah, chiranjeevi, nagarjuna]
        Collection c= m.values();
         S.o.pln(c)  [1000 , 800 , 1000 , 500].

          Set S1 = m.entrySet();

          Iterator itr = S1.iterator();
while (its. hasNext())
{
  Map.Entry  m1 = (map.Entry) its. next();
  S.o.pln (m1. getKey () + "----" +m1.getValue());
                                               ->  nagarjuna  500
 if (m1. getKey (). equals ("nagarjuna"))         venkatesh  1000
                                                   balaiah    800
    m1. Set Value (10000);                         chiranjeevi 1000
 }
S.o.pln (m);   } nagarjuna =10000, venkatesh =1000, balaiah = 800,
}                                                  chiranjeevi =1000}
}
```

3) Linked HashMap :-

→ It is the child class of. HashMap.

→ It is Exactly same as HashMap except the following differences

HashMap	Linked HashMap.
① The underlying D.S is HashTable	① The underlying D.S is 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 program if we are replacing HashMap with Linked HashMap. The following is the O/P.

{chiranjeevi =700, balaiah =800, venkatesh=1000, nagarjuna =500}

i.e insertion order is preserved

Note:-

→ The main application area of LinkedHashSet & LinkedHashMap s are cache applications implementation where duplication is not allowed & insertion order must be preserved.

(iii) IdentityHashMap :-

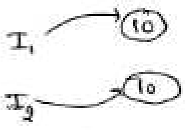
→ It is Exactly Same HashMap Except the following difference.

→ In the Case of HashMap to identify duplicate Keys JVM always uses .equals(), which is mostly ment for Content Comparision.

→ If we want to use == operator instead of .equals() to identify duplicate Keys we have to use IdentityHashMap. (== operator always ment for reference Comparision).

eg:-

```
HashMap m = new HashMap();
Integer i1 = new Integer(10);
Integer i2 = new Integer(10);
 m.put(i1, "pavan");
 m.put(i2, "Kalyan");
 S.o.pln(m); {10=Kalyan}
```



.equals() → Content
== → reference

$I_1 == I_2$ → false
I_1.equals(I_2) → True

→ In the above Code i_1 & i_2 are duplicate Keys because i_1.equals(i_2) returns true.

→ If we replace HashMap with IdentityHashMap then the o/p is {10=pavan, 10=Kalyan}

→ i_1 & i_2 are not duplicate Keys because $i_1 == i_2$ returns false.

WeakHashMap :-

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

→ In the Case of HashMap, Object is not ele elegible for g.c eventhough it doesn't have any external references if it is associated with HashMap. i.e, HashMap dominates Garbage Collector (g.c).

→ But In the Case of WeakHashMap Eventhough object associated with WeakHashMap, it is elegible for g.c, if it doesnot have any external references. i.e G.c dominates WeakHashMap.

eg:-
```
import java.util.*;
class WeakHashMapDemo
{
  p.s.v.m (String[] args) throws InterruptedException
  {
    HashMap m = new HashMap();
    Temp t = new Temp();
    m.put (t,"durga");
    S.o.pln(m);          {temp = durga}
     t = null;
    System.gc();
    Thread.sleep(5000);
     S.o.pln(m);
  }                {temp = durga}
}
```

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

o/p:

```
{temp = durga}
{temp = durga}
```

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

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

(ii) SortedMap (I) :-

→ If we want to represent a group of entries according to Some Sorting order then we should go for SortedMap. The Sorting should be done based on the keys but not based on the Values.

→ SortedMap interface is the child interface of Map.

→ SortedMap interface defines the following 6 Specific methods

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

(iii) TreeMap (C):-

→ The underly D.S is RED-BLACK Tree,

→ Insertion order is not preserved & all Entries are inserted according to some Sorting Order of Keys.

→ If we are depending on default natural Sorting order then the Keys should be Homogeneous & Comparable. otherwise we will get ClassCastException (CCE).

→ If we are defining our own Sorting order by Comparator then

The Keys need not be Homogeneous & Comparable.

→ There are no restrictions on Values, they Can be Heterogeneous & non-Comparable.

→ duplicate Keys are not allowed but values Can be duplicated.

null acceptance:-

→ for the Empty TreeMap as the first Entry with is null Key is allowed but after inserting that Entry if we are trying to insert any other Entry we will get NullpointerException (NPE).

→ for the NON-Empty TreeMap if we are trying to insert Entry with null key we will get NullpointerException (NPE)

→ There are no restructions on null values. i.e, we can use null any no. of times any where for Map values.

Constructors:-

(i) TreeMap t = new TreeMap()

-for default natural Sorting order.

(ii) TreeMap t = new TreeMap(Comparator c)

-for Customized Sorting order.

(iii) TreeMap t = new TreeMap(Map m)

(iv) TreeMap t = new TreeMap(SortedMap m)

Eg.-

```
import java.util.*;
Class TreeMapDemo3
{
  p.s.v.m(String[] args)
  {
    TreeMap m = new TreeMap();
      m.put(100, "zzz");
      m.put(103, "yyy");
      m.put(101, "xxx");
      m.put(104, 106);
      m.put(107, null);
      //m.put("FFFF", "xxx"); //CCE
     // m.put(null, "xxx"); //NPE
      S.o.pln(m);  {100 = zzz, 101= xxx, 103=yyy, 104=106, 107=null}
  }
}
```

O/p:-

```
{100 = zzz, 101 = xxx, 103=yyy, 104=106, 107 = null}
```

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

o/p:-
```
{zzz = 30 , xxx = 10 , LLL=40 ,AAA =20}
```

Hashtable(c):

→ The underlying datastructure is HashTable.

→ Heterogeneous objects are allowed for both keys & values

→ Insertion order is not preserved & it is based on HashCode of the keys.

→ null is not allowed for both key & values otherwise we will get NullpointerException (NPE).

→ duplicate keys are not allowed, but values can be duplicated.

→ All methods are synchronized & Hence HashTable object is ThreadSafe.

Constructor:

(i) Hashtable h = new Hashtable()

→ Creates an Empty Hashtable object with default initial capacity is 11 & default fillratio 75% (0.75).

(ii) Hashtable h = new Hashtable(int initialCapacity)

(iii) Hashtable h = new Hashtable(int initialCapacity, float fillratio)

(iv) Hashtable h = new Hashtable(Map m);

```
eg:- import java.util.*;
     Class HashtableDemo
     {
      p s v m(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+" ";
      }}
```

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

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

-from top to bottom & Right to Left

3) Properties(C):-

→ It is the child class of Hashtable

→ In our program if any thing which changes frequently(like database usernames, passwords, url) never reCommended to hardCode the value in the java program. Because for Every change, we have reCompile, rebuild, redeploye the application & Sometime even Server restart also required. Which Creates a big business impact to the client.

→ We have to Configuare those variables(~~properties~~ inside properties files & we have to read those values from javaCode.

→ the main advantage of this approach is, If any change in the properties file Just redeployement is enough which is not a business impact to the client.

Constructor:-

(i) Properties p = new Properties();

→ In the Case of Properties both key & value should be String type

Methods:-

*(i) String getProperty(String Propertyname)

→ returns the value associated with Specified property.

(ii)→ String setProperty(String pname, String pvalue);

→ to Set a new property.

(iii) String Enumeration propertyNames();

* (iv) Void load(InputStream is)

→ To load the properties from properties files into java properties-object.

(v) Void Store(OutputStream os, String Comment)

→ To update properties from properties object into properties file.

Eg:-
```
import java.util.*;
import java.io.*;
class PropertiesDemo
{
 P.S.v.m(String[] args) throws IOException
 {
  Properties p = new Properties();
  FileInputStream fis = new FileInputStream("abc.properties");
  p.load(fis);
  System.out.println(p);
  String s = p.getProperty("venki");
  S.o.pln(s);
  p.setproperty("nag", "999999");
  FileOutputStream fos = new FileOutputStream("abc.properties");
  p.Store(fos, "Updated by durga for SCJP Demo class");
 }
}
```

```
user = scott
venki = 8888
pwd = tiger
```

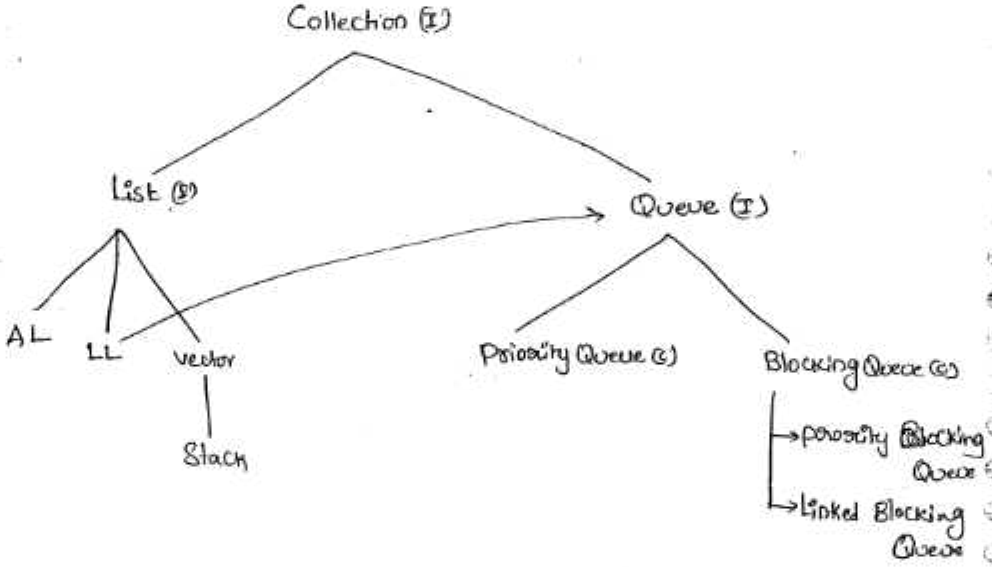
abc.properties

1.5 Version Enhancement:-

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.



→ Usually Queue follows FIFO (first in first out), But Based on our requirement we can change our order.

→ from 1.5 Version onwards LinkedList implements Queue Interface.

→ LinkedList Based implementation of Queue always follows FIFO

Queue Interface methods :-

(i) boolean offer(Object obj)

→ To add an object into the Queue.

(ii) Object peek();

→ To return head element of the Queue. If Queue is Empty then this method returns null.

(iii) Object element();

→ To return head element of the Queue. If Queue is Empty then we will get RuntimeException Saying NoSuchElementException

(iv) Object poll();

→ To remove & return head element of the Queue. If Queue is Empty then this method returns null.

(v) Object remove();

→ To remove & return head element of the Queue, if Queue is Empty then we will get RuntimeException Saying NoSuchElementExcepti

PriorityQueue(c) :-

→ This is the DataStructure to hold a group of individual Objects prior to processing According to Some priority.

→ The priority can be either default natural Sorting order or Customized Sorting order.

→ if we are depending on default natural Sorting Compulsory objects should be Homogeneous & Comparable otherwise we will get ClassCastException.

→ if we are defining our own Customized 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 :-

(i) PriorityQueue   q queue = new PriorityQueue();

→ Creates an Empty PriorityQueue with default initialCapacity "11" & priority order is default natural Sorting order.

(ii) PriorityQueue   q = new PriorityQueue(int initialCapacity);

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

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

(v) PriorityQueue q = new PriorityQueue(SortedSet s)

Eg:-

```
import java.util.*;
Class PriorityQueueDemo
{
  p.s.v.m(String[] args)
  {
    PriorityQueue q = new PriorityQueue();
    S.o.pln(q.peek()); //null
   // S.o.pln(q.element()); //NSE   NoSuchElementException
    for(int i=0 ; i<=10 ; i++)
    {
      q.offer(i);
    }
    S.o.pln(q); // [0, 1, 2, 3, 4, 5, ---- 10]
    S.o.pln(q.poll()); //0
    S.o.pln(q); // [1, 2, 3, 4, 5 --- 10]
  }
}
```

eg2:-

```
import java.util.*;
Class PriorityQueueDemo2
{
 p.s.v.m (String[] args)
 {
```

```
PriorityQueue q = new PriorityQueue(15, new MyComparator());
    q. offer("A");
    q. offer("Z");
    q. offer("L");
    q. offer("B");
    S.o.pln(q); // [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);
  }
}
```

o/p:- [Z , L, B, A]

** 1.6 Version Enhancements :-

(i) NavigableSet (I) :-

→ It is the child Interface of SortedSet.

→ This Interface defines Several methods to provide Support for navigation for the TreeSet object.

→ The following List of various methods present in NavigableSet.

(i) Ceiling(e) :-

→ returns the lowest element which is >= e

(ii) higher(e) :-

→ returns the lowest element which is >e.

(iii) floor(e) :-

→ returns highest element which is <=e.

(iv) lower(e) :-

→ returns the highest element which is <e.

(v) pollfirst() :-

→ remove & returns first element

(vi) pollLast() :-

→ remove & returns last element.

(vii) desendingSet() :-

→ returns the NavigableSet in reverse order.

```
Eg: import java.util.*;
    class NavigableSetDemo
    {
     P.S.v.m(String[] args)
     {
      TreeSet<Integer> t = new TreeSet<Integer>();
       t.add(1000);
       t.add(2000);
       t.add(3000);
       t.add(4000);
       t.add(5000);
       S.o.pln(t) ; [1000, 2000,
       S.o.pln(t.ceiling(2000)); 2000
       S.o.pln(t.higher(2000)); 3000
       S.o.pln(t.floor(3000)); 3000
       S.o.pln(t.lower(3000)); 2000
       S.o.pln(t.pollFirst()); 1000
       S.o.pln(t.pollLast()); 5000
       S.o.pln(t.descendingSet()), [4000 , 3000 , 2000]
       S.o.pln(t); [2000, 3000, 4000]
     }
    }
```

(ii) NavigableMap (I):-

→ It is the child interface of SortedMap to define Several method for Navigation purposes.

→ The following is the list of methods present in NavigableMap

(i) ceilingKey(e)

(ii) higherKey(e)

(iii) floorKey(e)

(iv) lowerKey(e)

(v) pollFirstEntry()

(vi) pollLastEntry()

(vii) desendingMap()

Eg:-

```
import java.util.*;
class NavigableMapDemo
{
 p.s.v.m (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");
   S.o.pln(t); {a=apple, b=banana, c=cat, d=dog, g=gun}
  S.o.pln(t.ceilingKey("c"));     c
  S.o.pln(t.higherKey("e"));     g
  S.o.pln(t.floorKey("e"));     d
  S.o.pln(t.lowerKey("e"));     d
  S.o.pln(t.pollFirstEntry());   a = apple
  S.o.pln(t.pollLastEntry());    g = gun
  S.o.pln(t.desendingMap());   {d=dog , c=cat , b=banana}
  S.o.pln(t);          {b=banana , c=cat , d=dog}
 }
}
```

Collections class

Collections Class:-

→ It is an utility class present in java.util package

→ It defines several utility methods for Collection implemented class objects

Sorting the elements of a list :-

→ Collections class defines the following methods to sort elements of a List.

① public static void sort(List l) :-

→ we can use these method to sort according to natural sorting order.

→ In this case comparsary elements should be Homogeonus & Comparable. otherwise we will get ClassCastException.

→ List should not contain null, otherwise we will get NullpointerException

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

→ To sort elements of a list according to customized sorting order

Searching the elements of a List :-

→ Collections class defines the following method to search elements of a List

① public static int binarySearch(List l, Object obj)

→ If the List is sorted according to natural sorting order then we have to use this method.

⑨ public static int binarySearch(List l, Object key, Comparator c)

→ If the List is Sorted according to Comparator then we have to use This method.

Conclusion :-

→ Internally binarySearch method uses BinarySearch algorithm.

→ Before Calling binarySearch() method Compulsary the List should be Sorted otherwise we will get unpredictable results.

→ Successfull Search returns index.

→ unsuccessfull Search returns insertion point

→ Insertion point is the Location where we Can place element in the Sorted List.

→ If the List is Sorted according to Comparator then at the time of Search also we should pass the Same Comparator otherwise we will get unpredictable results.

Ex:- To Search elements of List

```
import java.util.*;
class CollectionsSearchDemo
{
   p.s.v.m(String[] args)
   {
     ArrayList l = new ArrayList();
        l.add("Z");
        l.add("A");
        l.add("M");
        l.add ('K');
        l.add ('a');
        S.o.pln (l);    [Z, A, M, K, a]
      Collections.sort (l);
      S.o.pln(l);    | A | K | M | Z | a |
      S.o.pln (Collections. binarySearch(l, 'Z')); 3
      S.o.pln (Collections. binarySearch(l, 'J')); -2
    }
  }
```

-1	-2	-3	-4	-5	-6
A	K	M	Z	a	
0	1	2	3	4	

Exg!-

```
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.pln(l);   | 15 | 0 | 20 | 10 | 5 |
    Collections.sort( l, new MyComparator());
    S.o.pln(l);   | 20 | 15 | 10 | 5 | 0 |
    S.o.pln(Collections.binarySearch(l,10,new MyComparator())); //2
    S.o.pln(Collections.binarySearch(l,13,new MyComparator())) ; //-3
    S.o.pln(Collections. binarySearch( l,17)); // -6 unpredictable
  }
}
```

-1	-2	-3	-4	-5	-6
20	15	10	5	0	
0	1	2	3	4	

because it is not passing Comparator()

```
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 Contains n elements Range of Successfull Search

① Range of Successfull Search : 0 to $n-1$

② Range of unsuccessfull Search : $-(n+1)$ to -1

③ total Range : $-(n+1)$ to $n-1$

ex:-

-1	-2	-3	-4
10	20	30	
0	1	2	

Range of successful Search : 0 to 2

Range of unsuccessful Search : -4 to -1

Total Range : -4 to 2

Reversing the elements of a List:-

→ Collections class defines the following reverse method for Lists

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

ex:- To Reverse elements of List

```
import java.util.*;
Class CollectionsReverseDemo
{
   P.s.v.m(______)
   {
      AL l = new AL();
      l.add(15);
      l.add(0);
      l.add(20);
      l.add(10);
      l.add(5);
     S.o.pln(l);   | 15 | 0 | 20 | 10 | 5 |
     Collections.reverse(l);
     S.o.pln(l);   | 5 | 10 | 20 | 0 | 15 |
   }
}
```

reverse() Vs reverseOrder():-

→ We Can use reverse() method to reverse the elements of a List and this method Contain List arguement

→ Collections class defines reverse order method also to return Comparator object for reversing original Sorting order

Comparator c_1 = Collections.reverseOrder(Comparator c)

decending order

asending order

→ reverseOrder() method ~~Can take~~ Contains Comparator arguement whereas reverse() Contains List arguements.

Ex:- TO REVERSE ELEMENTS OF LIST

```
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);
        S.o.pln(l);   [15 | 0 | 20 | 10 | 5]
        Collections.reverse(l);
    }}  S.o.pln(l);   [5 | 10 | 20 | 0 | 15]
```

17/41

Arrays class

Arrays class :-

→ It is an utility class present in util package, To define Several utility methods for Arrays for both primitive Arrays & Object type Arrays

Sorting the elements of Array :-

→ Arrays class defines the following methods for this.

① public static void Sort(primitive[] p);

→ To Sort elements of primitive Array According to natural Sorting order

② public static void Sort(Object[] a)

→ To Sort elements of Object Array According to natural Sorting order.

→ In this Case Compulsary the elements should be Homogeneous & Comparable otherwise we will get ClassCasteException.

③ public static void Sort(Object[] a, Comparator c) :-

→ To sort elements of Object[] according to Customized Sorting order.

Note :-

primitive Arrays Can be Sorted only by natural Sorting order where as Object Arrays Can be Sorted either by natural Sorting order or by Customized Sorting order.

Ex:- To SORT elements of Arrays

<u>ArraysSortDemo.java</u>

```
import java.util.Arrays;
import java.util.Comparator;
Class ArraysSortDemo
{
  public static void main(String[] args)
  {
    int[] a = {10, 5, 20, 11, 6};
    S.o.pln("primitive Array before Sorting:");
    for(int a1 : a)
    {
      S.o.pln(a1);          10
    }                       5
                            20
                            11
                            6
    Arrays.Sort(a);
    S.o.pln("primitive Array After Sorting:");
    for(int a1 : a)
    {
      S.o.pln(a1);          5
    }                       6
                            10
                            11
                            20
    String[] s = {"A", "Z", "B"};
    S.o.pln("object Array Before Sorting:");
    for(String a2 : s)
    {
      S.o.pln(a2);      A
    }                   Z
                        B
    Arrays.Sort(s);
    S.o.pln(" object Array After Sorting:");
for(String a1 : s)
{
  S.o.pln(a1);        A
                      B
                      Z
}
Arrays.sort(s, new MyComparator());
S.o.pln("Object Array After Sorting by Comparator:");
for(String a1 : s)
{
  S.o.pln(a1);        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);
} }
```

Searching the elements of Array:-

→ Arrays class defines the following search methods for this.

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

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

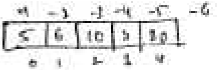
③ public static int binarySearch(Object[] o, Object key, Comparator c)

Note:

All rules of these binarySearch() method are Exactly Same as Collections class binarySearch() method.

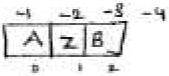
Ex:-

```
import java.util.*;
import static java.util.Arrays.*;
class ArraysSearchDemo
{
  p.s.v.m(______)
  {
    int[] a = {10, 5, 20, 11, 6};
    Arrays.sort(a);  // Sort by natural order
```

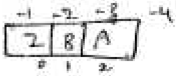


```
    S.o.pln(Arrays.binarySearch(a,6)); //1
    S.o.pln(Arrays.binarySearch(a,14)); // -5

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



```
    System.out.println(binarySearch(s,"Z")); //2
    S.o.pln(binarySearch(s, "S")); // -3
    Arrays.Sort(S, new MyComparator());
```



```
    S.o.pln(binarySearch(S, "Z", new MyComparator()); // 0
    S.o.pln(binarySearch(S, "S", new MyComparator()); // -2
    S.o.pln(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);
    }
}
```

Converting Arrays to List :-

⊙ public static List asList(Object[] a)

→ By using this method we are not creating an independent List Object just we are creating List view for the existing Array Object.

→ By using List reference if we perform any operation the changes will be reflected to the Array reference. Similarly, By using Array reference if we perform any changes those changes will be reflect to the List.

→ By using List reference we can't perform any operation which varies the size, (i.e, add & remove) otherwise we will get RuntimeException saying "unSupported-Operation-Exception" (USOE)

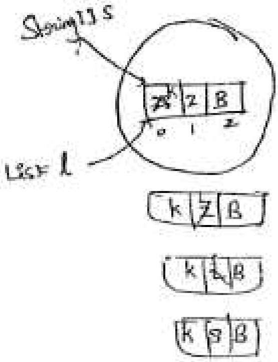
→ By using List reference we can perform replacement operation But replacement should be with the same-type of element only otherwise we will get RuntimeException saying "ArrayStoreException"

Ex:- To view Array IN List FORM.

ArrayAsListDemo.java

```
import java.util.*;

Class ArraysAsListDemo
{
 public static void main(String[] args)
 {
   String[] s = {"A", "Z", "B"};
   List l = Arrays.asList(s);
   S.o.pln(l); // [A, Z, B]
   s[0] = 'K';        [A, Z, B]  [K, Z, B]
   S.o.pln(l); // [K, Z, B]
   l.set(1, "L");    [K, Z, B]
                         L
   for(String s1 : s)
    S.o.pln(s1); //[K, L, B]
    l.add("durga"); R.E // USOE
    l.remove(2); R.E // USOE
    l.set(1, "S");    [K, L, B] => [K, S, B]
    l.set(1, 10); R.E // ArrayStoreException
 }
}
```


String[] s
A K
Z
B
0
1
2
List l
K Z B
K L B
K S B