Collections

Date / / Page

⇒ An Array is an indexed collection of fixed number of Homogeneous data elements.

⇒ The main Advantage of Arrays is we can represent multiple values by using single variable, so that Readability will be improved.

⇒ Limitations of Arrays :-

(1) Arrays are fixed in size i.e once we creates an array there is no chance of increasing or decreasing of size based on our requirement, Due to this to use Arrays, compulsory we should know the size in advance, which is not possible always.

(2) Array can hold only homogeneous data types elements.

```
Student[] s = new Student[10000];
S[0] = new Student(); ✓
S[1] = new Customer(); ✗
```

(i.e Incompatible types
found: Customer required: Student

⇒ We can solve this problem by using Object type Arrays.

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

(3) Arrays concept is not implemented based on some standard Data structure & Hence Readymade method support is not available, so for every requirement we have to write code explicitly, so that will increase complexity of program.

⇒ To overcome above problems of Arrays we should go for Collections concept.

(1) Collections are growable in nature. i.e based on our requirement we can increase / decrese the size

(2) Collection can hold both homogeneous / heterogeneous objects.

(3) Every Collection class is implemented based on some standard Data structure, hence for every requirement

readymade support is available.

⇒ So, Being a programmer, we are responsible to use those methods & not responsible to implement those methods.

✱ Differences b/w Arrays / Collections

Arrays	Collections
1) Arrays are fixed in size, i.e once we creates an Array, we can't change size.	Collections are growable & shrinkable in nature.
2) W.r.t Memory Arrays are not recommended to use.	W.r.t Memory Collections are recommended to use.
3) W.r.t performance Arrays are recommanded to use	W.r.t performance Collections are Not recommanded to use.
4) Arrays can hold only homogeneous data type elements	Collections can hold both homogeneous / heterogeneous elements
5) There is not underlying DS for Arrays.	Every collection class has standard underlying DS.
★ 6) Arrays can hold both primitives & objects.	Collections can hold only object but not primitives.

✱ Collection & Collection Framework

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

↓

To represent a group of individual object as a single entity, we have to use several [classes + interfaces] which is called collection Framework.

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 applicable for any Collection object.

↝ In general, Collection interface is considered as Root interface of Collection Framework

★ ↝ There is no concrete class, which implements Collection interface directly.

⇒ Difference b/w Collection & Collections

Collection → (Some as above definitions) → Interface

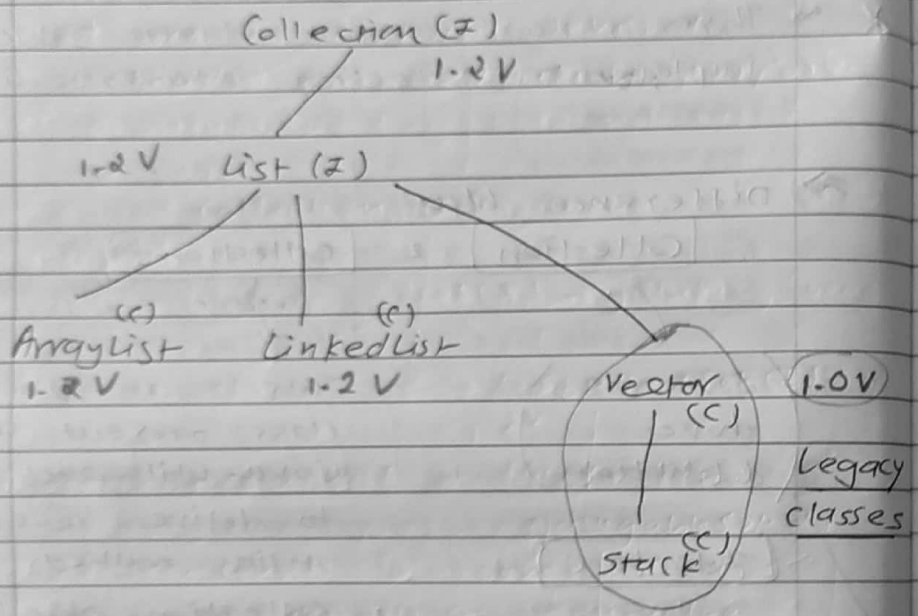
Collections → This is an Utility class present in java.util package to define several utility methods for Collection object like sorting, searching etc → Class

② List (I)

↝ Child interface of Collection (I).

↝ If we want to represent a group of individual object as a single entity, where duplicates are allowed & insertion order must be

preserved, then we should go for List(I).



Note: In 1.2V, Vector & Stack classes are re-engineered to implement List Interface

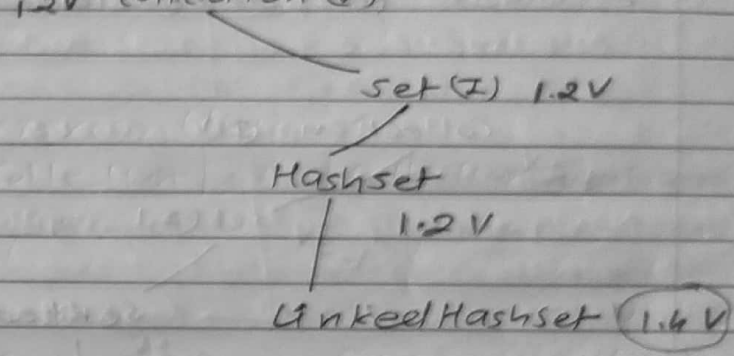
(3.) Set (I)

↝ Child interface of Collection (I)

↝ If we want to represent a group of individual object as a single entity where

- duplicates are Not allowed &
- Insertion order Not required

then we should go for Set (I).



(4.) SortedSet (I)

↝ Child interface of Set (I).

↝ If we want to represent a group of individual object as a single entity where

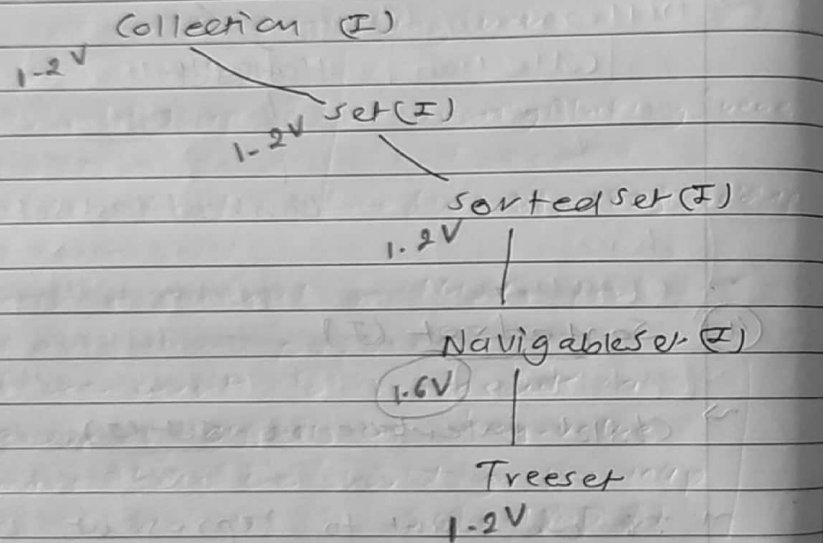
- duplicates are Not allowed &
- All objects should be inserted according to some sorting order

then we should go for sorted set.

(5.) NavigableSet (I)

~ child interface of SortedSet (I)

~ It contains several methods for Navigation purpose.



⇒ Differences b/w List & Set

List	Set
1.) Duplicates are allowed	Duplicate are Not allowed
2.) Insertion order preserved.	Insertion order Not preserved.

(6.) Queue (I)

⇒ child interface of Collection (I).

⇒ If we want to represent a group of individual objects prior to processing then we should go for queue(I).

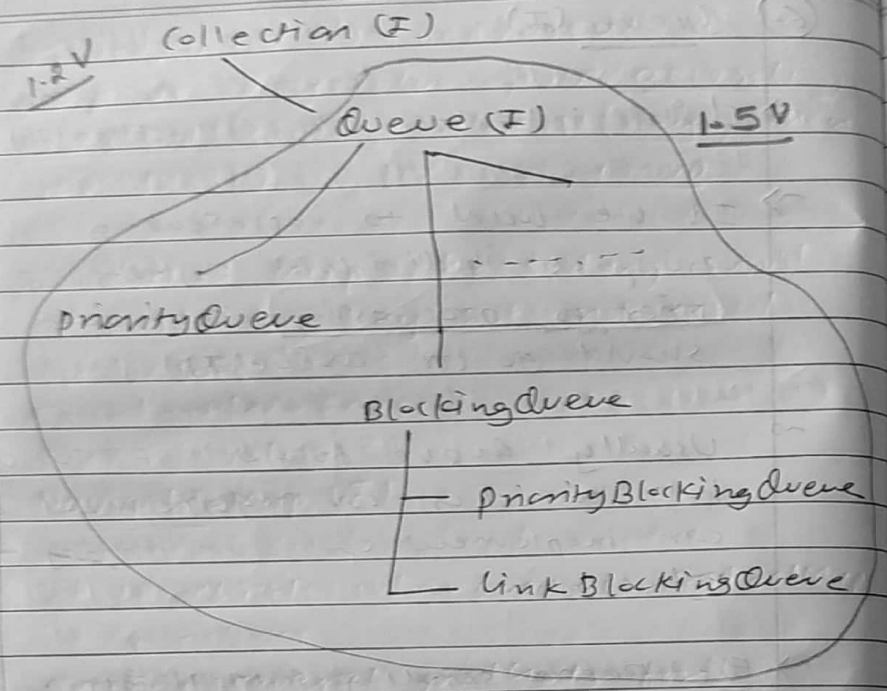
~ Usually Queue follows FIFO order but based on our requirement we can implement our own priority order also.

~ Ex. Before sending a Mail all Mail-ID we have to store in some Data Structure, In which order we add Mail-ID, In same order only Mail should be delivered, For this requirement Queue is best choice.

(Figure in next page)

(7.) Map (I)

Note: All the above interfaces (Collection, List, Set, SortedSet, NavigableSet & Queue) meant for representing a group of individual objects



It we want to represent a group of objects or key-value pairs then we should go for Map

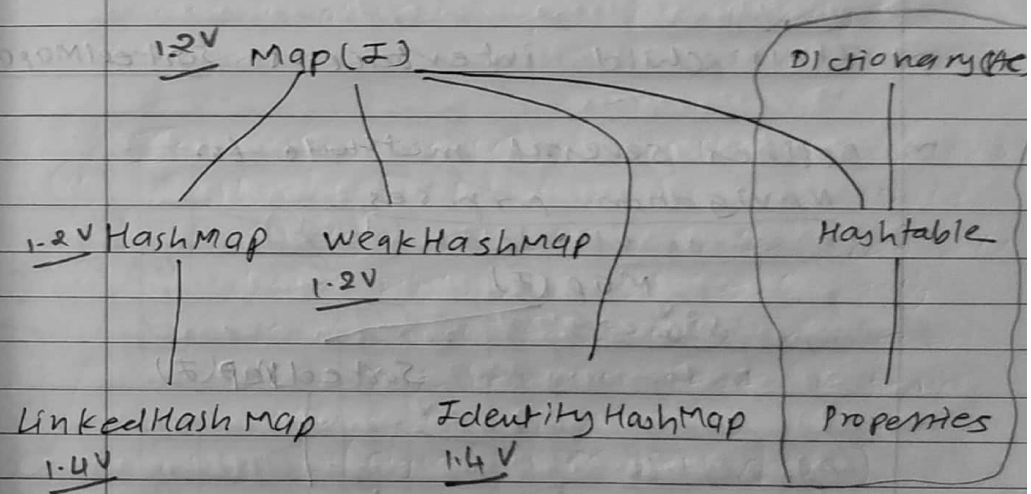
★ ⇝ Map is a part of Collection Framework, but not child interface of Collection(I)

⇝ It we want to represent a group of objects as key-value pair then we should go for Map

Key	Value
Roll NO	Name
101	A
102	B
103	C

⇝ Both key & value are objects only.

⇝ Duplicate keys are Not allowed but values can be duplicated.



(8.) SortedMap (I)

⇝ child interface of Map(I).

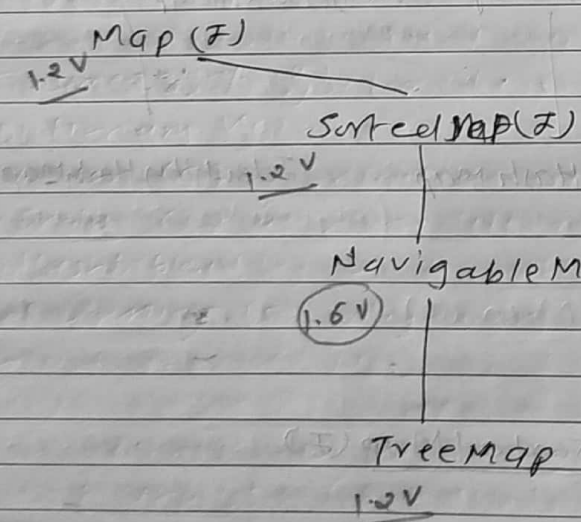
~ If we want to represent a group of key-value pair according to some sorting order of keys, then we should go for SortedMap.

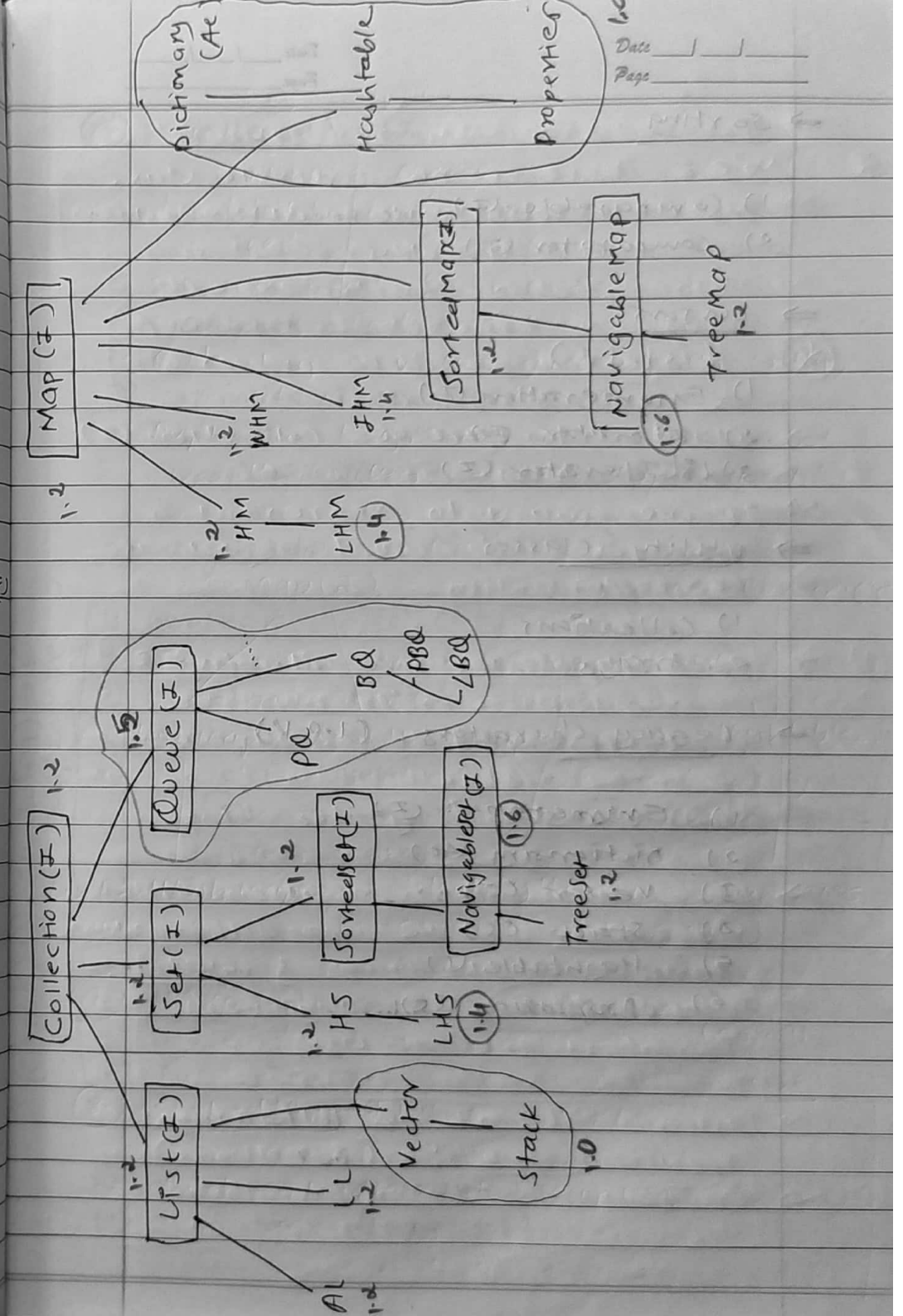
~ In SortedMap, the sorting should be based on key but not based on values.

(9.) NavigableMap (I)

~ It is child interface of SortedMap(I)

~ Defines several methods for Navigation purposes.





→ Sorting

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

→ Cursors

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

→ Utility classes

1) Collections
2) Arrays

→ Legacy Characters (1.0 V)

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

✱ 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 applicable for any collection object.

→ Methods : (12 Methods)

(Updation) → (6) Add/Remove

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()

(Check) → (4)

7) boolean contains (Object o)
8) boolean containsAll (Collection c)
9) boolean isEmpty()
10) int size()

11) Object[] toArray()
12) Iterator iterator()

★ Note :- There is no concrete class which implement Collection (I) directly

⊛ List (I)

⇝ Child interface of Collection (I)

⇝ If we want to represent a group of individual object as a single entity
where
- duplicates are Allowed &
- Insertion order must be preserved

then we should go for List.

⇝ we can preserve Insertion order via index & we can differentiate duplicate object by using index. Hence (index) will play very important role in List

⇝ List (I) defines the following specific methods:- (8 Methods)

1) void add (int index, Object o)
2) boolean addAll (int index, Collection c)
3) Object get (int index)
4) Object remove (int index)
5) Object set (int index, Object newObj)
   [To replace the element present at specified index with provided new Object & returns old object]
   ↗ Replace
6) int indexOf (Object o)
   [Returns index of First Occurence of 'o']
7) int lastIndexOf (Object o)

[Note :- No specific method for intermediate occurences we have to write logic code]

8) ListIterator listIterator();

① ArrayList

⇝ The underlying Data Structure is
" Re-sizeable Array "
or
" Growable Array "

↝ Duplicate objects are allowed.

↝ Insertion order is preserved.

↝ Heterogeneous objects are Allowed.
(Except TreeSet & TreeMap
Everywhere Heterogeneous
objects are allowed)

★ ↝ Null Insertion is possible.

→ **constructors :-**

① ↝ `Arraylist l = new Arraylist();`

creates an Empty Arrylist object with the default initial capacity (10).

once Arraylist reaches its max capacity then a new Arraylist object will be created with the

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

② ↝ `Arraylist l = new Arraylist (int InitialCapacity);`

creates an Empty Arraylist object with specified capacity.

③ ↝ `Arraylist l = new Arraylist (Collection c);`

creates an Equivalent Arraylist object for the given collection.

Ex:
```
class ArraylistDemo
{
    public static void main (Str )
    {
        Arraylist l = new Arraylist();
        l.add ("A");
        l.add (10);
        l.add ("A");
        l.add (null);
        S.O.P (l);          // [A, 10, A, null]

        l.remove(2);
        S.O.P (l);          // [A, 10, null]

        l.add(2, "M");
        l.add ("N");
        S.O.P (l);          // [A, 10, M, null, N]
    }
}
```

Note :- In ArrayList, when we try to print object directly, toString() method get called which is overrided in a way that it prints like [, ,]

In Map, we will get o/p like { key=value, key=value, ... }

→ usually we can use collection to hold & transfer object from location to another location. To provide support for this requirement, every collection class implements Serializable & Cloneable Interfaces

→ ArrayList & Vector classes implements RandomAccess Interface, so that any random element we can access with the same speed.

→ RandomAccess (I)

→ RandomAccess interface present in java.util package & it doesn't contain any method which is a marker interface, where required ability will be provided by JVM automatically

Ex.

```
ArrayList al = new ArrayList();
LinkedList ll = new LinkedList();

S.O.P(al instanceof Serializable); //true
S.O.P(ll instanceof Cloneable); //true
S.O.P(al instanceof RandomAccess); //true
S.O.P(ll instanceof RandomAccess); //false
```

→ When to use ArrayList? (or) (Best choice of ArrayList)

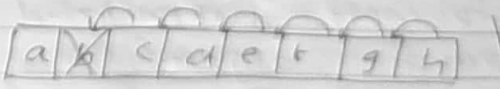
→ ArrayList is a best choice if our frequent operation is retrieval operation, because ArrayList implements RandomAccess interface.

Bcz, objects get stored in consecutive memory location, which can be easily identified directly via index

→ (Worst choice of ArrayList)

→ ArrayList is a worst choice if our frequent operation is insertion/deletion in the front/middle

(Bcz, if we remove element from front all elements from behind gets front side by shifting



)

Interview

⇒ Differences b/w ArrayList & Vector

ArrayList	Vector
1) Every method present in ArrayList is Non-synchronized.	Every method present in Vector is synchronized
2) At a time multiple threads are allowed to operate on ArrayList object hence it is Not Thread-safe	At a time only one thread is allowed to operate on Vector object hence it is Thread-safe.
3) Relatively performance is high, bcz threads are not required to wait to operate on ArrayList object	Relatively performance is low, bcz threads are required to wait to operate.
4) Introduced in 1.2 V so it is Non-legacy	Introduced in 1.0 V. so it is Legacy.

Interview

Q. How to get synchronized version of ArrayList object?

Ans: 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 al = new ArrayList();

List l = Collections.synchronizedList(al);
```

l → Synchronized

al → Non-synchronized

→ Similarly we can get synchronized version of Set & Map objects by using following methods of Collections class

```
public static Set synchronizedSet(Set s)

public static Map synchronizedMap(Map m)
```

(2) Linked List

→ The underlying Data structure is "Doubly Linked List"

→ Insertion order preserved.

→ Duplicate objects are allowed.

→ Heterogeneous object are allowed

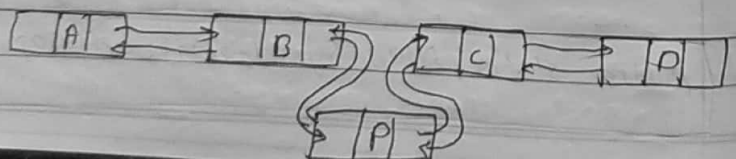
★ → null insertion is possible.

→ Linked list implements Serializable & Cloneable interfaces but (Not) RandomAccess

→ **Best choice :-**

If our frequent operation is insertion / deletion in the middle bcz, here shifting process not get done, here pointer only gets change

insert(2, "pk");



→ **Worst choice**

When our frequent operation is (retrieval), bcz, we can't access elements randomly, we know the address of first one only, so if we want to access 3rd element we have to go through 1st & 2nd element as they give us address of immediate next element only.

→ **Constructors :-**

→ As Linked list elements get stored in random places, so No capacity term needed.

① → `LinkedList ll = new LinkedList();`

Creates an empty Linked List object

② → `LinkedList ll = new LinkedList (Collection c);`

Creates an equivalent Linked list object for a given collection.

Date __/__/____
Page ____

⇒ LinkedList class-specific Methods

~ usually we can use LinkedList to develop stacks & queues, To provide support for requirement LinkedList class defines the following specific methods :-

1. void addFirst(Object o)
2. void addLast(Object o)
3. Object getFirst()
4. Object getLast()
5. Object removeFirst()
6. Object removeLast()

Ex:

```
class LinkedListDemo
{
    public static void main(String )
    {
        LinkedList ll = new LinkedList();

        ll.add("durga");
        ll.add(30);
        ll.add(null);
        ll.add("durga");

        s.o.p(ll);
        // [durga, 30, null, durga]

        ll.set(0, "software");   (Replace)
        s.o.p(ll);
        // [software, 30, null, durga]

        ll.add(0, "venky");   (Add)
        s.o.p(ll);
        // [venky, software, 30, null, durga]

        ll.removeLast();
        s.o.p(ll);
        // [venky, software, 30, null]

        ll.addFirst("ccc");
        s.o.p(ll);
        // [ccc, venky, software, 30, null]
    }
}
```

⇒ Differences b/w ArrayList & LinkedList

	ArrayList	LinkedList
1)	AL is a best choice if our frequent operation is retrieval	LL is a best choice if our frequent operation is insertion/deletion in middle.
2)	AL is a worst choice if our frequent op. is insertion/deletion in middle (∵ shifting)	LL is a worst choice if our frequent operation is retrieval (∵ NO RandomAccess)

3) In AL, elements are stored in consecutive memory location, so retrieval become easy	In LL, element won't be stored in consecutive Memory location, so, retrieval becomes complex

(3) Vector

→ Underlying DS is
"Re-sizeable Array" (or)
"Growable Array"

~ Insertion order preserved.

~ Duplicates are allowed

~ Homogeneous objects are allowed

★ ~ null insertion is possible

~ It implements Serializable, Cloneable & RandomAccess interfaces

★ ~ Every method present in vector is synchronized, & hence Vector object is thread safe (only difference when compared to ArrayList)

→ Constructors :

1) → `Vector v = new Vector();`

Creates empty Vector object with default initial capacity = 10

once vector reaches its max capacity then a new vector object will be created with new capacity

`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 incrementalCapacity);`

After filling initial capacity, rather than double the capacity, it increases the capacity by given (specified) incremental capacity.

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

creates an equivalent Vector object of a given collection.
This constructor meant for inter-conversion b/w collection objects

→ Vector specific Methods (in ▭)

→ To Add Objects :

1) add(Object o) ——— C
2) add(int index, Object o) —— L
3) addElement(Object o) —— V

→ To Remove 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 Get Objects :

1) Object get(int index) ——— L
2) Object elementAt(int index) —— V
3) Object firstElement() —— V
4) Object lastElement() —— V

→ Other Methods :-

1) int size() C
2) int capacity() V
3) Enumeration elements() V

→ Ex.

```
class VectorDemo
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        S.O.P(v.capacity());
        //10

        for(int i=1; i<=10; i++)
        {
            v.addElement(i);
        }
        S.O.P(v.capacity()); //10
        v.addElement("A");
        S.O.P(v.capacity()); //20
        S.O.P(v); //[1,2,3,...,10,A]
    }
}
```

Date __/__/____
Page _____

(4) Stack

~ Child class of Vector(class)

~ Specially designed class for LIFO (Last In First out) Order.

~ Constructor :-

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

~ Methods & (Stack specific)

1) Object push(Object o)
   To insert an object into stack

2) Object pop()
   To remove & return top of stack.

3) Object peek()
   To return top of stack without removing

4) boolean empty()
   To check is stack is empty or not

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

Date __/__/____
Page _____

	object	index
1	C	2
2	B	1
3	A	0

search("A"); ⇒ 3
search("z"); ⇒ -1

⇒ EX-

```
class StackDemo
{
    psvm(sh)
    {
        Stack s = new Stack();

        s.push("A");
        s.push("B");
        s.push("C");

        s.o.p(s); //[A, B, C]
        s.o.p(s.search("A")); //3
        s.o.p(s.search("z")); //-1
    }
}
```

✱ 3 cursors of Java

- If we want to get one by one from collection then we should go for cursor.
- There are 3 types of cursors available in java

1) Enumeration
2) Iterator
3) ListIterator

(1) ⇒ Enumeration :-

- we can use Enumeration to get object one by one from Legacy Collection Object.
- we can get Enumeration object from elements methods of Vector class.

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

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

→ methods :-

(1) public boolean hasMoreElements();

(2) public Object nextElement();

→ Ex.

```
class EnumerationDemo
{
    p s v m(Stm)
    {
        Vector v = new Vector();
        for(int i=0; i<=10; i++)
        {
            v.addElement(i);
        }
        S.O.P(V); //[0,1,2,3,...,10]

        Enumeration e = v.elements();

        while(e.hasMoreElements())
        {
            Integer I =
            (Integer) e.nextElement();

            if(I%2==0)
                S.O.P(I); //0 2 4 6 8 10
        }
        S.O.P(V); //[0,1,2,3,...,10]
    }
}
```

Date / / Page

⇝ Limitations :-

① we can apply Enumeration concept only for legacy classes & It is not a universal cursor.

② By using Enumeration, we can get only (read) access & we can't perform (remove) operation.

⇝ To overcome above limitations, we should go for Iterator.

② ⇝ Iterator (I) :-

⇝ we can apply Iterator concept for any Collection object & hence It is universal cursor.

⇝ By using Iterator, we can perform both (remove) & (read) operations.

⇝ we can get Iterator object by using iterator method of Collection Interface

```
public Iterator iterator()
```

Ex:
```
Iterator itr = c.iterator();
```
c = Any Collection Object

Date / / Page

⇝ Methods :-

(1) public boolean hasNext()

(2) public Object next()

(3) public void remove() Extra

⇝ Ex :-

```
class IteratorDemo
{
    public static void main(String[] )
    {
        ArrayList al = new ArrayList();
        for(int i=0; i<=10; i++)
        {
            al.add(i);
        }
        S.O.P(al); //[0,1,2,3,....,10]

        Iterator itr = al.iterator();
        while(itr.hasNext())
        {
            Integer I = (Integer)itr.next();
            If(I%2==0)
                S.O.P(I); 0,2,4,6,8,10
```

```
        else
            itr.remove();
        }System.out.println(al); //[0, 2, 4, 6, 8, 10]
    }
}
```

★ ★

⇒ Limitations :-

① By using Enumeration & Iterator we can always move only towards forward direction & we can't move towards Backward direction, These are single Direction cursor but Not Bi-directional cursor.

② By using Iterator, we can perform only read & remove operation, but we can't perform add & replace operation.

~ To overcome above limitations we should go for ListIterator;

③ ⇒ ListIterator (I) :-

~ By using ListIterator, we can move either to the forward direction or to the backward direction, Hence it is Bi-Directional cursor.

~ By using ListIterator we can perform replacement & addition of new objects in addition to read & remove operations.

~

```
public ListIterator listIterator()
```

we can get ListIterator object by using listIterator() method of List Interface.

```
ListIterator itr = l.listIterator();
                   ↓
                Any List object.
```

★ ~ ListIterator is a child Interface of Iterator & hence all methods present in Iterator by default available to ListIterator.

⇒ Methods :-

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

Forward movement

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

Backward movement

(7) public void remove()
(8) public void add(Object o)
(9) public void set(Object o)

Extra Operations

⇒ Ex.

```
class ListIteratorDemo
{
    public static void main(String...)
    {
        LinkedList ll = new LinkedList();
        ll.add("bala");
        ll.add("Venki");
        ll.add("chiru");
        ll.add("nag");

        S.O.P(ll); // [bala, Venki, chiru, nag]

        ListIterator itr = ll.listIterator();

        while(itr.hasNext())
        {
            String s = (String) itr.next();

            if(s.equals("venki"))
            {
                itr.remove();
                // [bala, chiru, nag]
            }
            else if(s.equals("nag"))
            {
                itr.add("chaitu");
                // [bala, chiru, nag, chaitu]
            }
            else if(s.equals("chiru"))
            {
                itr.set("charan");
                // [bala, charan, nag, chaitu]
            }
        }
    }
```

```
        }
    }
```
S.o.p(dl); [bala, charan, nag, chaitu]

⇒ The most powerful cursor is "ListIterator"

But its limitations is its only applicable for List objects.

⇒ Comparison table for cursors

property	Enumeration	Iterator	ListIterator
① Where we can apply?	only for Legacy classes	For Any collection objects	Only for List objects
② Is it Legacy?	yes (1.0 v)	NO (1.2 v)	NO (1.2 v)
③ Movement	Only Forward direction	Only Forward direction	Bi-directional (Forward + Backward)
④ Allowed Operations	only → Read	→ Read → Remove	→ Read → Remove → Add → Replace
⑤ How we can get?	By using elements() of Vector class	By using iterator() of Collection I	By using listIterator() of List I
⑥ Methods	2 Methods - hasMoreElements() - nextElement()	3 Methods - hasNext() - next() - remove()	9 Methods

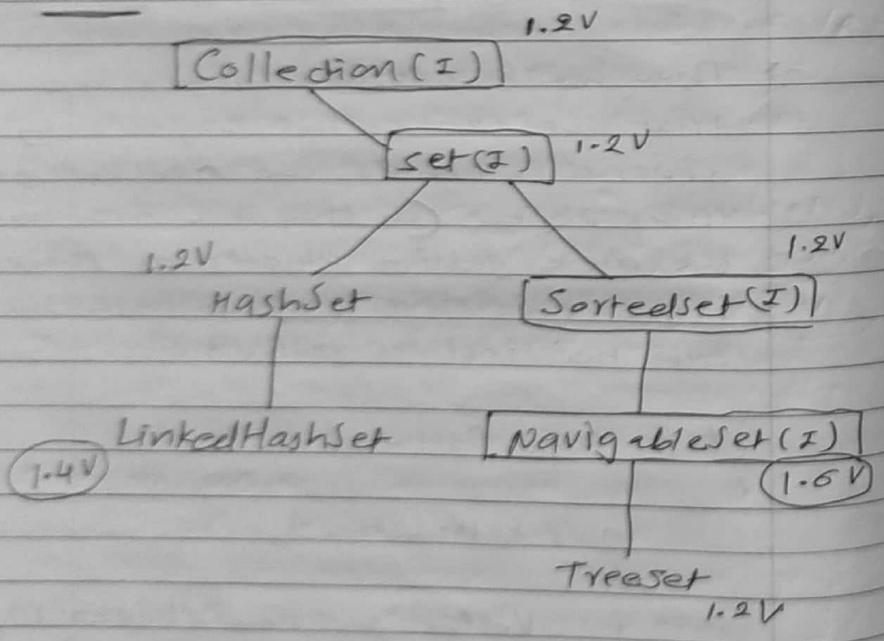
★ ⇒ Internal Implementations of Cursors

⇒ Though in previous pages we said that we get cursor objects by several methods, we are not getting their object (as they are Interfaces) we were getting objects of their implementation class, whose names can be find as follows:-

```
class CursorDemo
{
    psvm(sa)
    {
        Vector v = new Vector();

        Enumeration e = v.elements();
        Iterator itr = v.iterator();
```

```
ListIterator itr = v.listIterator();

s.o.p(e.getClass().getName());
// java.util.Vector$1

s.o.p(itr.getClass().getName());
// java.util.Vector$Itr

s.o.p(ltr.getClass().getName());
// java.util.Vector$ListItr
}
}
```

⑦ Set (I)



↝ Set is child interface of Collection (I)

↝ If we want to represent a group of individual objects as a single entity where

- duplicates are not allowed &
- Insertion order not preserved.

★ ↝ Set interface doesn't contain any new method & we have to use only collection method.

Note: Remember List (I) provides 8 extra specific methods apart from Collection (I) methods

① HashSet

↝ The underlying "Data Structure" is "Hash Table"

↝ Duplicate objects are Not Allowed.

↝ Insertion order is not preserved & It is based on "Hashcode" of objects

Date / / Page

- null insertion is possible (only once)
- Heterogeneous objects are Allowed
- Implements Serializable, Cloneable but Not RandomAccess interface.
- Hashset is a **best choice** if our frequent operation is search operation.

★ **Note:** In Hashset duplicates are not allowed, If we are trying to insert duplicate, we won't get any C.E. / R.E. but add() method simply returns false.

Eg. Hashset hs = new Hashset();
s.o.p(hs.add("A")); // (true)
s.o.p(hs.add("A")); // (false)

⇒ **Constructors :-**

① ⇒ `Hashset hs = new Hashset();`

Creates an Empty Hashset object with default initial capacity = 16 & default fill Ratio = 0.75

② ⇒ `Hashset hs = new Hashset(int initialCapacity);`

Creates an Empty Hashset object with specified initial capacity & default Fill Ratio = 0.75

③ ⇒ `Hashset hs = new Hashset(int initialCapacity, float fillRatio);`

Creates an Empty Hashset object with both specified values

④ ⇒ `Hashset hs = new Hashset(Collection c);`

Creates an equivalent Hashset for given collection object (Meant for interconversion)

⇒ **Fill Ratio / Load Factor**

- After filling how much ratio a new Hashset object will be created, this ratio is called Fill Ratio / Load Factor

Ex. Fill Ratio = 0.75 means After Filling 75% Ratio, A new Hashset object will created.

⇒ Ex.

```
class HashsetDemo
{
    p s v m(sh)
    {
        Hashset hs = new Hashset();

        hs.add("B");
        hs.add("C");
        hs.add("D");
        hs.add("Z");
        hs.add(null);
        hs.add(10);

★       s.o.p(h.add("Z"));//false

        s.o.p(hs); [null, D, B, C, 10, Z]
    }
}
```

This is totally random, we can't predict this

Here After Fill Ratio,

New capacity	=	current capacity	* 2

(2) LinkedHashset

⇒ It is the child class of Hashset. It is exactly same as Hashset (including constructors & methods) Except the following differences.

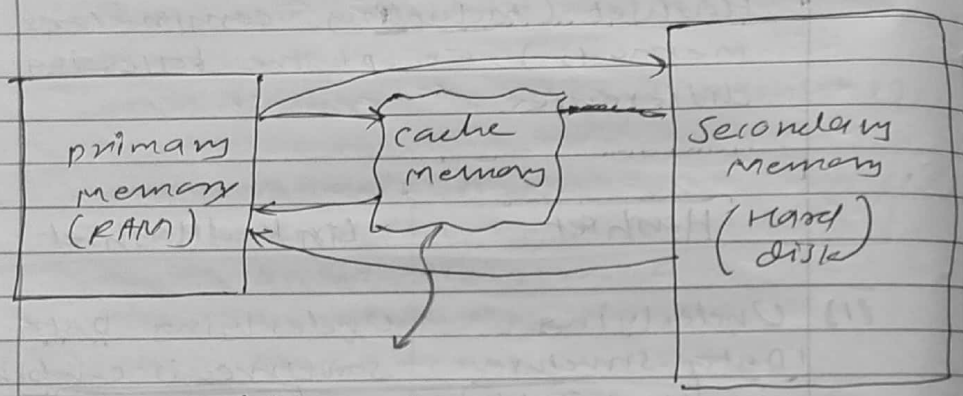
	Hashset	LinkedHashset
(1)	Underlying Data structure is HashTable	Underlying Data structure is combination of LinkedList & HashTable means Hybrid DS
(2)	Insertion order Not preserved	Insertion order preserved
(3)	Introduced in 1.2V	Introduced in 1.4V

⇒ In last program of Hashset, if we replace Hashset with LinkedHashset, then the o/p will be totally predictable bcz here insertion order get preserved

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

Date / / Page

⇝ In general, we can use LinkedHashSet to develop cache based application where duplicates are not allowed & ins. Insertion order preserved.



(Here also, duplicate not allowed & Insertion order preserved so, LinkedHashSet ⇒ Best choice)

(3.) SortedSet (I)

⇝ SortedSet(I) is a child interface of Set(I).

⇝ If we want to represent a group of individual Objects according to some sorting order without duplicates then we should go for SortedSet.

⇝ SortedSet(I) defines some 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 sortedset whose elements are less than obj)

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

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

★ (6) Comparator comparator()
(returns Comparator object that describes underlying sorting technique. If we are using "default" natural sorting order then we will get (null))
For, Numbers ⇒ Ascending order
String ⇒ Alphabetical order

Ex:

SortedSet

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

100
101
104
106
110
115
120

④ TreeSet

⇝ Underlying Data Structure is "Balanced Tree"

⇝ Duplicate objects are Not Allowed

⇝ Insertion order not preserved

⇝ Heterogeneous objects are not Allowed
If we do so, we will get ClassCastException

⇝ null insertion possible (But only once)

Until 1.6 V only

⇝ TreeSet implements Serializable, Cloneable but not RandomAccess interface

⇝ All objects will be inserted based on some sorting order, It may be "default sorting order" or "customized sorting order"

⇝ Constructors :-

① ⇝
```
TreeSet ts = new TreeSet();
```
creates an empty TreeSet object where elements will be inserted according to "default natural sorting order"

② ⇝
```
TreeSet ts = new TreeSet
                (Comparator c);
```
creates an empty TreeSet object where elements will be inserted according to "customized sorting order" specified by comparator object

③ ⇒ `Treeset ts = new Treeset (Collection c);`

created an Treeset object corresponding to given collection object & Here As collection object can be anything, so "Default sorting order" is followed

④ ⇒ `Treeset ts = new Treeset (SortedSet s);`

created Treeset Object corresponding given sortedset object & Here sorting order is same as sorting order followed in given sortedset object

Ex. 1

```
class TreesetDemo1
{
    p s v m (Sn)
    {
        Treeset ts = new
                     Treeset ();

        t.add ("A");
        t.add ("a");
        t.add ("B");
        t.add ("z");
        t.add ("4");

        // t.add (new Integer (10)); // CCE

        // t.add (null);

        s.o.p (t); // [A, B, L, Z, a]
    }
}
```

unicode of { capital letters ⇒ From 65
small letters ⇒ From 97 }

⇒ **null Acceptance :-**

Rule 1 :- For Non-Empty Treeset if we are trying to insert null, then we will get NPE.

Rule 2 :- For Empty Treeset, As a first element null is Allowed, but after inserting that null, if we trying to insert any other Object, then we will get NPE.

Note: BCZ, In Treeset AJ objects/elements get inserted they get sorted according to some order via comparison with each other, so comparison of any objects with null produces NPE.

★ ⇒ so, AJ having this much confusion, null is not accepted in Treeset from 1.7V onwards

(so, No null in Treeset from 1.7V)

★ Ex-2

```
class TreeSetDemo2
{ public static void main(String s)
  {TreeSet ts = new TreeSet();

   ts.add(new StringBuffer("A"));
   ts.add(new StringBuffer("Z"));
   ts.add(new StringBuffer("L"));
   ts.add(new StringBuffer("D"));

   S.O.P(ts);
  }
}
```

⇝ O/P: we will get ClassCastException

→ If we are depending on "Default Natural sorting order" compulsory the objects should be
- Homogeneous &
- Comparable

otherwise we will get R.E saying ClassCastException.

→ An object is said to be Comparable if and only if corresponding class implements "Comparable" Interface.

- String class & All wrapper classes already implemented "Comparable" but StringBuffer doesn't implement "Comparable" interface, hence we got CCE in above example.

⇝ Comparable (I)

★ ⇝ It is present in java.lang package.

⇝ It contains only 1 method
→ compareTo()

Syntax :

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

obj1.compareTo(obj2)

→ returns -ve
if obj1 has to come before obj2 (means obj1 < obj2)

→ returns +ve
if obj1 has to come after obj2 (means obj1 > obj2)

→ returns 0
if obj1 & obj2 are equal (means obj1 = obj2)

Note : String & other wrapper classes implemented "Comparable" interface & implement or define compareTo() method (which has to be done by them as they are concrete class (child) so, we can compare their objects directly using that method

Ex.3

```
class TreeSetCompareToDemo
{
    p s v m(Str)
    {
        S.O.P("A".compareTo("z")); // -25
        S.O.P("z".compareTo("A")); // 15
        S.O.P("A".compareTo("A")); // 0
        S.O.P("A".compareTo(null)); // RE: NPE
    }
}
```

★ ⇝ If we are depending on "Default Natural Sorting Order" then while adding objects into the TreeSet, JVM will call compareTo() method.

```
TreeSet ts = new TreeSet();
ts.add("K");
```

No comparison needed, As it is first object/element

(K)

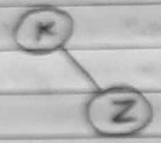
```
ts.add("Z");
```

comparison needed

JVM will call compareTo() like this...

"z".compareTo("k");

JVM will get +ve here...
So, it will add "z" After "k"



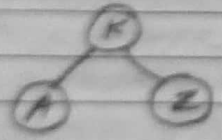
Here, noticeable thing is that when we try to add null JVM will try to call compareTo() method on null like this...

null.compareTo("k");

which is not possible

ts.add("A");

"A".compareTo("k"); (-ve)



ts.add('A');

→ "A".compareTo("A") ⇒ (+ve)

So,

→ "A".compareTo("A") ⇒ 0

So, No need of insertion
&
add() method returns false

S.O.P(ts); //[A, K, Z]

⇒ So, ... obj1.compareTo(obj2);

The object which is to be inserted

The object which is already inserted in treeset

Note :- If 'Default Natural Sorting Order'
- Not Available (like on StringBuffer)
or
- If we are not satisfied with 'Default Natural sorting order' (like we want reverse alphabetical order in string

then

we can go for "customized sorting".

which can be achievable using "comparator"

★

⇒ (Comparable) meant for "Default Natural Sorting Order"

whereas

(Comparator) meant for "customized sorting Order"

⇒ Comparator (I)

★ → Comparator present in (java.util) package.

→ It defines two (2) methods :-

1) compare()
2) equals()

1.)

public int compare(Object obj1, Object obj2)

→ returns -ve i/f/f obj1 has to come before obj2 (obj1 < obj2)

→ returns +ve i/f/f obj1 has to come after obj2 (obj1 > obj2)

→ returns 0 i/f/f obj1 & obj2 are same

2.)

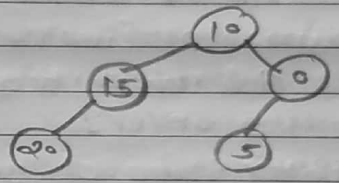
public boolean equals(Object obj)

→ whenever we are implementing Comparator (I) compulsory we should provide implementation for only compare() method & we are not required to provide implementation for equals() method, bcz it is already available to our class from object class through inheritance.

Q. WAP to insert Integer objects into Treeset, where sorting order is descending order

Ans:

```
class TreeSetDemo3
{
    p s v m(sn)
    {

line① ~~~~ TreeSet ts =
            new TreeSet(new MyComparator());

        ts.add(10);  → compare(P,10)
        ts.add(0);   → compare(15,10)
        ts.add(15);  → compare(5,10)
        ts.add(5);   → compare(5,0)
        ts.add(20);
        ts.add(20);

        S.O.P(ts); // [20,15,10,5,0]

    }
}

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



↝ At line①, if we are not passing our comparator object, then internally JVM will call compareTo() method which is meant for "Default Natural sorting order", In this case the o/p will be [0, 5, 10, 15, 20]

↝ At line①, If we are passing our comparator object then JVM will call compare() method.

↝ TreeSetDemo3 class will remain as it is, but the different possible combinations in compare() method which can change the o/p is as shown below :-

```
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        Integer I1 = (Integer)obj1;
        Integer I2 = (Integer)obj2;

(1)     return I1.compareTo(I2);

        // Default Natural sorting Order
        // [Assending 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);

        // [Assending 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 are considered as duplicate by JVM.]

        // [10]
    }
}
```

Q: WAP to insert string objects into TreeSet, where all elements should be inserted according to Reverse of Alphabetical order.

Ans:

```
class TreeSetDemo
{
    psvm(String[] sa)
    {
        TreeSet t = new TreeSet(new MyComparator());

        t.add("Raja");
        t.add("shobha");
```

```
t.add("Raja");
t.add("Ganga");
t.add("Ramu");

S.O.P(t); //[Shobha, Raja, Ramu,
                    Raja, Ganga]
    }
}

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);
    }
}
```

Q) class TreeSetDemo1
{
    p s v m (Str)
    {
        TreeSet t = new
            TreeSet(new MyComparator());

```
t.add(new StringBuffer("A"));
t.add(new StringBuffer("Z"));
t.add(new StringBuffer("K"));

S.O.P(t); //[A, K, Z]
    }
}

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

        return s1.compareTo(s2);
    }
}
```

Note: If we are depending on DNSO, compulsory the objects should be Homogeneous & comparable, otherwise we will get R.E. ClassCastException

⇒ If we are defining our own sorting by comparator then Object need not be comparable & homogeneous, so we can

add Heterogeneous & Non-Comparable objects also.

Q. WAP to insert String & StringBuffer objects into TreeSet, where sorting order is increasing length order if 2 objects having same length then consider their alphabetical order.

Ans:
```
class TreeSetDemo
{
    psvm(String[] s)
    {
        TreeSet t = new TreeSet(new MyComparator());
        t.add("A");
        t.add(new StringBuffer("ABC"));
        t.add(new StringBuffer("AA"));
        t.add("XX");
        t.add("ABCD");
        t.add("A");
        S.O.P(t); //[A, AA, XX, ABC, ABCD]
    }
}
```

```
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        int l1 = s1.length();
        int l2 = s2.length();
        if (l1 > l2)
        { return 1; }
        else if (l1 < l2)
        { return -1; }
        else
        { return s1.compareTo(s2); }
    }
}
```

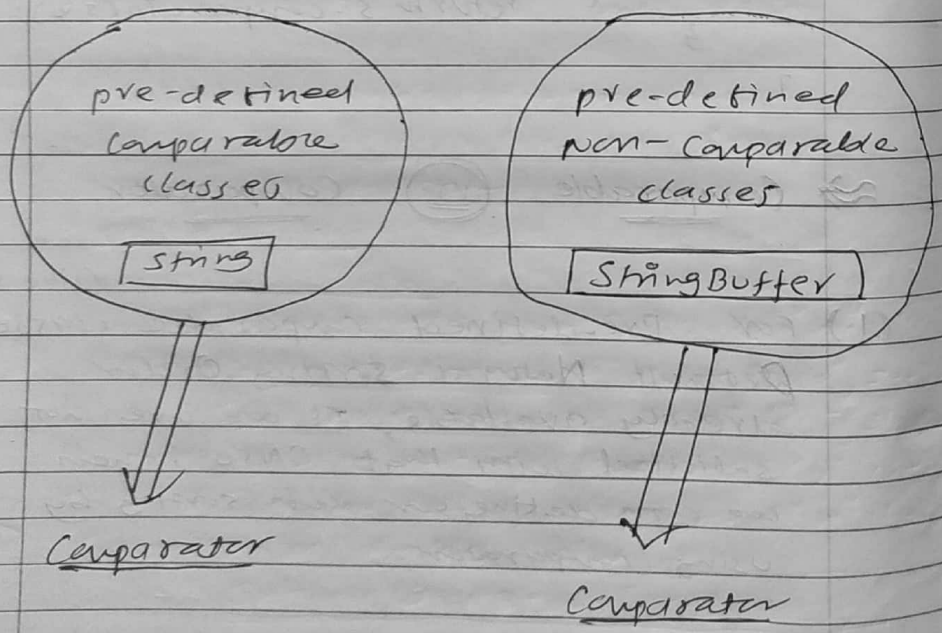
⇒ Comparable (vs.) Comparator

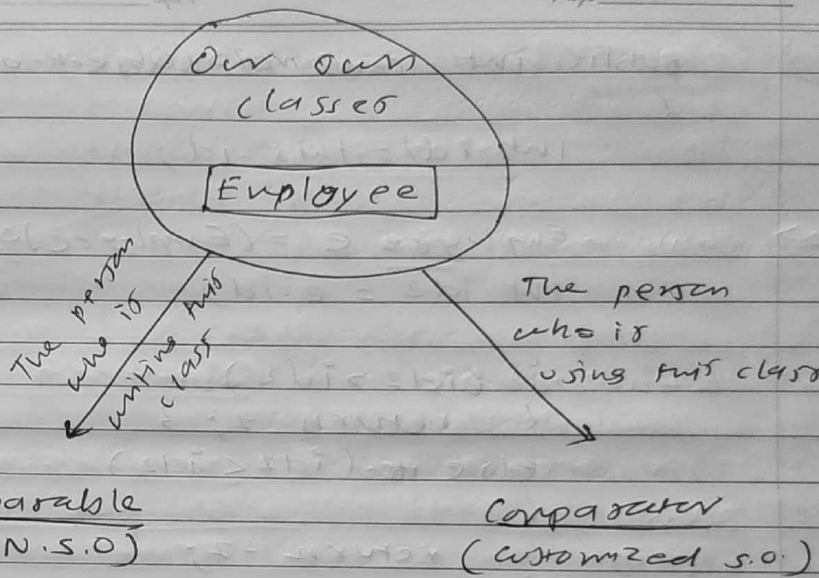
(1.) For Pre-defined comparable classes (like String) Default Natural Sorting Order already available, If we are not satisfied with that DNSO, then we can define our own sorting by using comparator

(2.) For pre-defined Non-comparable classes (like StringBuffer), DNSO not already available, we can define our own sorting by using comparator.

(3.) For our own classes (like Employee), the person who is writing the class is responsible to define DNSO by implementing Comparable(I).

The person who is using our class, if he is not satisfied with DNSO, then he can define his own sorting by using Comparator(I).





Ex. class Employee implements Comparable
{
String name;
int id;

Employee(String name, int id)
{
this.name = name;
this.id = id;
}

public String toString();
{
return name + "->" + id;
}

```
public int compareTo(Object obj)
{
    int id1 = this.id;

    Employee e = (Employee)obj;
    int id2 = e.id;

    if (id1 > id2)
    { return 1; }
    else if (id1 < id2)
    {
        return -1;
    }
    else
    { return 0; }
  }
}

class ComparableComparator
{
  p s v m(str)
  {

    Employee e1 = new Employee("n", 100);
    Employee e2 = new Employee("b", 200);
    Employee e3 = new Employee("c", 50);
    Employee e4 = new Employee("v", 150);
    Employee e5 = new Employee("n", 100);

    TreeSet t1 = new TreeSet();
    t1.add(e1);
    t1.add(e2);
    t1.add(e3);
    t1.add(e4);
    t1.add(e5);

    S.O.P(t1);
//[c->50, n->100, v->150, b->200]

    TreeSet t2 = new TreeSet(new MyComp(1));
    t2.add(e1);
    t2.add(e2);
    t2.add(e3);
    t2.add(e4);
    t2.add(e5);

    S.O.P(t2);
//[b->200, c->50, n->100, v->150]
  }
}

class MyComp 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);
    }
}
```

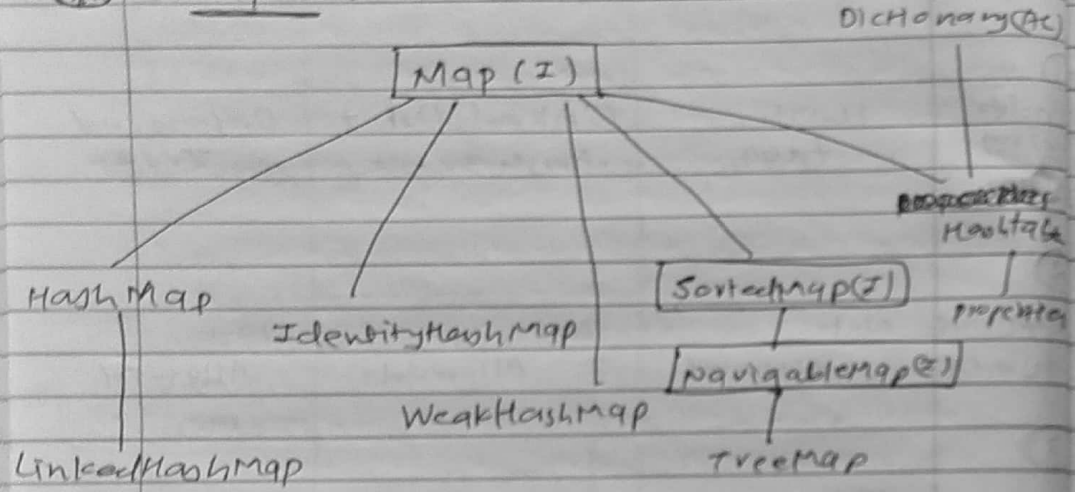
⇒ Comparison blw

Comparable	Comparator
★ 1) It is meant for DNSO	It is meant for customized S.O
2) present in java.lang.	present in java.util
3) It defines only 1 method :- compareTo()	It defines two (2) methods :- compare() equals()
4) String & All wrapper classes implements Comparable(I)	The only implemented classes of Comparator are 1.) Collator 2.) RuleBasedCollator

⇒ Comparison table blw Set classes

property	HashSet	LinkedHashSet	TreeSet
1) underlying D.S.	Hash table	Linked List + Hash table	Balanced Tree
2) Duplicate objects	Not Allowed	Not Allowed	Not Allowed
3) Insertion order	Not preserved (∵ hashcode based)	preserved	Not preserved (∵ sorting)
4) Sorting order	N.A.	N.A	Applicable
5) Hetro-genous objects	Allowed	Allowed	By default Not Allowed (But in special case like Comparator then allow)
6) null accep-tance	Allowed	Allowed	For empty TreeSet, as first element null → Allowed but only till 1.6v

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
101	Durga
102	Ravi
103	Shiva

3 → Entry

~ Both keys & values are objects only

~ Duplicate keys are not allowed, but values can be duplicated

~ Each key-value pair is called "Entry"

~ so, Map is considered as a collection of "Entry objects"

Map (I) Methods :-

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

- To add 1 key-value pair to the Map.
- If the key is already present then old value will be replaced with the new value & returns new value.

e.g. `S.O.P(m.put(101, "durga"));` //null
`S.O.P(m.put(102, "shiva"));` //null
`S.O.P(m.put(101, "Ravi"));` //durga

(Hashcode of old object)

(2) void putAll (Map m)

(3) Object get (Object key)

- Returns the value associated with specified key.

(4) Object remove (Object key)

- Removes the (entry) specified by key.

(5) boolean containsKey (Object key)

(6) boolean containsValue (Object value)

(7) boolean isEmpty()
(8) int size();
(9) void clear();

} same

★★

(10) Set keySet()
(11) Collection values()
(12) Set entrySet()

} Collection views of Map

Entry (I)

~ A Map is a group of key-value pairs & Each key-value pair is called an "Entry".

~ Hence, Map is called as collection of "Entry object".

~ without existing Map object there is no chance of existing Entry object.

~ Hence, Entry (I) is defined (inside) Map (I)

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

Entry specific Methods
- can apply only on Entry Objects

(1) HashMap

→ Underlying Data Structure is "HashTable"

~ Insertion order is not preserved & based on Hashcode of keys.

Date __/__/____
Page _____

- Duplicate keys are not allowed but values can be duplicated
- Heterogeneous objects are allowed for both key & value
- null is allowed for key (only once) null is allowed for values (for any number of times)
- HashMap implements Serializable & Cloneable interfaces but not RandomAccess
- HashMap is the best choice if our frequent operation is "searching"

Constructors :-

(1) `HashMap hm = new HashMap();`

creates an empty HashMap object with
default initial capacity = 16
default fill ratio = 0.75

(2) `HashMap hm = new HashMap(int initialCapacity);`

(3) `HashMap hm = new HashMap(int initialCapacity, float fillRatio);`

★ (4) `HashMap hm = new HashMap(Map hm);`

Interconversion b/w Map objects.

Ex.

```
class HashMapDemo
{
    p s v m(String[] args)
    {
        HashMap m = new HashMap();
        m.put("chiran", 700);
        m.put("bala", 800);
        m.put("venky", 200);
        m.put("nag", 500);

        s.o.p(m);
        // {chiran=700, bala=800, venky=200, nag=500}

        s.o.p(m.put("chiran", 1000));
        // 700
```

```
Set s = m.keySet();
s.o.p(s);

// [chiran, bala, venky, nag]

Collection c = m.values();
s.o.p(c);

// [1000, 800, 200, 500]

Set s1 = m.entrySet();
s.o.p(s1);

// [chiran=1000, bala=800,
     venky=200, nag=500]

Iterator itr = s1.iterator();

while(itr.hasNext())
{
   Map.Entry m =
      (Map.Entry) itr.next();

   s.o.p(m.getKey()+"---"
         m.getValue());
}
```

chiran --- 1000
bala --- 800
venky --- 200
nag --- 500

(In this program, all the o/p we got are totally unpredictable (order of o/p) ∵ HashMap Not preserve insertion order)

```
if(m1.getKey().equals("nag"))
{
    m1.setValue(10000);
}
}
s.o.p(m);

// {chiran=1000, bala=800,
    venky=200, nag=10000}

}
}
```

interview ★ ⇒ Differences blw HashMap & Hashtable

	HashMap	Hashtable
(1)	Every method present in HashMap is Not Synchronized	Every method present in Hashtable is Synchronized
(2)	At a time multiple thread are allowed to operate on HashMap object & Hence it Not Thread-safe	At a time only one thread is allowed to operate on Hashtable & Hence it is Thread safe

(3)	Relatively performance is high bcz threads are Not require to wait to operate on HashMap object	Relatively performance is low.
(4)	null is allowed for both key & value.	null is neither allowed in key nor in value, otherwise we will get NPE.
(5)	Introduced in 1.2V, so not legacy.	Introduced in 1.0V, so it is Legacy.

⇒ How to get synchronized version of HashMap?

```
HashMap m = new HashMap();

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

m1 → Synchronized

m → Non-Synchronized

(2) LinkedHashMap

⇒ Child class of HashMap

⇒ It is Exactly same as HashMap including Methods & Constructors except the following differences :-

	HashMap	LinkedHashMap
(1)	Underlying Data structure is "Hashtable"	Underlying Data structure is Combo of LinkedList + Hashtable (∵ Hybrid DS)
(2)	Insertion order is not preserved & it is based on Hashcode of keys.	Insertion order is preserved
(3)	Introduced in 1.2V	Introduced in 1.4V

⇒ In last HashMap program if we replace HashMap with LinkedHashMap then all the o/p are predictable bcz insertion order get preserved.

→ LinkedHashSet & LinkedHashMap are commonly used for developing "cache Based Applications"

(3) IdentityHashMap

★★ Note: In HashMap, For arranging objects JVM will use Hashcode of objects to arrange objects in particular order, so insertion order Not get preserved here.

→ But, For identifying Duplicate keys, JVM will use .equals() method.

→ It is exactly same as HashMap including methods & constructors except the following differences.

→ In the case of normal HashMap, JVM will use .equals() method to identify Duplicate keys (which is meant for content comparison)

→ But, In case of IdentityHashMap, JVM will use == operator to identify Duplicate keys (which is meant for reference comparison / address comparison)

Ex:

```
HashMap m1 = new HashMap();

IdentityHashMap m2 = new
                     IdentityHashMap();

Integer I1 = new Integer(10);

Integer I2 = new Integer(10);

m1.put(I1, "pavan");
m1.put(I2, "kalyan");

S.O.p(m1); //{10=kalyan}
```

I1.equals(I2) → true → Duplicate

```
m2.put(I1, "pavan");
m2.put(I2, "kalyan");

S.O.p(m2); //{10=pavan, 10=kalyan}
```

I1 == I2 → false → Duplicate

(4) WeakHashMap

→ It is exactly same as HashMap except the following differences:

→ In case of HashMap, Even though object doesn't have any reference, it is Not eligible for GC, if it associated with HashMap

i.e

HashMap Dominates Garbage Collector

→ But In case of WeakHashMap, If object doesn't have any references, it is eligible for GC, even though object associated with WeakHashMap.

i.e

GarbageCollector Dominates WeakHashMap

Ex.

```
class WeakHashMapDemo
{
    p s v m (String[] args)
    {
        HashMap m = new
                    HashMap();
        Temp t = new Temp();
        m.put(t, "durga");
        S.O.P(m);
        t = null;
        System.gc();
        Thread.sleep(5000);
        S.O.P(m);
    }
}

class Temp
{
    public String toString()
    {
        return "temp";
    }
    public void finalize()
    {
        S.O.P("Finalize Method called");
    }
}
```

O/P :-

```
{temp = durga}
After 5 seconds
{temp = durga}
```

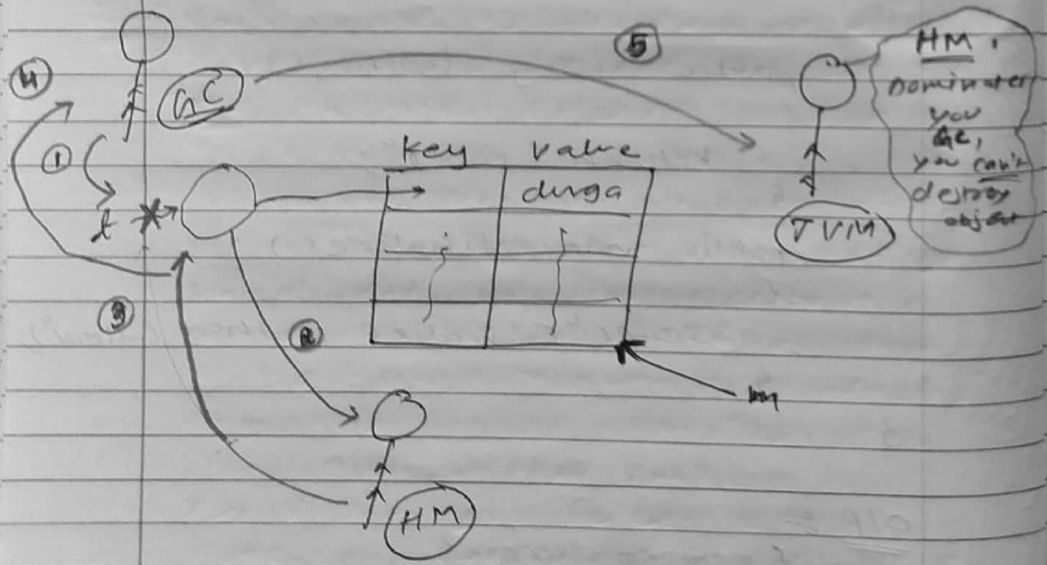
Date / / Page

→ In the above example, temp object Not eligible for GC bcz it is associated with HashMap.

→ In the above program, If we replace HashMap with WeakHashMap then temp object eligible for GC.

o/p:-

{temp = durga}
Finalize method call
After 5 seconds
{}



5. SortedMap (I)

→ child interface of Map(I).

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

→ Sorting is based on the key but Not based on value.

⇒ SortedMap defines the following Specific Methods:-

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

Ex. firstKey() ⇒ 101
lastKey() ⇒ 136

headMap(107) ⇒ {101=A, 103=B}
tailMap(125) ⇒ {125=E, 130=F}
subMap(104, 1025) ⇒ {104=C, 107=D}

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

SortedMap

comparator() ⇒ null

6. TreeMap

→ Underlying Data structure is

"RED-BLACK Tree"

→ Insertion order not preserved, it is based on some sorting order of keys.

→ Duplicate keys are Not Allowed but values can be duplicated.

→ If we are depending on DNSO then keys should be Homogeneous & comparable otherwise we will get R.E ClassCastExcp

If we are defining our own sorting order by comparator, then keys need not be Homogeneous & comparable, we can take Heterogeneous & non-comparable keys also.

- For values NO Rules.

Means whether we are using DNSO or customized.S.O, NO restrictions for values, we can take any

→ Null Acceptance :

(1) For Non-Empty TreeMap if we are trying to insert an entry with null key then we will get R.E (NPE)

(2) For Empty TreeMap, As a first entry with null key is allowed but after inserting that entry if we are trying to insert any other entry then we will get R.E (NPE)

• Note: Above null Acceptance rule applicable until 1.6V only

- From 1.7V onwards null is not Allowed for key

- But For values, we can use null any number of time, there is no restriction irrespective of version

→ Constructors :

(1)
```
TreeMap tm = new TreeMap();
```

for DNSO.

(2) `TreeMap tm = new TreeMap(Comparator c);`

for customized S.O

(3) `TreeMap tm = new TreeMap(SortedMap sm);`

(4) `TreeMap tm = new TreeMap(Map m);`

Ex. class TreeMapDemo
{
p s v m(Str)
{

TreeMap tm = new TreeMap();

tm.put(100, "zzz");
tm.put(103, "yyy");
tm.put(101, "xxx");
tm.put(104, 106);

//tm.put("FFF", "xxx"); //CCE:

//tm.put("null", "xxx"); //NPE

S.O.P(tm);
//{100=zzz, 101=xxx, 103=yyy, 104=106}
}
}

Ex. class TreeMapDemo2
{ main(){
TreeMap tm = new TreeMap(new MyComparator());

tm.put("xxx", 10);
tm.put("AAA", 20);
tm.put("zzz", 30);
tm.put("LLL", 40);

S.O.P(tm);
//{zzz=30, xxx=10, LLL=40, AAA=20}
}
}

class MyComparator implements Comparator
{

public int compare(Object obj1, Object obj2)
{
String s1 = obj1.toString();
String s2 = obj2.toString();

return s2.compareTo(s1);
}
}

7. Hashtable

- Underlying Data Structure "Hashtable"
- Insertion order is not preserved ∴ it is based on Hashcode of keys.
- Duplicate keys Not Allowed & values can be duplicated
- Heterogeneous Objects are allowed for both keys & values
- ★ null is Not Allowed for key & value otherwise we will get R.E. (NPE)

  (Here, Even value also can't be null)
- It implements serializable, cloneable interfaces but not RandomAccess
- Every method present in Hashtable is synchronized & hence Hashtable object is Thread-safe.
- Hashtable is best choice if our frequent operation is "search"

⇒ Constructors :-

(1) `Hashtable ht = new Hashtable();`

creates an Empty Hashtable object with
default initial capacity = (11)
default fill Ratio = 0.75

(2) `Hashtable ht = new Hashtable(int initialCapacity);`

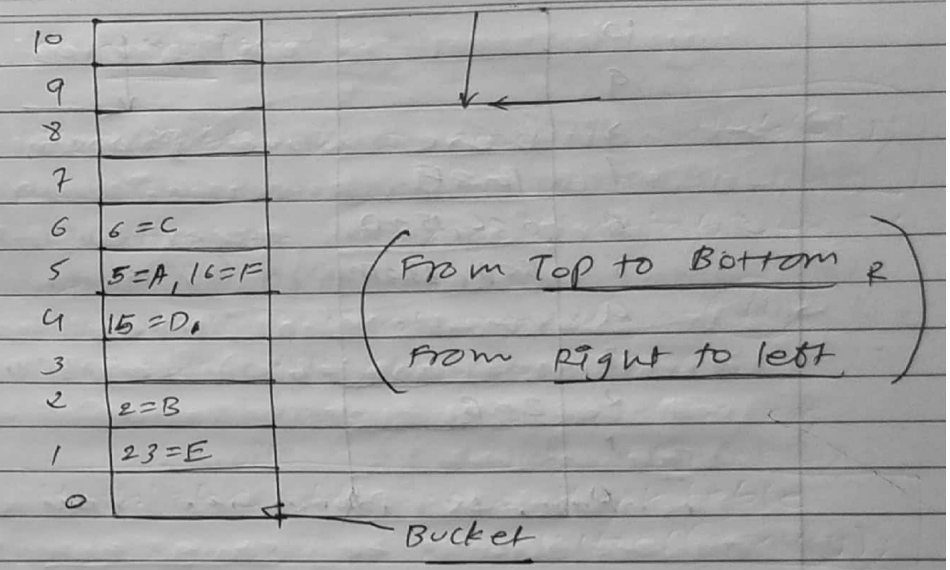
(3) `Hashtable ht = new Hashtable(int initialCapacity, float fillRatio);`

(4) `Hashtable ht = new Hashtable(Map m);`

EX.

```
class HashtableDemo
{
    p s v m(str)
    {
        Hashtable h = new
                      Hashtable();
```

```
h.put(new Temp(5), "A");
h.put(new Temp(2), "B");
h.put(new Temp(6), "C");
h.put(new Temp(15), "D");   //(15%11 = 4)
h.put(new Temp(23), "E");   //(23%11 = 1)
h.put(new Temp(16), "F");   //(16%11 = 5)

//h.put("durga", null); // NPE

System.out.println(h);
//{6=C, 16=F, 5=A, 15=D, 2=B, 23=E}
}
}

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



∴ o/p ⇒ {6=C, 16=F, 5=A, 15=D, 2=B, 23=E}

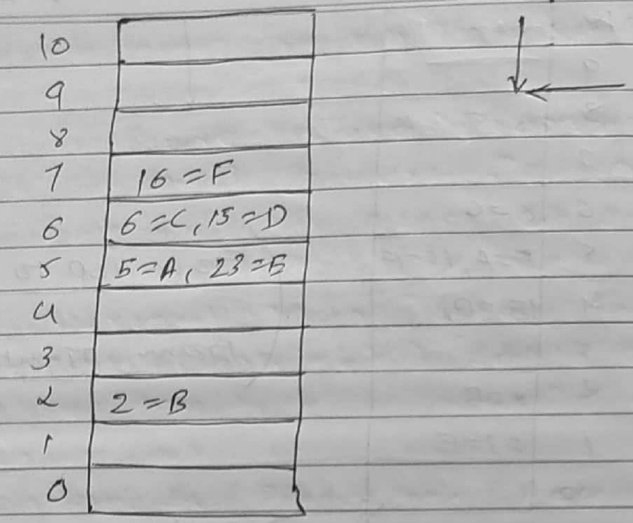
~ If we change hashCode() method of Temp class as

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

Then the o/p will be

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

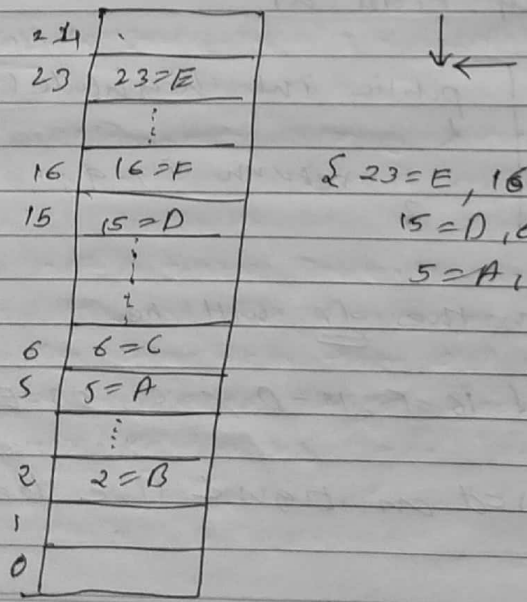
based on figure like below:-



⇝ If we configure initial capacity as 25, i.e

Hashtable h = new Hashtable(25);

with hashcode return 1; then o/p :-



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

(8.) Properties

⇝ In our program, if anything which changes frequently (like username, password, email, contact) are not recommended to hard-code in java program, bcz if there is change, to reflect that change → Compile, → Rebuild, → Redeploy application are required

Even sometimes → Server Restart also required

which creates a big business impact to client.

⇝ we can overcome this problem by using properties file, such type of variable things we have to configure in properties file. From properties file, we have to read into java program & 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 re-deploy is enough, which won't create

any business impact to client

⇒ we can use Java Properties Object to hold properties which are coming from properties file.

⇒ In Normal Map (like HashMap, TreeMap, Hashtable) key & Value can be any type.
But
In case of Properties key & value should be String type.

⇒ Constructor :

① `Properties p = new Properties();`

⇒ Methods :

① String setProperty(String pname, String pvalue);

(- To set a new Property.
- If the specified property already available then old value will be replaced with new value & returns old value)

② String getProperty(String pname);

(- To get value associated with specified property
- If the specified property not available then this method returns null.)

③ Enumeration propertyNames();

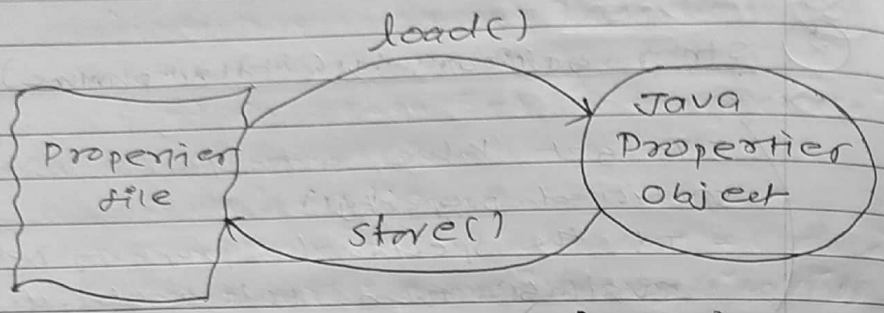
(- Returns all propertyNames present in properties file)

④ void load(InputStream is);

(- To load properties from properties file into java properties object)

⑤ void store(OutputStream os, String comment);

(- To store properties from Java properties object into properties file)



Abc.properties

↑ extension not important we can take any. like Abc.durga but above is conventional approach

Properties p = new Properties();

Initially abc.properties
user = scott
pwd = tiger
venki = 9999

EX.

```
class PropertiesDemo
{
    p s v m (str[] )
    {
        Properties p = new
                Properties();

        FileInputStream fis = new
            FileInputStream("abc.properties");

        p.load(fis);
        s.o.p(p);
// {pwd=tiger, user=scott, venki=9999}

        String s = p.getProperty("venki");
        s.o.p(s); // 9999

        p.setProperty("nag", "88888");

        FileOutputStream fos = new
                FileOutputStream("abc.properties");

        p.store(fos, "updated by chintan");
    }
}
```

Finally
abc.properties
#updated by chintan
#Date
nag = 88888
user = scott
pwd = tiger
venki = 9999

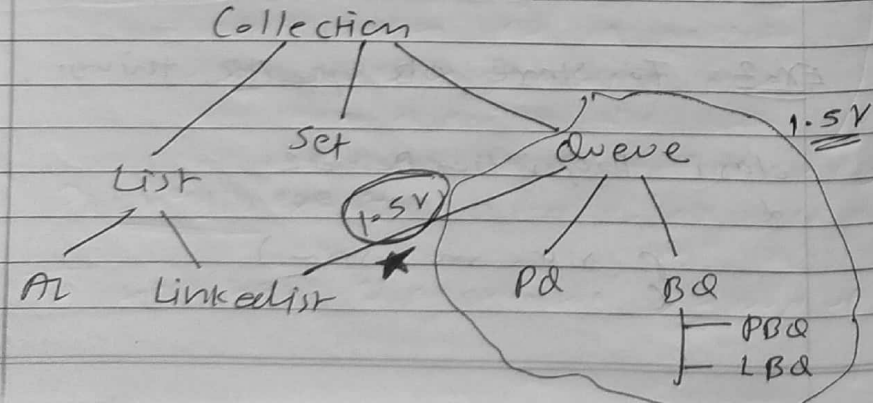
EX. 2 To store DB specific things.

```
class PropertiesDemo2
{
    p s v m (str[] )
    {
```

```
Properties p = new Properties();

FIS fis = new FIS("db.properties");

p.load(fis);

String url = p.getProperty("url");
String user = p.getProperty("user");
String pwd = p.getProperty("pwd");

Connection cn =
    DriverManager.getConnection
        (url, user, pwd);

.
// DB - specific things (code)
}
}
```

1.5V Enhancements - [Queue(I)]

→ Child interface of Collection(I).



→ If we want to represent a group of individual objects prior to processing, then we should go for queue. For Example, before sending sms, we have to store all Mob. No. in some DS. In which order we added Mob. No., In the same order msg should be deliverd, For this FIFO Requirement Queue is best choice.

→ Usually Queue follows FIFO order but based on our requirement we can implement our own priority order also (priority queue)

★ → From 1.5V onwards, LinkedList class also implements Queue interface.

★ → LinkedList based implementation of Queue always follows FIFO order.

→ Queue(I) Specific methods :-

① boolean offer(Object o)

(To add an object into the Queue)

② Object peek()

(To return head element of Queue.)

If queue is empty then this method returns null.

(3) Object element()

To return head element of queue But here, if queue is empty, this method raises
RE: NoSuchElementException

(4) Object poll()

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

(5) Object remove()

To remove & return head element of queue, But here if queue is empty then this method raises
RE: NoSuchElementException

(1) PriorityQueue :-

~ If we want to represent a group of individual objects prior to processing according to some priority then we should go for PriorityQueue.

~ The priority can be DNSO, (or) customized s.o. defined by Comparator

~ Insertion order is not preserved & it is based on some priority.

→ Duplicate objects are not allowed

→ If we are depending on DNSO compulsory the objects should be Homogeneous & Comparable otherwise we will get RE: ClassCastException

→ If we are defining our own sorting by comparator then objects need not to be Homogeneous & comparable.

→ null is not allowed even as first element also.

⇒ Constructors :-

① PriorityQueue q = new PriorityQueue();

creates an Empty priority queue with Default Initial capacity = (11)

All objects will be inserted according to DNSO.

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

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

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

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

Note: Remember, In PriorityQueue there is no constructor which only takes Comparator object, we have to provide Initial capacity also to achieve customized S.O

Ex.

```
class PriorityQueueDemo1
{
    p s v m(String[] args)
    {
        PriorityQueue q =
                new PriorityQueue();
        S.O.P(q.peek()); //null
        //S.O.P(q.element());
        // RE: NSEE
        for(int i=0; i<=10; i++)
        {
            q.offer(i);
        }
        S.O.P(q);
        //[0,1,2,3,4,5,6,7,8,9,10]
        S.O.P(q.poll());
        //0
        S.O.P(q);
        //[1,2,3,4,5,6,7,8,9,10]
    }
}
```

Note:- Some platforms won't provide proper support for thread priorities & priority

Date / / Page

Queues

Ex:

```
class PriorityQueueDemo2
{
    psvm(String)
    {
        PriorityQueue q = new PriorityQueue
        (15, new MyComparator());

        q.offer("A");
        q.offer("Z");
        q.offer("L");
        q.offer("B");

        S.o.p(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);
    }
}
```

Date / / Page

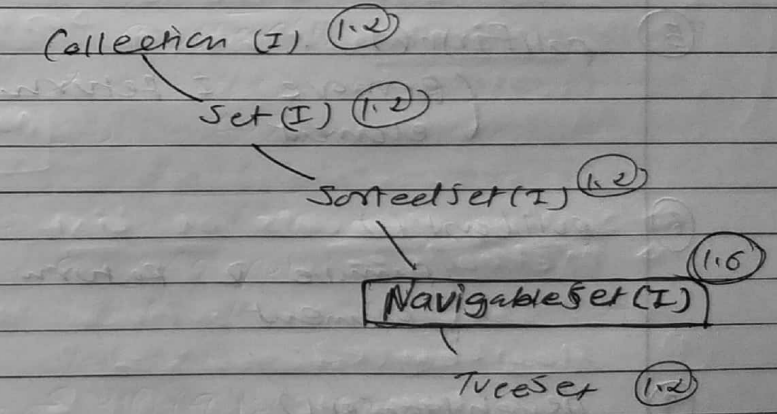
✱ 1.6 V Enhancements in Collection Framework

⇝ As a part of 1.6 V the following concepts introduced in Collection Framework.

1. NavigableSet
2. NavigableMap

(1) NavigableSet(I)

⇝ It is the child interface of SortedSet & it defines several methods for navigation purposes.



⇝ NavigableSet defines the following Methods:-

Date / / Page

1. floor(e)
   (Returns highest element which is <= e)

2. lower(e)
   (Returns highest element which is < e)

3. ceiling(e)
   (Returns lowest element which is >= e)

4. higher(e)
   (Returns lowest element which is > e)

5. pollFirst()
   (Remove & Return first element)

6. pollLast()
   (Remove & Return last element)

7. descendingSet()
   (Returns NavigableSet in reverse order)

[Remember - Don't take (=) in higher() & lower()]

Date / / Page

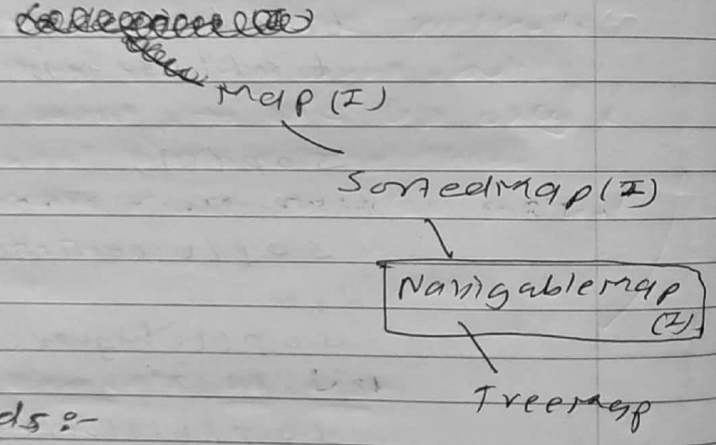
Ex-

```
class NavigableSetDemo
{
    p s v m(Str[])
    {
        TreeSet<Integer> t =
            new TreeSet<Integer>();

        t.add(1000);
        t.add(2000);
        t.add(3000);
        t.add(4000);
        t.add(5000);

        S.O.P(t);
        // [1000, 2000, 3000, 4000, 5000]
        S.O.P(t.ceiling(2000));
        // [2000]
        S.O.P(t.higher(2000));
        // [3000]
        S.O.P(t.floor(3000));
        // [3000]
        S.O.P(t.lower(3000));
        // [2000]
        S.O.P(t.pollFirst());
        // [1000]
        S.O.P(t.pollLast());
        // [5000]
        S.O.P(t.descendingSet());
        // [4000, 3000, 2000]
```

```
s.o.p(t);
// [2000, 3000, 4000]
}
}
```

(2) NavigableMap

- Child interface of SortedMap.
- It defines several methods for Navigation purposes.



→ Methods :-

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

} Same tasks as NavigableSet methods.

Ex.

```
class NavigableMapDemo
{
    p s v m (Str m)
    {
        TreeMap<String, String> t =
            new TreeMap<String, String>();

        t.put("b", "banana");
        t.put("c", "cat");
        t.put("a", "apple");
        t.put("d", "dog");
        t.put("g", "gun");

        s.o.p(t);
// {a=apple, b=banana, c=cat, d=dog, g=gun}
        s.o.p(t.ceilingKey("c"));
// c
        s.o.p(t.higherKey("e"));
// g
        s.o.p(t.floorKey("e"));
// d
        s.o.p(t.lowerKey("e"));
// d
        s.o.p(t.pollFirstEntry());
// a=apple
        s.o.p(t.pollLastEntry());
// g=gun
        s.o.p(t.descendingMap());
// {d=dog, c=cat, b=banana}
```

```
        s.o.p(l);
        // l = [banana, c=cat, d=dog]
    }
}
```

✱ Collections → Utility class - 1

→ Collections class several utility methods for Collection object like - searching, sorting, reversing etc.

Ex → Set, Queue have methods for sorting if they want, but if we want to use List only & we want to do sorting then Collections class provides method for that.

⇒ Sorting elements of List

→ Collections class defines the following two sort methods :-

① public static void sort(List l)

→ To sort based on DNSO

→ In this case, List should compulsory contain Homogeneous & comparable objects otherwise we will get (R.E. CCE)

→ List should not contain (null), otherwise we will get (R.E NPE)

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

→ To sort based on customized s.o.

→ These two are overloaded method provided by Collections class

Ex.

```
class CollectionsSortDemo
{
    p s v m(Str)
    {
        ArrayList al = new ArrayList();
        al.add("Z"); al.add("A");
        al.add("K"); al.add("N");
        //al.add(new Integer(10)); // CCE
        //al.add(null); // NPE
        s.o.p(al); // [Z, A, K, N]
        Collections.sort(al);
        s.o.p(al); // [A, K, N, Z]
    }
}
```

Ex-

```
class CollectionsDemo2
{
    p s v m (Str[] )
    {
        ArrayList al = new ArrayList();
        al.add("Z");
        al.add("A");
        al.add("K");
        al.add("L");

        S.O.P(al); // [Z, A, K, L]

        Collections.sort(al, new MyComparator());

        S.O.P(al); // [Z, L, K, A]
    }
}
class MyComparator implements
                    Comparator
{
    public int compare(Object obj1,
                       Object obj2)
    {
        String s1 = (String)obj1;
        String s2 = obj2.toString();

        return s2.compareTo(s1);
    }
}
```

⇒ Searching elements from List

→ Collections class defines the following binarySearch method

① `public static int binarySearch (List l, Object target)`

→ If the list is sorted according to DNSO then we have to use this method.

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

→ We have to use this method if the list is sorted according to customized S.O.

→ conclusions :-

→ The above search methods internally will use Binary search Algorithm.

→ successful search returns index, unsuccessful search returns insertion point.

→ insertion point is a location where we can place target element in sorted list

→ Before calling binarySearch() method, compulsory list should be sorted, otherwise we will get unpredictable result. we won't get any C.E/R.E.

→ If the list is sorted according to comparator then at the time of search operation also, we have to pass same comparator object, otherwise we will get unpredictable results.

Ex.

```
class CollectionsSearchDemo
{
    p s v m(Str[] )
    {
        ArrayList al = new ArrayList();
        al.add("Z");
        al.add("A");
        al.add("M");
        al.add("K");
        al.add("a");

        S.O.P(al); //[Z, A, M, K, a]
```

Collection.sort();

S.O.P(al); //[A, K, M, Z, a]

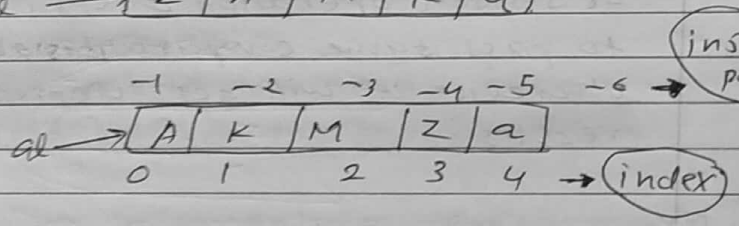
S.O.P(Collections.binarySearch(al, "Z")); //3

S.O.P(Collections.binarySearch(al, "J")); //-2

}

}

al → [Z | A | M | K | a]



Ex.

class CollectionsSearchDemo1

{

p s v m(Sm)

{

ArrayList al = new ArrayList();

al.add(15);

al.add(0);

al.add(20);

al.add(10);

al.add(5);

S.O.P(al); //[15, 0, 20, 10, 5]

Collections.sort(al, new MyComp());

S.O.P(al); //[20, 15, 10, 5, 0]

S.O.P(Collections.binarySearch(al, 10, new MyComp())); //2

S.O.P(Collections.binarySearch(al, 13, new MyComp())); //-3

S.O.P(Collections.binarySearch(al, 17)); //unpredictable

}

}

class MyComp implements Comparator

{

public int compare(Object obj1, Object obj2)

{

Integer i1 = (Integer)obj1;

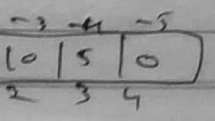
Integer i2 = (Integer)obj2;

return i2.compare(i1);

}

}

al → [15 | 0 | 20 | 10 | 5]



Date / / Page

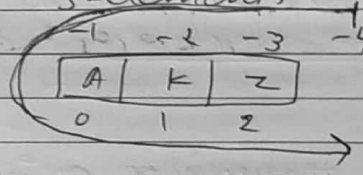
Note :- For the list of n elements, in the case of Binary search method.

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

Ex. 3-elements



1) Successful ⇒ 0 to 2
2) Unsuccessful ⇒ -4 to -1
3) Total ⇒ -4 to 2

⇒ Reversing element of List

⇒ Collections class defines the following reverse method to reverse elements of List

Date / / Page

① `public static void reverse(List l)`

Ex.
```
class CollectionsReverseDemo
{
    p s v m(String)
    {
        ArrayList al = new ArrayList();

        al.add(15);
        al.add(0);
        al.add(20);

        SO.p(al); // [15,0,20]

        Collections.reverse(al);

        S.O.p(al); // [20,0,15]
    }
}
```

⇒ reverse() vs reverseOrder() method

⇒ we can use reverse() method to reverse the order of elements of List

⇒ whereas we can use reverse order methods to get Reversed

Comparator.

(2) `Comparator c = Collections.reverseOrder(Comparator c);`

- Descending order
- Ascending order

✱ Arrays - utility class 2

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

⇒ Sorting elements of Array

→ Arrays class define the following sort methods to sort elements of primitive & object type array.

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

(To sort according to DNSO)

(2) `public static void sort (Object[] o)`

(To sort according to DNSO)

(3) `public static void sort (Object[] o, Comparator c)`

(To sort according to customized S.O.)

Ex.

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

class ArraysSortDemo
{
    p s v m (Str)
    {
        int[] a = {10, 5, 20, 11, 6};
        S.O.P("primitive Array
              before sorting:");
        for(int al:a)
        {
            S.O.P(al);
        }
        Arrays.sort(a);
```

```
S.O.P("primitive Array
        After sorting:");
for(int a1:a)
{
    S.O.P(a1);
}

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

S.O.P("Object Array Before
        sorting:");
for(String s2:s)
{
    S.O.P(s2);
}
Arrays.sort(s);

S.O.P("Object Array After
        sorting:");
for(String s2:s)
{
    S.O.P(s2);
}
Arrays.sort(s, new MyComparator());

S.O.P("Object Array After
        customized sorting:");
for(String s2:s)
{
    S.O.P(s2);
}
}
}
class MyComparator
        implements Comparator
{

    public int compare(Object o1,
                       Object o2)
    {

        String s1 = o1.toString();
        String s2 = o2.toString();

        return s2.compareTo(s1);
    }
}
```

O/P :-

```
Primitive Array Befor sorting :
10
5
20
11
6
primitive Array After sorting :
5
6
10
11
20
```

Object Array Before Sorting:
A
Z
B
Object Array After Sorting:
A
B
Z
Object Array After customized sorting:
Z
B
A

Note: We can sort primitive Arrays only based on DNSO

whereas

we can sort object Arrays either based on DNSO (or) based on customized SO

⇒ Searching elements from Array:-

Arrays class define the following binarysearch methods.

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

② public static int binarysearch (Object[] a, Object target)

③ public static int binarysearch (Object[] a, Object target, Comparator c)

Note: All wlers of Arrays class binarysearch methods are exactly same as Collections class binarysearch method.

Ex.

```
class ArraysSearchDemo
{
    p s v m(Sr a)
    {
        int[] a = {10, 5, 20, 11, 6};
        Arrays.sort(a);
        s.o.p(Arrays.binarysearch(a,6));
        // 1
        s.o.p(Arrays.binarysearch(a,14));
        // -5
```

| 10 | 5 | 20 | 11 | 6 |

-1	-2	-3	-4	-5	-6
5	6	10	11	20	
0	1	2	3	4	

A	z	B
-1	-2	-3

A	B	z
0	1	2

```
String[] s = {"A", "z", "13"};
Arrays.sort(s);

s.o.p(Arrays.binarysearch(s,"z"));
// 2
s.o.p(Arrays.binarysearch(s,"s"));
// -3
```

z	B	A
-1	-2	-3

z	B	A
0	1	2

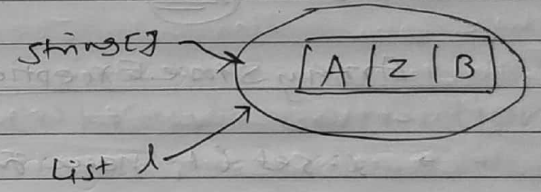
```
Arrays.sort(s, new MyComparator());
s.o.p(Arrays.binarysearch
        (s,"z", new MyComparator()));
// 0
s.o.p(Arrays.binarysearch
        (s,"s", new MyComparator()));
// -2
s.o.p(Arrays.binarysearch(s,"N"));
// Unpredictable result
    }
}
class MyComparator implements
                    Comparator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

        return s2.compareTo(s1);
    }
}
```

⇒ Conversion of Array to List

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

↝ strictly speaking, this method won't create an independent list object. For the existing Array we are getting List view

```
String[] s = {"A", "z", "s"};

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



↝ By using Array reference if we perform any change, it will reflect automatically to the List.

vice-versa also same.

```
s[0] = "k";
s.o.p(l);   [k, z, B]
```

Date / / Page

~ By using List reference, we can't perform any operation which varies the size, otherwise we will get (R.E.)

↓

UnsupportedOperationException.

→ l.add("M"); l.remove(1); } ✗

but

→ l.set(1, "N"); ✓

~ By using List reference, we are Not Allowed to replace with Heterogeneous Object otherwise we will get (R.E)

ArrayStoreException ←

→ l.set(1, new Integer(10)); ✗

Ex.

```
class ArraysAsListDemo
{
    p s v m(Str)
    {
        String[] s={"A","Z","B"};
        List l = Arrays.asList(s);
        s.o.p(l); //[A,Z,B]
        s[0] = "K";

        s.o.p(l); //[K,Z,B]
        l.set(1,"L");

        for(String s1:s)
        {
            s.o.p(s1); // K,L,B
        }

        //l.add("durga"); //UsOE
        // l.remove(2); //UsOE
        // l.set(1, new Integer(10)); //ASE
    }
}
```

✱ Concurrent Collection

✱ (Need) For Concurrent Collection :-

(1) Traditional Collection Object (like ArrayList, HashMap etc) can be accessed by Multiple Threads simultaneously and there may be a chance of Data inconsistency

problems & hence these are not Thread safe.

(2) Already existing Thread safe collection objects (like Vector, Hashtable, synchronizedList(), synchronizedSet(), synchronizedMap()) performance wise not up to the mark as only one thread can access at a time.

(3) Because for every operation Even for Read operation Also Total Collection will be locked by only one Thread at a time & it increases waiting time of Threads.

(4) While one Thread Iterating Collections, the other threads are Not allowed to Modify Collection object simultaneously If we are trying to Modify then we will get ConcurrentModificationException

R.E. ConcurrentModificationException

~ Hence, these Collection objects are Not suitable for scalable Multi-Threaded Applications.

~ To overcome these problems SUN people introduced Concurrent Collections in (1.5) version.

Ex.

```
class MyThread extends Thread
{
    static ArrayList al = new ArrayList();

    public void run()
    {
        try{
            Thread.sleep(2000);
        }
        catch(InterruptedException e){}

        s.o.p("child Thread updating List");
        l.add("D");
    }

    public static void main(Str)
    {
        l.add("A");
        l.add("B");
        l.add("C");
```

```
MyThread t = new MyThread();
t.start();

Iterator itr = l.iterator();
while (itr.hasNext())
{
    String s1 = (String) itr.next();

    S.O.P("Main Thread Iteration
          var & current object
          is : "+s1);

    Thread.sleep(3000);
}
S.O.P(1);
}
}
```

Olp :-

```
Java MyThread
```

R.E. — J.U.ConcurrentModificationException

⊛ Differences b/w Traditional (vs.) Concurrent Collections

- (1) Concurrent collections are always Thread-safe

(2) When compared with Traditional, Thread-safe collections performance is more bcz of different Locking Mechanism

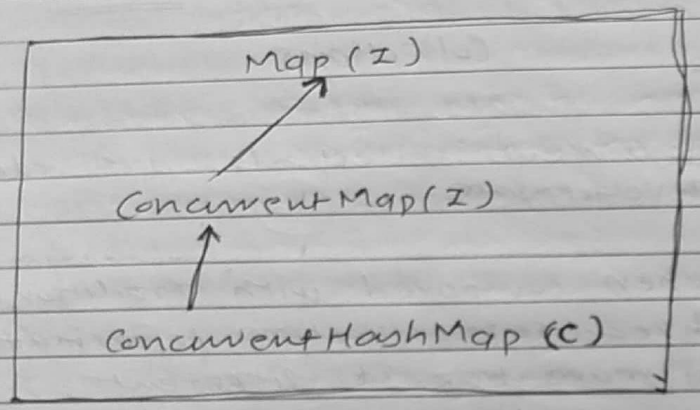
(3) While one Thread interacting collection the other Threads are allowed to modify collection in "safe Manner."

⇝ Hence, Concurrent Collections Never throw ConcurrentModificationExcp.

⇒ Important Concurrent classes are :-

1. Concurrent ~~Hashmap~~ HashMap
2. CopyOnWriteArrayList
3. CopyOnWriteArraySet

ConcurrentMap (I) :-



⇝ This interface defines 3 specific Methods :-

① Object putIfAbsent (Object key, Object Value)

To Add Entry to the map if the specified key is Not Already Available

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

put()	putIfAbsent()
If key is Already Available, old value will be replaced with New Value & Returns old value	If the key is Already present then Entry won't be added & Returns old Associated value.

Ex:

```
m.put (101, "durga");
m.put (101, "shiva");
s.o.p(m);      // 101 = shiva
```

```
m.put (101, "durga");
m.putIfAbsent (101, "shiva");
s.o.p(m);      // 101 = durga
```

Date / /
Page

② boolean remove (Object key,
Object value)

Removes Entry which have both key & value same as given

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

Ex.

```
import java.util.concurrent.
        ConcurrentHashMap;

class Test
{
    p s v m (Str )
    {
```

Date / /
Page

```
ConcurrentHashMap m =
    new ConcurrentHashMap();

m.put(101, "Durga");

m.remove(101, "Ravi");

// value not matched

s.o.p(m); // {101=Durga}

m.remove(101, "Durga");

s.o.p(m); // {}

    }
}
```

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

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

Date __/__/____
Page _____

```
else
{
    return false;
}
```

Ex:

```
import java.util.concurrent.
        ConcurrentHashMap;

class Test
{
    p s v m (sm)
    {
        ConcurrentHashMap m =
        new ConcurrentHashMap();

        m.put(101, "Durga");

        m.replace(101, "Ravi", "siva");

        s.o.p(m); //{101=Durga}

        m.replace(101, "Durga", "Ravi");

        s.o.p(m); // {101=Ravi}
    }
}
```

Concurrent Collections will be continued after several topic from here on..