Avishek (8210928619)

5

* Collections (136-151)

#136 Collections

20/Apr/23

① Introduction
② 9 key interfaces
  (i) Collection
  (ii) List
  (iii) Set
  (iv) SortedSet
  (v) NavigableSet
  (vi) Queue
  (vii) Map
  (viii) SortedMap
  (ix) NavigableMap
③ 3 Cousors of Java
  (i) Enumeration
  (ii) Iterator
  (iii) ListIterator
④ Comparator
⑤ 1.6 V Enhancements

* Introduction

* Arrays :-

An array is a indexed collection of fixed no. of homogeneous data elements.

→ Adv. → we can represent multiple values by using single variable.

→ Limitations :-

→ It is fixed in size. [we should know size in Advance]

→ Array can hold only homogeneous data type elements.

> Student s = new Student[1000];
> s[0] = new Student(); ✓
> s[1] = new Customer(); ✗ CE:

→ we can solve this problem by using Object[]

> Object [] a = new Object[1000];
> a[0] = new Student(); ✓
> a[1] = new Customer(); ✓

→ Arrays is not implemented based on some standard data structure & hence readymade method support is not available.

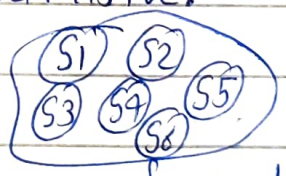
⇒ To solve array limitations we should go for collection concept.

* Collections

→ It is growable in nature. i.e based on our requirement we can increase (or) decrease the size.

→ Collections can hold both homogeneous & hetro-
-geneous ~~elem~~ objects.
→ Every collection class is implemented based on some standard data structure hence for every requirement readimade method support is availa

* Diff. blw Arrays	& Collection
① Fixed in size.	① Growable in nature.
② Memory wise not recommed	② Memory wise recommended.
③ Performance wise recomm.	③ Performance wise Not recomm
④ Only homogeneous element.	④ Both homo & hetrogeneous element allowed.
⑤ No underlying D.S. for arrays.	⑤ Some underlying D.S.
⑥ Readymade method support not available.	⑥ Readymade method support is available.
⑦ Applicable for primitive & object both.	⑦ Only for objects but not for primitive.



* Collection :-

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

* Collection framework :-

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

#137 Collections

20/Apr/23

Java | C++

Collection → Container

Collection framework → STL (Standard Template Lib)

* 9 key interfaces of collection framework

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

* 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 most common methods which are applicable for any collection object.

→ In general collection interface is consider as root interface of collection framework.

→ There is no concrete class which implements collection interface directly.

* diff b/w Collection & Collections

→ Collection is an interface, if we want to represent a group of individual objects as a single entity then we should go for collection.

→ Collections is an utility class present in java.util package to define several utility methods for collection objects (like sorting, searching etc).

* List (I)

→ It is child interface of collection.

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

→ Array List (1.2V)
→ Linked List (1.2V)
→ Vector } 1.0V (Legacy classes)
→ Stack

Note:- In 1.2V vector & stack classes are re-engineered

* Set (I)

→ It is child interface of collection.

→ If we want to represent a group of individual objects as a single entity where duplicates are not allowed & insertion order not required.

→ HashSet (1.2V)
→ LinkedHashSet (1.4V)

* SortedSet(I) (1.2V)
→ It is child interface of set.
→ If we want to represent a group of individual objects as a single entity where duplicates are not allowed & all object should be inserted acc to some sorting order.

* NavigableSet(I) (1.6V)
→ It is child interface of sorted set.
→ It contains several methods for navigation purposes.
→ TreeSet (1.2V)

* Diff blw List & Set

① Duplicates are allowed	① Duplicates are not allowed
② Insertion order preserved	② Insertion order not-preserved

* 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.
→ Usually queue follows FIFO order but based on our requirement we can implement our own priority order.
→ Priority queue (1.5V)
→ Blocking queue
  ↦ PriorityBlockingQueue
  ↦ LinkedBlockingQueue.

* Note :- All the above interfaces (Collection, List, Set, SortedSet, NavigableSet and Queue). ment for representing a group of individual objects.
→ If we want to represent a group of object as key-value pairs then we should go for map.

* Map (I)

→ Map is not child interface of collection.
→ If we want to represent a group of ~~individual~~ objects as key-value pairs then we should go for map.

eg.

Key	S.NO	101	102	103
value.	Name	durga	Ravi	Shiva

→ Both key & value are objects only.
→ Duplicates keys are not allowed but values can be duplicated.
→ HashMap (1.2V)
→ LinkedHashMap (1.4V)
→ WeakHashMap (1.2V)
→ IdentityHashMap (1.4V)
→ Hashtable (1.0V) (Legacy classes)
→ properties

* SortedMap(I) (1.2V)

→ It is child interface of map.

→ If we want to represent a group of key-value pairs Alc to some sorting order of keys.

→ In sorted map the sorting should be based on key but not based on values.

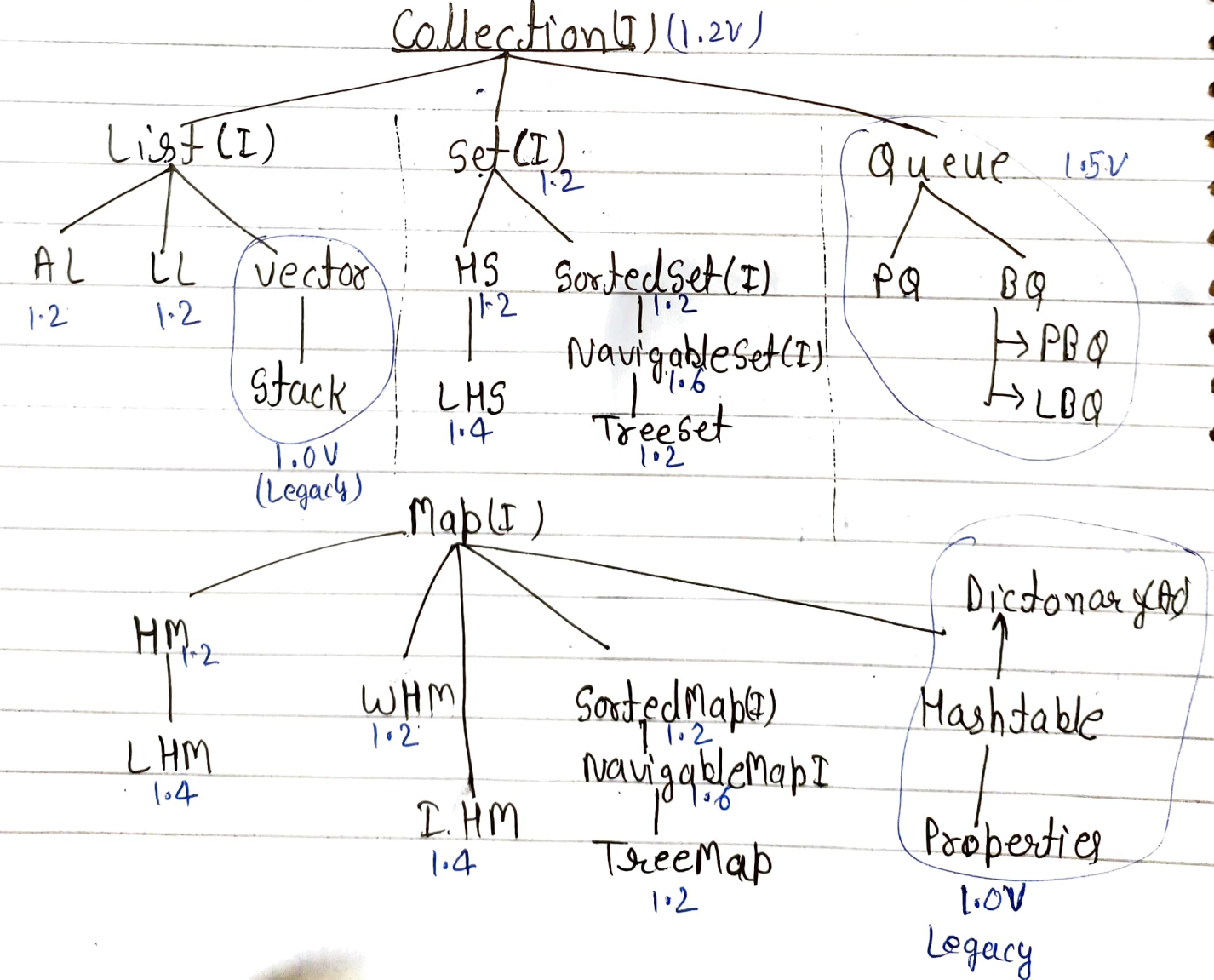
→

* NavigableMap (I) (1.6V)

→ It is child interface of sorted map.

→ It defines several methods for navigation purposes.

→ TreeMap (1.2V)



* Sorting
1. Comparable (I)
2. Comperator (I)

* Cursors
① Enumerations (I) (1.0V)
② Iterator (I)
③ List Iterator (I)

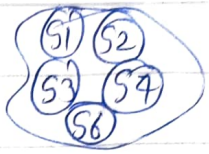
* utility classes
① collections
② Arrays

⇒ The following are legacy character present in Colle
→ Enumeration (I)
→ Dictionary (AC)
→ Vector (C)
→ Stack (C)
→ Hashtable (C)
→ Properties (C)

#138. Collection & Collections || List & set

20/Apr/23

* 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 are applicable for any collection object.

b contains (object o)	b add (object o)
b containsAll (collection c)	b addAll (collection c)
b isEmpty ()	b remove (object o)
int size ()	b removeAll (collection c)
	void clear ()
	b retainAll (collection c)

→ Object [] toArray ()

→ Iterator iterator ()

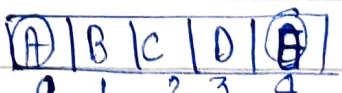
→ There is no concrete class which implements collection interface directly.

* List (I)

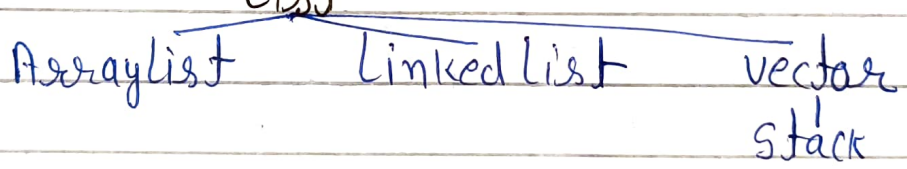
→ List is child interface of collection.

→ If we want to represent a group of individual objects as a single entity where duplicates are allowed & insertion order must be preserved.

→ we can preserve insertion order via index & we can differencate duplicate objects by using index.



→ Index plays very important role in list.
→ List interface defines the following specific methods.
> void add(int index, Object o)
> bool addAll(int index, Collection c)
> object get(int index)
> object remove(int index)
> object set(int index, object new)
↳ to replace the element present at specified index with provided object & returns old object.
> int indexOf(Object o)
returns index of first occurrence of 'o'
> int lastIndexOf(Object o)
> ListIterator listIterator();



* Arraylist
→ The underlying data structure is resizable array.
→ Duplicates are allowed.
→ Insertion order is preserved.
→ Heterogeneous objects are allowed. (except treeSet & treeMap everywhere it is allowed)
→ null insertion is possible.

* Constructors

① Arraylist l = new Arraylist();

→ it creates an empty arraylist object with default initial capacity 10.

→ Once Arraylist reaches it's max capacity then a new arraylist will be created with new capacity.

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

② Arraylist l = new ArrayList (int initialCapicity);

→ it creates an empty Arraylist object with specified initial capacity.

③ Arraylist l = new Arraylist (collection c);

→ It creates an equivalent Arraylist object for the given collection.

```
> class Arraylist Demo {
      p s v m(S[] args){
          Arraylist l = new Arraylist();
          l.add("A");
          l.add(10);
          l.add("A");
          l.add(null);
          Sopln(l)          // [A, 10, A, null]
          l.remove(2);
          Sopln(l);         // [A, 10, null]
          l.add(2, "M");
          l.add("N");
          Sopln(l);  // [A, 10, M, null, N]  }
}
```

#139 Collection Array list

20/Apr/23

→ Usually we use collections to hold & transfer objects from one location to another location (Contain
to provide support for this requirement every collection class by default implements serizable & clonable interfaces.

→ Arraylist & vector classes implements RandomAc
-ess interface so that any random element, we can access with the same speed.

* Random Access (I)

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

→ Arraylist is best choice if our frequent operat
-ion is retrival operation. (Random Access I)
→ Arraylist is worst choice if our requirement is insertion & deletion in middle.

* diff b/w Arraylist

① It's method is non-Synchronized
② Not Thread safe.
③ Performance is high.
④ 1.2 V & non-legacy

vector

① It's method is synchronized.
② It is thread safe.
③ Performance is low
④ 1.0 V & legacy

* How to get synchronized version of Arraylist obj.

> AL l = new AL();

List l1 = Collections.synchronizedList(l);
sync. ↙ ↘ non-sync.

→ We can get synchronized version of arraylist object by using .synchronizedList() method of collections class.

> public static List synchronizedList(List l)

→ Similarly, we can get synchronized version of set & map objects by using following methods.

> public static Set synchronizedSet(Set s)

> public static Map synchronizedMap(Map m).

* Linked List :-

→ It's underlying data structure is doubly linked list
→ Insertion order is preserved.
→ Duplicates objects are allowed.
→ Heterogenious object allowed.
→ null insertion is possible.
→ LinkedList implements Serizable, Cloneable interface but not RandomAccess.
→ It is best choice if our frequent operation is insertion & deletion in middle.
→ It is worst choice if our frequent operation is retrival operation.

* constructors :-

① LinkedList l = new LinkedList();
→ It creates an emply LinkedList object

② LinkedList l = new LinkedList(Collection c);
→ It creates an equivalent linked list object for the given collection.

* Linked List class specific method
→ usually we use Linked List to develope stack & queues to provide suppor for this requirement LL difines the following specific methods.

> void addFirst(object o)
> void addLast(object o)
> object getFirst()
> object getLast()
> object removeFirst()
> object removeLast()
> class LinkedListDemo{
    p s v m(S[] args){
    LinkedList l = new LinkedList();
    l.add("durga");
    l.add(30);
    l.add(null);
    l.add("durga"); // [durga, 30, null, durga]
    l.set(0,"Soft"); // [Soft, 30, null, durga]
    l.add(0, "venky");
    l.removeLast();
    l.addFirst("ccc");
    Sop(l);

#140 Collections Diff. b/w ArrayList & LinkedList

20|Apr|23

* diff b/w ArrayList & LinkedList	
① Retrival	① Insertion / deletion in middle.
② worst if insertion & deletion in middle.	② worst in retrival.
→ In arraylist elements will be stored in consuci-tive memory locations & hence retrival becomes easy.	→ In LinkedList the elemen won't be stored in consucitive memory location & retrival operation become difficult.

* Vector :-

→ Underlying data-structure is Resizeable & Growable array.
→ Insertion order is preserved.
→ Duplicates are allowed.
→ Heterogeneous object are allowed.
→ null insertion is possible.
→ It implements serizable, cloneable & Randam Access interfaces.
→ Every method present in vector is thread safe.

* constructars

① vector v = new Vector(); → initial capicity = 10

New Capacity = C.C * 2

② Vector v = new Vector (int initialCapacity) ;

③ Vector v = new vector (int init. Cap, int incremental);

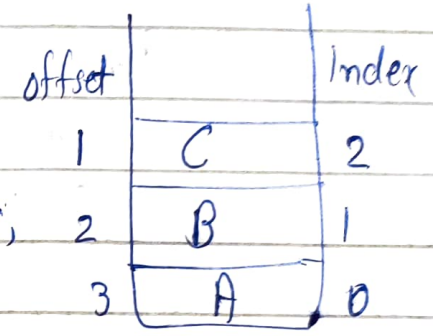
④ Vector v = new vector (Collection c);
→ inter conversion b/w vector & collection.

* Vector specific method :-
> add (Object o) ---C
> add (int idx, Object o) --L
> addElement (Object o) --V
> remove (Object o) --C
> removeElement (Object o) --V
> remove (int index) ---L
> removeElementAt (int index) ---V
> clear () ---C
> removeAllElement () ---V
> object get (int index) --L
> object elementAt (int index) --V
> object firstElement () --V
> Object lastElement () ---V
> int size ()
> int capacity ()
> Enumeration elements ()

```
> class VectorDemo {
      p s v m(S[] args) {
        Vector v = new Vector();
        Sopln(v.capacity()); //10
        for (int i = 1; i<=10; i++) {
            v.addElement(i); }
        Sopln(v.capacity()); //10
        v.addElement("A");
        Sopln(v.capacity()); //20
        Sopln(v); }
}
```

* stack
→ It is the child class of vector.
→ It is a specially designed class for Last In First Out (LIFO) order.
* Constructor
  Stack s = new stack();
* Methods
> object push(Object o)
    to insert an object into the stack
> object pop()
    to remove & return top of the stack.
> object peek()
    to return top of the stack without removal.

> boolean empty()
  returns true if the stack is empty.
> int search(Object o)
  returns offset if the element is available otherwise returns -1.




```
> class StackDemo {
    p s v m(S[] args) {
      Stack s = new Stack();
      S.push("A");
      S.push("B");
      S.push("C");
      Sopln(S); // [A,B,C]
      Sopln(S.search("A")); //3
      Sopln(S.search("Z")); //-1
```

* The 3 cursors of Java :-
→ If we want to get object one by one from collection then we should go for cursor.
① Enumeration
② Iterator
③ ListIterator.

* Enumeration :-

```
> Vector v = new Vector();
  for (int i=0; i<=10; i++) {
      a.addElement(i); }
  Sop(v); [0, 1, 2, 3, ---, 10]
  Enumeration e = v.elements();
  while (e.hasMoreElements()) {
      Integer I = (Integer) e.nextElement;
      if (I%2 ==0) {
          Sop(I); }  // 0 2 4 6 8 10
  } }
```

→ we can use enumeration to get objects one by one from legacy collection object.

→ we can create enumeration object by using elements() method of vector class.

```
> public Enumeration elements();
  eg Enumeration e = v.elements();
> public boolean hasMoreElements();
> public Object nextElement();
```

#141. Collections

20|Apr|23

* Limitations of Enumeration (1.0v)

→ We can apply enumeration concept only for legacy classes & it is not a universal cursor.

→ By using it we can get only read access & we can't perform remove operation.

* Iterator (I) :-

→ We can apply iterator concept for any collection object & hence it is universal cursor.

→ By using iterator we can perform both read & remove operations.

> public Iterator iterator()

eg:- Iterator itr = c.iterator();

Any collection object

* Methods

> public boolean hasNext()

> public Object next()

> public void remove()

```
> AL l = new AL();
  for(int i=0; i<=10; i++){
      l.add(i);
  }
  Sopln(l); // [0,1,2,---,10]
  Iterator itr = l.iterator();
  While(itr.hasNext()){
      Integer I = (Integer) itr.next();
```

```
if (I % 2 ==0) {
        Sop1n(I); }     // 0 2 4 6 8 10
else {
    itr.remove(I) }
Sop1n(l);               // [0, 2, 4, 6, 8, 10]
}
```

* Limitation of Iterator :-

→ By using enumeration & iterator we can always move only towards forward direction & we can't move backward direction.

→ These are single direction cursors.

→ By using iterator we can perform only read & remove operations & we can't perform replacement and addition of new objects.

→ To overcome above limitation we should go for list iterator.

* ListIterator (I);

→ By using listIterator we can move either to the forward direction or to the backward dir.

→ It is bi-directional cursor.

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

> public ListIterator listIterator()

eg:- ListIterator itr = l.listIterator();
Any List object.

* Methods

→ ListIterator is child interface of Iterator(I)
→ all methods of iterator is available for ListIterator.
→ It contains 9 methods.

> public boolean hasNext()
> public object next()
> public int nextIndex()

} forward movement

> public boolean hasPrevious()
> public object previous()
> public int ~~next~~previousIndex()

} Backward movement.

> public void remove()
> public void add(object o)
> public void set(Object o)

} extra operations.

```
> ~~public~~ class ListIteratorDemo{
    p s.v m (S[]a){
      LinkedList l = new LinkedList();
      l.add("bala");
      l.add("venk");
      l.add("chiru");
      l.add("nag");
      Sopln(l);
```

```
ListIterator itr = l.listIterator();
while (itr.hasNext()) {
    String s = (String) itr.next();
    if (s.equals("venki")) {
        itr.remove(); }
    else if (s.equals("nag")) {
        itr.add("chaitu"); }
    else if (s.equals("chiru")) {
        itr.set("charan"); }
}
sopln(l); }
}
```

→ The most powerful cursor is ListIterator. but its limitation is it is applicable only for list objects.

* Comparision table

property	Enumeration	Iterator	ListIterator
① where	only legacy class	Any collection obj	only for List obj
② Is it legacy?	Yes (1.0v)	NO (1.2v)	NO (1.2v)
③ Movement	Single direction	Single dir.	Bidirectional
④ operations	only read	Read / Remove	Read / Remove Replace / Add
⑤ How to get	elements() method	iterator() of coll.	listIterator()
⑥ Methods.	② hasMoreElements, nextElements	③ hasNext() next() remove()	9 methods.

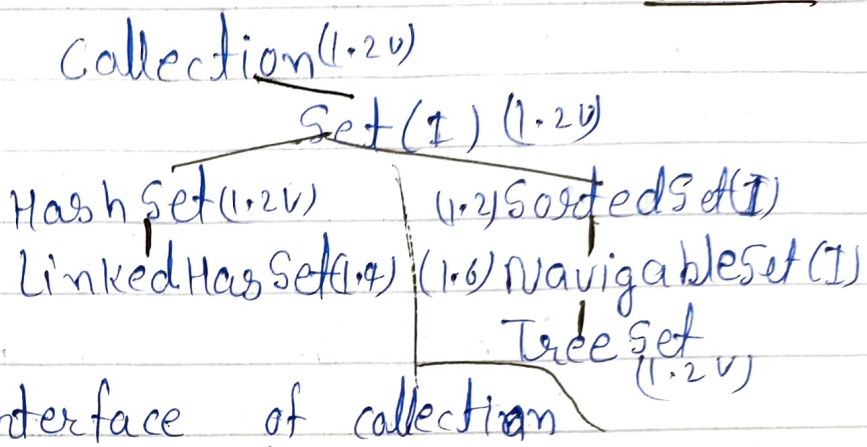
* Internal implementations of cursors

```
> class cursorDemo {
      p s v m (S [] args){
          vector v = new vector();
          Enumeration e = v.elements();
          Iterator itr = v.iterator();
          ListIterator litr = v.listIterator();
          Sopln( e.getClass().getName()); // .vector$1
          sopln(itr.getclass().getName()); //.vector$Itr
          Sopln(litr.getClass().getName(); .vector$ListItr
}
```

#142 Collection SetInterface Shorted Set

21/Apr/23

* Set



→ Set is child interface of collection
→ If we want to represent a group of individual objects as a single entity where duplicates are not allowed & insertion order is not preserved.
→ Set interface doesn't contain any new method & we have to use only collection interface methods.

* HashSet

→ The underlying data-structure is hash-table
→ Duplicates objects are not-allowed.
→ Insertion order is not preserved. & it is based on Hashcode of objects.
→ null & insertion is possible. (only once)
→ Heterogeneous objects are allowed.
→ It implements Serializable & Clonable but not Random Access (I)
→ HashSet is best choice if our frequent operation is search operation.

Note :- HashSet h = new HashSet();
Sopln(h.add("A")); ✓ true
Sopln(h.add("A")); ✗ false

→ If we are trying to insert duplicates then we won't get any CE (or) RE & add method simply returns false.

* Constructer

① HashSet h = new HashSet(); inital cap. = 16 ✓
Default Fill Ratio / Load factor : 0.75

② HashSet h = new HashSet(int initalCapacity);
③ HashSet h = new HashSet(int initialCap, float fillRatio);
④ HashSet h = new HashSet(Collection c);
It creates an equivalent hashset for the given collection.

* Fill Ratio (or) Load factor

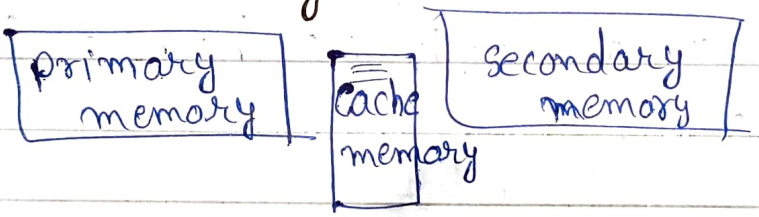
→ After filling how much Ratio, a new hashset object will be created, this ratio is called fill Ratio (or) load factor.

```
> class HashsetDemo {
    p s v m (S[] a){
      HashSet h = new HashSet();
      h.add("B");
      h.add("C");          Sopln(h.add"z")); // false
      h.add("z");          Sopln(h)//
      h.add(null);
      h.add(10);          }
```

* LinkedHashSet

HashSet	LinkedHashSet
→ Hashtable	→ LinkedList + Hashtable
→ It is the child class of	hashset.
→ It is exactly same as	hashset (including const & methods) except the following differences.
① Underlying DS Hashtable	① Underlying DS Hashtable + LinkedList.
② Insertion order not preserved	② Insertion order preserved.
③ 1.2V	③ 1.4V

* cache memory



note:- In General we can use LinkedHashSet to develope cache based applications where duplicates are not allowed & insertion order preserved.

* SortedSet :-

→ SortedSet is child interface of Set.

→ If we want to represent a group of individual objects acc to some sorting order without duplicates then we should go for SortedSet.

* Methods

① Object first();
returns first element of the SortedSet.

② object last();
returns last element.

③ SortedSet headSet(Object o);
returns sortedset whose ele. is less than obj.

④ SortedSet tailSet(Object o);
returns sortedset whose ele. 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 default ⇒ null)

Note:- Number ⇒ Ascending order
String ⇒ Alphabetical order

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

#143 Collections

22/04/23

* TreeSet

→ Under lying Data-structure Balanced tree.
→ Duplicates not allowed.
→ Insertion order not preserved.
→ Hetrogneous objects are not allowed. CE ⇒ CCE
→ null insertion possible (only once)
→ It implements serizable & clonable but not Random Access.
→ All elements will be inserted based on some sorting order.

* Constructors

① TreeSet t = new TreeSet (); → D.N.S.O
② TreeSet t = new TreeSet (Comparator c);
③ TreeSet t = new TreeSet (Collection c);
④ TreeSet t = new TreeSet (SortedSet s);

```
> class TreeSetDemo {
    p s v m(S[] a){
    TreeSet t = new TreeSet();
    t.add("A");
    t.add("a");
    t.add('B');
    t.add(new Integer(10)); // CCE
    t.add(null; // NPE
    Sop(t)  // [A, B, L, Z, a] }
}
```

* null acceptance

→ for non-Empty treeSet if we try to insert null RE: NPE.

→ For empty treeset as a first element null is allowed but after null if we try to insert any other then we will get RE: NPE.

*Note:- Untill 1.6 v null is allowed as first element to the empty Treeset but 1.7 v onwards null is not allowed even as a first element. ~~i.e "null"~~

> p s v m(S[] a){
TreeSet t = new TreeSet();
t.add(new StringBuffer("A"));
t.add(new SB("B"));
t.add(new SB("Z");) ⇒ CE: CCE
}

→ If we are depending on default natural sorting order compulsary object should be homogeneous & comparable. otherwise we will get RE: CCE.

→ comparable iff corresponding class implements comparable interface.

→ String & all wrapper classes implements comparable

* Comparable (I)

→ It is present in java.lang package & it contains only one method compareTo()

> public int compareTo(Object obj)

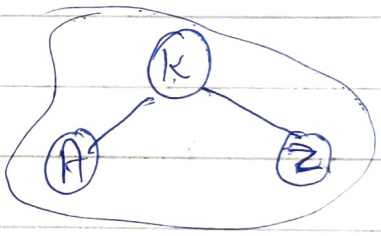
↳ [0, 1, -1]

> 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.

> Sopln("A".compareTo("Z")); -ve.
> Sopln("Z".compareTo("A")); +ve
> Sopln("A".compareTo("A")); 0
> Sopln("A".compareTo(null)); NPE

→ If we depending upon default sorting order then while adding object to the treeset JVM will call compareTo method.

> TreeSet t = new TreeSet();
  t.add("K");
  t.add("Z") ⇒ "Z".compareTo("K"); +ve
  t.add("A") ⇒ "A".compareTo("K"); -ve
  t.add("A") ⇒ "A".compareTo("K"); -ve
  0 ↳ "A".compareTo("A");
  Sopln(t); [A, K, Z]



> obj1.compareTo(obj2)
  The obj. which is to be inserted
  ↳ object which is already inserted

→ If default sorting order not available or we want different sorting order the we can go for customize sorting by using comparator.

#144 Collections.

22|Apr|23

→ Comparable ment for Default Natural Sorting order where as Comparator ment for customized Sorting order.

¥ Comparator

→ Comparator present in java.util package & it defines two methods.

① compare() ② Equals

① public int Compare (Object obj1, object obj2)

② public boolean equals (Object obj)

→ whenever we are implementing comparator interface compulsary we should provide implementation only for compare method & we are not require to provide impl. for equals() method. bcz it is already available to our class from object through inheritance.

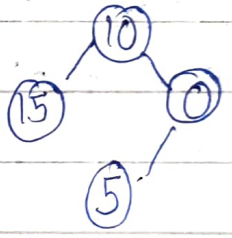
Q W.Ap. to insert Integer object in tree Set where sorting order is decending order.

```
> TreeSet t = new TreeSet();
  t.add(10);
  t.add(0);
  t.add(15);
  t.add(5);
  t.add(20);
  t.add(20);
  Sopln(t); }    [0, 5, 10, 15, 20]
```

```
class myComparator implements Comparator {
    public int comare(obj obj1, object obj2) {
        Integer I1 = (Integer) obj1;
        Integer I2 = (Integer) obj2;
        if (I1 < I2) {
            return +1;
        } else if (I1 > I2) {
            return -1;
        } else {
            return 0; }
    }
```

```
TreeSet k = new TreeSet(new myComparator);
k.add(10); ✓
k.add(0);  => +ve compare(0,10)
k.add(15); => -ve compare(15,10)
k.add(5);  => +ve compare(5,10)
           => -ve compare(5,0)
k.add(20); => +ve compare(20,10)
           => -ve compare(20,15)
k.add(20); => -ve compare(20,15)
           => -ve compare(20,15)
           => compare(20,20)

Sopln(t) ; [20, 15, 10, 5, 0]
```



> public int compare(Object obj1, object obj2) {
Integer I1 = (Integer) obj1;
Integer I2 = (Integer) obj2;
① return I1 compareTo(I2); Default sorting [0,5,10,15,20]

② return -I1.compareTo(I2); Dec.order [20,15,10,5,0]
③ return I2 compareTo(I1); Dec order [20,15,10,5,0]
④ return -I2 compareTo(I1); Asc ord [0,5,10,15,20]
⑤ return +1 ; [Insertion order] [10,0,15,5,20,20]
⑥ return -1 ; [Rev. of Int. order] [20,20,5,15,0,10]
⑦ return 0 ; (only 1st elem)

Q:- W.A.p. to insert string in a treeSet where all elements should be inserted Alc to reverse of alphabetical order.

> TreeSet t = new Treeset();
t.add("Roja");
t.add("ShobhaRani");
t.add("Rajakumari"); [Gangabhawani, Rajakumari,
t.add("GangaBhawani"); Ramulamma, Roja,
t.add("Ramulamma"); ShobhaRani J.
Sop(t);

#145 Collections

22/Apr/23

```
> class myComparator implements Comperator {
      public int compare(obj obj1, obj obj2){
          String s1 = (String)obj1;
          String s2 = obj2.toString();
          return -s1.compareTo(s2);
      }
  Treeset t = new Treeset(new myComparator());
  // [SobhaRani, Raja, Raja--, ---]
```

* W.A.p. to insert StringBuffer object in treeset where sorting order is alphabetical order.

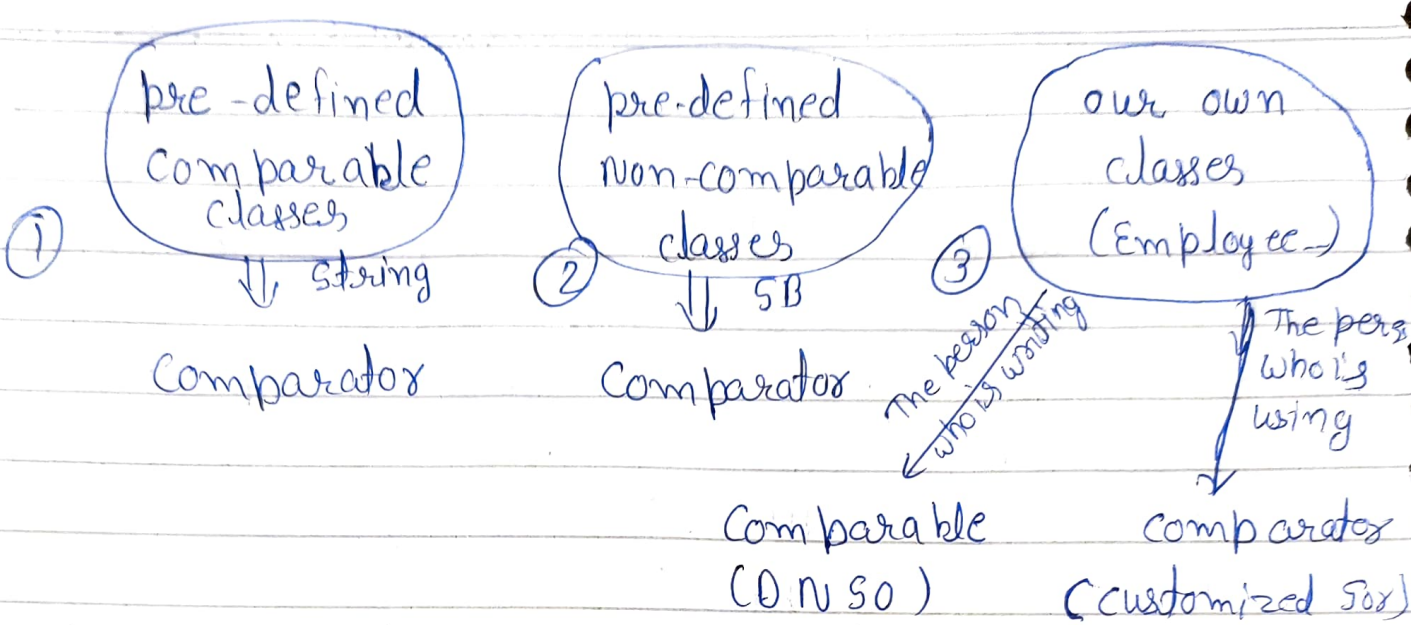
Ans:-
```
Treeset t = new Treeset(new myComp());
t.add(new SB("A"));
t.add(new SB("Z"));
t.add(new SB("K"));
t.add(new SB("L"));
sop\n(t);                    ["A","K","L","Z"]
```

```
> class myComperator implements Comperator {
      p int compare(obj1, obj2) {
          String s1 = obj1.toString();
          String s2 = obj2.toString();
          return s1.compareTo(s2);
  } }
```

Q. W.A.P. to insert String & String Buffer objects into treeset where sorting order is inc. length order, if two obj. having same length then consider their alpha. order.

```
> TreeSet t = new TreeSet(new MyComperator);
  t.add("A");
  t.add(new SB("ABC"));
  t.add(new SB("AA"));
  t.add("XX");
  t.add("ABCD");
  t.add("A");
  Sop(t);        [A, AA, XX, ABC, ABCD]

> class MyComperater implement comperator {
    public int compare(obj obj1, obj obj) {
      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 default natural sorting already available if we are not satisfied with that order then we can define our own sorting by using comparator.
2. For pre-defined non-comparable classes (like SB) default natural sorting order not-already available. we can define our own sorting by using comparator.
3. For our own classes ~~or~~ (like employee), the person who is writing the class is responsible to define default natural sorting order (DNSO) by implementing comparable interface.

   The person who is using our class, if he is not satisfied by it then he can define his own sorting by using comparator.

Q:-

```
> class Employee implements Comparable {
    String name;
    int eid;
    Employee (String name, int eid) {
        this.name = name;
        this.eid = eid;          }
    public String toString () {
        return name+"---" + eid; }
    public int compareTo (Object obj) {
        int eid1 = this.eid;
        Employee e = (Employee) obj;
        int eid2 = e.eid;
        if(eid1 < eid2){
            return -1;
        }else if (eid1 > eid2){
            return +1;  }
        else { return 0;}
    }
}

> class CompComp {
    p s v m(S[] a) {
        Employee e1= new Employee ("nag", 100);
        Employee e2 = new Employee ("bakaiah", 200);
        Employee e3 = new Employee("chiru", 50);
        Employee e4 = new Employee("venki", 150);
```

```
Employee e5 = new Employee("nag", 100);
TreeSet t = new TreeSet();
t.add(e1);
t.add(e2);
t.add(e3);
t.add(e4);
t.add(e5);
Sop(t);
TreeSet t1 = new TreeSet(new MyComparator());
t1.add(e1);
t1.add(e2);
t1.add(e3);
t1.add(e4);
t1.add(e5);
Sop(t1);
}
}
```

```
class MyComparator implements Comparator{
    public int compare(Object obj1, Object obj2){
        Employee e1 = (Employee) obj1;
        Employee e2 = (Employee) obj1;
        String s1 = e1.name;
        String s2 = e2.name;
        return s1.compareTo(s2);
    }
}
```

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

* Comparison of Comparable & Comparator

Comparable	Comparator
① It is ment for D.N.S.O.	① It is ment for customized SO
② Present in Java.lang pkg.	② Present in Java.util pkg.
③ It has only one method CompareTo().	③ It has 2 methods compare(), equals()
④ String & All wrapper Classess implement comparable interface.	④ The only implemented class are collator & RuleBased-Collator.

* Comparison of set implemented Classess.

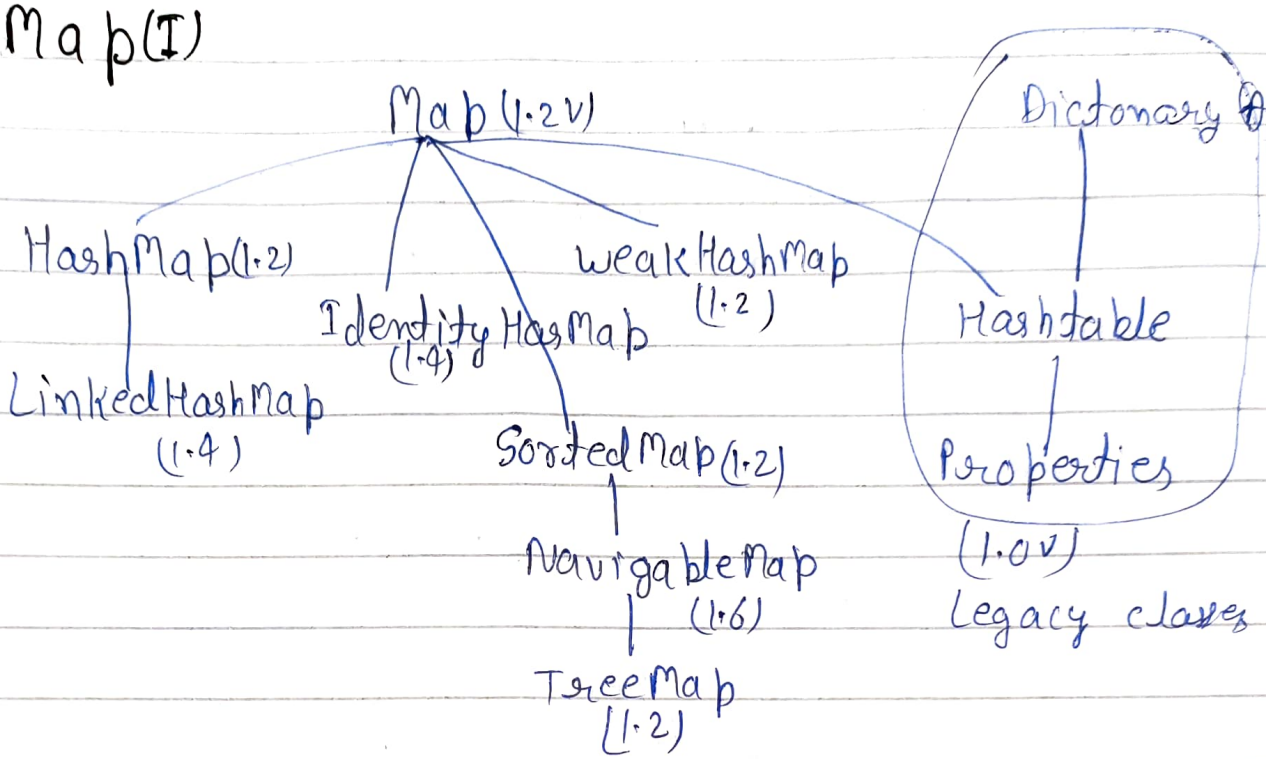
Property	HashSet	LinkedHashSet	Treeset
① Underlying DS	Hashtable	LL+Hashtable	Balanced tree
② Duplicate	Not allowed	Not allowed	Not allowed
③ Insertion ord.	Not preserved	Preserved	Not Preserved
④ Sorting order	N.A	N.A	Applicable
⑤ Heterogene-ous object	Allowed	Allowed	Not Allowed
⑥ null acceptance	Allowed	Allowed	For empty as first ele. null is allowed.

Hi 1.6 V
1.6 V ↗ not allowed

#146 Collections

23/Apr/23

* 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	101	102	103	104
value	Durga	Ravi	Shiva	Pawan

→ Both keys & values are object only.

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

→ Each key-value pair is entry hence map is considered as a collection of entry objects.

* methods

① Object put(Object key, Object value)

old value ← m.put(101, "durg");
null ← m.put(102, "Shiva");
null ← m.put(101, "Ravi");
durga ←

101 - ~~durga~~ Ravi
102 - Shiva

→ To add one key-value to map.
→ If the key is already present then old value will be replaced ~~by~~ with new value. & returns old value.

② void putAll(Map m);
③ object get(Object key);
④ object remove(object key);
⑤ boolean containsKey(Object key)
⑥ boolean containsValue(Object value)
⑦ boolean isEmpty()
⑧ int size()
⑨ void clear();

① Set keySet()
② Collection values() } Collection views of Map.
③ Set entrySet()

* Entry (I)
→ It is inner interface of Map.
→ without map entry can't exist.

```
> interface Map {
        interface Entry {
            ① object getKey()
            ② object getValue()
            ③ object setValue(object newo)
        }
}
```

} only on Entry obj

* HashMap *

→ The underlying D.S is Hashtable.
→ Insertion order is not preserved.
→ It is based on hash code of keys.
→ Duplicate keys are not allowed
→ Value can be duplicated.
→ Hetrogenas objects are allowed for both key & value
→ null allowed for key (only once)
→ null allowed for values (any no. of times)
→ It applies serizable & cloneable I. but not Random Access.
→ It is best choice for search operation.

* Constructor

① HashMap m = new HashMap();
→ 16 → 0.75
② HashMap m = new HashMap(int initialCap);
③ HashMap m = new HashMap(int iniCap, float fillRatio);
④ HashMap m = new HashMap(Map m1);

> p s v m(S[] args){
HashMap m = new HashMap();
m.put("chirn", 700);
m.put("balaiah", 800);
m.put("venkatesh", 200);
m.put("nagarjuna", 500);
Sop ln(m); {K=V, K=V, ---}
Sop ln(m.put("chirn", 1000)); //700

```
Set s = m.keySet();
Sop(s);
Collection c = m.values();
Sop(c);
Set s1 = m.entrySet();
Sop(s1);
Iterator itr = new Iterator();
while(itr.hasNext()){
    Map.Entry m1 = (Map.Entry) itr.next();
    Sop(m1.getKey() + "---" + m1.getValue());
    if(m1.getKey().equals("naga")){
        m.setValue(10000);      }
}
Sop(m); { k=v, ... }
```

* Diff b/w HashMap & Hashtable

HashMap	Hashtable
① Not synchronized	① Synchronized
② Multiple thread can operate.	② Only one thread can operate.
③ Not thread safe	③ Thread safe
③ Relatively performance is high	③ Rel. performance is low.
④ null is allowed for key & value	④ null is not allowed for key & value
⑤ It is non legacy (1.2V)	⑤ It is legacy (1.0V)

* How to get synchronized version of HashMap object.

→ HashMap m = new HashMap();

> Map m1 = Collections.**synchronizedMap(m)**;

m1 ↓ Synchronized

m ↓ non-Sync.

#147. Collections

23/Apr/23

* LinkedHashMap

→ It is the child class of HashMap.

HashMap	LinkedHashMap
① D.S Hashtable	① LinkedList + Hashtable
② Insertion order not preserved	② Insertion order preserved
③ 1.2 V	③ 1.4 V

```
> p s v m (S[] a){
      LinkedHashMap m = new LinkedHashMap();
      m.put ("ch", 700);
      m.put ("bala", 800);
      m.put ("venki", 200);
      m.put ("naga", 500);
      Sopln(m)
  }
```

→ LinkedHashset & LinkedHashMap are commanly used for developing cache based applications.

* diff b/w == & .equals()

→ "==" ment for reference comparision (address)

→ .equals() ment for content comparision

```
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);
Sopln(I1 == I2) ; false
Sopln(I1.equals(I2)); true
```

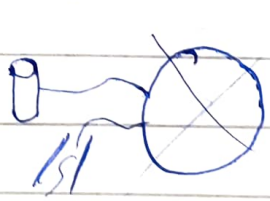
* IdentityHashMap

> HashMap m = new HashMap();
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);
m.put(I1, "Pawan");
m.put(I2, "Kalyan");
Sopln(m); { 10 = Kalyan }

$I_1 \rightarrow (10)$
$I_2 \rightarrow (10)$
10 = ~~Pawan~~ Kalyan
$I_1.equals(I_2)$ → true.

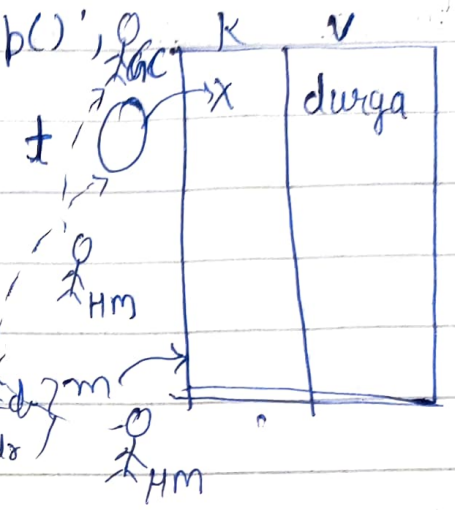
→ Same as HashMap except ~~one~~
→ In the HashMap JVM will use .equals() method to identify duplicate keys, which is ment for content comparision but
→ In case of IdentityHashMap JVM will use "==" operator to identify duplicate keys, which is ment for reference comparision (address comp.)
> IdentityHashMap m1 = new IdentityHashMap.
Sopln(m1); { 10 = Kalyan, 10 = Pawan }

* WeakHashMap



> class Temp {
public String toString(){
return "temp"); }
public void finalize(){
Sopln("finalize method called") }
}

> HashMap m = new HashMap();
Temp t = new Temp();
m.put(t, "durga");
Sopln(m); {t emp=durga}
t = null;
System.gc();
Thread.sleep(5000);
Sopln(m); {temp=durga}



> WeakHashMap m = new WeakHashMap();
> {temp=durg} > Finalize > {}

→ In the case of HashMap, eventhough object doesn't have any reference. it is not eligible for G.C. if it is associated with HashMap. i.e. Hash Map dominates GC. but

→ in the case of weakHashMap, if object does not contain any references. it is eligible for GC. eventhough object associated with weakHashMap. i.e. G.C dominates WeakHashMap.

#148 Collections

23|Apr|23

* SortedMap (I)

→ Child interface of Map.

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

→ Sorting is based on key but not on value.

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

* methods

> Object firstKey()
> Object lastKey()
> SortedMap headMap(Object key)
> SortedMap tailMap(Object key)
> SortedMap subMap(Object key1, Obj key2)
> Comparator comparator()

> firstKey() → 101 > lastKey → 136
> headMap(107) → {101=A, 103=B, 104=C}
> tailMap(107) → {107=D, 125=E, 136=F}
> subMap(103, 125) → {103=B, 104=C, 107=D}
> comparator() → null

* TreeMap

→ The underlying D.S is RED-BLACK Tree.

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

→ Duplicate keys ⇒ Not allowed
Duplicate values ⇒ Allowed

→ If we are depending on default natural sorting order then key should be homogenus & comparable. otherwise RE: CCE.

→ If we are defining our own sorting by comparator then the keys need not be homogenus & comparable.

→ No restrictions for values.

* null acceptance :-

→ for non-Empty TreeMap, if we trying to insert an entry with null key then RE: NPE.

→ For Empty TreeMap as the first entry with null key is allowed but after inserting that entry if we try to insert any other entry then RE: NPE.

Note:- Null accaptance applicable till 1.6 V

1.7 V ⇒ Null is not allowed for key.

→ For values ⇒ No restriction.

* Constructors

① TreeMap t = new TreeMap(); → DNSO of key.

② TreeMap t = new TreeMap(Comparator c); → CS.O

③ TreeMap t = new TreeMap(SortedMap m);

④ TreeMap t = new TreeMap(Map m);

```
> p s v m(S[]a){
>   TreeMap m = new TreeMap();
    m.put(100, "ZZ");
    m.put(103, "Y");
    m.put(101, "X");
    m.put(104, 106);
    Sop(m);
```

m.put("FFF","XXX"); CCE
m.put(null, "XXX"); NPE

```
> TreeMap t = new TreeMap(new MyComparator());
  t.put("X", 10);
  t.put("A",20);
  t.put("z", 30);
  t.put("L", 40);
  Sop(t);    }
```

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

#149. Collections

23/Apr/23

* Hashtable

→ underlying D.S is Hashtable.
→ Insertion order is not preserved & it is based on hashcode of keys.
→ Duplicate keys are not allowed & value can be duplicated.
→ Hetrogeneus keys are allowed for keys & values
→ null is not allowed for both keys & values.
→ It implements serizable & clonable interfaces but not Random access.
→ Every method present in it is synchronized.
→ It is thread safe.
→ It is best choice for search operation.

* Constructors.

① Hashtable h = new Hashtable();
[11] [0.75]
② Hashtable h = new Hashtable(int intCap);
③ Hashtable h = new Hashtable(int intCap, float fillRatio);
④ Hashtable h = new Hashtable(Map m);

```
> class Temp {
    int i;
    Temp(int i) {
      this.i = i; }
    public int hashCode() {
      return i;    }
```

```
public String toString(){
    return it" ";
}
```

```
> 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
> Sop(h);
```

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

From Top to bottom
From Right to Left

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

* Properties

→ In our program if anything which changes frequently (like username, Pass., email Id ...) are not recommended to hardcode in java. program bcz if there is any change. to reflect that change re-compilation, Rebuild, Redeploy application required.

→ Even sometime server re-start also require. which creates a big business to the client.

→ We can overcome this problem by using properties file.
→ Such type of variable things we have to configure in properties file.
→ From that properties file we have to read into java programm & we can use those properties
→ We can use java properties object to hold properties which are coming from properties file.
→ In normal map, key & value can be any type but in the case of properties key-value should be String type.

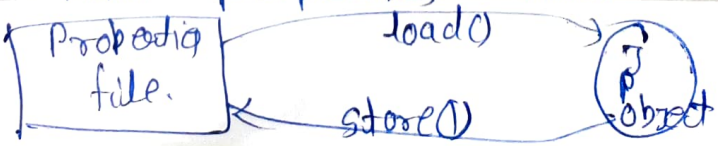
* Constructor

Properties p = new Properties();

* Methods

1) String getProperty(String pname);
2) String setProperty(String pname, String pvalue);
3) Enumeration propertyNames()

> void load(InputStream is)
→ to load properties from properties file into java properties object.
> void store(OutputStream os, String comment)
→ to store properties from java properties object into properties file.



io & util

```
class PropertiesDemo {
    p s v m(S[] args) throws Exception {
        Properties p = new Properties();
        FileInputStream fis = new FileInputStream
                              ("abc.properties")
        p.load(fis);
        Sopln(P);
        String s = p.getProperty("venki");
        Sopln(s);
        p.setProperty("nag", "8888");
        FileOutputStream fos = new FileOutputStream(
                               "abc.properties");
        p.store(fos, "Updated by Durga"); }
}
```

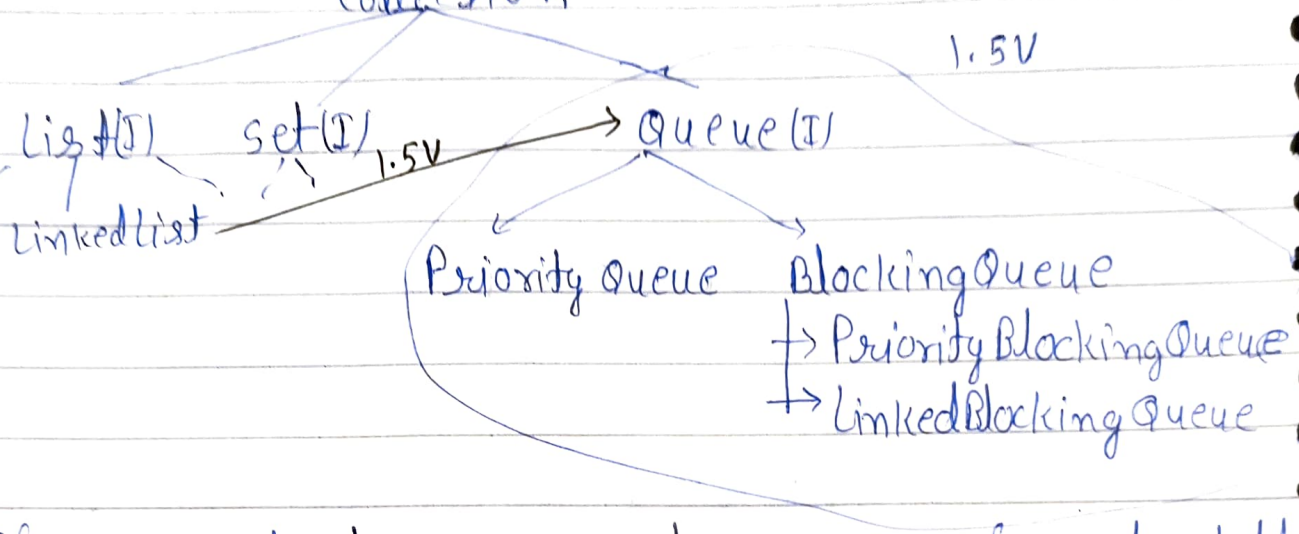
#150 Collection

23/Apr/23

* Queue

→ 1.5 V enhancement.

→ It is child interface of collection.



→ If we want to represent a group of individual objects ~~the~~ ~~to~~ prior to processing then we should go for Queue.

→ Message service.

→ Usually Queue follows First IN First Out (FIFO) but based on our requirement we can imple. our own priority order also (Priority Queue)

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

→ LinkedList based implementation of Queue always follow FIFO order.

* methods

① boolean offer (Object o)

→ to add element into the queue.

② Object peek()
to return head element of the queue, if Queue is empty then this returns null.

③ object element()
to return head element of the queue. if queue is empty then this method raises RE: NSEE

④ object poll()
to remove & return head element of the queue. if queue is empty then returns null.

⑤ Object remove()
to remove & return head element of the queue. if queue is empty then ~~returns~~ raises RE: NSEE.

* PriorityQueue :-

→ If we want to represent a group of individual objects prior to processing AIC to some priority then we should go for priorityQueue.
→ The priority can be either default Natural Sorting Order (DNSO) or customize S.O defined by comparator.
→ Insertion order not preserved & it is based on some priority.
→ Duplicates object are not allowed.
→ object should be homo. & comparable. other RE: CCE
→ If customized sorting the hetro. & ~~com~~ is allowed
→ null is not allowed.

* Constructor :-

> PriorityQueue q = new PriorityQueue();
> PriorityQueue q = new PriorityQueue(int ini.cap);
> PriorityQueue q = new PriorityQueue(int ini.cap, Comparator c);
> PriorityQueue q = new PriorityQueue(SortedSet s);
> PriorityQueue q = new PriorityQueue(Collection c);

```
> class PriorityQueue {
      p s v m(S[] args){
          PriorityQueue q = new PriorityQueue();
          Sopln(q.peek()); // null.
          Sopln(q.element()); // RE: NSEE.
          for(int i=0; i<=10; i++){
              q.offer(i)  }
          Sopln(q); // [0,1,2, --- 10]
          Sopln(q.poll); // 0
          Sopln(q) // [0,1,2,--,10] }
  }
```

→ Some platform won't provide proper support for thread priorities & priority Queues.

#151 Collections

* 1.6V Enhancements in Collection Framework.

→ Two concept introduced in 1.6 of Collection framework

① NavigableSet(I) ② NavigableMap(I)

* NavigableSet (I)

→ It is child interface of SortedSet.

→ It defines several method for navigable purposes

* methods

> floor(e)
  it returns highest ele. which is <=e

> lower(e)
  it returns highest ele. which is <e

> ~~lower~~

> ceiling(e)
  it returns lowest ele. which is >=e

> higher(e)
  it returns lowest ele. which is >e

> pollFirst()
  remove & return first element.

> pollLast()
  remove & return last element.

> descendingSet()
  it returns navigableSet in reverse order.

> TreeSet t = new TreeSet();
> TreeSet<Integer> t = new TreeSet<Integer>();

```
t.add(1000);
t.add(2000);
t.add(3000);
t.add(4000);
t.add(5000);
Sop1n(t);
Sop1n(t.ceiling(2000)); // 2000
Sop1n(t.lower(3000)); // 2000
Sop1n(t.higher(2000)); // 3000
Sop1n(t.floor(3000)); // 3000
Sop1n(t.pollFirst()); // 1000
Sop1n(t.pollLast()); // 5000
Sop1n(t.descendingSet()); // [4000, 3000, 2000]
Sop1n(t); // [2000, 3000, 4000]
```

* NavigableMap (I)

Map 1.2V
↓
SortedMap 1.2V
↓
NavigableMap 1.6V
↓
TreeMap 1.2V

→ It is child of SortedMap.
→ It defines several methods for navigation purposes.

* methods

> floorKey(e)
> lowerKey(e)
> ceilingKey(e)
> higherKey(e)
> pollFirstEntry()
> pollLastEntry()
> descendingMap()

> TreeMap<String, String> t = new TreeMap<String,St.>(
t.put("b", "banana");
t.put("c", "cat");
t.put("a", "apple");
t.put("d", "dog");
t.put("g", "gun");
Sopln(t);
Sopln(t.ceilingKey("c")); // c
Sopln(t.higherKey("e")); // g
Sopln(t.floorKey("e")); // d
Sopln(t.lowerKey("e")); // d
Sopln(t.pollFirstEntry()); // a = apple
Sopln(t.pollLastEntry()); // g = gun.
Sopln(t.descendingMap()); // {d=dog, c=, b=, a
Sopln(t); {b=, c=, d=}

*Collections

→ It defines several utility methods for Col. obj.

*Sorting elements of List

1. public static void sort(List l)
   - → Sort based on D.N.S.O.
   - → List should contain Homo. & comparable obj. (CCE)
   - → Should not contain null, o RE: NPE.

2. public static void sort(List l, Comparator c)
   to sort based on C.S.O.

```
> p s v m(S[] a){
    Arraylist l = new Arraylist();
    l.add("Z")
    l.add("A")
    l.add("K")
    l.add("N")
    Collections.sort(l);
    Sop(l);                                    }
   Collections.sort(l, new MyComparator()); }

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