Array:-

→ It is an indexed collection of fixed numbers of homogeneous data elements.

Adv:-

→ We can represent multiple values with a single variable.
→ So reusability of the code will be improved.

Limitations:-

→ Fixed in size.
→ Can hold only homogeneous data elements. (Can resolve by using Object Arrays)
→ Readymade method support is not available for Array.

⇒ To overcome the above limitations of Arrays we should go for collections.

Collections:-

→ These are growable in nature.
→ It can hold both homogeneous & Hetrogeneous elements.
→ Readymade method support is available for collections.

Arrays	Collections
→ It is fixed in size.	→ Growable in nature.
→ It can hold only homogeneous data.	→ It can hold both homogeneous & hetrogeneous elements.
→ There is no underlying data structure for Array.	→ Every collections class is implemented based on some standard data structure.
→ It can hold both primitive & object types.	→ It can hold only objects but no primitives.
→ w.r.t. memory not recommended to use.	→ memory → (✓)
→ w.r.t. performance recommended to use.	performance → (x)

2
a

Collection Framework :-

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

Collection :- (Interface)

→ 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.

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

Collection

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

List

→ Duplicates are allowed.

→ Insertion order preserved.

Collections

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

SET

→ Duplicates are not allowed.

→ Insertion order not preserved.

List Interface :-

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

Set Interface :-

→ It is the child interface of collection.
→ If we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion order not preserved then we should go for Set.

SortedSet Interface :-

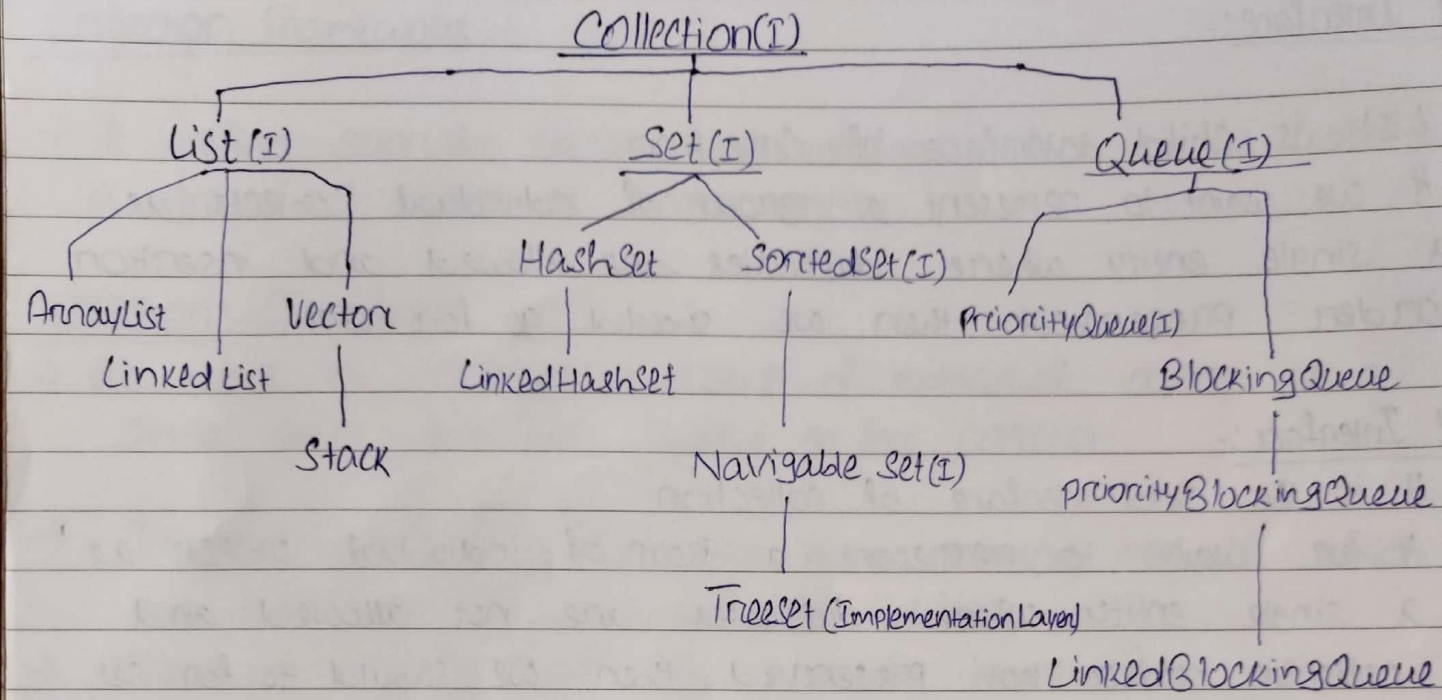
→ It is a child interface of Set.
→ If we want to represent a group of individual objects as a single entity where duplicates are not allowed but all objects should be inserted according to some sorting order then we should go for SortedSet.

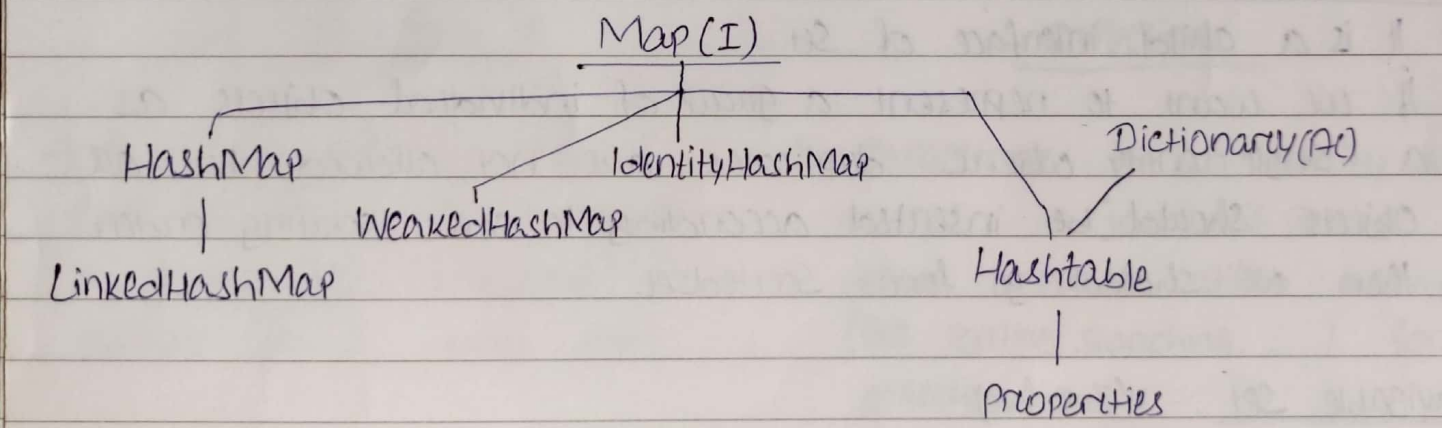
Navigable Set :- (Interface)

→ It is the child interface of SortedSet. It defines several methods for navigation purposes.

Queue :- (Interface)

→ It is child interface of collection.
→ If we want to represent a group of individual objects prior to processing then we should go for Queue.
→ (First in First out) is the best choice.





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

Cursors:
1. Enumeration (I)
2. Heriator (I)
3. ListIterator (I)

Utility classes:
1. Collections
2. Arrays

9 Key interfaces of Collection Framework:-

→ All of the above interfaces (Collection, List, Set, SortedSet, NavigableSet and Queue) meant for representing a group of individual objects.
→ If we want to represent a group of objects as key value pairs then we should go for Map interface.

Map : (Interface)

→ Map is not the child interface of collection.
→ If we want to represent a group of individual objects as key value pairs then should go for Map.
→ Both key and value are objects, duplicated keys are not allowed but values can be duplicated.

SortedMap:- (Interface)

→ It is the child interface of Map.
→ If we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap.

NavigableMap:- (Interface)

→ It is the child interface of sorted map, it defines several utility methods for navigation purpose.

Methods in Collection Interface :-

→ Collection interface doesn't contain any method to retrive objects there is no concrete class which implements collection class directly.

- → boolean add(Object o)
- → boolean addAll(Collection c)
- → boolean remove(Object o)
- → boolean removeAll(Collection c)
- → boolean retainAll(Collection c)
- → void clear()
- → boolean contains(Object o)
- → boolean containsAll(Collection c)
- → boolean isEmpty()
- → int size()
- → Object[] toArray()
- → Iterator iterator()

List Interface :-

→ We can differentiate duplicates by using index.

→ We can preserve insertion order by using index, hence index play very important role in the list interface.

- → void add(int index, Object o)
- → boolean addAll(int index, Collection c)
- → Object get(int index)
- → Object remove(int index)
- → Object set(int index, Object new)
- → int indexOf(Object o)
- → int lastIndexOf(Object o)
- → ListIterator listIterator()

ArrayList :-

→ The underlined data structure Resizable Array or Growable Array.
→ Duplicates 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 al = new ArrayList()
2. ArrayList al = new ArrayList(int initialCapacity)
3. ArrayList al = new ArrayList(Collection c)

Capacity:- ArrayList al = new ArrayList()

Creates an empty ArrayList object with default intial capacity 10. Once ArrayList reaches its maximum capacity then a new ArrayList will be created with new capacity = (currentcapacity * 3/2) +1

Example:-

```
ArrayList al = new ArrayList();
al.add("A");
al.add("10");
al.add("A");
al.add(null);
S.O.P.(al); // [A, 10, A, null]
al.remove(2);
S.O.P (al); // [A, 10, null]
al.add("2", "m");
al.add("N");          S.O.P(al); //[A, 10, M, null, N]
```

Serializable & Cloneable Interfaces:-

Usually we can use collections to hold and transfer objects from one place to another place, to provide support for this requirement every Collection already implements Serializable & Cloneable interfaces.

Random Access Interface:-

→ Present in java.util.package
→ It does not contain any methods and it is a Marker interface.
→ ArrayList & Vector classes implements RandomAccess interface so that we can access any random element with the same speed.

Example:-

```
ArrayList al1 = new ArrayList();
LinkedList al2 = new LinkedList();
S.O.P. (al1 instanceof Serializable); // true
S.O.P. (al2 instanceof Cloneable); // true
S.O.P. (al1 instanceOf RandomAccess); // true
S.O.P. (al2 instanceOf RandomAccess); // false
```

Usage:-

→ ArrayList is the best choice if our frequent operation is retrieval operation. (Because ArrayList implements RandomAccess Interface)

→ ArrayList is the worst choice if our frequent operation is insertion or deletion in the middle. (Because several shift operations are required).

ArrayList	Vector
→ Every method present in ArrayList is non synchronize.	→ Every method present in ~~LinkedList~~ Vector is synchronize.
→ At a time multiple threads are allowed to operate on ArrayList object and hence ArrayList is not thread safe.	→ At a time only one thread is allowed to operate on vector object is thread safe.
→ Performance is high.	→ Performance is low. (Threads required to wait to operate).

How to get synchronized version of ArrayList object ?

Ans:- By default ArrayList object is non-synchronized but we can get synchronized version of ArrayList by using Collections class synchronizedList() method.

Public static List synchronizedList(List I)

Non-Synchronized :- ArrayList I = new ArrayList();

Synchronized :- List I = Collections.synchronizedList(I);

Note:- Similarly we can get Synchronized version of Set, Map objects by using the following methods

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

LinkedList :-

→ The underlying data structure is Double Linked List
→ Insertion order is preserved.
→ Duplicates are allowed.
→ Heterogeneous objects are allowed.
→ Null insertion is possible.

LinkedList Constructors :-

* LinkedList ll = new LinkedList();
  Create an empty LinkedList Object.

* LinkedList ll = new LinkedList (Collection c);
  Creates an equivalent linkedlist object for the given collection.

LinkedList Methods :-

```
void addFirst();
void addLast();
object getFirst();
Object getLast();
Object removeFirst();
object removeLast();
```

LinkedList Usage :-

→ LinkedList implements Serializable and Clonable interfaces but not RandomAccess interface.
→ It is the best choice if our frequent operation is insertion or deletion in the middle.
→ It is the worst choice if our frequent operation is retrieval operation.

Demo Prog:-

```
LinkedList ll = new LinkedList();
ll.add("durga");
ll.add(30);
ll.add(null);
ll.add("durga");
ll.set(0, "Software");
ll.add(0, "venkey");
ll.addFirst("ccc");
S.O.P.(ll); // [ccc, venkey, Software, 30, null]
```

ArrayList	LinkedList
→ It is the best choice if our frequent operation is retrieval.	→ It is the best choice if our frequent operation is insertion and deletion.
→ It is the worst choice if our frequent operation is insertion or deletion.	→ It is the worst choice if our frequent operation is retrieval operation.
→ It implements RandomAccess interface.	→ It doesn't implement RandomAccess interface.
→ Underlying datastructure for ArrayList is resizable or growable Array.	→ Underlying data structure is Double Linked List.

Vector :-

→ The underlying data structure for the vector is resizable array or growable array.
→ Duplicate objects are allowed.
→ Insertion order is preserved.
→ Null insertion is possible.
→ Heterogeneous objects are allowed.
→ Vector class implemented Serializable, Clonable & RandomAccess Interface
→ Methods inside vector are synchronized.
→ Vector object is Thread-safe.
→ Best choice if the frequent operation is retrieval.

Constructors of Vector class :-

1) Vector v = new Vector();
Create an empty vector object with default initial capacity 10, once vector reaches its maximum capacity a new vector object will be created with [new capacity = 2 * current capacity]

2) Vector v = new Vector(int intialCapacity);
Creates an empty vector object with specified intial capacity.

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

4) Vector v = new Vector(Collection c);
Creates an equivalent vector object for the given collection

Methods:-

for adding objects:-

add(Object O) [from Collection - List(I)]
add(int index, Object O) [from List]
addElement(Object O) [from Vector]

for removing objects:-

Remove(Object O) – [from collection]
removeElement(Object O) – [from Vector]
remove(int index) – [from List]
RemoveElementAt(int index) – [from Vector]
clear() – [from collection]
removeAllElements() – [from Vector]

Other methods:-

int size(), int capacity(),
Enumeration elements();

for Accessing Elements:-

Object get(int index) [from collection]
Object elementAt(int index) [from Vector]
Object firstElement() [from Vector]
Object lastElement() [from Vector]

Prog:-

```
Vector V = new Vector();
S.O.P (V.capacity()); //10
V.add
```

Stack S = new Stack();

Stack :-

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

Stack Construction :-

```
Stack S = new Stack();
```

Methods in Stack :-

1) Object push(Object obj);
   - For inserting an object to the stack.
2) Object pop();
   - To removes and returns top of the stack.
3) Object peak();
   - To returns the top of the stack without removal of object
4) int search(Object obj);
   - If the specified object is available it returns its offset from top of the stack.
   - If the object is not available then it returns -1.

Demo prog :-

```
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
```

~~Enumeration~~ :-

Cursors :-

→ If we want to retrieve objects one by one from the Collection, then we should go for Cursors.
→ There are 3 types of cursors are available in java.
* Enumeration
* Iterator
* ListIterator

Enumeration :-

→ Introduced in 1.0 version. (Legacy)
→ We can use Enumeration to get objects one by one from the old collection objects (legacy collections)
→ We can create Enumeration object by using elements() methods of Vector class.

Public Enumeration elements();

Example: Enumeration e = V.elements();

Methods of Enumeration :-

* Public boolean hasMoreElements();
* public Object nextElement();

Prog :-

```
Vector V = new Vector();
for(int i=0; i<=10; i++){
    V.addElement(i);
}
S.O.P(V); //[0,1,2....10]
```

```
Enumeration e = V.elements();
while(e.hasMoreElements()){
Integer i = (Integer) e.nextElement();
if((i%2)=0)
    S.O.P(i); //[0,2,4,....10]
}
```

Iterator :-

→ We can apply Iterator concept for any Collection object hence it is universal cursor.

→ By using Iterator we can perform both read and remove operations.

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

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

Example :

```
Iterator itr = C.iterator();
```

* where C is any Collection Object

Methods in Iterator :-

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

Limitations of Iterator :-

→ By using Enumeration and Iterator we can move only towards forward direction and we can not move to the backward direction, and hence these are single direction cursors.

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

* To overcome this we should go for ListIterator.

Demo prog for Iterator:

```
ArrayList l = new ArrayList();
for (int i=0; i<10; i++){
    l.add(i);
}
S.O.P (l);
Iterator itr = l.iterator();
while (itr.hasNext()){
    Integer n = (Integer) itr.next();
    if (n%2 == 0)
        S.O.P (n); // 02468
}
S.O.P (l); // [0,1,2,....10]
}
```

ListIterator :-

→ By using ListIterator we can move either to the forward direction or to the backward direction, and hence ListIterator is bidirectional cursor.

→ By using ListIterator we can perform replacement and addition of new objects in addition to read and remove operation.

Methods in ListIterator :-

→ It is the child interface of Iterator & hence all methods of Iterator by default available to ListIterator.

→ It defines the following 9 methods

forward direction —
1. public boolean hasNext()
2. public void next()
3. public int nextIndex()

Backward direction —
4. public boolean hasPrevious()
5. public void previous()
6. public int previousIndex()

Other capability methods —
7. public void remove()
8. public void set(Object new)
9. public void add(Object new)

Note :- ListIterator is the most powerful cursor but its limitation is, it is applicable only for List implemented class objects and it is not a universal cursor.

Progm:-

```
LinkedList l = new LinkedList();
l.add("Rajesh");
l.add("Sunil");
l.add("Vinod");
l.add("Amitabh");
S.O.P (l); // [Rajesh, Sunil, Vinod, Amitabh]
ListIterator ltr = l.listIterator();
while(ltr.hasNext())
{
 String s = (String) ltr.next();
 if(s.equals("Vinod")){
     ltr.remove();
  }
 elseif(s.equals("Amitabh")
  {
     ltr.add("Abhishek");
  }
 elseif(s.equals("Sunil"))
 {
     ltr.set("Sanjay");
  }
}
S.O.P (l); // [Rajesh, Sanjay, Amitabh, Abhishek]
```

Comparision of Cursors :-

Property	Enumeration	Iterator	ListIterator
Applicable	only legacy class	Any collection	only List classes
Movement	only forward direction (single dir^n)	only forward direction	both forward and backward direction
Accessibility	Only read Access	Read or Remove	Read, remove, replace & addition of new objects
How to get it?	By using elements() method of vector class	By using iterator() method of collection (I)	By using ListIterator() method of List interface
Methods	2 Methods hasMoreElements() nextElement()	3 methods hasNext() next() remove()	9 methods

Implementation of class of cursors :-

Enumeration :- e.getClass().getName()
// java.util.Vector$1

Iterator :- itr.getClass().getName()
// java.util.Vector$Itr

ListIterator :- itr.getClass().getName()
// java.util.Vector$ListItr

Set :-

→ It is the child interface of collection.

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

→ Set interface does not contain any methods, so we have to use collection interface methods.

HashSet :-

→ The underlying data structure is Hashtable.
→ Duplicates are not allowed. If we are trying to insert duplicates, we won't get any compiletime or runtime error. add() method simply returns false.
→ Insertion order is not preserved and all objects will be inserted based on hash-code of objects.
→ Heterogeneous objects are allowed.
→ 'Null' insertion is possible.
→ Implements Serializable and clonable interfaces but not RandomAccess.
→ HashSet is the best choice, if our frequent operation is Search operation.

Constructors of HashSet :-

1) HashSet h = new HashSet();
Creates an empty HashSet object of intial capacity 16.
default fill Ratio is 0.75

2) HashSet h = new HashSet(int initialCapacity)

3) HashSet h = new HashSet(int initialCapacity, float LoadFactor)

4) HashSet h = new HashSet(Collection C)
for inter conversion between collection objects.

LoadFactor / fill Ratio :-

After loading the how much factor, a new Hashset object is created, that factor is called LoadFactor or Fill Ratio.

Progm :-

```
HashSet h=new HashSet();
h.add("B");
h.add("C");
h.add("D");
h.add("Z");
h.add("null");
h.add(10);
S.O.P (h.add("Z")); // false
S.O.P (h); // [null, D, B, C, 10, Z]
```

HashSet	LinkedHashSet
→ The underlying datastructure is Hash table.	→ The underlying datastructure is Hash table + Linked List. (i.e. hybrid datastructure)
→ Insertion order is not preserved.	→ Insertion order is preserved.

Note :- LinkedHashSet is the best choice to develop cache based applications, where duplicates are not allowed and insertion order must be preserved.

Progm for LinkedHashSet :-

```
LinkedHashSet lh = new LinkedHashSet();
lh.add("B");
lh.add("C");
lh.add("D");
lh.add("Z");
lh.add("null");
lh.add("10");
S.O.P.(lh.add("Z")); // false
S.O.P.(lh); // [B, C, D, Z, null, 10]
```

SortedSet (I) :-

→ It is the child interface of Set.

→ If we want to represent a group of individual objects according to some sorting order and duplicates are not allowed then we should go for SortedSet.

Methods of SortedSet :-

1) Object first() :- returns first element of the SortedSet.
2) Object last() :- returns last element of the SortedSet.
3) SortedSet headSet(Object obj) :- returns the sorted set whose elements are < obj.
4) SortedSet tailSet(Object obj) :- returns the SortedSet whose elements are >= obj

5) SortedSet subSet(Object obj1, Object obj2)
   - returns the sortedset whose elements are >= obj1 & < 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.

Example :

{100, 101, 103, 104, 107, 110, 115}

1. First() → 100
2. last() → 115
3. headSet (104) → [100, 101, 103]
4. tailSet (104) → [104, 107, 110, 115]
5. subSet (103, 110) → [103, 104, 107]
6. comparator() → null

Note :-

1. Default natural sorting order for numbers Ascending order and for String alphabatical order.

2. We can apply the above methods only on SortedSet implemented class objects. That is on the TreeSet object.

TreeSet :-

→ The underlying data structure for TreeSet is Balanced Tree.
→ Duplicates objects are not allowed.
→ Insertion order not preserved, but all objects will be inserted according to some sorting order.
→ Heterogeneous objects are not allowed.
→ If we are trying to insert heterogeneous objects then we will get runtime exception saying ClassCastException.
→ Null insertion is allowed, but only once.

TreeSet Constructors :-

1) TreeSet t = new TreeSet();
   → Creates an empty TreeSet object where elements will be inserted according to default natural sorting order.

2) TreeSet t = new TreeSet(Comparator c);
   → Creates an empty TreeSet object where elements will be inserted according to customized sorting order.

3) TreeSet t = new TreeSet(SortedSet s);

4) TreeSet t = new TreeSet(Collection c);

Ex:-

```
TreeSet t = new TreeSet();
t.add("A");
t.add("a");
t.add("B");
t.add("Z");
t.add("L");
// t.add(new Integer(10)); // ClassCastException
t.add(null); // NullPointerException
S.O.P(t); // [A, B, L, Z, a]
```

Null Acceptance:-

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

→ For Non empty TreeSet if we are trying to insert Null then we will get NullPointerException.

Note:-

→ If we are depending on default natural sorting order then objects should be homogeneous and comparable. Otherwise we will get runtime exception saying ClassCastException.

→ An object is said to be comparable if and only if the corresponding class implements java.lang.comparable interface.

→ String class and all wrapper classes already implements comparable interface. But StringBuffer does not implement comparable interface.

→ If we depending on default natural sorting order internally JVM will call compareTo() method will inserting objects to the TreeSet. Hence the objects should be comparable.

```
TreeSet t = new TreeSet();
t.add("B");
t.add("Z");     // "Z".compareTo("B"); +ve
t.add("A");     // "A".compareTo("B"); -ve
S.O.P(t);       // [A,B,Z]
```

Comparator Interface :-

→ We can use comparator to define our own sorting (customized sorting.)

→ Comparator Interface present in java.util Package.

→ It defines two methods, compare and equals.

1) public int compare(Object obj1, Object obj2)
   - → returns -ve if obj1 has to come before obj2.
   - → returns +ve if obj1 has to come after obj2.
   - → returns 0 if obj1 ~~has to com~~ & obj2 are equal

2) public boolean equals();

Note :-

→ Whenever we are implementing comparator interface, compulsory we should provide implementation for compare() method.

→ And implementing equals() method is optional, because it is already available in every java class from Object class through inheritance.

Comparable Interface :-

→ This interface present in java.lang package it contains only one method compareTO().

* Public int compareTO(Object obj)

Example :- obj1.compareTo(obj2)

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

→ If we ~~dont~~ are not satisfied with default natural sorting order or if the default natural sorting order is not already available then we can define our own customized sorting by using Comparator.

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

Example :-

1) S.O.P. ("A".compareTO("Z")); //-Ve
S.O.P. ("Z".compareTO("B")); //+Ve
S.O.P. ("A".compareTO("A")); // 0
S.O.P. ("A".compareTO(null)); NullPointerException

2) TreeSet t = new TreeSet();
t.add(new StringBuffer("A"));
t.add(new StringBuffer("B"));
S.O.P.(t); // ClassCastException

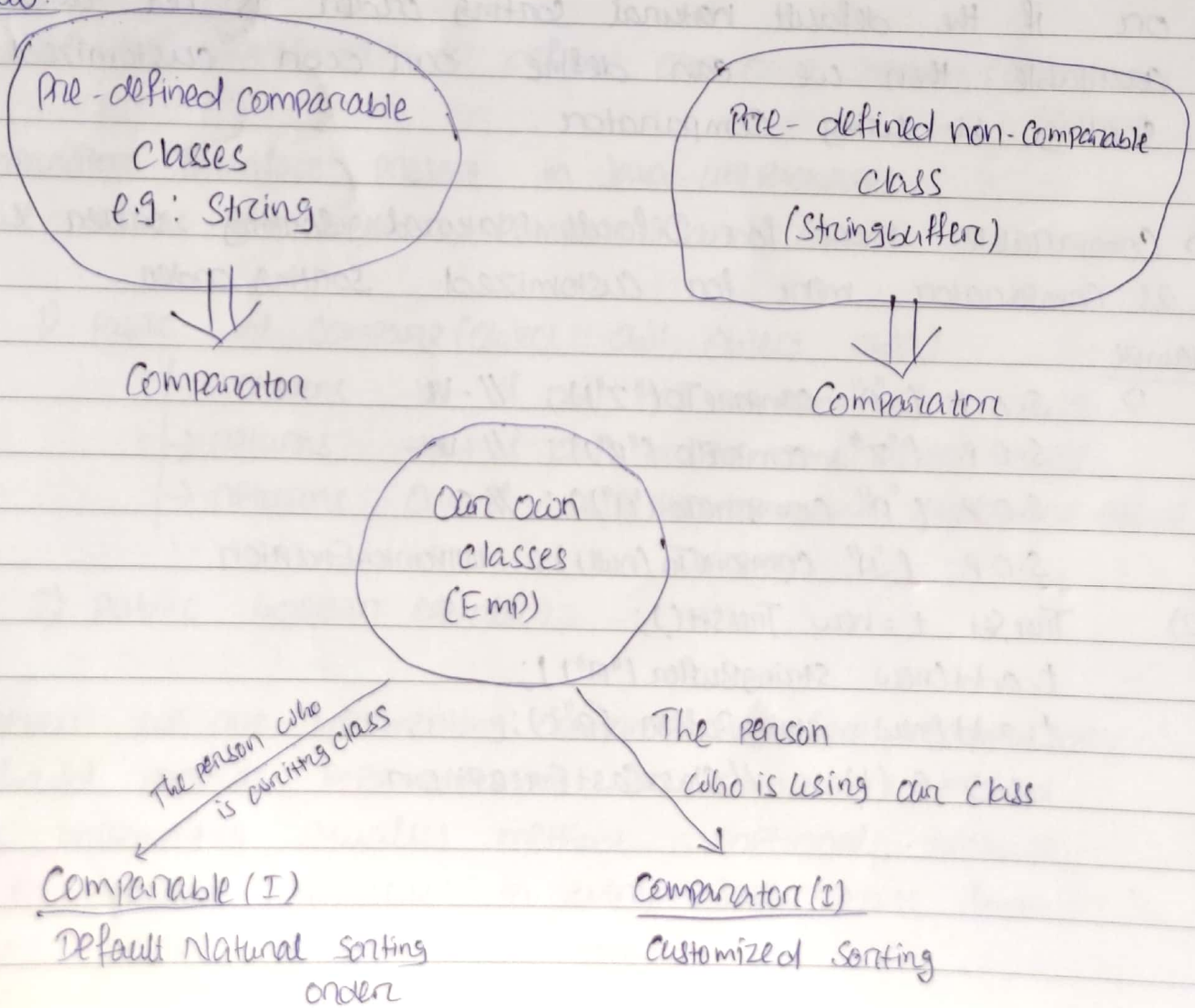
Comparable

→ It is meant for default natural sorting order.
→ Present in java.lang package.
→ This interface defines only one method compareTo().
→ All wrapper class and String class implement comparable interface.

Comparator

→ It is meant for customized sorting order.
→ present in java.util package
→ This interface defines two methods : compare()
& equals()
→ The only implemented classes of Comparator are Collator and RuleBasedCollator.

Workflow:-



Note:-

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

Comparision table of Set(I):-

Property	HashSet	LinkedHashSet	TreeSet
Underlying Data Structure	Hashtable	Hashtable + LinkedList	Balanced Tree
Insertion order	not preserved	preserved	not preserved
Sorting order	not applicable	not applicable	applicable
Heterogeneous objects	allowed	allowed	not allowed
Duplicate objects	not allowed	not allowed	not allowed
Null Acceptance	Allowed (only once)	Allowed (only once)	for empty TreeSet as first element allowed. Otherwise (case) we will get NullPointer Exception.

Map:-

→ Map is not child interface of collection.

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

	key	value
Entry ←	101	Durga
	102	Ravi
	103	Shiva
	104	Pavan

→ Both keys & values are objects only.

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

→ Each key value pair is called entry. Hence Map is considered as a collection of entry objects.

Methods:-

1. Object put (Object key, Object value)

→ To add one key value pair to the map.

→ If the key is already present then old value will be replaced with new value.

e.g:- null ← m.put(101, "durga");
null ← m.put(102, "Shiva");
durga ← m.put(101, "Ravi");

Ravi
101 - durga
102 - Shiva

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 associated with specified key.
5. boolean containsKey(Object key)
6. boolean containsValue(Object value)
7. boolean isEmpty()
8. int size()
9. void clear() :- all key value pairs will be removed

Collection views of Map :-

1. Set keySet() :- to get all the keys
2. Collection values() :- to get all the values
3. Set entrySet() :- to get all the entry

Entry(I) :-

→ Map is a group of key value pairs & each key value pair is called an entry. Hence Map is considered as a collection of entry objects.

→ Without existing Map object there is no chance of entry object Hence entry interface is defined inside Map interface.

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

} Entry specific methods and we can apply only on Entry object

HashMap:-

→ Underlying DS is HashTable.
→ Insertion order is not preserved & it is based on hashcode of keys.
→ 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 (any number of times).
→ HashMap implements Serializable & Clonable interface but not RandomAccess Interface.
→ It is the best choice if our frequent operation is search operation.

Constructors:-

① HashMap m = new HashMap();
default initCapacity 16, fillRatio → 0.75
② HashMap m = new HashMap(int initialCapacity);
③ HashMap m = new HashMap(int itiCap, Float fillRatio)
④ HashMap m = new HashMap(Map m)

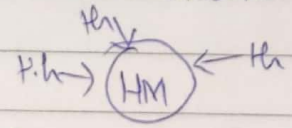
e.g.:-

```
HashMap m = new HashMap();
m.put("c", 700);
m.put("B", 800);
m.put("V", 200);
m.put("N", 500);
S.O.P(m);          // {k=v, k=v, ....}
S.O.P(m.put("c", 1000));   // 700
Set s = m.keySet();
S.O.P(s);          // [k, K]
Collection c = m.values();
S.O.P(c);
```

```
Set s1 = m.entrySet();
S.O.P. (s1) ; // [K=V, K=V, ...]
Iterator itr = s1.iterator();
while (itr.hasNext())
{
    Map.Entry m1 = (Map.Entry) itr.next();
    S.O.P (m1.getKey() + "..." + m1.getValue());
    if (m1.getKey() equals ("N"))
    {
        m1.setValue(10000);
    }
}
S.O.P. (m);
}
```

HashMap

→ Not synchronized
→ Not threadsafe.



→ Performance is high
→ null ← key
value

HashTable

→ Synchronized
→ threadsafe

→ Performance is low
→ null not allowed (NPE)