Collection Framework

Purpose: To represent many values by using single object.

Problem: To represent 10k values we declare 10k variables. By doing this the readability of the program is going to be down and this is worst kind of programming. To overcome this, we can go for arrays and collections.

Arrays: Student[] s = new Student[10000]; We can represent huge no of homogeneo us type of elements by using single object(variable) with arrays.

Problems with Arrays:

1. **Fixed in Size:** Once we create an array with some size there is no chance of increasing or decreasing the size of that array based on our requirement.
2. **Hold Homogeneous Elements:** Arrays can hold only homogeneous type of elements.

   Ex: s[0] = new Student();--(valid)

   s[1] = new Customer();--(CE: incompatible types found: Customer required: Student)

   Here the problem is that s[1] is expecting Student object but we are providing Customer object.

   Note: We can solve this problem by using **object type arrays**.

   Object[] a = new Object(10000); a[0] = new Student();-(valid) a[1] = new Customer();--(va lid)
3. **No Readymade Method Support:** Arrays are not implemented based on some standard data structure. So, it is developer's responsibility to write the code for every requirement like storing the elements based on some sorting order, search for an element is there or not in the array etc.
   Note: 1. To store the elements based on some sorting order we go for **TreeSet**.
   2. **contains( )** method is available in Collections to search an element.

4. To use arrays, it is better to know the size in advance, which is may not possible always.
   Note: If we know the size in advance, then it is highly recommended to go for arrays. Because performance wise arrays are better than collections.
   Ex:

al →	1	2	……	1 Crore
Index →	0	1	…….	9999999

Suppose, we have 1 Crore elements in an ArrayList. If we want to insert a new element, then a new ArrayList is created with 1Cr + 1 size, all 1Cr elements are copied from old ArrayList to

new ArrayList with the same reference variable (al), now the old object is eligible for garbage collection and the new element is inserted to new ArrayList. It takes huge amount of time.

Collection Framework: Collection framework defines several interfaces and classes which can be used to represent a group of individual objects as a single entity (i.e. collection).

Collection(I): Collection is a group of individual objects as a single entity. Collection is an interface which is used to represent a group of individual objects as a single entity.

Collections(C): Collections is a utility class presents in java.util package to define several utility methods like sorting, searching etc. in collection objects.

Ex: To sort an ArrayList Collections.sort(al); al → ArrayList Object

Arrays vs Collections:

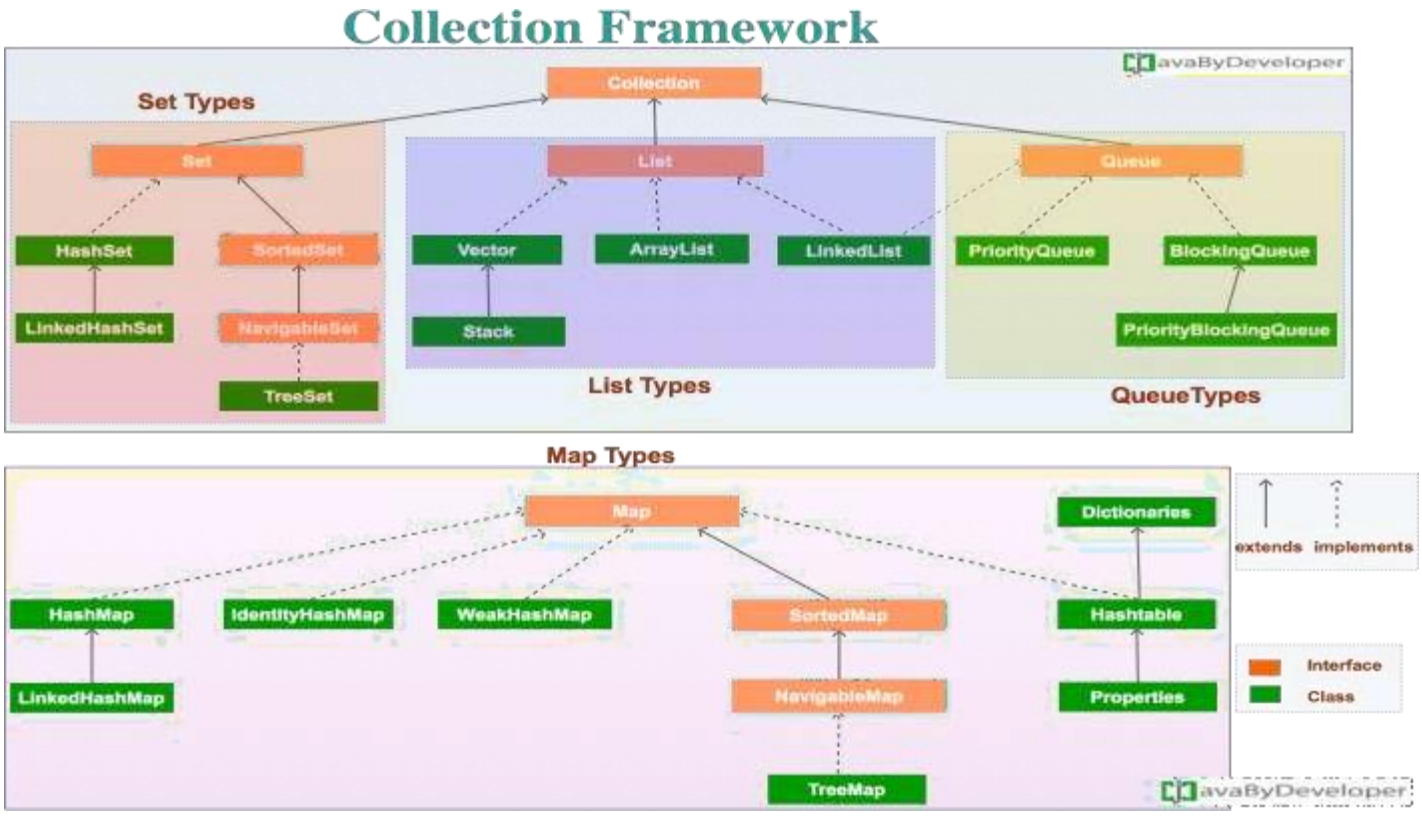
Arrays	Collections
1. Array are fixed in size.	1. Collections are growable in nature.
2. W.r.t memory arrays are not recommended.	2. W.r.t memory collections are recommended.
3. W.r.t performance arrays are recommended.	3. W.r.t performance collections are not recommended.
4. Arrays can hold only homogeneous elements.	4. Collections can hold both homogeneous and heterogeneous elements.
5. Arrays are not implemented based on any DS. So, readymade method support is not available.	5. Every collection class is implemented based on some standard DS. So, readymade method support is available.
6. Arrays can hold primitive types and reference types(objects).	6. Collections can hold only reference types(objects).
7. There are no predefined classes in Java API which represents arrays.	7. There are so many predefined classes are available in Java API (java.util package) which represents collections.
8. We can't use the cursors to process the array elements.	8. We can use cursors to process the collection elements.

9 Key Interfaces of Collection Framework:

1. Collection (1.2) 4. SortedSet (1.2) 7. Map (1.2)

2. List (1.2)
5. NavigableSet (1.6)
8. SortedMap (1.2)

3. Set (1.2)
6. Queue (1.5)
9. NavigableMap (1.6)



1. **Collection(I):** Collection interface is used to represent a group of individual objects as a single entity. It defines the most common methods which are applicable for many collection objects.

No.	Method	Description
1	public boolean add(Object element)	is used to insert an element in this collection.
2	public boolean addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	is used to delete an element from this collection.
4	public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	return the total number of elements in the collection.
7	public void clear()	removes the total no of element from the collection.
8	public boolean contains(Object element)	is used to search an element.
9	public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
10	public Iterator iterator()	returns an iterator.
11	public Object[] toArray()	converts collection into array.
12	public boolean isEmpty()	checks if collection is empty.
13	public boolean equals(Object element)	matches two collection.
14	public int hashCode()	returns the hashcode number for collection.

Note: There is no concrete class which represents Collection interface directly and it doesn't contain any method to retrieve objects.

Q) What is the difference between removeAll() and clear()? Which one to choose and when?

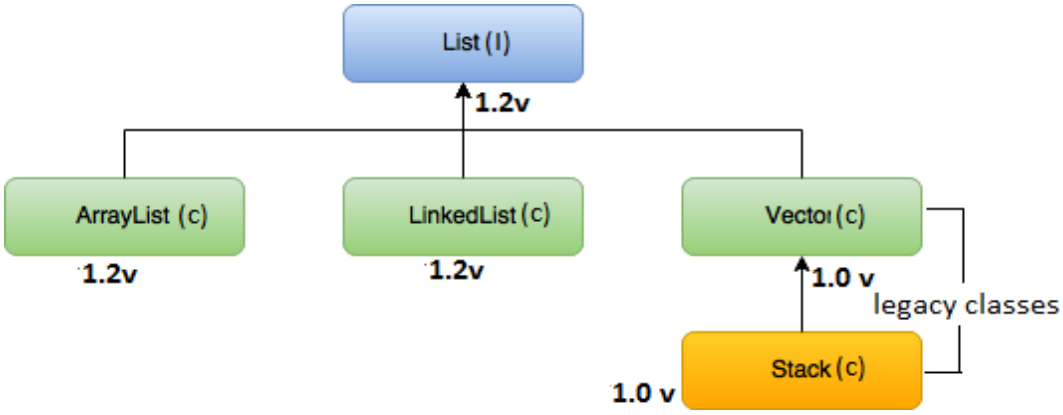
removeAll()	clear()
1. The removeAll() method removes the elements of a collection by comparing with the elements of other collection.	1. The clear() method blindly removes all the elements from a collection.
2. The return type is boolean.	2. The return type is void.
3.To perform removeAll() we require two collection objects.	3. To perform clear() we require only one collection object.
4.The removeAll() is slower than clear().	4. The clear() is faster than removeAll().
5. If we observe the implementation of removeAll(), the time complexity is O(n2).	5. If we observe the implementation of clear(), the time complexity is O(n).
6. Choose 'removeAll()' when you want to remove only certain elements from the collection by comparing with other collection.	6. Choose 'clear()' method when you want to remove all elements from the collection without any cross checks.

Ex: ArrayList a1 = new ArrayList(); a1.add(2); a1.add(5); a1.add(7);

ArrayList a2 = new ArrayList(); a2.add(1); a2.add(2); a2.add(7);

a1.removeAll(a2); //remove all elements from a1 if they exists in a2 SOP(a1); // **O/P:** 5
a1.clear(); // remove all elements from a1 and **O/P:** [ ]

2. List(I): List is the child interface of Collection interface. If we want to represent a group of individual objects as single entity where duplicates are allowed and insertion order must be preserved, then we go for List.



1. ArrayList(C): ArrayList is a class presents in java.util package which extends(inherits) from AbstractList class and implements from List, RandomAccess, Cloneable and Serializab le interfaces.

i) Properties:

- The underlying DS is resizable array or growable array.
- Duplicates are allowed and Insertion order is preserved.
- Heterogeneous elements are allowed and null insertion is possible.
- ArrayList is the best choice if our frequent operation is **retrieval operation** because ArrayList implements RandomAccess interface and ArrayList is the worst choice if our frequent operation is **insertion or deletion** in the middle because several shift operations will take huge amount of time.

Note: Except **TreeSet** and **TreeMap** everywhere heterogeneous elements are allowed.

ii) **Declaration:**

```
public class ArrayList<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, Serializable
```

iii) **Constructors:**

(a) ArrayList al = new ArrayList();

Creates an empty ArrayList with default initial capacity 10. Once ArrayList reaches its maximum capacity then a new ArrayList will be created with

```
new capacity = (current capacity * 3/2) + 1
```

& with same reference variable(al).

(b) ArrayList al = new ArrayList(int initial_capacity);

Creates an empty ArrayList with specified initial capacity.

(c) ArrayList al = new ArrayList(Collection c);

Creates an equivalent ArrayList for the given Collection.

Note:

a. Usually we use collections to hold and transfer objects from one location to another location(container) through network. To provide support for this requirement every collection class by default implements **Serializable** interface and to provide backup for the collection object every collection class implements **Cloneable** interface.

b. In Collections, only **ArrayList** and **Vector** classes are by default implements **RandomAccess** interface. So, that any random element we can access with same speed.

c. **RandomAccess:** RandomAcess interface presents in java.util package and it doesn't contain any methods, so it is a **marker interface**, where required ability will be provided by JVM automatically.

```
Ex: ArrayList l1 = new ArrayList();
    LinkedList l2 = new LinkedList();
    System.out.println(l1 instanceof Serializable);      // true
    System.out.println(l2 instanceof Serializable);      // true
    System.out.println(l1 instanceof Cloneable);         // true
    System.out.println(l2 instanceof Cloneable);         // true
    System.out.println(l1 instanceof RandomAccess);      // true
    System. out.println(l2 instanceof RandomAccess);     // false
```

ArrayList vs Vector:

ArrayList	Vector
1.Every method present in the ArrayList is non-synchronized.	1.Every method present in the Vector is synchronized.
2.At a time, multiple threads can operate on single ArrayList object and hence it is not thread safe.	2.At a time, only one thread allowed to operate on single vector object and hence it is thread safe.
3.Relatively performance is high because threads are not required to wait to operate on ArrayList object.	3.Relatively performance is low because threads are required to wait to operate on Vector object.
4.Introduced in 1.2 version and it is nonlegacy	4.Introduced in 1.0 version and it is legacy.

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.

Ex: ArrayList al = new ArrayList();

List l = Collections.synchronizedList(al);

Here, ArrayList object al is non-synchronized and List object l is synchronized.

Definition of synchronizedList() Method: public static List synchronizedList(List l){…..}

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

public static Set synchronizedSet(Set s){}

public static Map synchronizedMap(Map m){}

2. **LinkedList(C):** LinkedList is a class presents in java.util package which extends(inher its) from AbstractSequentialList class and implements from List, Deque, Cloneable and Serializable interfaces.

i) Properties:

- The underlying DS is Doubly LinkedList.
- Duplicates are allowed and Insertion order is preserved.
- Heterogeneous elements are allowed and null insertion is possible.
- If our frequent operation is **insertion** or **deletion** in the middle, then **LinkedList** is the best choice. Because the internal structure of the LinkedList. LinkedList is the worst choice if our frequent operation is **retrieval operation**.

ii) Declaration:

```
public class LinkedList<E> extends AbstractSequentialList<E> implements
List<E>, Deque<E>, Cloneable, Serializable
```

iii) Constructors:

(a) LinkedList l = new LinkedList(); Creates an empty LinkedList object

(b) LinkedList l = new LinkedList(Collection c);

Creates an equivalent LinkedList object for the given Collection.

iv) Methods of LinkedList:

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
void addFirst(Object o)	It is used to insert the given element at the beginning of a list.
void addLast(Object o)	It is used to append the given element to the end of a list.
int size()	It is used to return the number of elements in a list
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean contains(Object o)	It is used to return true if the list contains a specified element.
boolean remove(Object o)	It is used to remove the first occurence of the specified element in a list.
Object getFirst()	It is used to return the first element in a list.
Object getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.

Note: Usually LinkedList is used to develop Stacks and Queues.

ArrayList vs LinkedList:

ArrayList	LinkedList
1. ArrayList internally uses resizable array or dynamic array to store the elements.	1. LinkedList internally uses **doubly linked list** to store the elements.
2. Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	2. Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
3. ArrayList class can act as a list only because it implements List only.	3. LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4. ArrayList is the best choice if our frequent operation is **retrieval**.	4. LinkedList is the best choice if our frequent operation is **insertion or deletion** in the middle.
5. In ArrayList elements are stored in consecutive memory locations.	5. In LinkedList elements are not stored in consecutive memory locations.

3. Vector(C): Vector is a class presents in java.util package which extends(inherits) from AbstractList class and implements from List, RandomAccess, Cloneable and Serializab le interfaces.

i) Properties:

- The underlying DS is resizable array or growable array.
- Duplicates are allowed and Insertion order is preserved.
- Heterogeneous elements are allowed and null insertion is possible.
- Every method present in the vector is synchronized and hence vector object is thread safe.

ii) Declaration:

```
public class Vector<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, Serializable
```

iii) Constructors:

(a) Vector v = new Vector();

Creates an empty vector object with default initial capacity 10. Once Vector reaches its max capacity then a new vector will be crated with (double) capacity = current capacity*2.

(b) Vector v = new Vector(int initial_capacity);

Creates an empty Vector object with specified initial capacity.

(c) Vector v = new Vector(int initial_capacity, int incremental_capacity);

(d) Vector v = new Vector(Collection c);

Creates an equivalent vector object for the given Collection. This constructor meant for interconversion between collection objects.

iv) Methods of Vector:

- **To add objects**

  add(Object o) method from Collection(I)

  add(int index, Object o) method from List(I)

  addElement(Object o) method from Vector(C)

- **To remove objects**

  remove(Object o) method from Collection(I)

  clear() method from Collection(I)

  remove(int index) method from List(I)

  removeElement(Object o) method from Vector(C)

  removeElement(int index) method from Vector(C)

  removeAllElements() method from Vector(C)

- **To get objects**

  Object get(int index) method from List(I)

  Object elementAt(int index) method from Vector(C)

  Object firstElement() method from Vector(C)

  Object lastElement() method from Vector(C)

- **Other Methods**

  int size() – how many elements are present in the vector

  int capacity() – how many objects the vector can hold

  Enumeration elements() – to get the objects one by one from the collection

4. Stack(C): Stack is a class presents in java.util package which extends from Vector class. So, it is the child class of Vector and it is a specially designed for last-in-first-out (LIFO) order.

i) Declaration:
```
public class Stack<E> extends Vector<E>
```

ii) Constructors: Stack s = new Stack();

iii) Methods of Stack:

(a) Object push(Object o) – To insert an object into the Stack

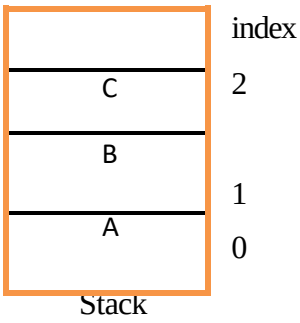
(b) Object pop(Object o) – To remove an object and return the top of the Stack

(c) Object peek() – To return top of the Stack without removal

(d) boolean empty() – Returns true if the Stack is empty

(e) int search(Object o) – Returns offset if the element is available otherwise returns -1

Ex:
```
Stack s = new Stack();
s.add('A');
s.add('B');
s.add('C');
SOP(s); // [A, B, C]
SOP(s.search('A')); // 3
SOP(s.search('Z')); // -1
```



Cursors: If you want to get object one by one from the collection then we should go for cursor. There are 3 types of cursors available in Java. 1. Enumeration 2. Iterator 3. ListIterator

1. Enumeration(I): 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();
```

Ex: Enumeration e = v.elements();

Methods:(i) public boolean hasMoreElements(); (ii) public object nextElement();

Limitations of Enumeration:

(i) We can apply Enumeration concept only for legacy classes and it is not a universal cursor.

(ii) By using Enumeration, we can get only read access and we can't perform remove operation

To overcome above limitations, we should go for Iterator

2. Iterator(I): We can apply Iterator concept for any collection object and hence it is universa l 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()
```

Ex: Iterator itr = c,iterator(); // c is any collection object

Methods: (i) public boolean hasNext(); (ii) public Object next(); (iii) public void remove();

Limitations of Iterator:

(i) By using Enumeration and Iterator we can always move only towards direction and we can't move towards backward direction these are single direction cursors but not bidirectional cursors.

(ii) By using Iterator, we can perform only read and remove operations and we can't perform replacement and addition of new objects

To overcome above limitations, we should go for ListIterator.

3. ListIterator: By using ListIterator we can move either forward direction or backward direction and hence it is bidirectional cursor. By using ListIterator we can perform replacement and addition of new objects along with read and remove operations.

We can create ListIterator by using listIterator() method of List interface.

```
public ListIterator listIterator();
```

Ex: ListIterator ltr = l.listIteratorI(); // 'l' is any List object

Methods: ListIterator is the child interface of Iterator and hence all methods present on Iterator by default available to ListIterator. ListIterator defines the following 9 methods

Forward Operation	Backward Operation	Other Operation
(i) public boolean hasNext()	(iv) public boolean hasPrevious()	(vii)public void add(Object o)
(ii) public Object next()	(v) public Object previous()	(viii) public void remove()
(iii) public int nextIndex()	(vi) public int previousIndex()	(ix) public void set(Object o)

Note: The most powerful cursor is ListIterator but its limitation is it is applicable only for List objects.

Comparison between Enumeration, Iterator and ListIterator:

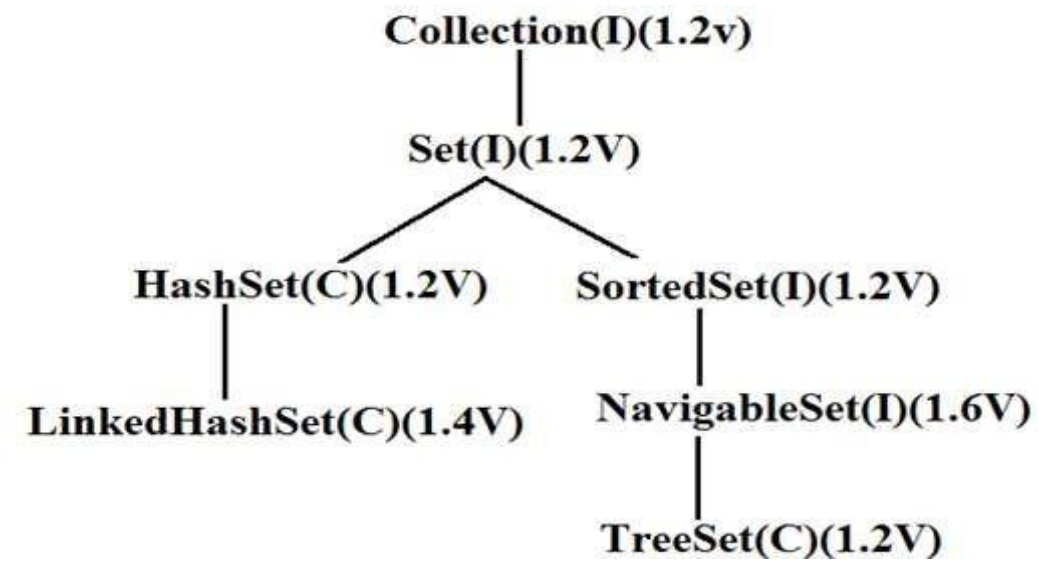
Property	Enumeration	Iterator	ListIterator
1.Where we can apply	Only for legacy classes	For any collection object	For only List object
2.Is it legacy?	Yes (1.0v)	No (1.2v)	No (1.2v)
3.Movement	Single directional (forward)	Single directional (forward)	Bidirectional (forward & backward)
4.Allowed Operations	Only Read	Read & Remove	Read, Remove, Add, Replace
5.How can we get?	By using elements() method of Vector(C)	By using iterator() method of Collection(I)	By using ListIterator() of List(I)
6.Methods	2 methods	3 methods	9 methods

Q: Enumeration, Iterator and ListIterator are the interfaces. How you get objects of interfaces?

We are not creating objects of interfaces, we are getting its internal implemented class objects.

Ex:
```
Internal Implementation of cursors
Vector v = new Vector();
Enumeration e = v.elements();
Iterator itr = v.iterator();
ListIterator ltr = v.listIterator();
System.out.println(e.getClass().getName()); // java.util.Vector$1
System.out.println(itr.getClass().getName()); // java.util.Vector$Itr
System.out.println(ltr.getClass().getName());// java.util.Vector$ListItr
```
That is, anonymous inner class(1) present in java.util.Vector, Itr inner class present in java.util.Vector and ListItr inner class present in java.util.Vector.

3.Set(I): Set is the child interface of Collection interface. If we want to represent a group of individual objects as single entity where duplicates are not allowed and insertion order not preserved, then we go for Set.



Note: Set interface doesn't contain any new method and we must use only Collection interface methods.

1. HashSet(C): HashSet is a class presents in java.util package which extends(inherits) from AbstractSet class and implements from Set, Cloneable and Serializable interfaces.

i) Properties:

- The underlying DS is Hash table.
- Duplicate objects are not allowed, Insertion order is not preserved and it is based on hash code of objects.
- Heterogeneous elements are allowed and null insertion is possible (only once).
- If our frequent operation is **search**, then **HashSet** is the best choice.

Note: In HashSet duplicates are not allowed if we are trying to insert duplicates then we won't get any compile time or run time errors and add() method simply returns **false**.

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

ii) Declaration:

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>,
Cloneable, Serializable
```

iii) Constructors:

(a) HashSet h = new HashSet(); Creates an empty HashSet object with default initial capacity 16 and default fill ratio 0.75.

(b) HashSet h = new HashSet(int initial_capacity); Creates an empty HashSet object with specified initial capacity and default fill ratio 0.75

(c) HashSet h = new HashSet (int initial_capacity, float fill_ratio);

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

Creates an equivalent HashSet object for the given Collection. This constructor meant for inter conversion between collection objects.

Fill Ratio (or) Load Factor: After filling how much ratio a new HashSet object will be created, this ratio is called Fill Ratio. For example fill ratio 0.75 means after filling 75% ratio a new HashSet object will be created automatically.

2. LinkedHashSet(C): LinkedHashSet is a class presents in java.util package which extends (inherits) from HashSet class and implements from Set, Cloneable and Serializable interfaces.

HashSet vs LinkedHashSet: It is exactly same as HashSet including constructors and methods except the following differences.

HashSet	LinkedHashSet
1.The underlying DS is Hash table.	1. The underlying DS is LinkedList and Hash table
2.Insertion order not preserved.	2. Insertion order preserved.
3.Introduced in 1.2v	3. Introduced in 1.4v

Note: In general, we can use LinkedHashSet to develop cache based applications where duplicates are not allowed and insertion order preserved.

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

SortedSet interface defines the following specific methods.

(a) Object first() - returns first element of the SortedSet

(b) Object last() - returns last element of the SortedSet

(c) SortedSet headSet(Object obj) – returns SortedSet whose elements are < obj

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

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

(f) Comparator comparator() – return Comparator object that describes underlying sorting technique like ascending, descending etc. If we are using default natural sorting order, then we will get **null**.

Ex: Consider a set as {100,101,104,106,110,115,120}

(a) first() – 100 (b) last – 120 (c) headSet(106) – {100,101,104} (d) tailSet(106) – {106,110,115,120} (e) subSet(101,115) – {101,104,106,110} (e) comparator() – null

TreeSet(C): TreeSet is a class presents in java.util package which extends(inherits) from AbstractSet class and implements from NavigableSet, Cloneable and Serializable interfaces.

i) Properties:

- The underlying DS is Balanced tree.
- Duplicate objects are not allowed, Insertion order is not preserved.
- Heterogeneous elements are not allowed, if we try to insert then we will get RE as **ClassCastException** and null insertion is not allowed (in 1.7 on wards).
- **Note:** For TreeSet null insertion is possible (only once) until 1.6 version. Until 1.6 version, for a nonempty TreeSet if we are trying to insert null then we will get NullPointerException and for an empty TreeSet as a first element null is allowed, but after inserting that null if we are trying to insert any other element then we will get Runtime exception as NullPointerException.
- All objects will be inserted based on some sorting order, it may be default natural sorting order or customized sorting order.

ii) Declaration:

```
public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>,
Cloneable, Serializable
```

iii) Constructors:

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

(b) TreeSet t = new TreeSet (Comparator c); Creates an empty TreeSet object where the elements will be inserted according to customized sorting order specified by comparator object.

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

Creates an equivalent TreeSet object for the given Collection.

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

Creates a TreeSet object for the given SortedSet.

Note:

```
TreeSet t = new TreeSet();
t.add("A");  // valid
t.add(new StringBuffer("B"));  // CE: ClassCastException
```

If we are depending on default natural sorting order compulsory the objects should be homogeneous and comparable otherwise we will get RE as ClassCastException. An object is said to be comparable iff corresponding class implements Comparable interface. String class and all wrapper classes implement Comparable interface but StringBuffer class doesn't imple ment Comparable interface. Hence, we got ClassCastException in the above example.

Comparable(I): It is present in java.lang package and it contains only one method compareTo().

```
public int compareTo(Object o)
```

Ex: obj1.compareTo(obj2); //compareTo() method is used to compare two objects.

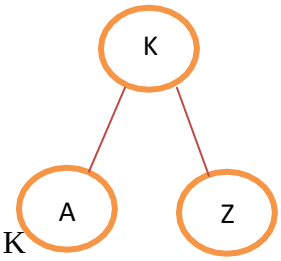
obj1 the object which is to be inserted, obj2 the object which is already inserted.

(a) returns -ve value if obj1 comes before obj2 (b) returns +ve value if obj1 comes after obj2

(c) returns 0 if obj1 is equals to obj2

Note: (i) SOP("A".compareTo(null)); // RE: NullPointerException

(ii) If we are depending on default natural sorting order then while adding objects into the TreeSet JVM will call comapreTo() method.



Ex: TreeSet t = new TreeSet();

t.add("K"); // It is the 1st element, hence comparison is not required.

t.add("Z"); // "Z".compareTo("K") – returns -ve value i.e. Z should come after K

t.add("A"); // "A".compareTo("K") – returns +ve value i.e. A should come before K

t.add("A"); // "A".compareTo("K") – returns +ve value i.e. A should come before K and "A".compareTo("A") – returns 0 i.e. Both are equal. Hence, "A" is not inserted for 2nd time. SOP(t); // [A,K,Z]

(iii) If default sorting order not available or if we are not satisfied with default natural sorting order then we can go for customized sorting by using Comparator.

Comparable meant for default natural sorting order where as Comparator meant for customized sorting order

Comparator(I):

Comparator present in java.util package and it defines 2 methods Compare() and equals().

1. `public int compare(Object obj1, Object obj2)`

Ex: compare(obj1, obj2); //compare() method is used to compare two objects.

obj1 the object which is to be inserted, obj2 the object which is already inserted.

(a) returns -ve value if obj1 comes before obj2 (b) returns +ve value if obj1 comes after obj2

(c) returns 0 if obj1 is equals to obj2

2. `public boolean equals(Object o)` returns true is both are equal else returns false.

Note: (i) When ever we are implementing Comparator interface complusory we should provide implementation only for compare() method and we are not required to provide implementation for equals() method because it is already available from Object class through inheritance.

(ii) If we are depending on default natural sorting order compulsory the objects should be homogeneoues and comparable. Otherwise we will get RE as ClassCastException. If we are defing our own sorting by Comparator then objects need not be comparable and homogeneous.i.e. we can add hetrerogeneous non comparable objects also.

Comparable vs Comparator:

1. For predefined comparable classes like String, default natural sorting order already available. If we are not satisfied with that default natural sorting order, then we can define our own sorting by using Comparator.

2. For predefined noncomparable classes like StringBuffer, default natural sorting order 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 default natural sorting order by implementing Comparable interface. The person who is using our class, if he is not satisfied with default natural sorting order then he can his own sorting by using Comparator.

Comparison of Comparable Comparator:

Comparable(I)	Comparator(I)
1. Comparable meant for default natural sorting order.	1. Comparator meant for customized sorting order.
2. Comparable presents in java.lang package.	2. Comparable presents in java.util package.
3. It contains only one method i.e. compareTo()	3. It contains 2 methods i.compare() ii.equals()

4. By default all wrapper classes and String class implement Comparable interface.	4. By default Collator, RulBasedCollator classes implement Comparable interface.

Comparison of Set interface implemented classes:

Property	HashSet	LinkedHashSet	TreeSet
1.Underlying DS	Hash table	LinkedList + Hash table	Balanced Tree
2.Duplicates	Not allowed	Not allowed	Not allowed
3.Insertion Order	Not preserved	Preserved	Not Preserved
4.Sorting Order	NA	NA	Applicable
5.Heterogeneous Objects	Allowed	Not allowed	Not allowed
6.null acceptance	Allowed	Allowed	For empty TreeSet as 1st element null is allowed.

5. NavigableSet(I): NavigableSet is the child interface of SortedSet and it defines several methods for navigation purposes.

NavigableSet interface defines the following specific methods.

(a) floor(e) - returns highest element which is <= e

(b) lower(e) - returns highest element which is > e

(c) ceiling(e) - returns lowest element which is >= e

(d) higher(e) - returns lowest element which is > e

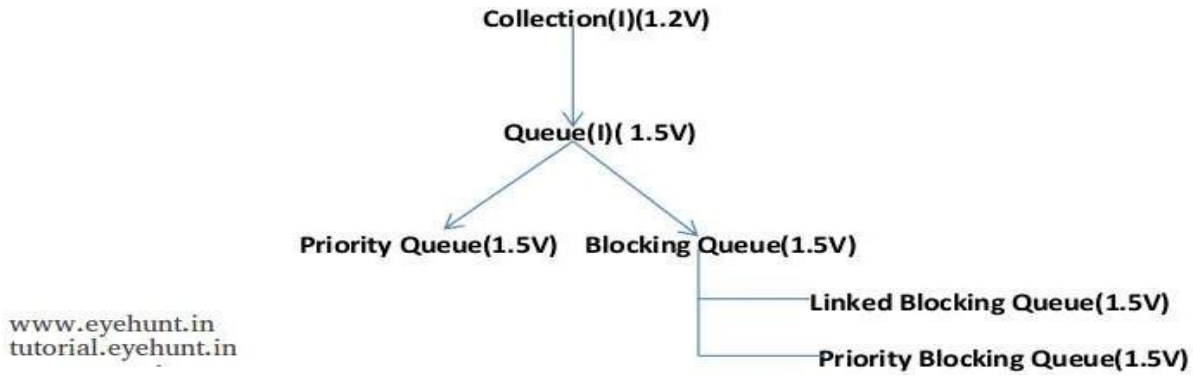
(e) pollFirst() – remove and return first element

(f) pollLast() – remove and return last element

(g) descendingSet() – returns NavigableSet in reverse order

```
Ex:   TreeSet<Integer> t = new TreeSet<Integer>();
      t.add(100);   t.add(200);   t.add(300);   t.add(400);   t.add(500);
      SOP(floor(300)); // 300     SOP(lower(300)); // 200     SOP(ceiling(200)); // 200
      SOP(higher(200)); // 300    SOP(pollFirst()); // 100    SOP(pollLast()); // 500
      SOP(descendingSet()); // [400, 300, 200]                 SOP(t); // [200, 300, 400]
```

6. Queue(I): Queue is the child interface of Collection. If we want to represent a group of individual objects prior to processing, then we should go for Queue. For example, before sending SMS messages all mobile numbers we have to store in some DS. In which order we added mobile numbers in the same order only messages should be delivered. For this FIFO requirement Queue is the best choice.



Usually Queue follows FIFO order but based on our requirement we can implement our own priority order also(PriorityQ ueue). From 1.5v on wards LinkedList class also impleme nts Queue interface. LinkedList based implementation of Queue always follows FIFO order.

Queue interface defines the following specific methods.

(a) boolean offer(Object o) – to add an object into the Queue

(b) Object peek() – returns head element of the Queue. If the Queue is empty returns null.

(c) Object element() – returns head element of the Queue. If the Queue is empty raise

RE: NoSuchElementException.

(d) Object poll() - remove &returns head element of the Queue. If the Queue is empty returns null.

(e) Object remove() - remove & returns head element of the Queue. If the Queue is empty raise

RE: NoSuchElementException.

PriorityQueue(C): PriorityQueue is a class presents in java.util package which extends(inherits) from AbstractQueue class and implements from Serializable interface. 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 either default natural sorting order or customized sorting order defined by Comparator.

i) Properties:

- Insertion order is not preserved and it is based on some priority.
- Duplicate objects are not allowed.
- If we are depending on default natural sorting order compulsory objects should be homogeneous and comparable otherwise we will get RE as ClassCastExcept ion.
- If we are defining our own sorting by Comparator then objects need not be homogeneous and comparable, Null is not allowed even as the first element also.

ii) Declaration:

```
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable
```

iii) Constructors:

(a) PriorityQueue q = new PriorityQueue(); Creates an empty PriorityQueue with default initial capacity 11 and all objects will be inserted according to default natural sorting order.

(b) PriorityQueue q = new PriorityQueue(int initial_capacity);

(c) PriorityQueue q = new PriorityQueue(int initial_capacity, Camparator c);

Creates an empty PriorityQueue with given initial capacity and all objects will be inserted according to customized sorting order.

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

Creates an equivalent PriorityQueue object for the given SortedSet.

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

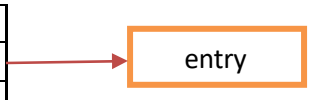
Creates an equivalent PriorityQueue object for the given Collection.

Note: Some OS platforms won't provide proper support for PriorityQueues and thread priorities.

7. Map(I):

Map is **not** child interface of Collection. If we want represent a group of objects as key value pairs ten we should go for Map interface.

Key	Value
101	A
102	B
103	C
104	A



Both keys and values are objects only. Duplicate keys are not allowed but duplicate values are allowed. Each key value pair is called entry. Hence, Map is considered as a collection of entry objects.

Methods:

(a) Object put(Object key, Object value) – to add one key value pair to the Map. If the key is already present, then old value will be replaced with new value and returns old value.

Ex: m.put(101, "A"); // Add this entry to the Map and returns null

m.put(101, "B");// Replace 'A' with 'B', has same key and returns 'A'

(b) void putAll(Map p) - Add the specified Map to current Map /Ex: m1.add(m2)

(c) Object get(Object key) – Returns the value associated with specified key

(d) Object remove(Object key) – Removes the entry associated with specified key

(e) boolean containsKey(Object key) - Search the specified key in the Map

(f) boolean containsValue(Object value) - Search the specified value in the Map

(g) boolean isEmpty()

(h) int size()

(i) void clear() – All key value pairs will be removed from the Map

Collection Views of Map Methods:

(j) Set keySet() - returns the Set containing all the keys

(k) Collection values() – returns the Set containing all the values

(l) Set entrySet() - return the Set view containing all the keys and values

Entry(I): Entry is the sub interface of Map. We will be accessed it by Map.Entry name. A Map is a group of key value pairs and each key value pair is called an entry. Hence, Map is considered as a collection of entry objects. Without existing Map object, there is no chance of existing entry object.

Methods:

(a) Object getKey() – used to obtain key

(a) Object getValue() - used to obtain value

(a) Object setValue(Object o) - used to replace the old value with new value

HashMap(C): HashMap is a class presents in java.util package which extends(inherits) from AbstractMap class and implements from Map, Cloneable, Serializable interfaces.

i) Properties:

- The underlying DS is Hash table.
- Insertion order is not preserved and it is based on hash code of keys.
- Duplicate keys are not allowed, but values can be duplicated.
- Heterogeneous objects are allowed for both key and value.
- Null is allowed for key (only once). Null is allowed for value (any no of tomes).
- HashMap is the best choice if our frequent operation is search operation.

ii) Declaration:

```
public class HashMap<K, V> extends AbstractMap<K, V> implements Map<K, V>,
Cloneable, Serializable
```

iii) Constructors:

(a) HashMap m = new HashMap();

Creates an empty HashMap with default initial capacity 16 and default fill ratio 0.75.

(b) HashMap m = new HashMap(int initial_capacity);

(c) HashMap m = new HashMap(int initial_capacity, float fill_ratio);

(d) HashMap m1 = new HashMap(Map m2);

Creates HashMap object 'm1' by using the elements of the given Map object m2.

HashMap vs Hashtable:

HashMap	Hashtable
1. Every method present in HashMap is non-synchronized.	1. Every method present in Hashtable is synchronized.

2. At a time multiple threads are allowed to operate on HashMap object and hence it is not thread safe.	2.At a time only one thread is allowed to operate on Hashtable and hence it is thread safe.
3.Relatively performance is high because threads are not required to wait to operate on HashMap object.	3.Relatively performance is low because threads are required to wait to operate on Hashtable object.
4. null is allowed for both key and value.	4. null is not allowed for keys and values otherwise we will get NullPointerException.
5.Introduced in 1.2 version and it is not legacy.	5. Introduced in 1.0 version and it is legacy.

Q: How to get synchronized version of HashMap object?

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

Ex: HashMap m = new HashMap();

Map m1= Collections.synchronizedMap(m);

Here, HashMap object 'm' is non-synchronized and Map object 'm1' is synchronized.

Definition of synchronizedMap() Method: public static Map synchronizedMap(Map m){…..}

LinkedHashMap(C): LinkedHashMap is a class presents in java.util package which extends (inherits) from HashMap class and implements from Map, Cloneable and Serializable interfaces.

HashSet vs LinkedHashSet: It is exactly same as HashMap including constructors and methods except the following differences.

HashMap	LinkedHashMap
1.The underlying DS is Hash table.	1. The underlying DS is LinkedList and Hash table
2.Insertion order not preserved and it is based on hash code of keys.	2. Insertion order preserved.
3.Introduced in 1.2v	3. Introduced in 1.4v

Note: In general, we can use LinkedHashSet and LinkedHashMap to develop cache based applications where duplicates are not allowed and insertion order preserved.

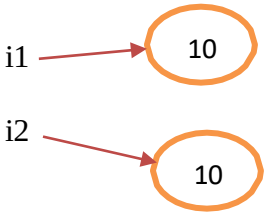
== vs equals():

In general, == operator meant for reference comparison (address comparison) whereas equals() method meant for content comparison.

Ex: Integer i1 = new Integer(10);

Integer i2 = new Integer(10);

SOP(i1 == i2); // false SOP(i1.equals(i2)); // true



IdentityHashMap(C): It is exactly same as HashMap (including methods and constructors) except the following difference.

In the case of normal HashMap JVM will use equals() method to identify duplicate keys which is meant for content comparison whereas the IdentityHashMap JVM will use == operator to identify duplicate keys which is meant for reference comparison(address comparison).

Ex:

HashMap m = new HashMap();
Integer i1 = new Integer(10);
Integer i2 = new Integer(10);
m.put(i1, "Pawan");
m.put(i2, "Kalyan");
SOP(m); // {10 = Kalyan}
i1 and i2 are duplicate keys because HashMap use equals() method i.e. i1.equals(i2) returns true.
	IdentityHashMap m = new IdentityHashMap();
	Integer i1 = new Integer(10);
	Integer i2 = new Integer(10);
	m.put(i1, "Pawan");
	m.put(i2, "Kalyan");
	SOP(m); // {10 = Pawan, 10 = Kalyan}
	i1 and i2 are not duplicate keys because IdentityHashMap use == operator i.e. i1 == i2 returns false.

WeakHashMap(C): It is exactly same as HashMap except the following difference

In the case of HashMap even though object doesn't have any reference it is not eligible for gc if it is associated with HashMap i.e. HashMap dominated garbage collector.

But in the case of WeakHashMap if object doesn't contain any references it is eligible for gc even though object associated with WeakHashMap i.e. garbage collector dominates WeakHashMap.

```
public class HashMapDemo {
 public static void main(String[] args)
 throws InterruptedException {
  HashMap m = new HashMap();
  Temp t = new Temp();
  m.put(t, "Sunny");
  System.out.println(m);
  t = null;
  System.gc();
  Thread.sleep(5000);
  System.out.println(m);
 }
}
OP: {temp = Sunny} {temp = Sunny}
```

```
public class WeakHashMapDemo {
 public static void main(String[] args)
 throws InterruptedException {
  WeakHashMap m = new WeakHashMap();
  Temp t = new Temp();
  m.put(t, "Sunny");
  System.out.println(m);
  t = null;
  System.gc();
  Thread.sleep(5000);
  System.out.println(m);
 }
}
OP: {temp = Sunny} { }
```

In the above example in the case of HashMap, 'temp' object is not eligible for gc because it is associated with HashMap. In this case o/p is {temp = Sunny} {temp = Sunny}. In the case of WeakHashMap 'temp'object is eligible for gc. In this case o/p is {temp = Sunny} { }.

8. SortedMap(I): SortedMap is the child interface of Map interface. If we want to represent a group of key value pairs according to some sorting order of keys, then we should go for SortedMap.

SortedMap interface defines the following specific methods.

(a) Object firstKey() - returns first element of the SortedMap

(b) Object lastKey() - returns last element of the SortedMap

(c) SortedSet headMap(Object key) – returns SortedMap whose elements are < key

(d) SortedSet tailMap(Object key) - returns SortedMap whose elements are >= key

(e) SortedSet subMap(Object key1, Object key2) - returns SortedMap whose elements are >= key1 and < key2

(f) Comparator comparator() – return Comparator object that describes underlying sorting technique like ascending, descending etc. If we are using default natural sorting order, then we will get **null**.

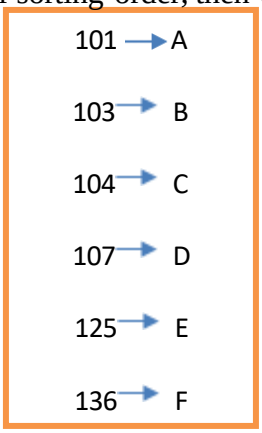
Ex: (a) firstKey() – 101 (b) lastKey() – 136

(c) headMap(107) – {101 = A,103 = B, 104 = C}

(d) tailMap(107) – {107 = D, 125 = E, 136 = F}

(e) subMap(103,125) – {103 = B, 104 = C, 107 = D}

(f) comparator() – null



9. NavigableMap(I): NavigableMap is the child interface of SortedMap. It defines several methods for navigation purposes. NavigableMap defines the following methods.

(a) floorKey(e) - returns highest element which is <= e

(b) lowerKey(e) - returns highest element which is > e

(c) ceilingKey(e) - returns lowest element which is >= e

(d) higherKey(e) - returns lowest element which is > e

(e) pollFirstEntry() – remove and return first element

(f) pollLastEntry() – remove and return last element

(g) descendingMap() – returns NavigableMap in reverse order

TreeMap(C): TreeMap is a class presents in java.util package which extends(inherits) from AbstractMap class and implements from NavigableMap, Cloneable, Serializable interfaces.

i) Properties:

- The underlying DS is Red-Black tree.
- Insertion order is not preserved and it is based on some sorting order of keys.
- Duplicate keys are not allowed, but values can be duplicated.
- If we are depending on default natural sorting order, then keys should be homogeneous and comparable otherwise we will get RE as ClassCastException.
- If we are defining our own sorting by Comparator, then keys need not be homogeneous and comparable. We can take heterogeneous non-comparable objects also.
- Whether we are depending on default natural sorting or customized sorting order there are no restrictions for values. We can take heterogeneous non-comparable objects also.
- **Null Acceptance:**

  (i) For non-empty TreeMap if we are trying to insert an entry will 'null' key then we will get RE as NullPointerException.

  (ii) For empty TreeMap as the 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 RE as NullPointerException.

  Note: The above 'nu**l**' acceptance rule applicable until 1.6v only. From 1.7v on wards 'nu**l**' is not a**l**owed for key. But for values we can use 'nu**l**' any no of times there is no restriction whether it is 1.6v or 1.7v

ii) Declaration:

```
public class TreeMap<K, V> extends AbstractMap<K, V> implements
NavigableMap<K, V>, Cloneable, Serializable
```

iii) Constructors:

(a) TreeMap t = new TreeMap();

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

(b) TreeMap t = new TreeMap(Comparator c);

Creates an empty TreeMap object where the elements will be inserted according to customized sorting order specified by comparator object.

(c) TreeMap t = new TreeMap(Map m);

Creates an equivalent TreeMap object for the given Map.

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

Creates a TreeMap object for the given SortedMap.

Hashtable(C): Hashtable is a class presents in java.util package which extends(inherits) from Dictionary class and implements from Map, Cloneable, Serializable interfaces.

i) Properties:

- The underlying DS is Hash table.
- Insertion order is not preserved and it is based on hash code of keys.
- Duplicate keys are not allowed, but values can be duplicated.
- Heterogeneous objects are allowed for both keys and values
- 'nu**l**' is not a **l**owed for both key and values otherwise we will get RE as NullPointerException.
- Every method present in Hash table is synchronized and hence Hashtable object is thread safe.
- Hashtable is the best choice if our frequent operation is search operation.

ii) Declaration:

```
public class Hashtable<K, V> extends Dictionary<K, V> implements Map<K, V>,
Cloneable, Serializable
```

iii) Constructors:

(a) Hashtable h = new Hashtable(); Creates an empty Hashtable object with default initial capacity 11 and default fill ratio 0.75.

(b) Hashtable h = new Hashtable(int initial_capacity); Creates an empty Hashtable object with specified initial capacity and default fill ratio 0.75

(c) Hashtable h = new Hashtable(int initial_capacity, float fill_ratio);

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

Creates an equivalent Hashtable object for the given Map. This constructor meant for inter conversion between map objects.

Properties(C): In our program, if anything which changes frequently (like username, password, mail id, mobile no) are not recommended to hard code in Java program because if there is any change to reflect that change recompilation, rebuild, and redeploy application are required. Even some times server restart is also required which creates a big business impact to the client.

We can overcome this problem by using properties file, such type of variable things we have to configure in the properties file. From that properties file we have to read into Java program and we can use those properties.

The main advantage of this approach is if there is a change in properties file to reflect that change just redeployment is enough which won't create any business impact to the client.

We can use Java properties object to hold properties which are coming from properties file.

In normal Map (like HashMap, TreeMap, Hashtable) key and value can be any type but in the case of properties key and value should be String type.

Constructor: Properties p = new Properties();

Methods:

(a) 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 and returns old value.

(b) String getProperty(String pname) – to get value associated with specified property.

If the specified property is not available, then it returns 'nu**l**'.

(c) Enumeration propertyNames() - returns all property names of properties object.

(d) void load(InputStream is) – to load properties from properties file into Java properties object using InputStream class.

(e) void load(Reader r) – to load properties from properties file into Java properties object using Reader class.

(f) void store(OutputStream os, String comment) – to store properties from Java properties object into properties file along with the comment.

(g) void storeToXML(OutputStream os, String comment) - to store properties from Java properties object into XML file along with the comment by using OutputStream.

(h) void storeToXML(Writer r, String comment, String encoding) - to store properties from Java properties object into XML file along with the comment and specified encoding by using Writer.

Collections(C): Collections class defines several utility methods for collection objects like sorting, searching, reversing etc.

1. Sorting Elements of List: Collections class defines the following 2 sort methods.

1. public static void sort(List l) – To sort based on default natural sorting order.

In this case List, should compulsory contain homogeneous and comparable objects otherwise we wi **l** get RE as ClassCastException. List should not contain 'nu **l**' otherwise we wi **l** get RE as NullPointerException.

2. public static void sort(List l, Comparator c) – To sort based on customized sorting order.

2. Searching Elements of List: Collections class defines the following binary search methods.

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

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

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

We have to use this method if the list is sorted according to customized sorting order.

Conclusions:

- The above search methods internally will use binary search algorithm.
- Successful search returns index and Unsuccessful search returns insertion point.
- Insertion point is the location where we can place the target element in the sorted list.
- Before calling binarySearch() method compulsory list should be sorted otherwise we will get unpredictable results.
- 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: ArrayList al = new ArrayList(); (Before Sort)

al.add("Z"); al.add("A"); al.add("M"); (After Sort)

al.add("K"); al.add("a");

Collections.sort(al);

	Z	A	M	K	a
index	0	1	2	3	4
	A	K	M	Z	a
insertion point	-1	-2	-3	-4	-5

Collections.binarySearch(al, "Z");//returns 'Z' index after sorting bcoz successful search i.e. 3

Collections.binarySearch(al, "J"); //returns 'J' insertion point after sorting because unsuccessful search i.e. -2 (J comes after A i.e. in 'K's place whose insertion point is -2 after sorting).

Note: For the list of 'n' elements, in the case of binarySearch() method

(i) Successful search result range: 0 to n-1 (ii) Unsuccessful search result range: -(n+1) to -1

(iii) Total result range: -(n+1) to n-1

3. Reversing Elements of List:

public static void reverse(List l) - to reverse elements of list

reverse() vs reverseOrder(): We can use reverse() method to reverse order of elements of list whereas we can use reverseOrder() method to get reversed comparator.

Comparator c1 = Collections.reverseOrder(Comparator c);

If 'c1' meant for descending order, then 'c' meant for ascending order and vice versa.

Arrays: Arrays class is a utility class to define several utility methods for Array objects.

1. Sorting Elements of Array:

Arrays class define the following sort methods to sort elements of primitive and object type arrays.

(i) public static void sort(Primitive[] p) – to sort according to natural sorting order.

(ii) public static void sort(Object[] a) – to sort according to natural sorting order.

(iii) public static void sort(Object[] a, Comparator c) – to sort according to customized sorting order.

Note: We can sort primitive arrays only based on default natural sorting order whereas we can sort object arrays either based on default natural sorting order or based on customized sorting order.

2. Searching the Elements of Array: Arrays class defines the following binarySearch() methods.

(i) public static int binarySearch(Prinitive[] p, Primitive target)

(ii) public static int binarySearch(Object[] a, Object target)

(iii) public static int binarySearch(Object[] a, Object target, Comparator c)

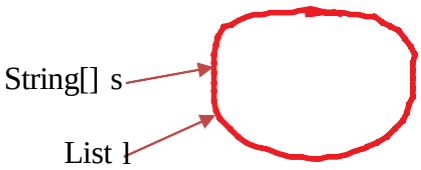
Note: All rules of Arrays class binarySearch() methods are exactly same as Collections class binarySearch() methods.

3. Conversion of Array to List: public static List asList(Object[] a)

Strictly speaking this method won't create an independent list object. For the existing array we are getting list view.

Ex: String[] s = {“A”,”Z”,”B”};

List l = Arrays.asList(s);



Conclusions:

- By using array reference if we perform any change automatically that change will be reflected to the list and by using list reference if we perform any change automatically that change will be reflected to the array.
- By using list reference we can’t perform any operation which varies the size otherwise we will get RE as UnsupportedOperationException.

  l.add(“M”); l.remove(1); //invalid RE: UnsupportedOperationException

  l.set(1,”N”); //valid

- By using list reference we are not allowed to replace with heterogeneous objects otherwise we will get RE as ArrayStoreException.

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