Collections Framework

Module 15: Collections Framework	
1.	Introduction to Collections Framework
2.	Array & its limitations, Arrays Vs Collections
3.	Collection
4.	10 key interfaces of Collections Framework a) Collection (I) b) List (I) – ArrayList, LinkedList, Vector, Stack c) Set (I) – HashSet, LinkedHashSet d) SortedSet (I) e) NavigableSet (I) – TreeSet f) Queue (I) – PriorityQueue g) Deque (I) h) Map (I) – HashMap, LinkedHashMap, Hashtable, WHM, IHM i) SortedMap (I) j) NavigableMap (I) – TreeMap
5.	Sorting (Comparable vs Comparator interface)
6.	Cursors (Enumeration, Iterator, ListIterator interface)
7.	Utility classes (Arrays, Collections)

1. Introduction to Collections Framework

- A Collections framework is a unified architecture for representing & manipulating collections, enabling collections to be manipulated independently of implementation details.
- All collections frameworks contain the following:
  a) **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In Object oriented language, interfaces generally form a hierarchy.
  b) **Implementations:** These are the concrete implementations of the collections interfaces. In essence, they're reusable data structures.
  c) **Algorithms:** These are the methods that perform useful computations, such as searching & sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic i.e. same method can be used on many different implementations of the appropriate collection interface.

Apart from the Java collections framework, the best – known examples of collections frameworks are the C++ Standard Template library (STL).

Benefits of the Java Collections Framework

1. **Reduces programming effort**: By providing useful data structures & algorithms so you don't have to write them yourself.
2. **Increases program speed & quality:** This collections framework provides high – performance, high – quality implementations of useful data structures & algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing our own data structures, you'll have more time to devote to improving program's quality & performance.
3. **Allows interoperability between unrelated**: The collection interfaces are the vernacular by which APIs pass collections back & forth. If my network administration API furnished a collection of node names & if your GUI toolkit expects a collection of column headings, our APIs will interoperated seamlessly, even though they were written independently.
4. **Reduces the effort required to learn & to use new APIs**
5. **Reduces the effort required to design & implement:** Designers & Implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
6. **Fosters software reuse:** Fosters software reuse by providing a standard interface for collections & algorithms with which to manipulate them.

2. Array

- An array is an indexed collection of fixed number of homogenous data elements.
- The main advantage of Array is we can represent multiple values by using single variable so that readability of the code will be improved.
- **Limitations of Array are**
  a) Fixed in size i.e., once we create an array, there is no chance of increasing/decreasing the size based on our requirement. Due to this, to use Array's concept, compulsory we should know the size in advance which may not possible always.
  b) Array can hold only homogenous datatype elements. (Though we can use **Object** type array)
  c) Array concept is not implemented based on some standard data Structure & hence readymade method support is not available i.e. for every requirement, we have to write the code explicitly which increases complexity of programming.
- To overcome above limitations of Array, we should go for **Collection** concept.
- **Advantages of Collection are**
  a) Collections are growable in nature i.e. based on our requirement we can increase/decrease the size.
  b) Collections can hold both homogenous & heterogeneous elements.
  c) Every collection class is implemented based on some standard data Structure hence for every requirement, readymade method support is available.

Array Vs Collection

No.	**Array**	**Collection**
1.	Fixed in size.	Growable in nature.
2.	With respect to memory, Arrays are not recommended to use.	With respect to memory, Collections are recommended to use.
3.	With respect to performance, Arrays are recommended to use.	With respect to performance, Collections are not recommended to use.
4.	Arrays can hold only homogenous datatype elements.	Collections can hold both homo & heterogeneous datatype elements.
5.	No underlying data structure hence no readymade methods are available.	Every collection class has underlying data structure hence readymade methods are available.
6.	Arrays can hold both primitive & objects.	Collections can hold only object types but not primitive.

3. Collection

- A **collection** – sometimes called a **container** – is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate & communicate aggregate data.
- Typically, they represent data items that form a natural group, such as poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).
- The collection interfaces are divided into 2 groups i.e. **java.util.Collection** & **java.util.Map**

4. 10 key interfaces of Java Collections Framework

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

Fig – ***Collection interface Hierarchy***

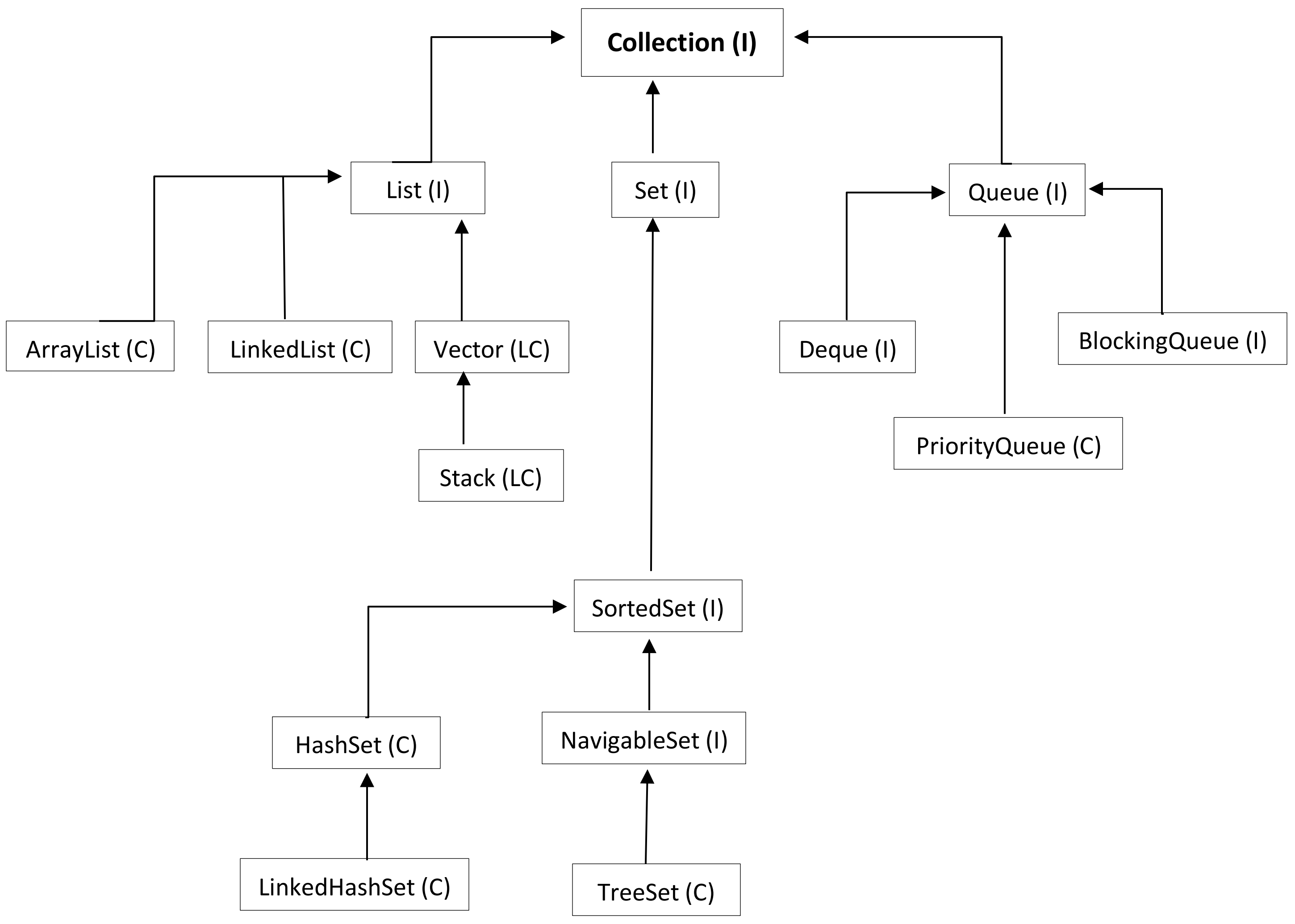
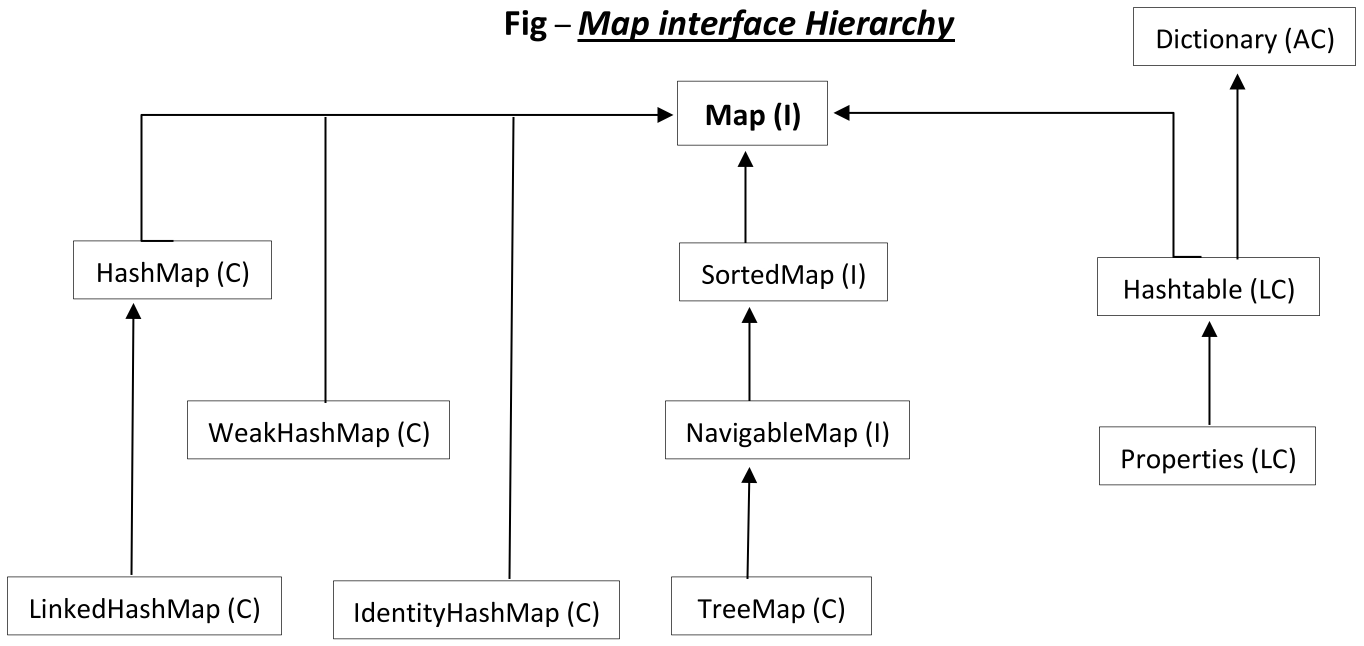


Fig – ***Map interface Hierarchy***



1. Collection Interface

```
public interface Collection<E> extends Iterable<E>
```

- **All Superinterfaces:** Iterable<E>
- **All known Subinterfaces** – BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, **Deque**<E>, **List**<E>, **NavigableSet**<E>, **Queue**<E>, **Set**<E>, **SortedSet**<E>, TransferQueue<E>

- Collection interface is considered as root interface of Collections framework.
- If we want to represent a group of individual objects as a single entity then we should go for **Collection**.
- Collection interface is typically used to pass Collections around & manipulate them where maximum generality is desired.
- By convention all general – purpose collection implementations have a constructor that takes a Collection argument. This constructor, known as a ***conversion constructor***, initializes the new collection to contain all the elements in the specified collection, whatever the given collection's sub interface or implementation type.
- Collection interface defines the most common methods which are applicable for any collection object.
- The JDK doesn't provide any direct implementations of Collection interface.

No.	Methods	Description
1.	boolean add (E e)	Adds an object to a collection.
2.	boolean addAll (Collection <? Extends E>)	Adds a group of Objects to the collection.
3.	void clear ()	Clears all objects from collection.
4.	boolean contains (Object obj)	Checks a particular object is available or not.
5.	boolean containsAll (Collection <?> col)	Checks a group of objects available or not.
6.	boolean isEmpty ()	Checks whether the collection is empty or not.
7.	boolean remove (Object obj)	Removes a particular object from a collection.
8.	boolean removeAll (Collection<?> col)	Removes a group of Objects from the collection. **Removes duplicates as well.**
9.	default boolean removeIf (Predicate<? Super E> p)	Remove all the elements of the collection that satisfy the given predicate
10.	boolean retainAll (Collection <?> col)	Retains only the elements in the collection that are present in the specified collection. Removes all the elements that are not present in the specified collection from existing collection.
11.	int size ()	No. of objects in the collection.
12.	Iterator <E> iterator ()	To get object one by ne from the collection.
13.	default Spliterator<E> spliterator ()	Creates a spliterator over the elements in the collection.
14.	default Stream<E> stream ()	Returns a sequential stream with this collection as its source.
15.	default Stream<E> parallelStream ()	Returns a possibly parallel Stream with this collection as its source.
16.	Object [] toArray ()	Converts the collection to Array.
17.	<T> T [] toArray (T [])	Returns an array containing all the elements in this collection; the runtime type of the returned array is that of the specified array.

Note: There is also a **Collections class** in **java.util** package having utility methods for Collection objects (like sorting, searching etc.)

The following are the legacy classes & interfaces in Collections Framework

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

2. List interface

public interface List<E> **extends** Collection<E>

- **All Super interfaces –** Collection <E>, Iterable <E>
- **All Known Implementing classes –** AbstractList, AbstractSequentialList, **ArrayList**, **LinkedList**, **Stack**, **Vector**, RoleList, RoleUnresolvedList, AttributeList, CopyOnWriteArrayList

- It is child interface of Collection interface.
- 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.
- We can preserve insertion order via index & we can differentiate duplicate objects by using index, hence index will play very important role in List.

No.	Methods	Description
1.	void add (int index, E element)	Inserts the specified element at the specified position in the list.
2.	boolean addAll (int index, Collection <? extends E> col)	Inserts all of the elements in the specified collection into the list at the specified position onwards.
3.	E remove (int index)	Removes the element from the specified position.
4.	E get (int index)	Returns the element at the specified position.
5.	E set (int index, E element)	Replaces the element at the specified position with the given element. The only diff b/w **add** (int index, E element) & **set** (int index, E element) is return type.
6.	int indexOf (Object obj)	Returns the index of 1st occurrence of the specified element or -1 incase element not found in list.
7.	int lastIndexOf (Object obj)	Returns the index of last occurrence of the specified element or -1 incase element not found in list.
8.	ListIterator<E> listIterator ()	Returns a list iterator over the elements in the list.
9.	Spliterator<E> spliterator ()	Creates a spliterator over the elements in the list.
10.	List<E> subList (int fromIndex, int toIndex)	Returns a view of the portion of the list b/w the specified range.
11.	default void sort (Comparator<? super E> col)	Sorts the list according to the order induced by the specified Comparator. Internally, the implementation is stable, adaptive, iterative merge sort.
12.	default void replaceAll (UnaryOperator<E> operator)	Replaces each element of the list with the result of applying the operator to that element.

List interface implementation classes

a) ArrayList**<E>** class extends **AbstractList<E>** implements **List<E>, RandomAccess,** Cloneable, Serializable
b) LinkedList**<E>** class extends **AbstractSequentialList<E>** implements **List<E>, Deque<E>,** Cloneable, Serializable
c) Vector**<E>** class extends **AbstractList<E>** implements **List<E>, RandomAccess,** Cloneable, Serializable
d) Stack**<E>** class extends **Vector<E>**

List interface implementation classes

a) ArrayList class

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

- **All implemented interfaces** – Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess
- **Direct known subclasses –** AttributeList, RoleList, RoleUnresolvedList

- The underlying data structure is **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.
- By default, ArrayList is non – synchronized but we can get synchronized version of ArrayList object by using synchronizedList () method of **Collections** class.

```
public static List synchronizedList (List l)          // synchronized version of List
public static Set synchronizedSet (Set s)             // synchronized version of Set
public static Map synchronizedMap (Map m)             // synchronized version of Map

e.g.,   ArrayList unsyncList = new ArrayList ();
        List syncList = Collections.synchronizedList(unsyncList);
```

- Usually, we can use Collections to hold & transfer objects from one location to another (container). To provide support for this requirement, every Collection class by default implements Serializable & Cloneable interface.
- **ArrayList is the best choice** if our frequent operation is retrieval operation (as it implements RandomAccess interface).
- **ArrayList is the worst choice** if our frequent operation is insertion or deletion in the middle (because of shift operation).

Constructors

a) **ArrayList l = new ArrayList ();**
- creates an empty ArrayList object with **default initial capacity of 10.**
- Once ArrayList reaches its max capacity then a new ArrayList object will be created with

New Capacity = (Current Capacity * 3/2) + 1

b) **ArrayList l = new ArrayList (int initialCapacity)** – creates an empty ArrayList Object with specified initial capacity.

c) **ArrayList l = new ArrayList (Collection c);**
- creates an equivalent ArrayList object for the given collection.
- This constructor is meant for inter conversion between Collection objects.

RandomAccess interface

- It is present in java.util package & it does not contain any methods.
- It is a **marker interface**, where required ability internally automatically provided by JVM.
- Only **ArrayList** & **Vector** classes implements **RandomAccess** interface so that any random element we can access with the same speed.

Imp. Point

- All the methods that are in **ArrayList** i/f already have their declaration in **List** i/f that's why using List reference to hold ArrayList object can use all ArrayList specific method.

```
List l = new ArrayList ();
l.add("A"); l.add(10);
System.out.println(l);                    // [A, 10]
System.out.println(l.indexOf(10));        // 1
```

```
import java.util.*;

class ListCollectionDemo {
  public static void main (String [] args) {
    ArrayList l = new ArrayList ();
    l.add("A"); l.add (10); l.add("A"); l.add(null);
    System.out.println("List: " + l);
    System.out.println("List size: " + l.size ());
    l.set (1, 11);
    System.out.println("List after set (1, 11): " + l);
    l.remove (3);
    System.out.println("List after remove (3): " + l);
    List l1 = List.of("M", "A", "N", "B", "D");
    l.addAll(l1);
    System.out.println("l1: " + l1 + "; List addAll(l1): " + l);
    System.out.println("indexOf(A): " + l.indexOf ("A"));
    System.out.println("lastIndexOf(A): " + l.lastIndexOf ("A"));
    System.out.println("containsAll(l1): " + l.containsAll (l1));
    List subList = l.subList (3, 6);
    System.out.println("List: " + l + "; subList (3,6): " + subList);
    l.removeAll (subList);
    System.out.println("removeAll(subList) List: " + l);
    l.retainAll (l1);
    System.out.println("l1: " + l1 + "; retainAll(l1) List: " + l);

    Object [] objArray = l.toArray ();
    System.out.print("List to l.toArray (): ");
    for (Object obj : objArray)
      System.out.print(obj + ", ");

  }
}
```

Output:

```
List: [A, 10, A, null]
List size: 4
List after set (1, 11): [A, 11, A, null]
List after remove (3): [A, 11, A]
l1: [M, A, N, B, D]; List addAll(l1): [A, 11, A, M, A, N, B, D]
indexOf(A): 0
lastIndexOf(A): 4
containsAll(l1): true
List: [A, 11, A, M, A, N, B, D]; subList (3,6): [M, A, N]
removeAll(subList) List: [11, B, D]
l1: [M, A, N, B, D]; retainAll(l1) List: [B, D]
List to l.toArray (): B, D
```

b) LinkedList class

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

- **All Implemented interfaces –** Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

- The underlying data structure is **Doubly Linked list** (List + Deque).
- Insertion order is preserved.
- Duplicate Objects are allowed.
- Heterogenous objects are allowed.
- null Insertion is possible.
- LinkedList does not implement RandomAccess interface.
- LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.
- LinkedList is the worst choice if our frequent operation is retrieval operation.

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.
   - This constructor is meant for inter conversion between Collection objects.

No.	Methods	Description
1.	void addFirst (E e)	Inserts the specified element at the beginning of the list.
2.	void addLast (E e)	Inserts the specified element to the end of the list.
3.	E getFirst ()	Returns the first element in the list.
4.	E getLast ()	Returns the last element in the list.
5.	E removeFirst ()	Removes & returns the first element from the list.
6.	E removeLast ()	Removes & returns the last element from the list.

ArrayList Vs LinkedList

No.	ArrayList	LinkedList
1.	ArrayList is the best choice if our frequent operation is retrieval operation because it implements RandomAccess interface.	LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.
2.	It is worst choice if our frequent operation is insertion/ deletion in the middle due to shift operation.	It is the worst choice if our frequent operation is retrieval operation.
3.	In ArrayList, the elements will be stored in consecutive memory locations & hence retrieval operation will become easy.	In LinkedList, the elements won't be stored in consecutive memory locations & hence retrieval operation will become difficult.

```
import java.util.*;

class LinkedListDemo {
   public static void main(String[] args) {
      LinkedList l = new LinkedList();
      l.add("sam"); l.add(30); l.add(null); l.add("sam");
      System.out.println("LinkedList: " + l);

      l.set(0, "will");
      System.out.println("LinkedList after set(): " + l);

      l.add(0, "smith");
      System.out.println("LinkedList after add(): " + l);

      l.removeLast();
      System.out.println("LinkedList after removeLast(): " + l);

      l.addFirst("mike");
      System.out.println("LinkedList after addFirst(): " + l);
   }
}
```

Output:

```
LinkedList:                       [sam, 30, null, sam]
LinkedList after set():           [will, 30, null, sam]
LinkedList after add():           [smith, will, 30, null, sam]
LinkedList after removeLast():    [smith, will, 30, null]
LinkedList after addFirst():      [mike, smith, will, 30, null]
```

c) Vector class

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

- **All implemented interfaces –** Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess
- **Direct known subclasses –** Stack

- Everything is same as ArrayList<E> except Vector objects are synchronized (Thread – safe)
- The underlying data structure is Resizable array or Growable array.
- Insertion order is preserved.
- Duplicates are allowed.
- Heterogenous objects are allowed.
- null insertion is possible.
- It implements RandomAccess interface.
- **Every method present in the vector is synchronized & hence vector object is thread – safe.**

Constructors

a) **Vector v = new Vector ()** – creates an empty vector object with initial capacity of 10.
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)** – creates an empty vector object with specified initial_capacity & how much increment in size of vector is specified by incremental_Capacity.
d) **Vector v = new Vector (Collection c);**
   - creates an equivalent vector object for the given collection.
   - This constructor is meant for inter conversion between Collection objects.

No.	Methods	Description
1.	synchronized void addElement (E e)	Adds the specified element at the end of the vector.
2.	synchronized void removeElement (Object obj)	Removes the first occurrence of the argument from the vector.
3.	synchronized void removeElementAt (int index)	Deletes the component at the specified index & each component in the vector will be shifted backward to one index smaller.
4.	synchronized void removeAllElements ()	Removes all components from the vector & sets its size to zero.
5.	synchronized E elementAt (int index)	Returns the component at the specified index.
6.	synchronized E firstElement ()	Returns the 1st element in the vector
7.	synchronized E lastElement ()	Returns the last element in the vector
8.	synchronized int size ()	Returns the no. of components in the vector
9.	synchronized int capacity ()	Returns the current capacity of the vector
10.	Enumeration<E> elements ()	Returns the enumeration of the components of the vector

ArrayList Vs Vector

No.	ArrayList	Vector
1.	Every method present in the ArrayList is non synchronized.	Every method present in the vector is synchronized.
2.	At a time, multiple threads are allowed to operate on ArrayList object & hence it is not thread – safe.	At a time, only on thread is allowed to operate on vector object & hence it is thread – safe.
3.	Relatively performance is high because threads are not required to wait to operate on ArrayList object.	Relatively performance is low because threads are required to wait to operate on Vector object.
4.	Introduced in 1.2 V & it is non – legacy.	It is introduced in 1.0 V & it is legacy.

```
import java.util.*;

class HelloWorld {
   public static void main(String[] args) {
      Vector v = new Vector();
      System.out.println("Vector Capacity: " + v.size());

      for(int i = 0; i < 10; i++)
         v.addElement(i);

      System.out.println("Vector: " + v + "; Vector Capacity: " + v.capacity());
      v.addElement("A");
      System.out.println("Vector after addElement(): " + v + ";\nVector Capacity: " + v.capacity());
      System.out.println("Vector size: " + v.size());
   }
}
```

```
Vector Capacity: 0
Vector: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]; Vector Capacity: 10
Vector after addElement(): [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A];
Vector Capacity: 20
Vector size: 11
```

d) Stack class

```
public class Stack<E> extends Vector<E>
```

- **All implemented interfaces** – Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess
- It is the child class of Vector.
- It is a specially designed class for **Last In First Out (LIFO)** order.

Constructors

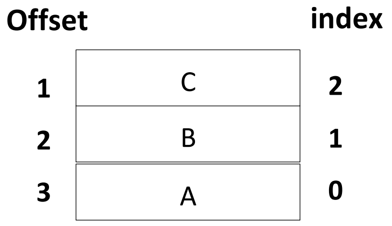
a) Stack s = new Stack () – creates an empty stack object.

No.	Methods	Description
1.	E push (E item)	Pushes an item onto the top of the stack.
2.	synchronized E pop ()	Removes the object at the top of the stack & returns that object as the value.
3.	synchronized E peak ()	Looks at the object at the top of this stack without removing it from the stack.
4.	boolean empty ()	Tests if the stack is empty
5.	synchronized int search ()	Returns the 1 – based position where an object is on this stack.

```
class StackDemo {
  public static void main(String[] args) {
    Stack s = new Stack();
    s.push("A"); s.push("B"); s.push("C");
    System.out.println("Stack: " + s);
    System.out.println("offset(A): " + s.search("A"));
    System.out.println("offset(Z): " + s.search("Z"));
  }
}
```

Output:

```
Stack:      [A, B, C]
offset(A):  3
offset(Z):  -1
```



3. Set interface

```
public interface Set <E> extends Collection<E>
```

- **All Superinterfaces**: Collection<E>, Iterable<E>
- **All known Subinterfaces**: NavigableSet<E>, SortedSet<E>
- **All known Implementing classes**: AbstractSet, **HashSet**, **LinkedHashSet**, **TreeSet**, ConcurrentHashMap.KeySetView, ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, JobStateReasons

- It is the child interface of Collection interface.
- If we want to represent a group of individual objects as a single entity **where duplicates are not allowed** & **insertion order not required to be preserved** then we should go for Set interface.
- Set interface does not contain any new method & we have to use only Collection interface method.

List Vs Set

No.	List	Set
1.	Duplicates are allowed.	Duplicates are not allowed.
2.	Insertion order preserved.	Insertion order not preserved.

4. SortedSet interface

```
public interface SortedSet <E> extends Set<E>
```

- **All Superinterfaces** – Collection<E>, Iterable<E>, Set<E>
- **All known Subinterfaces** – NavigableSet<E>
- **All known implementing classes** – TreeSet, ConcurrentSkipListSet

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

No.	Method	Description
1.	E first ()	Returns the first (lowest) element currently in the set
2.	E last ()	Returns the last (highest) element currently in the set
3.	SortedSet<E> headSet (E toElement)	Returns a view of the portion of the set whose elements are strictly less than toElement.
4.	SortedSet<E> tailSet (E fromElement)	Returns a view of the portion of the set whose elements are greater than or equal to fromElement.
5.	SortedSet<E> subSet (E fromElement, E toElement)	Returns a view of the portion of the set whose elements range from fromElement inclusive, to toElement exclusive.
6.	Comparator<? super E> comparator ()	Returns the comparator used to order the elements in the set or null if the set uses the Comparable natural ordering of its elements.

5. NavigableSet interface

```
public interface NavigableSet<E> extends SortedSet<E>
```

- **All Superinterfaces** – Collection<E>, Iterable<E>, Set<E>, SortedSet<E>
- **All known implementing classes** – **TreeSet**, ConcurrentSkipListSet

- It is the child interface of SortedSet interface.
- It contains several methods for Navigation purposes.

No.	Method	Description
1.	E ceiling (E e)	Returns the least element in the set greater than or equal to the given element or null if there is no such element.
2.	E floor (E e)	Returns the greatest element in the set less than or equal to the given element or null if there is no such element.
3.	E higher (E e)	Returns the least element in the set strictly greater than the given element, or null if there is no such element.
4.	E lower (E e)	Returns the greatest element in the set strictly less than the given element, or null if there is no such element.
5.	E pollFirst ()	Retrieves & removes the first (lowest) element, or returns null if this set is empty.
6.	E pollLast ()	Retrieve & removes the last (highest) element, or returns null if this set is empty.
7.	NavigableSet<E> descendingSet ()	Returns a reverse order view of the elements contained in this set.
8.	Iterator<E> descendingIterator ()	Returns an iterator over the elements in this set, in descending order.

```
import java.util.*;

class NavigableSetDemo {
   public static void main (String [] args) {
      TreeSet<Integer> t = new TreeSet<> ();
      t.add(1000); t.add(2000); t.add(3000); t.add(4000); t.add(5000);
      System.out.println("TreeSet: " + t);                              // [1000, 2000, 3000, 4000, 5000]
      System.out.println("Ceiling (2000): " + t.ceiling(2000));           //  2000
      System.out.println("Higher (2000): " + t.higher(2000));             //  3000
      System.out.println("Floor (3000): " + t.floor(3000));               //  3000
      System.out.println("Lower (3000): " + t.lower(3000));               //   2000
      System.out.println("PollFirst: " + t.pollFirst());                  //   1000
      System.out.println("PollLast: " + t.pollLast());                    //   5000
      System.out.println("DescendingSet: " + t.descendingSet());   //   [4000, 3000, 2000]
      System.out.println("TreeSet after operations: " + t);            //   [2000, 3000, 4000]
   }
}
```

Set Implemented classes

a) HashSet
b) LinkedHashSet
c) TreeSet

TreeSet Vs HashSet Vs LinkedHashSet

No.	Property	TreeSet	HashSet	LinkedHashSet
1.	Underlying Data Structure	Balanced Tree	Hashtable	Hashtable + LinkedList
2.	Duplicate objects	Not allowed	Not allowed	Not allowed
3.	Insertion order	Not Preserved	Not Preserved	Preserved
4.	Sorting order	Applicable	Not applicable	Not applicable
5.	Heterogeneous object	Not allowed (default)	Allowed	Allowed
6.	Null Acceptance	Not allowed	Not allowed	Not allowed

a) HashSet

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

- **All implemented Interfaces** – Serializable, Cloneable, Iterable<E>, Collection<E>, Set<E>
- **Direct known subclasses** – **LinkedHashSet**, JobStateReasons

- The underlying data structure is Hashtable (actually a HashMap instance).
- Duplicate objects are not allowed & if we're trying to insert duplicates, we won't get any compile time error or runtime exception, add () method simply returns false.
- Insertion order is not preserved & it is based on HashCode of objects.
- null Insertion is not possible.
- Heterogeneous objects are allowed.
- HashSet is best choice if our frequency operation is Search operation. This class offers constant time performance for the basic operations (add, remove, contains & size).

➢ By default, HashSet is non – synchronized but we can get synchronized version of **HashSet** object by using **synchronizedSet ()** method of Collections class.

```
public static Set synchronizedSet (Set s)          // synchronized version of Set
e.g.,   Set s = Collections.synchronizedSet (new HashSet ());
```

Constructors

a) **HashSet h = new HashSet ()** – creates a new, empty set with default initial capacity 16 & load factor or fill ratio of 0.75.

b) **HashSet h = new HashSet (int initial_Capacity)** – creates a new, empty set with the specified initial capacity & default load factor of 0.75.

c) **HashSet h = new HashSet (int initial_Capacity, float fillRatio)** – creates a new, empty set with the specified initial capacity & the specified load factor.

d) **HashSet h = new HashSet (Collection c)**
- creates an equivalent HashSet for the given collection.
- This constructor is meant for interconversion between Collection object.

Fill Ratio/ Load factor: After filling, with how much ratio a new HashSet object will be created, this ratio is called Fill Ratio.

e.g. Fill Ratio 0.75 means after filling 75% of a HashSet object, a new HashSet object will be created automatically.

b) LinkedHashSet

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

➢ **All implemented interfaces** – Serializable, Cloneable, Iterable<E>, Collection<E>, Set<E>

➢ It is child class of HashSet.
➢ The underlying data structure is a combination of Hash table & linked list.
➢ This implementation differs from HashSet in that it maintains a doubly – linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion – order).
➢ It is exactly same as HashSet including constructors & methods except the following differences:

No.	HashSet	LinkedHashSet
1.	The underlying data structure is Hashtable.	The underlying data structure is a combination of Hashtable & LinkedList.
2.	Insertion order not preserved.	Insertion order preserved.
3.	Introduced in 1.2 version.	Introduced in 1.4 version.

➢ In general, we can use LinkedHashSet to develop cache-based applications where duplicates are not allowed & insertion order is preserved.

c) TreeSet

```
public abstract class AbstractSet<E> extends AbstractCollection<E>
                                            implements Set<E>

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

- **All Implemented interfaces:** Serializable, Cloneable, Iterable<E>, Collection<E>, **NavigableSet**<E>, **Set**<E>, **SortedSet**<E>

- TreeSet uses a Tree for storage & is basically an implementation of a self – balancing binary search tree like a Red – Black tree.
- Therefore, operation like add, remove & search take O(log N) time.
- Since a set doesn't retain insertion order, the navigable set interface provides the implementation to navigate through the Set.
- The elements are ordered using their natural ordering or by a Comparator provided at set creation time, depending on which constructor is used.
- Operations like printing N elements in the sorted order takes O(N) time.
- Duplicate Objects are not allowed.
- Insertion order not preserved.
- null Insertion is not possible.
- TreeSet implements Serializable & Cloneable interface but not RandomAccess interface.
- Heterogenous objects are not allowed otherwise we will get Runtime exception saying ClassCastException because Comparation is done between same type of objects.
- All objects will be inserted based on some sorting order. It may be default natural sorting order or customized sorting order.
- **TreeSet** serves as an excellent choice for storing large amounts of sorted information which are supposed to be accessed quickly because of its faster access & retrieval time.

- By default, TreeSet is non – synchronized but we can get synchronized version of **TreeSet** object by using **synchronizedSet ()** method of Collections class.

```
public static Set synchronizedSet (Set s)          // synchronized version of Set
e.g.,   Set s = Collections.synchronizedSet (new TreeSet ());
```

- Methods in TreeSet<E> is same as in Collection interface & NavigableSet interface.

Constructors

a) TreeSet t = new TreeSet ()
- creates an empty TreeSet object where the elements will be inserted according to default natural sorting order.
- All elements inserted into the set must implement the Comparable interface & all such elements must be mutually comparable.

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 a new TreeSet containing the elements in the specified collection, sorted according to the natural ordering of its elements.

d) TreeSet t = new TreeSet (SortedSet<E> s) – creates a new TreeSet containing the same elements & using the same ordering as the specified sorted set.

Note: The ordering maintained by a set (whether or not an explicit comparator is provided) must be *consistent with equals* if it is to correctly implement the Set interface. (See Comparable or Comparator for a precise definition of *consistent with equals*.) This is so because the Set interface is defined in terms of the equals operation, but a TreeSet instance performs all element comparisons using its **compareTo** (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal. The behaviour of a set *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Set interface.

6. Queue interface

```
public interface Queue<E> extends Collection<E>
```

- **All Superinterfaces** – Collection<E>, Iterable<E>
- **All known Subinterfaces** – BlockingDeque<E>, BlockingQueue<E>, **Deque**<E>, TransferQueue<E>
- **All known Implementing classes** – **LinkedList**, AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, **PriorityQueue**, SynchronousQueue

- It is the child interface of Collection interface.
- 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 also (PriorityQueue)
- From 1.5 version, LinkedList class also implements Queue interface. LinkedList based implementation of Queue always follow FIFO.
- **Summary of Queue methods:**

	Throws exception	***Returns special value***
Insert	add (e)	offer (e)
Remove	remove ()	poll ()
Examine	element ()	peek ()

No.	**Methods**	**Description**
1.	boolean add (E e)	Inserts the specified element into the queue without violating capacity restrictions & returns true & throw **IlegalStateException** if no space is available.
2.	boolean offer (E e)	Inserts the specified element into the queue without violating capacity restrictions
3.	E element ()	Returns head element of the queue. If queue is empty, then this method raises RE: NoSuchElementException.
4.	E peek ()	Returns head element of the queue. If queue is empty, then this method returns null.
5.	E remove ()	Removes & returns head element of the queue. If queue is empty, then this method raises RE: NoSuchElementException.
6.	E poll ()	Removes & returns head element of the queue. If queue is empty, this method returns null.

Priority Queue

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

public abstract class AbstractQueue<E> extends AbstractCollection<E>
                                       implements Queue<E>
```

- **All Superinterfaces** – Serializable, Iterable<E>, Collection<E>, Queue<E>
- 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 i.e., defined by Comparator.
- Insertion order is not preserved & it is based on some priority.
- Duplicated objects are not allowed.
- If we're depending on default natural sorting, then compulsory the objects should be homogeneous & Comparable but if we're depending on Comparator sorting order then objects need not to be homogenous & Comparable.
- This implementation provides O(log(n)) time for the enqueuing and dequeuing methods (offer, poll, remove and add); linear time for the remove (Object) and contains (Object) methods; and constant time for the retrieval methods (peek, element, and size).
- Some platforms won't provide proper support for Thread priorities & PriorityQueues

Constructors

a) **PriorityQueue pq = new PriorityQueue ()** – creates a PriorityQueue with default initial capacity as 11 & all objects will be inserted in default natural sorting order.
b) PriorityQueue pq = new PriorityQueue (int initialCapacity)
c) PriorityQueue pq = new PriorityQueue (int initialCapacity, Comparator c)
d) PriorityQueue pq = new PriorityQueue (Comparator<? Extends E> c)
e) PriorityQueue pq = new PriorityQueue (Collection<? extends E> c)
f) PriorityQueue pq = new PriorityQueue (SortedSet<? Extends E> c)
g) PriorityQueue pq = new PriorityQueue (PriorityQueue<? Extends E> c)

Note: Some platforms won't provide proper support for Thread priorities & PriorityQueues.

```
import java.util.*;

class PriorityQueueDemo {
   public static void main (String [] args) {
      PriorityQueue q = new PriorityQueue ();
      System.out.println(q.peek());
      //System.out.println(q.element());
      for (int i = 0; i < 10; i++)
         q.offer(i);
      System.out.println("PriorityQueue: " + q);
      System.out.println(q.poll());
      System.out.println("PriorityQueue: " + q);
   }
}
```

Output:

```
null
PriorityQueue: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0
PriorityQueue: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
class PriorityQueueDemo {
   public static void main(String[] args) {
      PriorityQueue q = new PriorityQueue(15, new MyComparator());
      q.offer("A"); q.offer("Z"); q.offer("L"); q.offer("B");
      System.out.println("PriorityQueue: " + q);                    // [Z, L, B, A]
   }
}

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

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

7. Deque interface

```
public interface Deque<E> extends Queue<E>
```

- **All Superinterfaces** – Collection<E>, Iterable<E>, Queue<E>
- **All known Subinterfaces** – BlockingDeque<E>
- **All known Implementing classes** – ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque, LinkedList

- A linear collection that supports element insertion & removal at both ends. The name deque is short for "double ended queue".

- **Imp. Point:** When a deque is **used as a queue**, FIFO (First – In – First – Out) behaviour result i.e., Elements are added at the end of the deque & removed from the beginning.
- **Imp. Point:** When a deque is **used as a stack**, LIFO (Last – In – First – Out) behaviour result i.e., Elements are pushed & popped from the beginning of the deque. **Deque should be used as stack instead of legacy Stack class.**

- **Summary of Deque methods:**

	First Element (Head)		**Last Element (Tail)**	
	Throws exception	***Special value***	***Throws exception***	***Special value***
Insert	addFirst (e)	offerFirst (e)	addLast (e)	offerLast (e)
Remove	removeFirst ()	pollFirst ()	removeLast ()	pollLast ()
Examine	getFirst ()	peekFirst ()	getLast ()	peekLast ()

Queue Method	***Equivalent Deque Method***
add (e)	addLast (e)
offer (e)	offerLast (e)
remove ()	removeFirst ()
poll ()	pollFirst ()
element ()	getFirst ()
peek ()	peekFirst ()

Comparison of Queue & Deque methods

Stack Method	***Equivalent Deque Method***
push (e)	addFirst (e)
pop ()	removeFirst (e)
peek ()	peekFirst ()

Comparison of Stack & Deque methods

- A Deque that additionally supports Blocking operations that wait for the deque to become non – empty when retrieving an element, & wait for space to become available in the deque when storing an element.

8. Map interface

public interface Map<K, V>

- **All known Subinterfaces** – Bindings, **ConcurrentMap**<K, V>, ConcurrentNavigableMap<K, V>, LogicalMessageContext, MessageContext, **NavigableMap**<K, V>, SOAPMessageContext, **SortedMap**<K, V>

- **All known implementing classes** – **AbstractMap**, Attributes, AuthProvider, **ConcurrentHashMap**, CurrentSkipListMap, EnumMap, **HashMap**, **Hashtable**, **IdentityHashMap**, **LinkedHashMap**, PrinterStateReasons, **Properties**, Provider, RenderingHints, SimpleBindings, TabularDataSupport, **TreeMap**, UIDefaults, **WeakHashMap**

- Map is not child interface of Collection interface.
- If we want to represent a group of objects as key – value pairs then we should go for Map interface.

e.g.

Key	**Value**
101	Sam
102	Ravi

- Both key & value are objects only & duplicate keys are not allowed but values can be duplicated.
- Each key – value pair is called **Entry**; hence ***Map is collection of Entry objects.***
- The Map interface provides three collection views, which allows a map's content to be viewed as a set of keys, collection of values, or set of key – value mappings.
- The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the **TreeMap** class maintains order, others like the **HashMap** class don't.
- Nested class (Inner interface inside Map<K, V>)

  public static interface Map.Entry<K, V>

  - Methods inside Map.Entry interface
    1. **K getKey () –** returns the key corresponding to this entry.
    2. **V getValue () –** returns the value corresponding to this entry.
    3. **V setValue () –** replaces the value corresponding to this entry with the specified value.

No.	**Methods**	**Description**
1.	V put (K key, V value)	Associates the specified value with the specified key in the map.
2.	void putAll (Map<? extends K, ? extends V> m)	Copies all of the mapping from the specified map to this map.
3.	V get (Object key)	Returns the values to which the specified key is mapped, or null if this map contains no mapping for the key.
4.	V remove (Object key)	Removes the mapping for a key from this map if it is present
5.	boolean containsKey (Object key)	Returns true if this map contains a mapping for specified key.
6.	boolean containsValue (Object value)	Returns true if this map maps one or more keys to the specified value.
7.	boolean isEmpty ()	Returns true if this map contains no key – value mappings.
8.	int size ()	Returns the no. of key – value mappings in the map.
9.	void clear ()	Removes all of the mappings from the map
10.	Set<Map.Entry<K, V>> entrySet ()	Returns a Set view of the mappings contained in this map.
11.	Set<K> keySet ()	Returns a Set view of the keys contained in this map.
12.	Collection<V> values ()	Returns a Collection view of the values contained in the map.
13.	default V compute (K key, BiFunction<? super K, ? super V, ? extends V> remapping function)	Attempts to compute a mapping for the specified key & its current mapped value (or null if there is no current mapping).
14.	default V computeIfAbsent (K key, Function<? super K, ? extends V> remapping function)	If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function & enters it into this map unless null.
15.	default V computeIfPresent (K key, BiFunction<? super K, ? super V, ? extends V> remapping function)	If the value of the specified key is present & non – null, attempts to compute a new mapping given the key & its current mapped value.

16.	default V merge (K key, V value, BiFunction<? super K, ? super V, ? extends V> remapping function)	If the specified key is not already associated with a value or is associated with null, associates it with the given non – null value.
17.	default void replaceAll (BiFunction<? super K, ? super V, ? extends V> function	Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

Entry interface

- Without existing Map object, there is no chance of existing Entry object hence Entry interface is defined inside Map interface.

```
interface Entry<K, V> {
   K getKey ();
   V getValue ();
   V setValue (V value);
}
```

9. SortedMap interface

```
public interface SortedMap<K, V> extends Map<K, V>
```

- All Superinterfaces – Map<K, V>
- All known Subinterfaces – ConcurrentNavigableMap<K, V>, NavigableMap<K, V>
- All known Implementing classes – ConcurrentSkipListMap, TreeMap

- It is the child interface of Map interface.
- If we want to represent a group of Object as a group of key – value pairs according to some sorting order of keys (using Comparator), then we should go for SortedMap.
- In SortedMap, the sorting should be based on key but not based on value.

No.	**Methods**	**Description**
1.	K firstKey ()	Returns the first (lowest) key currently in this map.
2.	K lastKey ()	Returns the last (highest) key currently in this
3.	SortedMap<K, V> headMap (K toKey)	Returns a view of the portion of this map whose keys are strictly less than toKey.
4.	SortedMap<K, V> tailMap (K fromKey)	Returns a view of the portion of this map where keys are greater than or equal to fromKey.
5.	SortedMap<K, V> submap (K fromKey, K toKey)	Returns a view of the portion of this map whose key range from fromKey(inclusive) to toKey(exclusive)
6.	Comparator<? super K> comparator ()	Returns the comparator used to order the keys in this map or null if this map uses the natural ordering of its keys.

10. NavigableMap interface

- It is the child interface of Sorted Map.
- It defines several methods for Navigation purpose.
- A SortedMap extended with navigation methods returning the closet matches for a given search targets.

No.	Methods	Description
1.	K ceilingKey (K key)	Returns the least key greater than or equal to the given key, or null if there is no such key.
2.	K higherKey (K key)	Returns the least key strictly greater than the given key, or null if there is no such key.
3.	K floorKey (K key)	Returns the greatest key less than or equal to the given key, or null if there is no such key.
4.	K lowerKey (K key)	Returns the greatest key strictly less than the given key, or null if there is no such key.
5.	Map.Entry<K, V> pollFirstEntry ()	Removes & returns a key – value mapping associated with the least key in this map, or null if the map is empty.
6.	Map.Entry<K, V> pollLastEntry ()	Removes & returns a key – value mapping associated with the greatest key in this map, or null if the map is empty.
7.	NavigableMap<K, V> descendingMap ()	Returns a reverse order view of the mapping

```
import java.util.*;

class NavigableMapDemo {
   public static void main (String [] args) {
      TreeMap<String, String> t = new TreeMap<> ();
      t.put("b", "banana"); t.put("c", "cat");
      t.put("a", "apple");  t.put("d", "dog");
      t.put("f", "fan");    t.put("g", "gun");
      System.out.println("TreeMap: " + t);
      System.out.println("ceilingKey(c): " + t.ceilingKey("c"));
      System.out.println("higherKey(e): " + t.higherKey("e"));
      System.out.println("floorKey(e): " + t.floorKey("e"));
      System.out.println("lowerKey(e): " + t.lowerKey("e"));
      System.out.println("pollFirstEntry (): " + t.pollFirstEntry());
      System.out.println("pollLastEntry (): " + t.pollLastEntry());
      System.out.println("descendingMap: " + t.descendingMap());
      System.out.println("TreeMap after operations: " + t);

   }
}
```

Output:

```
TreeMap: {a=apple, b=banana, c=cat, d=dog, f=fan, g=gun}
ceilingKey(c): c
higherKey(e): f
floorKey(e): d
lowerKey(e): d
pollFirstEntry (): a=apple
pollLastEntry (): g=gun
descendingMap: {f=fan, d=dog, c=cat, b=banana}
TreeMap after operations: {b=banana, c=cat, d=dog, f=fan}
```

Map interface implemented classes

a) HashMap
b) LinkedHashMap
c) TreeMap
d) WeakHashMap
e) IdentityHashMap
f) Hashtable

a) HashMap

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

- **Direct known subclasses** – LinkedHashMap, PrinterStateReasons

- HashMap is called HashMap because it uses a technique called Hashing. Hashing is a technique of converting a large String into small String that represents the same string. A shorter value helps in indexing & faster searches.
- HashSet also uses HashMap internally.
- The underlying data structure is Hashtable.
- The HashMap class is roughly like Hashtable, except that HashMap is unsynchronized & permits null key (only once) & null value (any no. of times).
- Insertion order is not preserved & it's based on Hashcode of keys.
- Duplicate keys are not allowed but values can be duplicated.
- This implementation provides O (1) or constant time for get & put operations, assuming the hash function disperses the elements properly among the buckets.

- Iteration over HashMap requires time proportional to the "capacity" of the HashMap instance (the no. of buckets) plus its size (the no. of key – value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.
- Methods same as Map interface methods.

- By default, HashMap is non – synchronized but we can get synchronized version of HashMap by using synchronizedMap () method of Collections classes.

```
public static Map synchronizedMap (Map m)       // synchronized version of Map
e.g.,   Map m = Collections.synchronizedMap (new HashMap ());
```

Constructors

a) **HashMap m = new HashMap** () – constructs an empty HashMap with the default initial capacity (16) & the default load factor (0.75)
b) **HashMap m = new HashMap** (int initialCapacity) – constructs an empty HashMap with the specified initial capacity & the default load factor (0.75)
c) **HashMap m = new HashMap** (int initialCapacity, float LoadFactor);
d) **HashMap m = new HashMap** (Map<? extends K, ? extends V> m);

- Performance of HashMap depends on 2 parameters
  1. **Initial capacity** – It is the capacity of HashMap at the time of its creation (i.e., no. of buckets a HashMap can hold when the HashMap is instantiated). Initial default capacity is 2^4 = 16 i.e., it can hold 16 key – value pairs.
  2. **Load Factor** – It is the percent value of the capacity after which the capacity of HashMap is to be increased. (i.e., the percentage fill of buckets after which Rehashing takes place). In Java, default load factor is 0.75 i.e., the rehashing takes place after filling 75% of the capacity.
  3. **Threshold** – Product of Load Factor & initial Capacity. In java, it's by default (0.75 * 16 = 12) i.e., rehashing takes place after filling 75% of the capacity.
  4. **Rehashing** – It is the process of doubling the capacity of HashMap after it reaches its threshold. In java, HashMap continues to rehash in the following sequence: 2^4, 2^5, 2^6, …. so on.

- As a general rule, the default load factor (0.75) offers a good tradeoff b/w time & space cost. Higher values decrease the space overhead but increase the lookup cost. The expected no. of entries in the map & its load factor should be taken into account when setting its initial capacity, so as to minimize the no. of rehash operations.
- If initial capacity is greater than the maximum no. of entries divided by the load factor, no rehash operations will ever occur.
- If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.
- **Note**: Using many keys with same hashcode () is a sure way to slow down performance of any Hashtable. To ameliorate impact, when keys are comparable, this class may use comparison order among keys to help break ties.

Internal Structure of HashMap

- **Internally HashMap contains an array of Node & a node is represented as a class that contains 4 fields:**
  int **hash**, K **key**, V **value**, Node **next**: reference to next node.

Node<K,V>
int hash K key V value Node<K,V> next

- It can be seen that the node is containing a reference to its own object. So, it's a **linked list.**
- **Major update:** From Java 8 onwards, Java has started using Self Balancing BST instead of a linked list for chaining. The advantage of self – balancing BST is, we can get the worst case (when every key maps to same slot) search time as O (log N).
- Before understanding the internal working of HashMap, we must be aware of hashCode () & equals () method.
  1. **equals ()**: It checks the equality of 2 objects. It compares the key, whether they are equal or not. If we override the equals () method, then it's mandatory to override the hashCode () method.
  2. **hashCode ():** It returns the memory reference of the object in integer form. The value received from this method is used as bucket number. The bucket no. is the address of the element inside the map. HashCode of null key is 0.
  3. **Buckets:** Array of the node is called bucket. Each node has a data structure like a LinkedList (self-balancing BST from java 8). More than one node can share the same bucket.

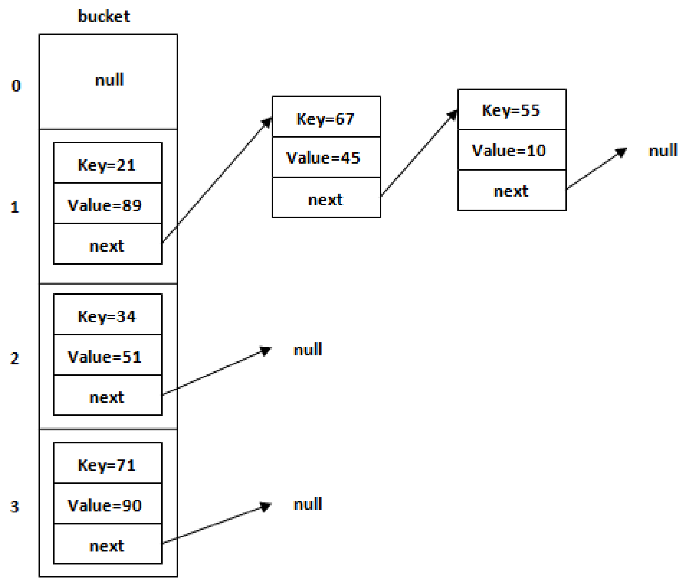


Figure: Allocation of nodes in Bucket

Let's take example & understand HashMap internal functioning

1. Insert key, value pair in HashMap

- We use put () method to insert the key & the value pair in the HashMap. The default size is 16 (0 to 15)
- e.g.,

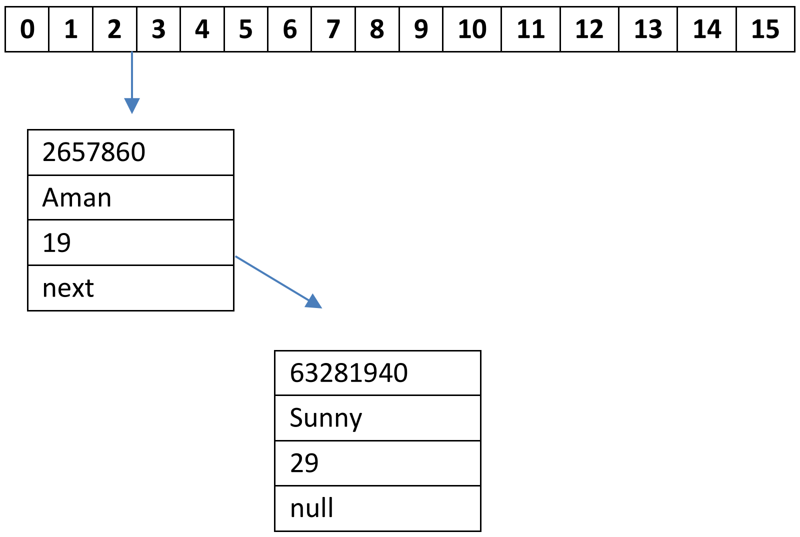
```
HashMap<String, Integer> map = new HashMap<> ();
map.put ("Aman", 19);   // Suppose hashcode (Aman) = 2657860
map.put ("Sunny", 29); // Suppose hashcode (Sunny) = 63281940
map.put ("Ritesh", 39);
```

- Calculating the index using the formula: (n is the size of array)

  Index = hashcode(key) & (n – 1)

  e.g.,
  Index (Aman) = hashcode (Aman) * (16 – 1) = 2657860 * 15 = 4
  Index (Sunny) = hashcode (Sunny) * (16 – 1) = 63281940 * 15 = 4

- This is the case of **Hash Collision** i.e.; the calculated index value is same for 2 or more keys.
  In this case, equals () method checks that keys are equal or not
  - If keys are same, replace the value with the current value.
  - Otherwise, connect this node object to the existing node object through the LinkedList. Hence both the keys will be stored at index 4.



2. get () method in HashMap

- When get (K key) method is called, it calculates the hashcode of the key then using Index formula, index for the key will be generated.
- For e.g., suppose index is 4 for key Aman, then get () method search for the index value 4.
  - If both keys are equal, then it returns the value
  - Else check for the next element in the node if it exists.

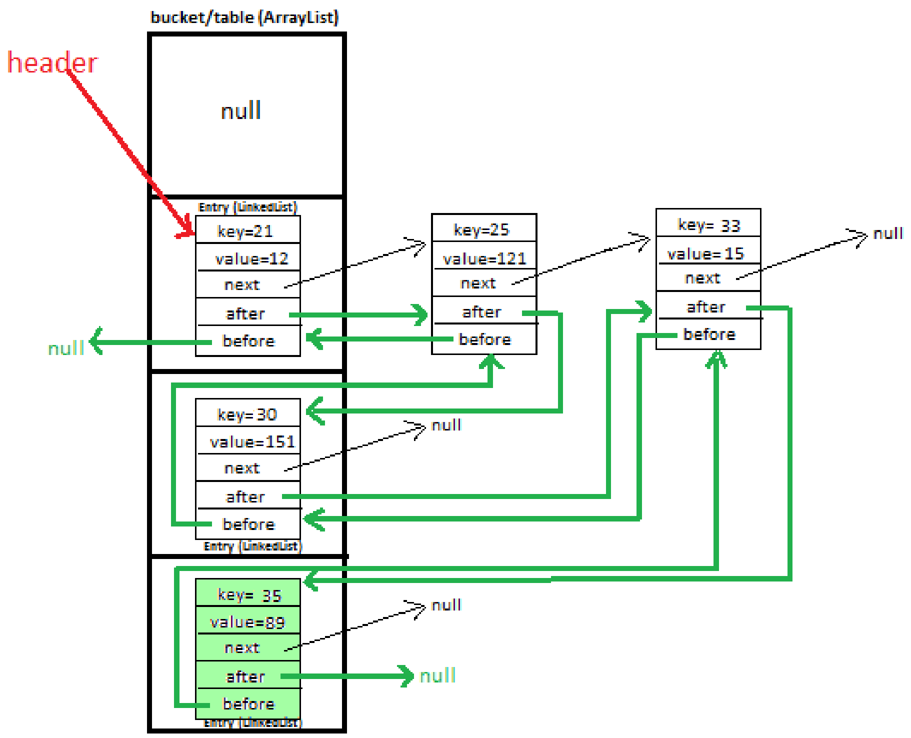
b) LinkedHashMap

```
public class LinkedHashMap<K, V> extends HashMap<K, V>
                                 implements Map<K, V>
```

- The LinkedHashMap class is just like HashMap with an additional feature of maintaining an order of elements inserted into it.
- Hashtable & doubly – linked list implementation of the Map interface, with predictable iteration order. This implementation differs from the HashMap in that it maintains a doubly – linked list running through all of its entries.
- This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion order)
- This implementation spares its clients from the unspecified, generally chaotic ordering provided by HashMap (and Hashtable), without incurring the increased cost associated with TreeMap. It can be used to produce a copy of a map that has same order as the original, regardless of the original map's implementation.
- Each node of the LinkedHashMap is represented as:

Hash	Key	Value	Next	Before/head	After/tail

- **Hash:** All the input keys are converted into a hash which is a shorter form of the key so that the search & insertion are faster.
- **Key:** Since this class extends HashMap, the data is stored in the form of a key – value pair. Therefore, this parameter is the key to the data.
- **Value:** For every key, there is value associated with it.
- **Next:** Since the LinkedHashMap stores the insertion order, this contains the address to the next node of the LinkedHashMap.
- **Previous:** This parameter contains the address of the previous node of the LinkedHashMap.



c) TreeMap

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

- A TreeMap is implemented using a Red – black tree, which is a type of self – balancing BST.
- This provides efficient performance for common operations such as adding, removing, & retrieving elements, with an average time complexity of O (log N)

- The TreeMap in Java is a concrete implementation of the **java.util.SortedMap** interface. It provides an ordered collection of key – value pairs, where the keys are ordered based on their natural order or a custom Comparator passed to the constructor.
-
- By default, TreeMap is non – synchronized but we can get synchronized version of TreeMap by using **synchronizedSortedMap** () method of Collections classes.

```
public static Map synchronizedMap (Map m)       // synchronized version of Map
e.g.,   Map m = Collections.synchronizedMap (new TreeMap ());
```

Constructors in TreeMap

1. TreeMap t = new TreeMap ();
2. TreeMap t = new TreeMap (Comparator<? super K> comparator);
3. TreeMap t = new TreeMap (Map<? extends K, ? extends V> m);
4. TreeMap t = new TreeMap (SortedMap< K, ? extends V> m);

d) WeakHashMap

```
public class WeakHashMap<K, V> extends AbstractMap<K, V>
                                       implements Map<K, V>
```

- It's exactly same as HashMap except the following difference:
  - In the case of HashMap, even though Object doesn't have any reference, it's not eligible for garbage collection (GC) if the object is associated with HashMap i.e., HashMap dominates Garbage Collector.
  - In the case of WeakHashMap, if object doesn't contain any references, it's eligible for Garbage collection even though object associated with WHM i.e., Garbage Collector dominates WHM.

e) IdentityHashMap

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

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

```
e.g., Integer i1 = new Integer (10);
Integer i2 = new Integer (10);
System.out.println (i1 == i2);          // false
System.out.println (i1. equals(i2));    // true
```

- It's exactly same as HashMap (including methods & constructors) except the following difference:
  - In case of Normal HashMap, JVM will use .equals() method to identify duplicate keys, which is meant for content comparison.
  - But in case of IdentityHashMap, JVM will use "==" operator to identify duplicate keys which is meant for reference comparison (address comparison)

  E.g.,

```
IdentityHashMap m = new IdentityHashMap ();
Integer i1 = new Integer (10);
Integer i2 = new Integer (10);
m.put (i1, "Shivam");
m.put (i2, "Raj");
System.out.println (m);        //        {10 = Shivam, 10 = Raj}
```

f) Hashtable

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

- This class implements a hash table, which maps keys to values.
- Insertion order not preserved
- Duplicate keys are not allowed.
- Every method present in Hashtable is synchronized & hence Hashtable object is thread – safe. If a thread – safe implementation is not needed, it is recommended to use HashMap in place of Hashtable.
- Hashtable is the best choice, if our frequent operation is Search operation.

Constructors

1. Hashtable h = new Hashtable ();
2. Hashtable h = new Hashtable (int initialCapacity);
3. Hashtable h = new Hashtable (int initialCapacity, float loadFactory);
4. Hashtable h = new Hashtable (Map<? extends K, ? extends V> t);

Dictionary

```
public abstract class Dictionary<K, V> extends Object
```

- The Dictionary class is an abstract class & can't be instantiated directly. Instead, it provides the basic operations for accessing the key – value pairs stored in the collection, which are implemented by its concrete subclass java.util.Hashtables

Properties

```
public class Properties extends Hashtable<Object, Object>
```

- In our program, if anything which changes frequently (like user_name, password, mail_id etc.) are not recommended to hard code in Java program because if there is any change, to reflect that change, recompilation, rebuild & redeploy application are required even sometime server restart also required which creates a big business impact to the client.
- We can overcome this problem by using Properties file. We can assign our frequently changing properties in Properties file & from there we can read into Java Program.
- 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.
- **Constructor in Properties class**
    - Properties p = new Properties ();

No.	Methods	Description
1.	String getProperty (String key)	Searches for the property with the specified key in this property list.
2.	Object setProperty (String key, String value)	Sets a new Property. Calls the Hashtable method put.
3.	Enumeration propertyNames ()	Returns all property names present in Properties object.
4.	void load (InputStream is)	Loads properties from Properties file into java Properties object.
5.	void store (OutputStream os, String comment)	Stores Properties from Java Properties object into Properties file.

5. Sorting (Comparable & Comparator i/f)

- If we are depending on default natural sorting order, compulsory **the objects should be Homogenous & Comparable** otherwise, we will get Runtime exception saying ClassCastException.

- An Object is said to be **Comparable** if corresponding class implements Comparable interface.
  e.g. String class & all wrapper classes already implements Comparable interface but StringBuffer class doesn't implement Comparable interface; hence we will get ClassCastException.

- Comparable interface is present in java.lang package & it contains only 1 method

  i.e.
```
public int compareTo (Object obj)
```

- Method explanation
```
obj1.compareTo (obj2)
```

  - **obj1** – The object which is to be inserted.
  - **obj2** – The object which is already inserted.
  - This method returns ***negative no. if obj1 has to come before obj2.***
  - This method returns ***positive no. if obj1 has to come after obj2.***
  - This method returns ***zero if obj1 equals to obj2 i.e. duplicate.***

- If we're depending on default natural sorting order, then while adding objects into the TreeSet, JVM will call compareTo () method.

- If we are not satisfied with default natural sorting or default natural sorting is not available then we should go for customized sorting by using Comparator interface.

- **Comparator interface** is present in java.util package & it defines 2 methods

  a) int *compare* (Object obj1, Object obj2)
  - This method returns ***negative no. if obj1 has to come before obj2.***
  - This method returns ***positive no. if obj1 has to come after obj2.***
  - This method returns ***zero if obj1 equals to obj2 i.e. duplicate.***

  b) boolean *equals* (Object obj)

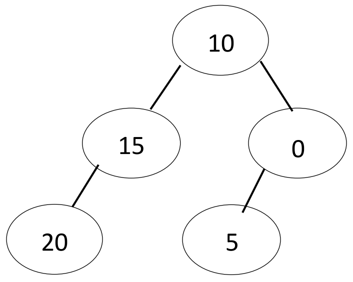
- **Imp. Point:** Whenever we're implementing **Comparator** interface**, compulsory we should provide implementation only for compare () method** & **we're not required to provide implementation for equals () method** because it is already available to our class from Object class through Inheritance.

Comparable Vs Comparator

No.	Comparable	Comparator
1.	It is meant for default natural sorting order.	It is meant for customized sorting order.
2.	Present in java.lang package.	Present in java.util package.
3.	It defines only one method i.e. compareTo ()	It defines 2 methods i.e. compare () & equals ()
4.	String & all wrapper classes implements Comparable interface.	The only implemented classes of Comparator interface are Collator, RuleBasedCollator

```
import java.util.*;

class HelloWorld {
    public static void main(String[] args) {
        TreeSet t = new TreeSet(new MyComparator());
        t.add(10); t.add(0); t.add(15); t.add(5); t.add(20); t.add(20);
        System.out.println(t);
    }
}

class MyComparator implements Comparator {
    public int compare (Object obj1, Object obj2) {
        Integer i1 = (Integer) obj1;
        Integer i2 = (Integer) obj2;

        // Way 1
         If (i1 < i2)
             return 1;                   // i1 comes after i2 (reverse of default natural sorting)
         else if (i1 > i2)
             return -1;                 // i1 comes before i2
         else
            return 0;

        // Way 2
        return i1.compareTo(i2)  or  -i2.compareTo(i1);                // Ascending order [0, 5, 10, 15, 20]

       // Way 3
       return -i1.compareTo(i2)  or  i2.compareTo(i1);               // Descending order [20, 15, 10, 5, 0]

       // Way 4
       return 1;              // Insertion order  [10, 0, 15, 5, 20, 20]

      // Way 5
      return -1;            // reverse of insertion order [20, 20, 5, 15, 0, 10]
    }
}
```

V.V.Imp. Using Way 1, let's understand what's going on



TreeSet t = new TreeSet (new MyComparator ()); // Way1
t.add (10)   // will act as root node
t.add (0)   => compare (0, 10) => return 1 (i.e., 0 will be inserted as right node in tree)
t.add (15) => compare (15, 10) => return -1 (i.e., 15 will be inserted as left node in tree)
t.add (5)   => compare (5, 10) => return 1 (i.e., 5 will be inserted as right node to 10)
(0 is already right to 10)
=> compare (5, 0)   => return -1 (i.e., 5 will be inserted as left node to 0)
t.add (20) => compare (20, 10) => return -1 (i.e., 20 will be inserted as left node to 10)
(15 is already left to 10)
=> compare (20, 15) => return -1 (i.e., 20 will be inserted as left node to 15)
t.add(20) => compare (20, 10) => return -1 (20 as left node to 10)
=> compare (20, 15) => return -1 (20 as left node to 15)
=> compare (20, 20) => return 0

6. Cursors of Java Collections Framework

- If we want to get Objects one by one from a collection, then we should go for cursor.
- There are 3 types of cursors available in Java
  a) Enumeration Interface
  b) Iterator Interface
  c) ListIterator

a) Enumeration Interface

- We can use Enumeration to get objects one by one from legacy collection object.
- We can create Enumeration object by using element () method of Vector class.

- **Methods**
  1. boolean **hasMoreElements** () – tests if the enumeration contains more elements.
  2. E **nextElement** () – returns the next element of the enumeration.

- Limitations of Enumeration
  1. We can apply enumeration concept only for legacy classes & it is not universal cursor.
  2. By using Enumeration, we can get only read access & we can’t perform remove operation.

- To overcome these limitations, we should go for **Iterator**.

b) Iterator Interface

- We can apply Iterator concept for any collection object; hence it is Universal Cursor.
- By using Iterator, we can perform both read & remove operations.
- We can create Iterator object by using iterator () method present in Collection interface.
- **Methods**
  1. boolean **hasNext** () – returns true if the iteration has more elements.
  2. E **next** () – returns the next element in the iteration.
  3. void **remove** () – removes from the underlying collection the last element returned by the iterator.

- Limitations of Enumeration
  1. By using Enumeration & Iterator, we can always move only towards forward direction not backward direction. So, these are single direction cursors not bidirectional cursor.
  2. By using Iterator, we can get only read & remove operation & we can’t perform replacement or addition of new Objects.

- To overcome these limitations, we should go for **ListIterator**.

c) ListIterator Interface

- It is child interface of Iterator; hence all methods of Iterator by default available to the ListIterator.
- ListIterator is a bidirectional cursor.
- By using ListIterator, we can perform replacement & addition of new objects along with read & remove operations.
- We can create ListIterator by using listIterator () method of List interface.
- **Methods**

No.	Methods	Description
1.	void add (E element)	Inserts the specified element into the list.
2.	boolean hasNext ()	Returns true if the list iterator has more elements while traversing the list in forward direction.
3.	boolean hasPrevious ()	Return true if the list iterator has more elements while traversing the list in reverse direction.
4.	E next ()	Returns the next element in the list & advances the cursor position.
5.	int nextIndex ()	Returns the index of the element that would be returned by a subsequent call to next ().
6.	E previous ()	Returns the previous element in the list & moves the cursor position backwards.
7.	int previousIndex ()	Returns the index of the element that would be returned by a subsequent call to previous ().
8.	void remove ()	Removes the last element returned by next () or previous () from the list.
9.	void set (E element)	Replaces the last element returned by next() or previous () with the specified element.

No.	Property	Enumeration	Iterator	ListIterator
1.	Where we can apply?	Only for legacy classes	For any Collection objects	Only for List objects
2.	Is it legacy?	Yes	No	No
3.	Movement	Forward direction	Forward direction	Bidirectional
4.	Allowed operations	Only read	Read, Remove	Read, Remove, Add, Replace
5.	How can we get?	By using element () method of Vector class.	By using iterator () method of Collection interface.	By using listIterator () method of List interface.

```
import java.util.*;

class JavaCursorDemo {
   public static void main (String [] args) {
      Vector v = new Vector ();
      ArrayList l = new ArrayList ();
      for (int i = 0; i < 10; i++) {
         v.addElement(i);
         l.add(i);
      }

      System.out.println("Vector: " + v);
      System.out.print("Enumerated vector: ");
      Enumeration e = v.elements();
      while(e.hasMoreElements()) {
         Integer i = (Integer)e.nextElement();
         If (i % 2 == 0)
            System.out.print(i + ", ");
      }
```

```
        System.out.println();
        System.out.println("ArrayList: " + l);
        Iterator itr = l.iterator();
        System.out.print("Iterated list: ");
        while(itr.hasNext()) {
           Integer i = (Integer)itr.next();
           If (i % 2 == 0)
              System.out.print(i + ", ");
           else
              itr.remove();
        }
        System.out.println();
        System.out.println("List after odd no. removal: " + l);

        LinkedList ll = new LinkedList ();
        ll.add("bala"); ll.add("venki"); ll.add("nag"); ll.add("chiru");
        System.out.println("LinkedList: " + ll);
        ListIterator ltr = ll.listIterator();
        while(ltr.hasNext()) {
           String s = (String)ltr.next();
           if(s.equals("venki"))
              ltr.remove();
           else if (s.equals("nag"))
              ltr.add("chaitu");
           else if (s.equals("chiru"))
              ltr.set("charan");
        }
        System.out.println("ListIterated LinkedList: " + ll);

    }
}
```

Output:

```
Vector:                          [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Enumerated vector:                0, 2, 4, 6, 8,
ArrayList:                       [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Iterated list:                    0, 2, 4, 6, 8,
List after odd no. removal:      [0, 2, 4, 6, 8]
LinkedList:                      [bala, venki, nag, chiru]
ListIterated LinkedList:         [bala, nag, chaitu, charan]
```

```
import java.util.*;

class CursorDemo {
    public static void main (String [] args) {
        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
    }
}
```

7. Utility classes (Collections, Arrays)

Collections

- Collections class defines several utility methods for Collection objects like sorting, searching, reversing etc.
- **Sorting methods of Collection class**

a) public static void sort (List l)

- To sort based on default Natural sorting order.
- In this case, List should compulsory contain homogenous & Comparable objects otherwise we will get Runtime Exception: ClassCastException.
- List should not contain null otherwise we will get Runtime Exception: NullPointerException

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

- To sort based on Customized sorting order.

```
import java.util.*;

class CollectionsDemo {
   public static void main (String [] args) {
      ArrayList l = new ArrayList ();
      l.add("Z"); l.add("A"); l.add("K"); l.add("N");
      // l.add (new Integer (10));                    // RE: ClassCastException
      // l.add(null);                                  // RE: NullPointerException
      System.out.println("Before Sorting: " + l);
      //Collections.sort(l);                          // [A, K, N, Z]
      Collections.sort(l, new MyComparator ());       //  [Z, N, K, A]
      System.out.println("After Sorting: " + l);

   }
}

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

- **Searching elements of List**

a) public static int binarySearch (List l, Object target)

- If the list is sorted according to Default natural sorting order, then we should use this method.

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

- If the list is sorted according to some customized sorting order, then we should use this method.

c) Conclusions

- The above search method internally will use binary search algorithm
- Successful search returns index while unsuccessful search returns **Insertion Point**. **(Insertion point is the location where we can place 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, we have to pass same comparator object otherwise we will get unpredictable results.

Insertion Point:	-1	-2	-3	-4	-5	-6
	A	K	M	Z	a	
Index:	0	1	2	3	4	

```
import java.util.*;

class BinarySearchDemo {

  public static void main (String [] args) {
     ArrayList l = new ArrayList ();
     l.add("Z"); l.add("A"); l.add("M"); l.add("K"); l.add("a");
     System.out.println("Before Sorting: " + l);                    // [Z, A, M, K, a]
     Collections.sort(l);
     System.out.println("After Sorting: " + l);                     //  [A, K, M, Z, a]
     System.out.println(Collections.binarySearch(l, "Z"));          //  3
     System.out.println(Collections.binarySearch(l, "J"));          //  -2
     System.out.println(Collections.binarySearch(l, "b"));          //   -6
  }
}
```

Insertion Point:	-1	-2	-3	-4	-5	-6
	20	15	10	5	0	
Index:	0	1	2	3	4	

```
class BinarySearchDemo {

   public static void main (String [] args) {
     ArrayList l = new ArrayList ();
     l.add(15); l.add(0); l.add(20); l.add(10); l.add(5);
     System.out.println("List: " + l);                              // List: [15, 0, 20, 10, 5]
     Collections.sort(l, new MyComparator ());
     System.out.println("SortedList: " + l);                        // SortedList: [20, 15, 10, 5, 0]

     System.out.println(Collections.binarySearch(l, 10, new MyComparator ()));    //  2
     System.out.println(Collections.binarySearch(l, 13, new MyComparator ()));    //  -3
     System.out.println(Collections.binarySearch(l, 17));           // unpredictable (-6)
  }
}

class MyComparator implements Comparator {
  public int compare (Object obj1, Object obj2) {
     Integer i1 = (Integer)obj1;
     Integer i2 = (Integer)obj2;
     return i2.compareTo(i1);
  }
}
```

➢ **Reversing elements of List**

```
Collections.reverse (List l)
Comparator c1 = Collections.reverseOrder (Comparator c)
```

a) public static void reverse (List l)

- We can use reverse () method to reverse order of elements of List.
- We can use reverseOrder () method to get reversed comparator.

```
import java.util.*;

class CollectionsReverseDemo {
   public static void main (String [] args) {
      ArrayList l = new ArrayList ();
      l.add(15); l.add(0); l.add(20); l.add(10); l.add(5);
      System.out.println("List: " + l);                    // List: [15, 0, 20, 10, 5]
      Collections.reverse(l);
      System.out.println("Reversed List: " + l);     // Reversed List: [5, 10, 20, 0, 15]
   }
}
```

Arrays

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

➢ **Sorting elements of Array**

a) public static void **sort (Primitive [] p)** – to sort according to Natural sorting order
b) public static void **sort (Object [] o)** – to sort according to Natural sorting order
c) public static void **sort (Object [] o, Comparator c)** – to sort according to customized sorting order.

➢ **Searching the element of Array**

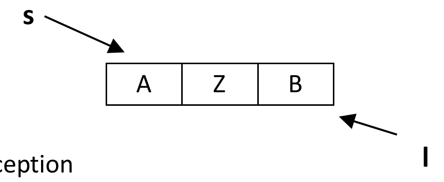
a) public static int **binarySearch** (Primitive [] p, Primitive target)
b) public static int **binarySearch** (Object [] o, Object target)
c) public static int **binarySearch** (Object [] o, Object target, Comparator c)

➢ **Conversion of Array to List**

a) public static List **asList** (Object [] o)

- Strictly speaking, this method won't create an independent List object. For the existing array, we're getting List view.

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


 System.out.println(l);          //  [A, Z, B]
// l.add("C");  l.remove(1);    // RE: UnsupportedOperationException
 s[0] = "C";
 System.out.println(l);         // [C, Z, B]
```



➢ By using **List reference**, we can't perform any operation which varies the size otherwise we will get **RE: UnsupportedOperationException.**

➢ By using **Array reference**, we can perform any change & automatically that change will be reflected to the List.

Collections & Arrays Utility Methods

No.	Methods
1.	public static void **sort** (List l)
2.	public static void **sort** (List l, Comparator c)
3.	public static int **binarySearch** (List l, Object target)
4.	public static int **binarySearch** (List l, Object target, Comparator c)
5.	public static void **reverse** (List l)
6.	public static Comparator **reverseOrder** (Comparator c)
7.	public static void **sort** (Primitive [] p)
8.	public static void **sort** (Object [] o)
9.	public static void **sort** (Object [] o, Comparator c)
10.	public static int **binarySearch** (Primitive [] p, Primitive p)
11.	public static int **binarySearch** (Object [] o, Object target)
12.	public static int **binarySearch** (Object [] o, Object target, Comparator c)
13.	public static List **asList** (Object [] o)