Collection Interface:-

The Collection interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have. These methods are summarized in the following table.

Because all collections implement Collection, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an **UnsupportedOperationException**.

Sr.No.	Method & Description
1	**boolean add(Object obj)** Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
2	**boolean addAll(Collection c)** Adds all the elements of c to the invoking collection. Returns true if the operation succeeds (i.e., the elements were added). Otherwise, returns false.
3	**void clear( )** Removes all elements from the invoking collection.
4	**boolean contains(Object obj)** Returns true if obj is an element of the invoking collection. Otherwise, returns false.
5	**boolean containsAll(Collection c)** Returns true if the invoking collection contains all elements of **c**. Otherwise, returns false.
6	**boolean equals(Object obj)** Returns true if the invoking collection and obj are equal. Otherwise, returns false.
7	**int hashCode( )** Returns the hash code for the invoking collection.
8	**boolean isEmpty( )**

	Returns true if the invoking collection is empty. Otherwise, returns false.
9	**Iterator iterator( )** Returns an iterator for the invoking collection.
10	**boolean remove(Object obj)** Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.
11	**boolean removeAll(Collection c)** Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
12	**boolean retainAll(Collection c)** Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
13	**int size( )** Returns the number of elements held in the invoking collection.
14	**Object[ ] toArray( )** Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
15	**Object[ ] toArray(Object array[ ])** Returns an array containing only those collection elements whose type matches that of array.

List Interface:-

The List interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.

- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.

- In addition to the methods defined by **Collection**, List defines some of its own, which are summarized in the following table.
- Several of the list methods will throw an UnsupportedOperationException if the collection cannot be modified, and a ClassCastException is generated when one object is incompatible with another.

Sr.No.	Method & Description
1	**void add(int index, Object obj)** Inserts obj into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	**boolean addAll(int index, Collection c)** Inserts all elements of **c** into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
3	**Object get(int index)** Returns the object stored at the specified index within the invoking collection.
4	**int indexOf(Object obj)** Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.
5	**int lastIndexOf(Object obj)** Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.
6	**ListIterator listIterator( )** Returns an iterator to the start of the invoking list.
7	**ListIterator listIterator(int index)** Returns an iterator to the invoking list that begins at the specified index.
8	**Object remove(int index)**

	Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
9	**Object set(int index, Object obj)** Assigns obj to the location specified by index within the invoking list.
10	**List subList(int start, int end)** Returns a list that includes elements from start to end.1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Set Interface:-

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.

The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

The methods declared by Set are summarized in the following table −

Sr.No.	Method & Description
1	**add( )** Adds an object to the collection.
2	**clear( )** Removes all objects from the collection.
3	**contains( )** Returns true if a specified object is an element within the collection.
4	**isEmpty( )** Returns true if the collection has no elements.

5	**iterator( )** Returns an Iterator object for the collection, which may be used to retrieve an object.
6	**remove( )** Removes a specified object from the collection.
7	**size( )** Returns the number of elements in the collection.

SortedSet Interface:-

The SortedSet interface extends Set and declares the behavior of a set sorted in an ascending order. In addition to those methods defined by Set, the SortedSet interface declares the methods summarized in the following table −

Several methods throw a NoSuchElementException when no items are contained in the invoking set. A ClassCastException is thrown when an object is incompatible with the elements in a set.

A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the set.

Sr.No.	Method & Description
1	**Comparator comparator( )** Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned.
2	**Object first( )** Returns the first element in the invoking sorted set.
3	**SortedSet headSet(Object end)** Returns a SortedSet containing those elements less than end that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.

4	**Object last( )** Returns the last element in the invoking sorted set.
5	**SortedSet subSet(Object start, Object end)** Returns a SortedSet that includes those elements between start and end.1. Elements in the returned collection are also referenced by the invoking object.
6	**SortedSet tailSet(Object start)** Returns a SortedSet that contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

Map Interface:-

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

- Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
- Several methods throw a NoSuchElementException when no items exist in the invoking map.
- A ClassCastException is thrown when an object is incompatible with the elements in a map.
- A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.
- An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map.

Sr.No.	Method & Description
1	**void clear( )** Removes all key/value pairs from the invoking map.
2	**boolean containsKey(Object k)** Returns true if the invoking map contains **k** as a key. Otherwise, returns false.
3	**boolean containsValue(Object v)**

	Returns true if the map contains **v** as a value. Otherwise, returns false.
4	**Set entrySet( )** Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map.
5	**boolean equals(Object obj)** Returns true if obj is a Map and contains the same entries. Otherwise, returns false.
6	**Object get(Object k)** Returns the value associated with the key **k**.
7	**int hashCode( )** Returns the hash code for the invoking map.
8	**boolean isEmpty( )** Returns true if the invoking map is empty. Otherwise, returns false.
9	**Set keySet( )** Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
10	**Object put(Object k, Object v)** Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
11	**void putAll(Map m)** Puts all the entries from **m** into this map.
12	**Object remove(Object k)** Removes the entry whose key equals **k**.
13	**int size( )** Returns the number of key/value pairs in the map.

14	**Collection values( )** Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Map Entry Interface:-

The Map.Entry interface enables you to work with a map entry.

The **entrySet( )** method declared by the Map interface returns a Set containing the map entries. Each of these set elements is a Map.Entry object.

Following table summarizes the methods declared by this interface −

Sr.No.	Method & Description
1	**boolean equals(Object obj)** Returns true if obj is a Map.Entry whose key and value are equal to that of the invoking object.
2	**Object getKey( )** Returns the key for this map entry.
3	**Object getValue( )** Returns the value for this map entry.
4	**int hashCode( )** Returns the hash code for this map entry.
5	**Object setValue(Object v)** Sets the value for this map entry to **v**. A ClassCastException is thrown if **v** is not the correct type for the map. A NullPointerException is thrown if **v** is null and the map does not permit null keys. An UnsupportedOperationException is thrown if the map cannot be changed.

Sorted Map Interface:-

The SortedMap interface extends Map. It ensures that the entries are maintained in an ascending key order.

Several methods throw a NoSuchElementException when no items are in the invoking map. A ClassCastException is thrown when an object is incompatible with the elements in a map. A NullPointerException is thrown if an attempt is made to use a null object when null is not allowed in the map.

The methods declared by SortedMap are summarized in the following table −

Sr.No.	Method & Description
1	**Comparator comparator( )** Returns the invoking sorted map's comparator. If the natural ordering is used for the invoking map, null is returned.
2	**Object firstKey( )** Returns the first key in the invoking map.
3	**SortedMap headMap(Object end)** Returns a sorted map for those map entries with keys that are less than end.
4	**Object lastKey( )** Returns the last key in the invoking map.
5	**SortedMap subMap(Object start, Object end)** Returns a map containing those entries with keys that are greater than or equal to start and less than end.
6	**SortedMap tailMap(Object start)** Returns a map containing those entries with keys that are greater than or equal to start.

Enumeration Interface:-

The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.

This legacy interface has been superceded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes such as Vector and Properties, is used by several other API classes, and is currently in widespread use in application code.

The methods declared by Enumeration are summarized in the following table −

Sr.No.	Method & Description
1	**boolean hasMoreElements( )** When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.
2	**Object nextElement( )** This returns the next object in the enumeration as a generic Object reference.

LinkedList Class:-

The LinkedList class extends AbstractSequentialList and implements the List interface. It provides a linked-list data structure.

Following are the constructors supported by the LinkedList class.

Sr.No.	Constructor & Description
1	**LinkedList( )** This constructor builds an empty linked list.
2	**LinkedList(Collection c)** This constructor builds a linked list that is initialized with the elements of the collection **c**.

Apart from the methods inherited from its parent classes, LinkedList defines following methods −

Sr.No.	Method & Description
1	**void add(int index, Object element)**

	Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index > size()).
2	**boolean add(Object o)** Appends the specified element to the end of this list.
3	**boolean addAll(Collection c)** Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException if the specified collection is null.
4	**boolean addAll(int index, Collection c)** Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null.
5	**void addFirst(Object o)** Inserts the given element at the beginning of this list.
6	**void addLast(Object o)** Appends the given element to the end of this list.
7	**void clear()** Removes all of the elements from this list.
8	**Object clone()** Returns a shallow copy of this LinkedList.
9	**boolean contains(Object o)** Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).
10	**Object get(int index)**

	Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
11	**Object getFirst()** Returns the first element in this list. Throws NoSuchElementException if this list is empty.
12	**Object getLast()** Returns the last element in this list. Throws NoSuchElementException if this list is empty.
13	**int indexOf(Object o)** Returns the index in this list of the first occurrence of the specified element, or -1 if the list does not contain this element.
14	**int lastIndexOf(Object o)** Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
15	**ListIterator listIterator(int index)** Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
16	**Object remove(int index)** Removes the element at the specified position in this list. Throws NoSuchElementException if this list is empty.
17	**boolean remove(Object o)** Removes the first occurrence of the specified element in this list. Throws NoSuchElementException if this list is empty. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
18	**Object removeFirst()**

	Removes and returns the first element from this list. Throws NoSuchElementException if this list is empty.
19	**Object removeLast()** Removes and returns the last element from this list. Throws NoSuchElementException if this list is empty.
20	**Object set(int index, Object element)** Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
21	**int size()** Returns the number of elements in this list.
22	**Object[] toArray()** Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.
23	**Object[] toArray(Object[] a)** Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.

Array List Class:-

The ArrayList class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.

Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

Following is the list of the constructors provided by the ArrayList class.

Sr.No.	Constructor & Description

1	**ArrayList( )** This constructor builds an empty array list.
2	**ArrayList(Collection c)** This constructor builds an array list that is initialized with the elements of the collection **c**.
3	**ArrayList(int capacity)** This constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

Apart from the methods inherited from its parent classes, ArrayList defines the following methods −

Sr.No.	Method & Description
1	**void add(int index, Object element)** Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index > size()).
2	**boolean add(Object o)** Appends the specified element to the end of this list.
3	**boolean addAll(Collection c)** Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException, if the specified collection is null.
4	**boolean addAll(int index, Collection c)** Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null.
5	**void clear()** Removes all of the elements from this list.

6	**Object clone()** Returns a shallow copy of this ArrayList.
7	**boolean contains(Object o)** Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element **e** such that (o==null ? e==null : o.equals(e)).
8	**void ensureCapacity(int minCapacity)** Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
9	**Object get(int index)** Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
10	**int indexOf(Object o)** Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
11	**int lastIndexOf(Object o)** Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
12	**Object remove(int index)** Removes the element at the specified position in this list. Throws IndexOutOfBoundsException if the index out is of range (index < 0 || index >= size()).
13	**protected void removeRange(int fromIndex, int toIndex)** Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
14	**Object set(int index, Object element)**

	Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
15	**int size()** Returns the number of elements in this list.
16	**Object[] toArray()** Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.
17	**Object[] toArray(Object[] a)** Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.
18	**void trimToSize()** Trims the capacity of this ArrayList instance to be the list's current size.

HashSet Class:-

HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.

A hash table stores information by using a mechanism called **hashing**. In hashing, the informational content of a key is used to determine a unique value, called its hash code.

The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

Following is the list of constructors provided by the HashSet class.

Sr.No.	Constructor & Description
1	**HashSet( )** This constructor constructs a default HashSet.
2	**HashSet(Collection c)** This constructor initializes the hash set by using the elements of the collection **c**.

3	**HashSet(int capacity)** This constructor initializes the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
4	**HashSet(int capacity, float fillRatio)** This constructor initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments. Here the fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded.

Apart from the methods inherited from its parent classes, HashSet defines following methods −

Sr.No.	Method & Description
1	**boolean add(Object o)** Adds the specified element to this set if it is not already present.
2	**void clear()** Removes all of the elements from this set.
3	**Object clone()** Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
4	**boolean contains(Object o)** Returns true if this set contains the specified element.
5	**boolean isEmpty()** Returns true if this set contains no elements.
6	**Iterator iterator()** Returns an iterator over the elements in this set.

7	**boolean remove(Object o)** Removes the specified element from this set if it is present.
8	**int size()** Returns the number of elements in this set (its cardinality).

Linked HashSet Class:-

This class extends HashSet, but adds no members of its own.

LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set.

That is, when cycling through a LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

Following is the list of constructors supported by the LinkedHashSet.

Sr.No.	Constructor & Description
1	**HashSet( )** This constructor constructs a default HashSet.
2	**HashSet(Collection c)** This constructor initializes the hash set by using the elements of the collection **c**.
3	**LinkedHashSet(int capacity)** This constructor initializes the capacity of the linkedhashset to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
4	**LinkedHashSet(int capacity, float fillRatio)** This constructor initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments.

TreeSet Class:-

TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in a sorted and ascending order.

Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

Following is the list of the constructors supported by the TreeSet class.

Sr.No.	Constructor & Description
1	**TreeSet( )** This constructor constructs an empty tree set that will be sorted in an ascending order according to the natural order of its elements.
2	**TreeSet(Collection c)** This constructor builds a tree set that contains the elements of the collection **c**.
3	**TreeSet(Comparator comp)** This constructor constructs an empty tree set that will be sorted according to the given comparator.
4	**TreeSet(SortedSet ss)** This constructor builds a TreeSet that contains the elements of the given SortedSet.

Apart from the methods inherited from its parent classes, TreeSet defines the following methods −

Sr.No.	Method & Description
1	**void add(Object o)** Adds the specified element to this set if it is not already present.
2	**boolean addAll(Collection c)** Adds all of the elements in the specified collection to this set.
3	**void clear()**

	Removes all of the elements from this set.
4	**Object clone()** Returns a shallow copy of this TreeSet instance.
5	**Comparator comparator()** Returns the comparator used to order this sorted set, or null if this tree set uses its elements natural ordering.
6	**boolean contains(Object o)** Returns true if this set contains the specified element.
7	**Object first()** Returns the first (lowest) element currently in this sorted set.
8	**SortedSet headSet(Object toElement)** Returns a view of the portion of this set whose elements are strictly less than toElement.
9	**boolean isEmpty()** Returns true if this set contains no elements.
10	**Iterator iterator()** Returns an iterator over the elements in this set.
11	**Object last()** Returns the last (highest) element currently in this sorted set.
12	**boolean remove(Object o)** Removes the specified element from this set if it is present.
13	**int size()** Returns the number of elements in this set (its cardinality).

14	**SortedSet subSet(Object fromElement, Object toElement)** Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
15	**SortedSet tailSet(Object fromElement)** Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

HashMap Interface:-

The HashMap class uses a hashtable to implement the Map interface. This allows the execution time of basic operations, such as get( ) and put( ), to remain constant even for large sets.

Following is the list of constructors supported by the HashMap class.

Sr.No.	Constructor & Description
1	**HashMap( )** This constructor constructs a default HashMap.
2	**HashMap(Map m)** This constructor initializes the hash map by using the elements of the given Map object **m**.
3	**HashMap(int capacity)** This constructor initializes the capacity of the hash map to the given integer value, capacity.
4	**HashMap(int capacity, float fillRatio)** This constructor initializes both the capacity and fill ratio of the hash map by using its arguments.

Apart from the methods inherited from its parent classes, HashMap defines the following methods −

Sr.No.	Method & Description

1	**void clear()** Removes all mappings from this map.
2	**Object clone()** Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
3	**boolean containsKey(Object key)** Returns true if this map contains a mapping for the specified key.
4	**boolean containsValue(Object value)** Returns true if this map maps one or more keys to the specified value.
5	**Set entrySet()** Returns a collection view of the mappings contained in this map.
6	**Object get(Object key)** Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.
7	**boolean isEmpty()** Returns true if this map contains no key-value mappings.
8	**Set keySet()** Returns a set view of the keys contained in this map.
9	**Object put(Object key, Object value)** Associates the specified value with the specified key in this map.
10	**putAll(Map m)** Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.

11	**Object remove(Object key)** Removes the mapping for this key from this map if present.
12	**int size()** Returns the number of key-value mappings in this map.
13	**Collection values()** Returns a collection view of the values contained in this map.

TreeMap Class:-

The TreeMap class implements the Map interface by using a tree. A TreeMap provides an efficient means of storing key/value pairs in sorted order, and allows rapid retrieval.

You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in an ascending key order.

Following is the list of the constructors supported by the TreeMap class.

Sr.No.	Constructors & Description
1	**TreeMap( )** This constructor constructs an empty tree map that will be sorted using the natural order of its keys.
2	**TreeMap(Comparator comp)** This constructor constructs an empty tree-based map that will be sorted using the Comparator comp.
3	**TreeMap(Map m)** This constructor initializes a tree map with the entries from **m**, which will be sorted using the natural order of the keys.
4	**TreeMap(SortedMap sm)**

	This constructor initializes a tree map with the entries from the SortedMap **sm**, which will be sorted in the same order as **sm**.

Apart from the methods inherited from its parent classes, TreeMap defines the following methods −

Sr.No.	Method & Description
1	**void clear()** Removes all mappings from this TreeMap.
2	**Object clone()** Returns a shallow copy of this TreeMap instance.
3	**Comparator comparator()** Returns the comparator used to order this map, or null if this map uses its keys' natural order.
4	**boolean containsKey(Object key)** Returns true if this map contains a mapping for the specified key.
5	**boolean containsValue(Object value)** Returns true if this map maps one or more keys to the specified value.
6	**Set entrySet()** Returns a set view of the mappings contained in this map.
7	**Object firstKey()** Returns the first (lowest) key currently in this sorted map.
8	**Object get(Object key)** Returns the value to which this map maps the specified key.
9	**SortedMap headMap(Object toKey)** Returns a view of the portion of this map whose keys are strictly less than toKey.

10	**Set keySet()** Returns a Set view of the keys contained in this map.
11	**Object lastKey()** Returns the last (highest) key currently in this sorted map.
12	**Object put(Object key, Object value)** Associates the specified value with the specified key in this map.
13	**void putAll(Map map)** Copies all of the mappings from the specified map to this map.
14	**Object remove(Object key)** Removes the mapping for this key from this TreeMap if present.
15	**int size()** Returns the number of key-value mappings in this map.
16	**SortedMap subMap(Object fromKey, Object toKey)** Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
17	**SortedMap tailMap(Object fromKey)** Returns a view of the portion of this map whose keys are greater than or equal to fromKey.
18	**Collection values()** Returns a collection view of the values contained in this map.

LinkedHash Map Class:-

This class extends HashMap and maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating a LinkedHashMap, the elements will be returned in the order in which they were inserted.

You can also create a LinkedHashMap that returns its elements in the order in which they were last accessed.

Following is the list of constructors supported by the LinkedHashMap class.

Sr.No.	Constructor & Description
1	**LinkedHashMap( )** This constructor constructs a default LinkedHashMap.
2	**LinkedHashMap(Map m)** This constructor initializes the LinkedHashMap with the elements from the given Map class **m**.
3	**LinkedHashMap(int capacity)** This constructor initializes a LinkedHashMap with the given capacity.
4	**LinkedHashMap(int capacity, float fillRatio)** This constructor initializes both the capacity and the fill ratio. The meaning of capacity and fill ratio are the same as for HashMap.
5	**LinkedHashMap(int capacity, float fillRatio, boolean Order)** This constructor allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If Order is true, then access order is used. If Order is false, then insertion order is used.

Apart from the methods inherited from its parent classes, LinkedHashMap defines the following methods −

Sr.No.	Method & Description
1	**void clear()** Removes all mappings from this map.
2	**boolean containsKey(Object key)** Returns true if this map maps one or more keys to the specified value.

3	**Object get(Object key)** Returns the value to which this map maps the specified key.
4	**protected boolean removeEldestEntry(Map.Entry eldest)** Returns true if this map should remove its eldest entry.

Vector:-

Vector implements a dynamic array. It is similar to ArrayList, but with two differences −

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Following is the list of constructors provided by the vector class.

Sr.No.	Constructor & Description
1	**Vector( )** This constructor creates a default vector, which has an initial size of 10.
2	**Vector(int size)** This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size.
3	**Vector(int size, int incr)** This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.
4	**Vector(Collection c)** This constructor creates a vector that contains the elements of collection c.

Apart from the methods inherited from its parent classes, Vector defines the following methods −

Sr.No.	Method & Description
1	**void add(int index, Object element)** Inserts the specified element at the specified position in this Vector.
2	**boolean add(Object o)** Appends the specified element to the end of this Vector.
3	**boolean addAll(Collection c)** Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
4	**boolean addAll(int index, Collection c)** Inserts all of the elements in in the specified Collection into this Vector at the specified position.
5	**void addElement(Object obj)** Adds the specified component to the end of this vector, increasing its size by one.
6	**int capacity()** Returns the current capacity of this vector.
7	**void clear()** Removes all of the elements from this vector.
8	**Object clone()** Returns a clone of this vector.
9	**boolean contains(Object elem)** Tests if the specified object is a component in this vector.
10	**boolean containsAll(Collection c)** Returns true if this vector contains all of the elements in the specified Collection.

11	**void copyInto(Object[] anArray)** Copies the components of this vector into the specified array.
12	**Object elementAt(int index)** Returns the component at the specified index.
13	**Enumeration elements()** Returns an enumeration of the components of this vector.
14	**void ensureCapacity(int minCapacity)** Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
15	**boolean equals(Object o)** Compares the specified Object with this vector for equality.
16	**Object firstElement()** Returns the first component (the item at index 0) of this vector.
17	**Object get(int index)** Returns the element at the specified position in this vector.
18	**int hashCode()** Returns the hash code value for this vector.
19	**int indexOf(Object elem)** Searches for the first occurence of the given argument, testing for equality using the equals method.
20	**int indexOf(Object elem, int index)** Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method.
21	**void insertElementAt(Object obj, int index)**

	Inserts the specified object as a component in this vector at the specified index.
22	**boolean isEmpty()** Tests if this vector has no components.
23	**Object lastElement()** Returns the last component of the vector.
24	**int lastIndexOf(Object elem)** Returns the index of the last occurrence of the specified object in this vector.
25	**int lastIndexOf(Object elem, int index)** Searches backwards for the specified object, starting from the specified index, and returns an index to it.
26	**Object remove(int index)** Removes the element at the specified position in this vector.
27	**boolean remove(Object o)** Removes the first occurrence of the specified element in this vector, If the vector does not contain the element, it is unchanged.
28	**boolean removeAll(Collection c)** Removes from this vector all of its elements that are contained in the specified Collection.
29	**void removeAllElements()** Removes all components from this vector and sets its size to zero.
30	**boolean removeElement(Object obj)** Removes the first (lowest-indexed) occurrence of the argument from this vector.
31	**void removeElementAt(int index)** removeElementAt(int index).

32	**protected void removeRange(int fromIndex, int toIndex)** Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
33	**boolean retainAll(Collection c)** Retains only the elements in this vector that are contained in the specified Collection.
34	**Object set(int index, Object element)** Replaces the element at the specified position in this vector with the specified element.
35	**void setElementAt(Object obj, int index)** Sets the component at the specified index of this vector to be the specified object.
36	**void setSize(int newSize)** Sets the size of this vector.
37	**int size()** Returns the number of components in this vector.
38	**List subList(int fromIndex, int toIndex)** Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
39	**Object[] toArray()** Returns an array containing all of the elements in this vector in the correct order.
40	**Object[] toArray(Object[] a)** Returns an array containing all of the elements in this vector in the correct order; the runtime type of the returned array is that of the specified array.
41	**String toString()** Returns a string representation of this vector, containing the String representation of each element.

42	**void trimToSize()** Trims the capacity of this vector to be the vector's current size.

Stack:-

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

```
Stack( )
```

Apart from the methods inherited from its parent class Vector, Stack defines the following methods −

Sr.No.	Method & Description
1	**boolean empty()** Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
2	**Object peek( )** Returns the element on the top of the stack, but does not remove it.
3	**Object pop( )** Returns the element on the top of the stack, removing it in the process.
4	**Object push(Object element)** Pushes the element onto the stack. Element is also returned.
5	**int search(Object element)** Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned.

HashTable:-

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.

However, Java 2 re-engineered Hashtable so that it also implements the Map interface. Thus, Hashtable is now integrated into the collections framework. It is similar to HashMap, but is synchronized.

Like HashMap, Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Following is the list of constructors provided by the HashTable class.

Sr.No	Constructor & Description
1	**Hashtable( )** This is the default constructor of the hash table it instantiates the Hashtable class.
2	**Hashtable(int size)** This constructor accepts an integer parameter and creates a hash table that has an initial size specified by integer value size.
3	**Hashtable(int size, float fillRatio)** This creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward.
4	**Hashtable(Map < ? extends K, ? extends V > t)** This constructs a Hashtable with the given mappings.

Apart from the methods defined by Map interface, Hashtable defines the following methods −

Sr.No	Method & Description
1	**void clear( )** Resets and empties the hash table.
2	**Object clone( )** Returns a duplicate of the invoking object.

3	**boolean contains(Object value)** Returns true if some value equal to the value exists within the hash table. Returns false if the value isn't found.
4	**boolean containsKey(Object key)** Returns true if some key equal to the key exists within the hash table. Returns false if the key isn't found.
5	**boolean containsValue(Object value)** Returns true if some value equal to the value exists within the hash table. Returns false if the value isn't found.
6	**Enumeration elements( )** Returns an enumeration of the values contained in the hash table.
7	**Object get(Object key)** Returns the object that contains the value associated with the key. If the key is not in the hash table, a null object is returned.
8	**boolean isEmpty( )** Returns true if the hash table is empty; returns false if it contains at least one key.
9	**Enumeration keys( )** Returns an enumeration of the keys contained in the hash table.
10	**Object put(Object key, Object value)** Inserts a key and a value into the hash table. Returns null if the key isn't already in the hash table; returns the previous value associated with the key if the key is already in the hash table.
11	**void rehash( )** Increases the size of the hash table and rehashes all of its keys.
12	**Object remove(Object key)**

	Removes the key and its value. Returns the value associated with the key. If the key is not in the hash table, a null object is returned.
13	**int size( )** Returns the number of entries in the hash table.
14	**String toString( )** Returns the string equivalent of a hash table.

Collection Algorithm:-

The collections framework defines several algorithms that can be applied to collections and maps.

These algorithms are defined as static methods within the Collections class. Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.

The methods defined in collection framework's algorithm are summarized in the following table −

Sr.No.	Method & Description
1	**static int binarySearch(List list, Object value, Comparator c)** Searches for value in the list ordered according to **c**. Returns the position of value in list, or -1 if value is not found.
2	**static int binarySearch(List list, Object value)** Searches for value in the list. The list must be sorted. Returns the position of value in list, or -1 if value is not found.
3	**static void copy(List list1, List list2)** Copies the elements of list2 to list1.
4	**static Enumeration enumeration(Collection c)** Returns an enumeration over **c**.

5	**static void fill(List list, Object obj)** Assigns obj to each element of the list.
6	**static int indexOfSubList(List list, List subList)** Searches list for the first occurrence of subList. Returns the index of the first match, or .1 if no match is found.
7	**static int lastIndexOfSubList(List list, List subList)** Searches list for the last occurrence of subList. Returns the index of the last match, or .1 if no match is found.
8	**static ArrayList list(Enumeration enum)** Returns an ArrayList that contains the elements of enum.
9	**static Object max(Collection c, Comparator comp)** Returns the maximum element in **c** as determined by comp.
10	**static Object max(Collection c)** Returns the maximum element in **c** as determined by natural ordering. The collection need not be sorted.
11	**static Object min(Collection c, Comparator comp)** Returns the minimum element in **c** as determined by comp. The collection need not be sorted.
12	**static Object min(Collection c)** Returns the minimum element in **c** as determined by natural ordering.
13	**static List nCopies(int num, Object obj)** Returns num copies of obj contained in an immutable list. num must be greater than or equal to zero.
14	**static boolean replaceAll(List list, Object old, Object new)** Replaces all occurrences of old with new in the list. Returns true if at least one replacement occurred. Returns false, otherwise.

15	**static void reverse(List list)** Reverses the sequence in list.
16	**static Comparator reverseOrder( )** Returns a reverse comparator.
17	**static void rotate(List list, int n)** Rotates list by **n** places to the right. To rotate left, use a negative value for **n**.
18	**static void shuffle(List list, Random r)** Shuffles (i.e., randomizes) the elements in the list by using **r** as a source of random numbers.
19	**static void shuffle(List list)** Shuffles (i.e., randomizes) the elements in list.
20	**static Set singleton(Object obj)** Returns obj as an immutable set. This is an easy way to convert a single object into a set.
21	**static List singletonList(Object obj)** Returns obj as an immutable list. This is an easy way to convert a single object into a list.
22	**static Map singletonMap(Object k, Object v)** Returns the key/value pair k/v as an immutable map. This is an easy way to convert a single key/value pair into a map.
23	**static void sort(List list, Comparator comp)** Sorts the elements of list as determined by comp.
24	**static void sort(List list)** Sorts the elements of the list as determined by their natural ordering.

25	**static void swap(List list, int idx1, int idx2)** Exchanges the elements in the list at the indices specified by idx1 and idx2.
26	**static Collection synchronizedCollection(Collection c)** Returns a thread-safe collection backed by **c**.
27	**static List synchronizedList(List list)** Returns a thread-safe list backed by list.
28	**static Map synchronizedMap(Map m)** Returns a thread-safe map backed by **m**.
29	**static Set synchronizedSet(Set s)** Returns a thread-safe set backed by **s**.
30	**static SortedMap synchronizedSortedMap(SortedMap sm)** Returns a thread-safe sorted set backed by **sm**.
31	**static SortedSet synchronizedSortedSet(SortedSet ss)** Returns a thread-safe set backed by **ss**.
32	**static Collection unmodifiableCollection(Collection c)** Returns an unmodifiable collection backed by **c**.
33	**static List unmodifiableList(List list)** Returns an unmodifiable list backed by the list.
34	**static Map unmodifiableMap(Map m)** Returns an unmodifiable map backed by **m**.
35	**static Set unmodifiableSet(Set s)** Returns an unmodifiable set backed by **s**.

36	**static SortedMap unmodifiableSortedMap(SortedMap sm)** Returns an unmodifiable sorted map backed by **sm**.
37	**static SortedSet unmodifiableSortedSet(SortedSet ss)** Returns an unmodifiable sorted set backed by **ss**.

Iterator:-

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

In general, to use an iterator to cycle through the contents of a collection, follow these steps −

- Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
- Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
- Within the loop, obtain each element by calling next( ).

For collections that implement List, you can also obtain an iterator by calling ListIterator.

The Methods Declared by Iterator

Sr.No.	Method & Description
1	**boolean hasNext( )** Returns true if there are more elements. Otherwise, returns false.

2	**Object next( )** Returns the next element. Throws NoSuchElementException if there is not a next element.
3	**void remove( )** Removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).

The Methods Declared by ListIterator

Sr.No.	Method & Description
1	**void add(Object obj)** Inserts obj into the list in front of the element that will be returned by the next call to next( ).
2	**boolean hasNext( )** Returns true if there is a next element. Otherwise, returns false.
3	**boolean hasPrevious( )** Returns true if there is a previous element. Otherwise, returns false.
4	**Object next( )** Returns the next element. A NoSuchElementException is thrown if there is not a next element.
5	**int nextIndex( )** Returns the index of the next element. If there is not a next element, returns the size of the list.
6	**Object previous( )** Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.

7	**int previousIndex( )** Returns the index of the previous element. If there is not a previous element, returns -1.
8	**void remove( )** Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) is invoked.
9	**void set(Object obj)** Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).

Comparable:-

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.

The Comparator interface defines two methods: compare( ) and equals( ). The compare( ) method, shown here, compares two elements for order −

The compare Method

```
int compare(Object obj1, Object obj2)
```

obj1 and obj2 are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned.

By overriding compare( ), you can alter the way that objects are ordered. For example, to sort in a reverse order, you can create a comparator that reverses the outcome of a comparison.

The equals Method

The equals( ) method, shown here, tests whether an object equals the invoking comparator −

```
boolean equals(Object obj)
```

obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false.

Overriding equals( ) is unnecessary, and most simple comparators will not do so.