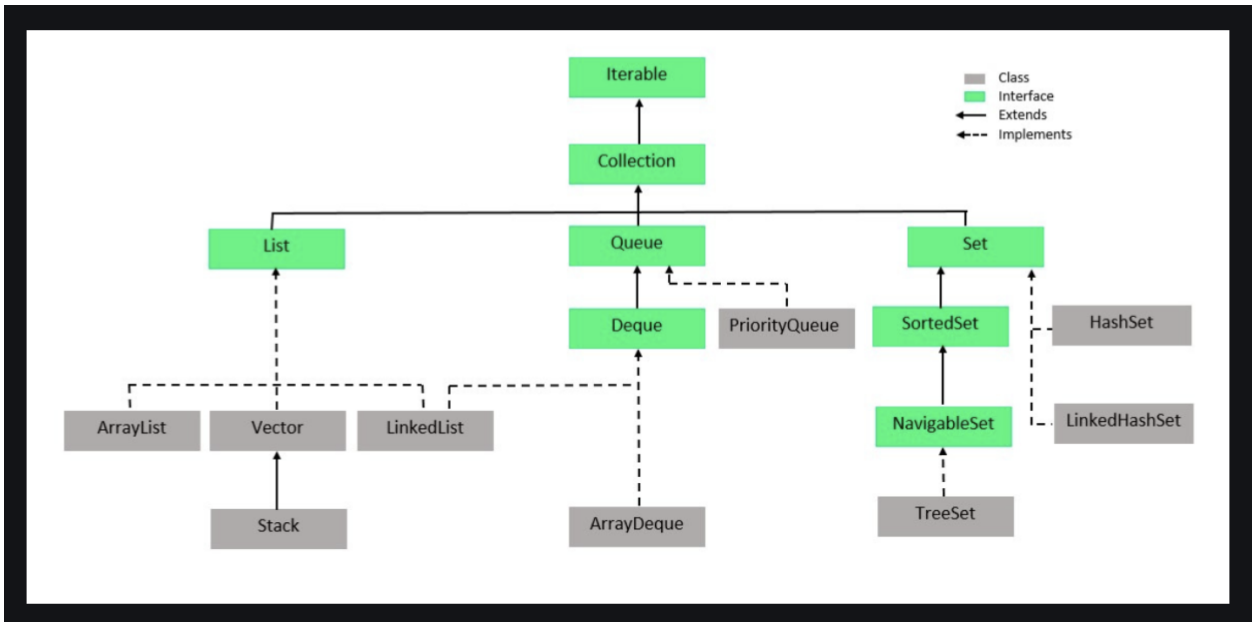
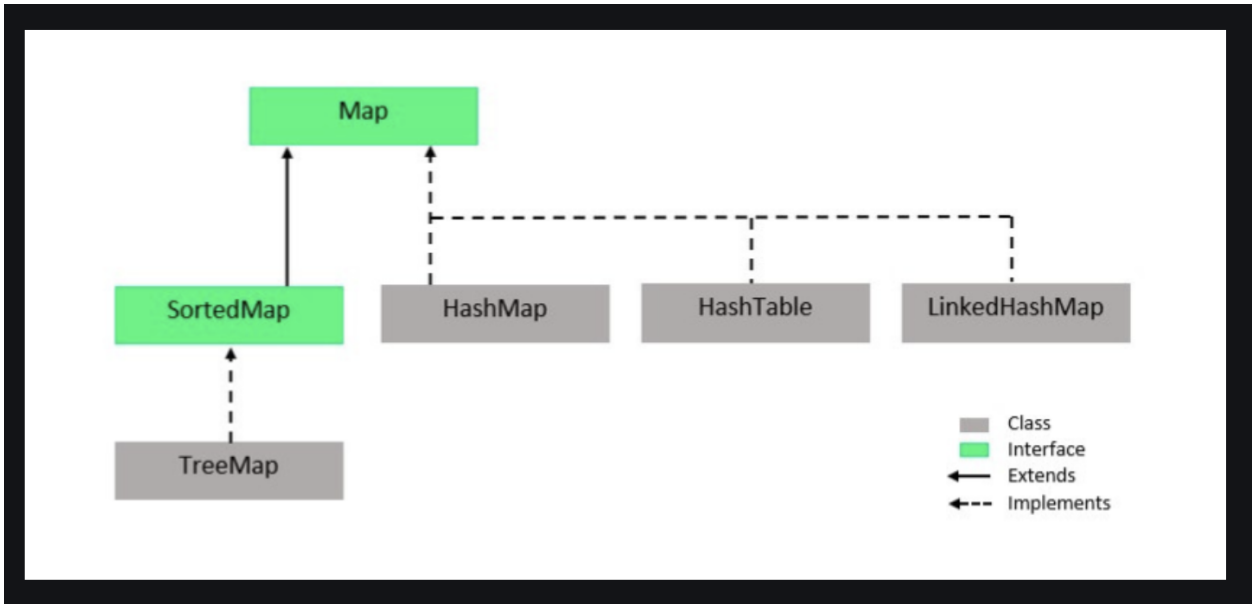
Collections

Collections Hierarchy




Map
SortedMap
HashMap
HashTable
LinkedHashMap
TreeMap
Class
Interface
Extends
Implements

1.The 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.

Example

Following is an example to explain few methods from various class implementations of the above collection methods –

```
import java.util.*;
public class CollectionsDemo {

   public static void main(String[] args) {
      // ArrayList
      List a1 = new ArrayList();
      a1.add("Zara");
      a1.add("Mahnaz");
      a1.add("Ayan");
      System.out.println(" ArrayList Elements");
      System.out.print("\t" + a1);

      // LinkedList
      List l1 = new LinkedList();
      l1.add("Zara");
      l1.add("Mahnaz");
      l1.add("Ayan");
      System.out.println();
      System.out.println(" LinkedList Elements");
      System.out.print("\t" + l1);

      // HashSet
      Set s1 = new HashSet();
      s1.add("Zara");
      s1.add("Mahnaz");
      s1.add("Ayan");
      System.out.println();
      System.out.println(" Set Elements");
```

```
      System.out.print("\t" + s1);

      // HashMap
      Map m1 = new HashMap();
      m1.put("Zara", "8");
      m1.put("Mahnaz", "31");
      m1.put("Ayan", "12");
      m1.put("Daisy", "14");
      System.out.println();
      System.out.println(" Map Elements");
      System.out.print("\t" + m1);
   }
}
```

This will produce the following result –

Output

```
ArrayList Elements
        [Zara, Mahnaz, Ayan]
 LinkedList Elements
        [Zara, Mahnaz, Ayan]
 Set Elements
        [Ayan, Zara, Mahnaz]
 Map Elements
        {Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}
```

2.Java - The 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.

Example

The above interface has been implemented in various classes like ArrayList or LinkedList, etc. Following is the example to explain few methods from various class implementation of the above collection methods –

```
import java.util.*;
public class CollectionsDemo {

   public static void main(String[] args) {
      List a1 = new ArrayList();
      a1.add("Zara");
      a1.add("Mahnaz");
      a1.add("Ayan");
      System.out.println(" ArrayList Elements");
      System.out.print("\t" + a1);

      List l1 = new LinkedList();
      l1.add("Zara");
      l1.add("Mahnaz");
      l1.add("Ayan");
      System.out.println();
      System.out.println(" LinkedList Elements");
      System.out.print("\t" + l1);
   }
}
```

This will produce the following result –

Output

```
ArrayList Elements
        [Zara, Mahnaz, Ayan]
 LinkedList Elements
        [Zara, Mahnaz, Ayan]
```

3.Java - The 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.

Example

Set has its implementation in various classes like HashSet, TreeSet, LinkedHashSet.

Following is an example to explain Set functionality –

```
import java.util.*;
public class SetDemo {

  public static void main(String args[]) {
     int count[] = {34, 22,10,60,30,22};
     Set<Integer> set = new HashSet<Integer>();
     try {
        for(int i = 0; i < 5; i++) {
           set.add(count[i]);
        }
        System.out.println(set);

        TreeSet sortedSet = new TreeSet<Integer>(set);
        System.out.println("The sorted list is:");
        System.out.println(sortedSet);

        System.out.println("The First element of the set is: "+ (Integer)sortedSet.first());
```

```
        System.out.println("The last element of the set is: "+ (Integer)sortedSet.last());
      }
      catch(Exception e) {}
   }
}
```

This will produce the following result –

Output

```
[34, 22, 10, 60, 30]
The sorted list is:
[10, 22, 30, 34, 60]
The First element of the set is: 10
The last element of the set is: 60
```

4.Java - The 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.

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.

Example

SortedSet have its implementation in various classes like TreeSet. Following is an example of a TreeSet class –

```
import java.util.*;
public class SortedSetTest {

   public static void main(String[] args) {
      // Create the sorted set
      SortedSet set = new TreeSet();

      // Add elements to the set
      set.add("b");
      set.add("c");
      set.add("a");
```

```
      // Iterating over the elements in the set
      Iterator it = set.iterator();

      while (it.hasNext()) {
        // Get element
        Object element = it.next();
        System.out.println(element.toString());
      }
   }
}
```

This will produce the following result –

Output

```
a
b
c
```

5.Java - The 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.

Example

Map has its implementation in various classes like HashMap. Following is an example to explain map functionality –

```
import java.util.*;
public class CollectionsDemo {

   public static void main(String[] args) {
      Map m1 = new HashMap();
      m1.put("Zara", "8");
      m1.put("Mahnaz", "31");
      m1.put("Ayan", "12");
      m1.put("Daisy", "14");

      System.out.println();
      System.out.println(" Map Elements");
      System.out.print("\t" + m1);
   }
}
```

This will produce the following result –

Output

```
Map Elements
        {Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}
```

6.Java - The 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.

Example

Following is an example showing how Map.Entry can be used –

```
import java.util.*;
public class HashMapDemo {

   public static void main(String args[]) {
      // Create a hash map
      HashMap hm = new HashMap();

      // Put elements to the map
      hm.put("Zara", new Double(3434.34));
      hm.put("Mahnaz", new Double(123.22));
      hm.put("Ayan", new Double(1378.00));
      hm.put("Daisy", new Double(99.22));
```

```
      hm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = hm.entrySet();

      // Get an iterator
      Iterator i = set.iterator();

      // Display elements
      while(i.hasNext()) {
        Map.Entry me = (Map.Entry)i.next();
        System.out.print(me.getKey() + ": ");
        System.out.println(me.getValue());
      }
      System.out.println();

      // Deposit 1000 into Zara's account
      double balance = ((Double)hm.get("Zara")).doubleValue();
      hm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " + hm.get("Zara"));
   }
}
```

This will produce the following result –

Output

```
Daisy: 99.22
Ayan: 1378.0
Zara: 3434.34
Qadir: -19.08
Mahnaz: 123.22

Zara's new balance: 4434.34
```

7.Java - The SortedMap 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.

Example

SortedMap has its implementation in various classes like TreeMap. Following is the example to explain SortedMap functionlaity –

```
import java.util.*;
public class TreeMapDemo {

   public static void main(String args[]) {
      // Create a hash map
      TreeMap tm = new TreeMap();

      // Put elements to the map
      tm.put("Zara", new Double(3434.34));
      tm.put("Mahnaz", new Double(123.22));
      tm.put("Ayan", new Double(1378.00));
      tm.put("Daisy", new Double(99.22));
      tm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = tm.entrySet();

      // Get an iterator
      Iterator i = set.iterator();
```

```
      // Display elements
      while(i.hasNext()) {
        Map.Entry me = (Map.Entry)i.next();
        System.out.print(me.getKey() + ": ");
        System.out.println(me.getValue());
      }
      System.out.println();

      // Deposit 1000 into Zara's account
      double balance = ((Double)tm.get("Zara")).doubleValue();
      tm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " + tm.get("Zara"));
   }
}
```

This will produce the following result –

Output

```
Ayan: 1378.0
Daisy: 99.22
Mahnaz: 123.22
Qadir: -19.08
Zara: 3434.34

Zara's new balance: 4434.34
```

8.Java - The 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.

Example

Following is an example showing usage of Enumeration.

```
import java.util.Vector;
import java.util.Enumeration;

public class EnumerationTester {

  public static void main(String args[]) {
    Enumeration days;
    Vector dayNames = new Vector();

    dayNames.add("Sunday");
    dayNames.add("Monday");
```

```
      dayNames.add("Tuesday");
      dayNames.add("Wednesday");
      dayNames.add("Thursday");
      dayNames.add("Friday");
      dayNames.add("Saturday");
      days = dayNames.elements();

      while (days.hasMoreElements()) {
        System.out.println(days.nextElement());
      }
    }
}
```

This will produce the following result –

Output

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

The Collection Classes

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table –

Sr.No.	Class & Description
1	AbstractCollection Implements most of the Collection interface.
2	AbstractList Extends AbstractCollection and implements most of the List interface.
3	AbstractSequentialList Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
4	LinkedList Implements a linked list by extending AbstractSequentialList.
5	ArrayList Implements a dynamic array by extending AbstractList.
6	AbstractSet Extends AbstractCollection and implements most of the Set interface.
7	HashSet

	Extends AbstractSet for use with a hash table.
8	LinkedHashSet Extends HashSet to allow insertion-order iterations.
9	TreeSet Implements a set stored in a tree. Extends AbstractSet.
10	AbstractMap Implements most of the Map interface.
11	HashMap Extends AbstractMap to use a hash table.
12	TreeMap Extends AbstractMap to use a tree.
13	WeakHashMap Extends AbstractMap to use a hash table with weak keys.
14	LinkedHashMap Extends HashMap to allow insertion-order iterations.
15	IdentityHashMap

	Extends AbstractMap and uses reference equality when comparing documents.

Java - The Collection Algorithms

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.

Example

Following is an example, which demonstrates various algorithms.

```
import java.util.*;
public class AlgorithmsDemo {

   public static void main(String args[]) {

      // Create and initialize linked list
```

```
      LinkedList ll = new LinkedList();
      ll.add(new Integer(-8));
      ll.add(new Integer(20));
      ll.add(new Integer(-20));
      ll.add(new Integer(8));

      // Create a reverse order comparator
      Comparator r = Collections.reverseOrder();

      // Sort list by using the comparator
      Collections.sort(ll, r);

      // Get iterator
      Iterator li = ll.iterator();
      System.out.print("List sorted in reverse: ");

      while(li.hasNext()) {
        System.out.print(li.next() + " ");
      }
      System.out.println();
      Collections.shuffle(ll);

      // display randomized list
      li = ll.iterator();
      System.out.print("List shuffled: ");

      while(li.hasNext()) {
        System.out.print(li.next() + " ");
      }

      System.out.println();
      System.out.println("Minimum: " + Collections.min(ll));
      System.out.println("Maximum: " + Collections.max(ll));
   }
}
```

This will produce the following result –

Output

```
List sorted in reverse: 20 8 -8 -20
List shuffled: 20 -20 8 -8
```

Minimum: -20
Maximum: 20

ArrayList 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.

Example

The following program illustrates several of the methods supported by ArrayList –

```
import java.util.*;
public class ArrayListDemo {

   public static void main(String args[]) {
      // create an array list
      ArrayList al = new ArrayList();
      System.out.println("Initial size of al: " + al.size());

      // add elements to the array list
      al.add("C");
      al.add("A");
      al.add("E");
      al.add("B");
      al.add("D");
      al.add("F");
      al.add(1, "A2");
      System.out.println("Size of al after additions: " + al.size());
```

```
      // display the array list
      System.out.println("Contents of al: " + al);

      // Remove elements from the array list
      al.remove("F");
      al.remove(2);
      System.out.println("Size of al after deletions: " + al.size());
      System.out.println("Contents of al: " + al);
   }
}
```

This will produce the following result –

Output

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

Java - The HashMap Class

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.

Example

The following program illustrates several of the methods supported by this collection –

```
import java.util.*;
public class HashMapDemo {

   public static void main(String args[]) {

      // Create a hash map
      HashMap hm = new HashMap();

      // Put elements to the map
      hm.put("Zara", new Double(3434.34));
      hm.put("Mahnaz", new Double(123.22));
      hm.put("Ayan", new Double(1378.00));
      hm.put("Daisy", new Double(99.22));
      hm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = hm.entrySet();

      // Get an iterator
      Iterator i = set.iterator();

      // Display elements
      while(i.hasNext()) {
        Map.Entry me = (Map.Entry)i.next();
        System.out.print(me.getKey() + ": ");
        System.out.println(me.getValue());
      }
      System.out.println();

      // Deposit 1000 into Zara's account
      double balance = ((Double)hm.get("Zara")).doubleValue();
      hm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " + hm.get("Zara"));
   }
}
```

This will produce the following result –

Output

```
Daisy: 99.22
Ayan: 1378.0
Zara: 3434.34
```

Qadir: -19.08
Mahnaz: 123.22

Zara's new balance: 4434.34

Java - The 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.

Example

The following program illustrates several of the methods supported by LinkedList –

```
import java.util.*;

public class LinkedListDemo {


   public static void main(String args[]) {

      // create a linked list

      LinkedList ll = new LinkedList();



      // add elements to the linked list

      ll.add("F");

      ll.add("B");
```

```
ll.add("D");

ll.add("E");

ll.add("C");

ll.addLast("Z");

ll.addFirst("A");

ll.add(1, "A2");

System.out.println("Original contents of ll: " + ll);



// remove elements from the linked list

ll.remove("F");

ll.remove(2);

System.out.println("Contents of ll after deletion: " + ll);



// remove first and last elements

ll.removeFirst();

ll.removeLast();

System.out.println("ll after deleting first and last: " + ll);



// get and set a value
```

```
      Object val = ll.get(2);

      ll.set(2, (String) val + " Changed");

      System.out.println("ll after change: " + ll);

   }
}
```

This will produce the following result –

Output

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]

Contents of ll after deletion: [A, A2, D, E, C, Z]

ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

How to Use an 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.

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( ).

Example

Here is an example demonstrating both Iterator and ListIterator. It uses an ArrayList object, but the general principles apply to any type of collection.

Of course, ListIterator is available only to those collections that implement the List interface.

```
import java.util.*;

public class IteratorDemo {


  public static void main(String args[]) {

    // Create an array list

    ArrayList al = new ArrayList();
```

```
    // add elements to the array list
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");

    // Use iterator to display contents of al
    System.out.print("Original contents of al: ");
    Iterator itr = al.iterator();

    while(itr.hasNext()) {
      Object element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // Modify objects being iterated
    ListIterator litr = al.listIterator();
```

```
while(litr.hasNext()) {
   Object element = litr.next();
   litr.set(element + "+");
}
System.out.print("Modified contents of al: ");
itr = al.iterator();

while(itr.hasNext()) {
   Object element = itr.next();
   System.out.print(element + " ");
}
System.out.println();

// Now, display the list backwards
System.out.print("Modified list backwards: ");

while(litr.hasPrevious()) {
   Object element = litr.previous();
   System.out.print(element + " ");
```

```
    }

    System.out.println();

  }

}
```

This will produce the following result –

Output

```
Original contents of al: C A E B D F

Modified contents of al: C+ A+ E+ B+ D+ F+

Modified list backwards: F+ D+ B+ E+ A+ C+
```

Sr.No.	Iterator Method & Description
1	Using Java Iterator Here is a list of all the methods with examples provided by Iterator and ListIterator interfaces.

How to Use a Comparator ?

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.

Example

```
import java.util.*;

class Dog implements Comparator<Dog>, Comparable<Dog> {
   private String name;
   private int age;
   Dog() {
   }

   Dog(String n, int a) {
      name = n;
      age = a;
   }

   public String getDogName() {
      return name;
   }

   public int getDogAge() {
      return age;
   }

   // Overriding the compareTo method
   public int compareTo(Dog d) {
      return (this.name).compareTo(d.name);
   }

   // Overriding the compare method to sort the age
   public int compare(Dog d, Dog d1) {
      return d.age - d1.age;
   }
}

public class Example {

   public static void main(String args[]) {
      // Takes a list o Dog objects
      List<Dog> list = new ArrayList<Dog>();

      list.add(new Dog("Shaggy", 3));
```

```
      list.add(new Dog("Lacy", 2));
      list.add(new Dog("Roger", 10));
      list.add(new Dog("Tommy", 4));
      list.add(new Dog("Tammy", 1));
      Collections.sort(list);   // Sorts the array list

      for(Dog a: list)   // printing the sorted list of names
        System.out.print(a.getDogName() + ", ");

      // Sorts the array list using comparator
      Collections.sort(list, new Dog());
      System.out.println(" ");

      for(Dog a: list)   // printing the sorted list of ages
        System.out.print(a.getDogName() +"  : "+ a.getDogAge() + ", ");
   }
}
```

This will produce the following result –

Output

```
Lacy, Roger, Shaggy, Tammy, Tommy,
Tammy  : 1, Lacy  : 2, Shaggy  : 3, Tommy  : 4, Roger  : 10,
```