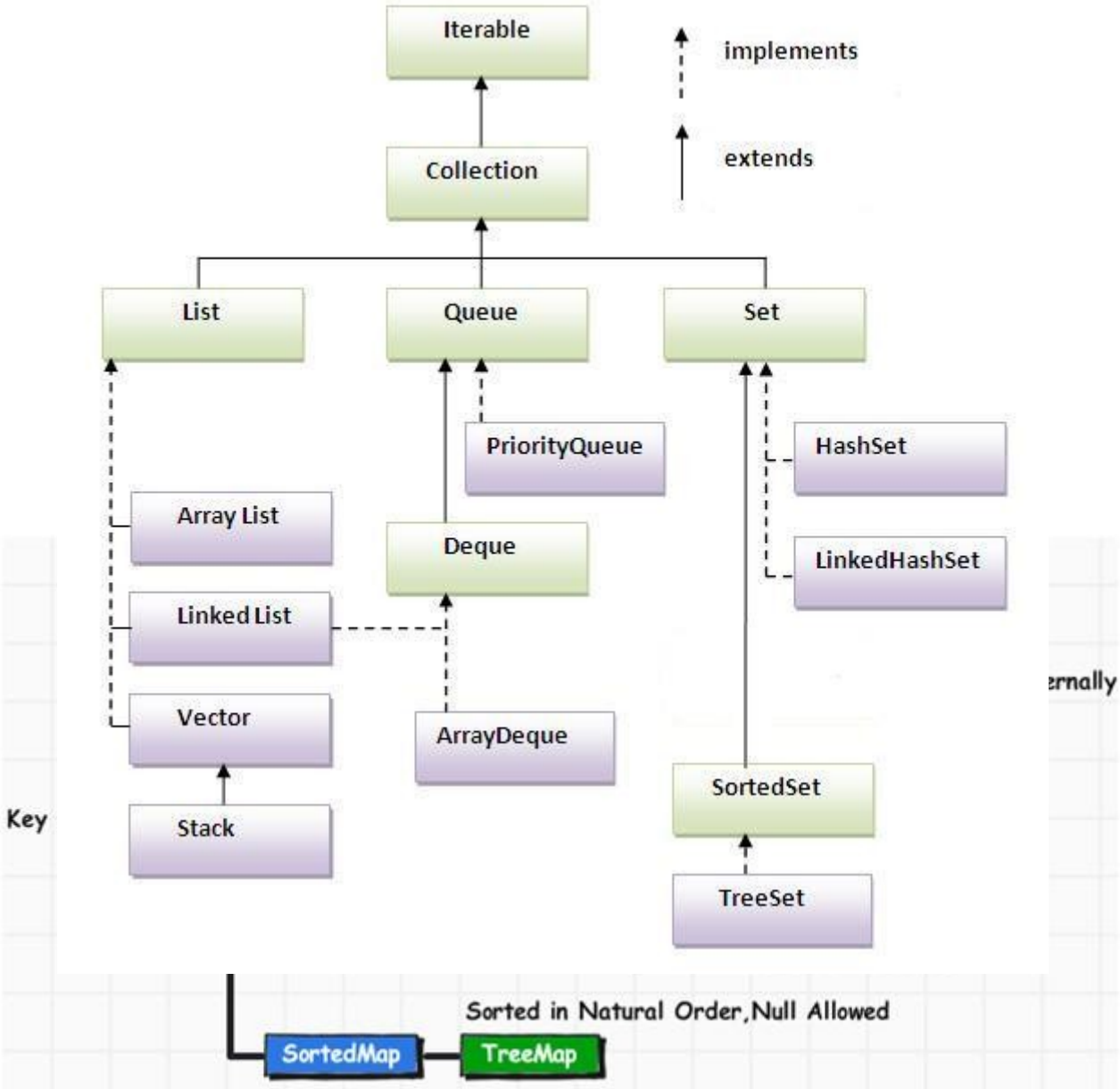
15. COLLECTION FRAMEWORK

Collections are used to store, retrieve, manipulate, and communicate aggregate data.

1. Collection Framework:

- **Interfaces**
- **Implementations**
- **Algorithms**

2. Collection Hierarchy:



15. COLLECTION FRAMEWORK

3. Benefits of the Java Collection Framework:

- Reduces programming effort:
- Increases program speed and quality:
- Allows interoperability among unrelated APIs:
- Reduces effort to learn and to use new APIs:
- Reduces effort to design new APIs:
- Fosters software reuse:

4. Java Collection Interfaces:

1. List interface:
- Ordered Collection
- may contain duplicate elements.
- User has precise control where to insert element and can access elements by their position.

2. Set interface:
- can not contain duplicate elements
- unordered

3. SortedSet interface:
- unordered
- sorted on insertion order
- no duplicates

4. NavigableSet interface:
- extends SortedSet
- retrieval of elements based on closet match to given value or values

5. Queue interface:
- order elements in FIFO manner.
- new elements are inserted at the tail
- Priority queue class: no FIFO

6. Deque interface:
- extends Queue
- both as FIFO and LIFO.
- New elements inserted, retrieved and removed at both ends

5. Interfaces Independant of Collection:

1. Map interface:
- maps keys to values
- no duplicate keys allowed but value may be duplicate
- each key can map to atmost one value

2. SortedMap interface:

- sorted by key
- maps keys to values
- no duplicates
- each key can map to atmost one value

6. Interfaces for Traversing Collection:

1. Iterator:

- Interface
- to traverse through the collection
- to obtain the object of type Iterator, we have to call the iterator method.

Syntax:

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove();
}
```

2. ListIterator:

- Interfaces
- extends Iterator interfaces
- used to traverse in both directions forward and backward
- can be used to traverse only the Collection those implement List interface.

Syntax :

```
public interface ListIterator<E> extends Iterator<E>{
        boolean hasNext();
        boolean hasPrevious();
        E next();
        E previous();
        void remove();
        void set(E e);
        void add(E e);
        int nextIndex();
        int previousIndex();
}
```

3. Enumeration:

- used to traverse Legacy classes like Vector, hashtable
- can not remove the element from collection as no remove method is provided
- can not travesre in backward direction

Syntax:

```
public interface Enumeration<E>
{
        boolean hasMoreElements();
        E nextElement();
}
```

Other than these interfaces we can use the enhanced for to only traverse the collection

Enhanced for: (not interface)

- traverse a collection using for loop

```
for (Object o : collection)
   System.out.println(o);
```

Use Iterator instead of the for-each construct when you need to:

- Remove the current element. Because the for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.
- Iterate over multiple collections in parallel.

7. Bulk Operations:

- containAll : returns boolean
- addAll
- removeAll
- retainAll
- clear

Example:

Collection 1: a,b,c,d
Collection 2: a,b,c,d,e,f.

1. containAll : true
2.removeAll : e,f
3.retainAll : a,b,c,d

8.Java Collections class:

Collections is a utility class which contains the static methods like search, sort, shuffle which can be operated for the collection . These methods may throw NullPointer exception if collection provided to them is null.

9. List Interface and Classes:

1. ArrayList class:

- oredered, may contain duplicate members
- efficient for searching
- not suitable for manipulation: add, remove, insert

Constructors:

ArrayList( )
ArrayList(Collection<? extends E> c)
ArrayList(int capacity)

Methods of ArrayList class :

There are number of methods available which can be used directly using object of ArrayList class. Some of the mportant methods are:

- **add(Object o)**: This method adds an object o to the arraylist.
  obj.add("hello");
  This statement would add a string hello in the arraylist at last position.

- **add(int index, Object o)**: It adds the object o to the array list at the given index.
  obj.add(2, "bye");
  It will add the string bye to the 2nd index (3rd position as the array list starts with index 0) of array list.

- **remove(Object o)**: Removes the object o from the ArrayList.
  obj.remove("Chaitanya");
  This statement will remove the string “Chaitanya” from the ArrayList.

- **remove(int index)**: Removes element from a given index.
  obj.remove(3);
  It would remove the element of index 3 (4th element of the list – List starts with o).

- **set(int index, Object o)**: Used for updating an element. It replaces the element present at the specified index with the object o.
  obj.set(2, "Tom");
  It would replace the 3rd element (index =2 is 3rd element) with the value Tom.

- **int indexOf(Object o)**: Gives the index of the object o. If the element is not found in the list then this method returns the value -1.
  int pos = obj.indexOf("Tom");
  This would give the index (position) of the string Tom in the list.

- **Object get(int index)**: It returns the object of list which is present at the specified index.
  String str= obj.get(2);

Function get would return the string stored at 3rd position (index 2) and would be assigned to the string "str". We have stored the returned value in string variable because in our example we have defined the ArrayList is of String type. If you are having integer array list then the returned value should be stored in an integer variable.

- **int size()**: It gives the size of the ArrayList – Number of elements of the list.
  int numberofitems = obj.size();

- **boolean contains(Object o)**: It checks whether the given object o is present in the array list if its there then it returns true else it returns false.
  obj.contains("Steve");
  It would return true if the string "Steve" is present in the list else we would get false.

- **clear():** It is used for removing all the elements of the array list in one go. The below code will remove all the elements of ArrayList whose object is obj.
  obj.clear();

Example: simple add, remove example

```
package filehandling;
import java.util.*;
public class SimpleArrayListDemo {

    public static void main(String[] args) {

   ArrayList<String> obj = new ArrayList<String>();

   /*This is how elements should be added to the array list*/
   obj.add("Ajeet");
   obj.add("Harry");
   obj.add("Chaitanya");
   obj.add("Steve");
   obj.add("Anuj");

   /* Displaying array list elements */

     System.out.println("Currently the array list has following elements:"+obj);

   /*Add element at the given index*/
   obj.add(0, "Rahul");
   obj.add(1, "Justin");

   /*Remove elements from array list like this*/
   obj.remove("Chaitanya");
   obj.remove("Harry");
```

```
        System.out.println("Updated array list is:"+obj);

        /*Remove element from the given index*/
        obj.remove(1);

        System.out.println("Updated array list is:"+obj);

        //Sorting arraylist

        Collections.sort(obj);
        System.out.println("After sorting:"+obj);
    }
}
```

Output:

Currently the array list has following elements:[Ajeet, Harry, Chaitanya, Steve, Anuj]
Updated array list is:[Rahul, Justin, Ajeet, Steve, Anuj]
Updated array list is:[Rahul, Ajeet, Steve, Anuj]
After sorting:[Ajeet, Anuj, Rahul, Steve]

Example: drawing cards

```
public class CardDemo {

        public static void main(String[] args) {

                Scanner sc=new Scanner(System.in);
                String[] suit = new String[] {"spades", "hearts", "diamonds", "clubs"};
                String[] rank = new String[] {"ace", "2", "3", "4","5", "6", "7", "8", "9", "10", "jack",
"queen", "king" };

                List<String> deck=new ArrayList<String>();

                for (int i = 0; i < suit.length; i++)
                {
                        for (int j = 0; j < rank.length; j++)

                        {
                           deck.add(rank[j] + " of " + suit[i]);
                        }
                }
                Collections.shuffle(deck);
                Iterator<String> it=deck.iterator();

                while(it.hasNext())
                {
                         System.out.println(it.next());
```

```
                }
                System.out.println("Deck size: "+deck.size());
                System.out.println("Enter number of hands for deal");
                int numhands=sc.nextInt();
                System.out.println("Enter number of cards per hand for deal");
                int numcards=sc.nextInt();

                if((numhands*numcards)>deck.size())
                {
                        System.out.println("No enough cards..");
                }

                else
                {
                        for(int i=0;i<numhands;i++)
                        {
                        List<String> list= dealHand(deck, numcards);
                        System.out.println("Hand "+(i+1)+" "+list);
                        }
                }

        }
        public static List<String> dealHand(List<String> deck, int numhands)
        {
                List<String> handView=deck.subList(deck.size()-numhands, deck.size());
                List<String> hand=new ArrayList<String>(handView);
                handView.clear();
                return hand;
        }
}
```

Output:

```
queen of clubs
4 of spades
king of diamonds
10 of diamonds
queen of hearts
10 of clubs
queen of spades
8 of clubs
10 of spades
8 of diamonds
king of clubs
8 of hearts
5 of hearts
7 of hearts
king of spades
8 of spades
5 of clubs
```

```
4 of diamonds
2 of spades
jack of clubs
6 of clubs
queen of diamonds
6 of hearts
5 of diamonds
3 of clubs
6 of spades
7 of diamonds
4 of clubs
ace of hearts
10 of hearts
2 of clubs
3 of hearts
9 of hearts
9 of diamonds
king of hearts
ace of clubs
7 of spades
jack of hearts
2 of hearts
5 of spades
jack of spades
ace of spades
7 of clubs
3 of diamonds
4 of hearts
3 of spades
ace of diamonds
2 of diamonds
9 of spades
9 of clubs
jack of diamonds
6 of diamonds
Deck size: 52
Enter number of hands for deal
6
Enter number of cards per hand for deal
5
Hand 1 [2 of diamonds, 9 of spades, 9 of clubs, jack of diamonds, 6 of diamonds]
Hand 2 [7 of clubs, 3 of diamonds, 4 of hearts, 3 of spades, ace of diamonds]
Hand 3 [jack of hearts, 2 of hearts, 5 of spades, jack of spades, ace of spades]
Hand 4 [9 of hearts, 9 of diamonds, king of hearts, ace of clubs, 7 of spades]
Hand 5 [4 of clubs, ace of hearts, 10 of hearts, 2 of clubs, 3 of hearts]
Hand 6 [6 of hearts, 5 of diamonds, 3 of clubs, 6 of spades, 7 of diamonds]
```

List algorithms (methods from Collections class) :

- **sort :** sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A stable sort is one that does not reorder equal elements.)
- **shuffle :** randomly permutes the elements in a List.

- **reverse :** reverses the order of the elements in a List.
- **rotate :** rotates all the elements in a List by a specified distance.
- **swap :** swaps the elements at specified positions in a List.
- **replaceAll :** replaces all occurrences of one specified value with another.
- **fill :** overwrites every element in a List with the specified value.
- **copy :** copies the source List into the destination List.
- **binarySearch :** searches for an element in an ordered List using the binary search algorithm.
- **indexOfSubList :** returns the index of the first sublist of one List that is equal to another.
- **lastIndexOfSubList :** returns the index of the last sublist of one List that is equal to another.

Example :

Sorting of ArrayList<Integer> /*Collections.sort( )method can be used for sorting the Integer ArrayList as well.*/

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ArraListSortDemo {

      public static void main(String args[]){
        List<Integer> arraylist = new ArrayList<Integer>();
        arraylist.add(11);

       arraylist.add(2);
        arraylist.add(7);
        arraylist.add(3);

        /* ArrayList before the sorting*/
        System.out.println("Before Sorting:");

        for(int counter: arraylist){      // enhanced for
                System.out.println(counter);
            }

        /* Sorting of array list using Collections.sort*/
        Collections.sort(arraylist);

        /* ArrayList after sorting*/
        System.out.println("After Sorting:");
        for(int counter: arraylist){
                System.out.println(counter);
            }
      }
}
```

Output:

```
Before Sorting:
11
2
7
3
After Sorting:
2
3
7
11
```

2. LinkedList class:

- implements doubly linked list
- ordered
- may contain duplicates
- suitable for manipulation(add, remove, insert)
- implements List, Queue and Deque interface

Constructors:

LinkedList( )
LinkedList(Collection<? extends E> c)

Example:

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

    public static void main(String[] args) {

    /* Linked List Declaration */
    LinkedList<String> linkedlist = new LinkedList<String>();

    /*add(String Element) is used for adding
     * the elements to the linked list*/
    linkedlist.add("Item1");
    linkedlist.add("Item5");
    linkedlist.add("Item3");
    linkedlist.add("Item6");
    linkedlist.add("Item2");

    /*Display Linked List Content*/
    System.out.println("Linked List Content: " +linkedlist);

    /*Add First and Last Element*/
```

```
        linkedlist.addFirst("First Item");
        linkedlist.addLast("Last Item");
        System.out.println("LinkedList Content after addition: " +linkedlist);

        /*This is how to get and set Values*/
        Object firstvar = linkedlist.get(0);
        System.out.println("First element: " +firstvar);
        linkedlist.set(0, "Changed first item");
        Object firstvar2 = linkedlist.get(0);
        System.out.println("First element after update by set method: " +firstvar2);

        /*Remove first and last element*/
        linkedlist.removeFirst();
        linkedlist.removeLast();
        System.out.println("LinkedList after deletion of first and last element: " +linkedlist);

        /* Add to a Position and remove from a position*/
        linkedlist.add(0, "Newly added item");
        linkedlist.remove(2);
        System.out.println("Final Content: " +linkedlist);
    }
}
```

Output:

Linked List Content: [Item1, Item5, Item3, Item6, Item2]
LinkedList Content after addition: [First Item, Item1, Item5, Item3, Item6, Item2, Last Item]
First element: First Item
First element after update by set method: Changed first item
LinkedList after deletion of first and last element: [Item1, Item5, Item3, Item6, Item2]
Final Content: [Newly added item, Item1, Item3, Item6, Item2]

3. Vector class:

- legacy class
- implements a growable array of objects.
- elements of Vector can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.
- maintains the insertion order which means it displays the elements in the same order, in which they got added to the Vector
- it is synchronized

Three ways to create vector class object:

- **Method 1:**

  Vector vec = new Vector();

It creates an empty Vector with the default initial capacity of 10. It means the Vector will be re-sized when the 11th elements needs to be inserted into the Vector.

- **Method 2:**

  Vector object= new Vector(int initialCapacity)

  Vector vec = new Vector(3);

  It will create a Vector of initial capacity of 3.

- **Method 3:**

  Vector object= new vector(int initialcapacity, capacityIncrement)

  Vector vec= new Vector(4, 6)

  Here we have provided two arguments. The initial capacity is 4 and capacityIncrement is 6. It means upon insertion of 5th element the size would be 10 (4+6) and on 11th insertion it would be 16(10+6).

Complete Example of Vector in Java:

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

public class VectorDemo {

  public static void main(String args[]) {
    /* Vector of initial capacity(size) of 2 */
    Vector<String> vec = new Vector<String>(2);


    /* Adding elements to a vector*/
    vec.addElement("Apple");
    vec.addElement("Orange");
    vec.addElement("Mango");
    vec.addElement("Fig");

    /* check size and capacityIncrement*/
    System.out.println("Size is: "+vec.size());
    System.out.println("Default capacity increment is: "+vec.capacity());

    vec.addElement("fruit1");
    vec.addElement("fruit2");
    vec.addElement("fruit3");

    /*size and capacityIncrement after two insertions*/
    System.out.println("Size after addition: "+vec.size());
    System.out.println("Capacity after increment is: "+vec.capacity());

    /*Display Vector elements*/
    Enumeration<String> en = vec.elements();
    System.out.println("\nElements are:");
    while(en.hasMoreElements())
```

```
        System.out.print(en.nextElement() + " ");

    }
}
```

Output:

Size is: 4
Default capacity increment is: 4
Size after addition: 7
Capacity after increment is: 8

Elements are:
Apple Orange Mango Fig fruit1 fruit2 fruit3

Difference between ArrayList and Vector :

ArrayList	Vector
ArrayList is **not synchronized**.	Vector is **synchronized**.
ArrayList is **not a legacy** class, it is introduced in JDK 1.2.	Vector is a **legacy** class.
ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or waiting state until current thread releases the lock of object.
ArrayList uses **Iterator** interface to traverse the elements.	Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also.

10. Set Interface and Classes:

1. HashSet:

- Unordered,
- no duplicate elements,
- uses hashtable

Points to Note about HashSet:

1. HashSet doesn't maintain any order, the elements would be returned in any random order.
2. HashSet doesn't allow duplicates. If you try to add a duplicate element in HashSet, the old value would be overwritten.
3. HashSet allows null values however if you insert more than one nulls it would still return only one null value.
4. HashSet is non-synchronized.

5. The iterator returned by this class is fail-fast which means iterator would throw ConcurrentModificationException if HashSet has been modified after creation of iterator, by any means except iterator's own remove method.

Example: using iterator

```
import java.util.HashSet;
import java.util.Iterator;

public class HashSetIteratorDemo {

        public static void main(String[] args) {

    // Create a HashSet
    HashSet<String> hset = new HashSet<String>();

    //add elements to HashSet
    hset.add("Chaitanya");
    hset.add("Rahul");
    hset.add("Tim");
    hset.add("Rick");
    hset.add("Harry");

    Iterator<String> it = hset.iterator();
    while(it.hasNext()){
      System.out.println(it.next());
    }
  }
}
```

Output:

```
Chaitanya
Rick
Harry
Rahul
Tim
```

Example: using enhanced for

```
import java.util.HashSet;

public class HashSetIteratorDemo {

        public static void main(String[] args) {

    // Create a HashSet
    HashSet<String> hset = new HashSet<String>();
```

```
    //add elements to HashSet
    hset.add("Chaitanya");
    hset.add("Rahul");
    hset.add("Tim");
    hset.add("Rick");
    hset.add("Harry");

    for (String temp : hset) {
       System.out.println(temp);
     }
  }
}
```

Output:
Chaitanya
Rick
Harry
Rahul
Tim

Example : to Print out all distinct elements from input.

```
public class HashSetDemo {

        public static void main(String[] args) {
                // TODO Auto-generated method stub

                Scanner sc=new Scanner(System.in);
                List<String> list=new ArrayList<String>();
                System.out.println("Enter String: ");
                for(int i=0;i<15;i++)
                {
                        list.add(sc.next());
                }
                System.out.println("List with duplicates:"+list);
                Set<String> set=new HashSet<String>(list);
                System.out.println("List without duplicates and unordered"+set);
        }
```

Output:

Enter String:
i
saw
he
saw
we

```
saw
they
saw
then
they
went
and
we
went
so
List with duplicates:[i, saw, he, saw, we, saw, they, saw, then, they, went, and, we, went, so]
List without duplicates and unordered[they, so, he, and, went, i, then, saw, we]
```

2. LinkedHashSet:

- implemented as hashtable with linked list running in it.
- follows insertion ordered

Example:

```
import java.util.LinkedHashSet;
import java.util.Set;

public class LinkedHashSetDemo {

   public static void main(String args[]) {
      // LinkedHashSet of String Type
      Set<String> lhset = new LinkedHashSet<String>();
      // Adding elements to the LinkedHashSet
      lhset.add("Z");
      lhset.add("PQ");
      lhset.add("N");
      lhset.add("O");
      lhset.add("KK");
      lhset.add("FGH");
      System.out.println(lhset);

      // LinkedHashSet of Integer Type
      Set<Integer> lhset2 = new LinkedHashSet<Integer>();

      // Adding elements
      lhset2.add(99);
      lhset2.add(7);
      lhset2.add(0);
      lhset2.add(67);
      lhset2.add(89);
      lhset2.add(66);
      System.out.println(lhset2);
```

```
    }
}
```

Output:
[Z, PQ, N, O, KK, FGH]
[99, 7, 0, 67, 89, 66]

11. SortedSet Interface and Classes:

- extends Set interface
- have a extra property that elements will be in sorted manner.

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();

    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

1. TreeSet:

- unordered (insertion unordered)
- sorted (alphabetically ordered)
- no duplicates
- slower than HashSet
- implements SortedSet interface

Example: Program to print the word list in alphabetical order.

```
public class TreeSetDemo {

        public static void main(String[] args) {
                // TODO Auto-generated method stub

                Scanner sc=new Scanner(System.in);
                List<String> list=new ArrayList<String>();
                System.out.println("Enter String: ");
                for(int i=0;i<15;i++) {
                        list.add(sc.next());
                }
```

```
        System.out.println("List with duplicates:"+list);
        Set<String> set=new TreeSet<String>(list);
        System.out.println("List without duplicates and sorted"+set);
    }
```

Set Interface Bulk Operations:

Bulk operations are particularly well suited to Sets; when applied, they perform standard set-algebraic operations. Suppose s1 and s2 are sets. Here's what bulk operations do:

- s1.containsAll(s2) — returns true if s2 is a **subset** of s1. (s2 is a subset of s1 if set s1 contains all of the elements in s2.)
- s1.addAll(s2) — transforms s1 into the **union** of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set.)
- s1.retainAll(s2) — transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.)
- s1.removeAll(s2) — transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

Example: The program to sort out non-duplicate and duplicate words

```
public class DuplicateNonduplicate {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Scanner sc=new Scanner(System.in);
        List<String> list=new ArrayList<String>();
        Set<String> set=new HashSet<String>(list);
        Set<String> dup=new HashSet<String>();
        System.out.println("Enter String: ");
        for(int i=0;i<15;i++) {
            list.add(sc.next());
        }
        System.out.println("List with duplicates:"+list);

        for(int i=0;i<list.size();i++) {
            if(!(set.add(list.get(i)))) {
                dup.add(list.get(i));
            }
        }
        set.removeAll(dup);
        System.out.println("List of non-duplicates"+set);

        System.out.println("List of duplicates:"+dup);
        sc.close();
```

```
        }
}
```

Difference between HashSet and TreeSet

- HashSet gives better performance (faster) than TreeSet for the operations like add, remove, contains, size etc. HashSet offers constant time cost while TreeSet offers log(n) time cost for such operations.
- HashSet does not maintain any order of elements while TreeSet elements are sorted in ascending order by default.

12. Queue Interface and classes:

- FIFO
- elements are insterted at end and removed from first
- extra operations: insertion - add(e) , removal – remove() , inspection – element()

```
public interface Queue<E> extends Collection<E> {
   E element();
   boolean offer(E e);
   E peek();
   E poll();
   E remove();
}
```

Queue Interface Structure:

Type of Operation	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Priority queues: order the elements according to thier values
Bounded queues: restricts the number of elements that it holds, java.util.concurrent do not allow insertion of null**.**

1. PriorityQueue class:

- implements Queue interface
- does not permit null elements
- it is unbounded
- not synchronized

Example:

```
public class PriorityQueueDemo {

	public static void main(String[] args) {

		Scanner sc=new Scanner(System.in);
		ArrayList<Integer> values=new ArrayList<Integer>();
		System.out.println("Enter elements:");
		for(int i=0;i<5;i++)
		{
			values.add(sc.nextInt());
		}
		Queue<Integer> pq=new PriorityQueue<Integer>(values);
		System.out.println(pq);
		sc.close();
	}
}
```

In the following example program, a queue is used to implement a countdown timer.

```
import java.util.*;

public class QueueDemo {
   public static void main(String[] args) throws InterruptedException {

      Scanner sc= new Scanner(System.in);
      System.out.println("Enter timer value: ");
     int time =sc.nextInt();

     Queue<Integer> queue = new LinkedList<Integer>();

     for (int i = time; i >= 0; i--)
        queue.add(i);

     while (!queue.isEmpty()) {
        System.out.println(queue.remove());
        Thread.sleep(1000);
     }
    sc.close();
   }
}
```

13. Deque Interface and Classes:

- double ended queue
- insertion and removal from both ends
- implements both stack and queue at the same time
- duplicates allowed

The methods provided by Deque:

Insertion:
- addFirst, offerFirst
- addLast, offerLast

Removal:
- removeFirst, pollFirst
- removeLast, pollLast
- **removeFirstOccurence:** removes the first occurance of specified element if exists.
- **removeLastOccurence:** removes the Last occurance of specified element if exists. The return type is boolean

Retrieve:
- getFirst, peekFirst
- getLast, peekLast

1. LinkedList class:

- impelements List, Queue and Deque interfaces.

Example: followiong example shows the demo of Deque.

```
public class DequeueDemo {

	public static void main(String[] args) {
		Scanner sc=new Scanner(System.in);
		Deque<Integer> dq=new LinkedList<Integer>();
		System.out.println("Enter elements to add at first");
		for(int i=0;i<5;i++)
		{
			dq.addFirst(sc.nextInt());
		}
		System.out.println("Enter elements to add at last");
		for(int i=0;i<5;i++)
		{
			dq.addLast(sc.nextInt());
		}
		System.out.println(dq);
		System.out.println("Remove the element from head");
		dq.removeFirst();
		System.out.println(dq);
		System.out.println("Remove the element from trail");
		dq.removeLast();
		System.out.println(dq);
		System.out.println("Enter the element to remove its first occurance.");
		dq.removeFirstOccurrence(sc.nextInt());
```

```
                System.out.println(dq);
                System.out.println("Enter the element to remove its last occurance.");
                dq.removeLastOccurrence(sc.nextInt());
                System.out.println(dq);
        }
}
```

Output:

```
Enter elements to add at first
2
6
2
7
3
Enter elements to add at last
7
4
8
5
6
[3, 7, 2, 6, 2, 7, 4, 8, 5, 6]
Remove the element from head
[7, 2, 6, 2, 7, 4, 8, 5, 6]
Remove the element from trail
[7, 2, 6, 2, 7, 4, 8, 5]
Enter the element to remove its first occurance.
8
[7, 2, 6, 2, 7, 4, 5]
Enter the element to remove its last occurance.
7
[7, 2, 6, 2, 4, 5]
```

List Vs Set:

- List is an ordered collection it maintains the insertion order, which means upon displaying the list content it will display the elements in the same order in which they got inserted into the list. Set is an unordered collection, it doesn't maintain any order. There are few implementations of Set which maintains the order such as LinkedHashSet (It maintains the elements in insertion order).
- List allows duplicates while Set doesn't allow duplicate elements. All the elements of a Set should be unique if you try to insert the duplicate element in Set it would replace the existing value.
- List allows any number of null values. Set can have only a single null value at most.
- ListIterator can be used to traverse a List in both the directions(forward and backward)

However it can not be used to traverse a Set. We can use Iterator (It works with List too) to traverse a Set.

- List interface has one legacy class called Vector whereas Set interface does not have any legacy class.

14. Map Interfaces and Classes:

- maps keys to values
- can not contain duplicate keys
- key maps to atmost one value

Methods:
basic operations : put, get, remove, containskey, containsValue, size, empty
bulk operations : putAll, clear
collection views : keySet, entrySet, values

Collection Views:

The Collection view methods allow a Map to be viewed as a Collection in these three ways:

- keySet : the Set of keys contained in the Map.
- values : The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.
- entrySet : the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry, the type of the elements in this Set.

The Collection views provide the only means to iterate over a Map.

1. To iterate keyset:

Using for:

```
for (KeyType key : m.keySet())
   System.out.println(key);
```

Using iterator:

```
// Filter a map based on some
// property of its keys.
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )
   if (it.next().isBogus())
      it.remove();
```

2. To iterate entryset (key and value):

Using for:

```
for (Map.Entry<KeyType, ValType> e : m.entrySet())
   System.out.println(e.getKey() + ": " + e.getValue());
```

Using iterator:

```
Iterator<Entry<String, Integer>> it=stdlist.entrySet().iterator();

while(it.hasNext())
   {
        Entry<String, Integer> mp=it.next();
        String key=mp.getKey();
        System.out.println(key);
        int val=mp.getValue();
        System.out.println(val);
   }
```

1. HashMap:

- unordered,
- no duplicates
- uses hashtable to store keys
- It is similar to the Hashtable class except that it is unsynchronized and permits nulls.
- It is used for maintaining key and value mapping.

Example: group students by passing or failing grades:

```
public class HashMapDemo {

        public static void main(String[] args) {
                Scanner sc=new Scanner(System.in);
                String grade;

                Map<String,Integer> stdlist=new HashMap<String, Integer>();
                int passp=40;
                Map<String,String> gradelist=new HashMap<String, String>();
                System.out.println("Enter Student ID and percetange rerpectively");

                for(int i=0;i<5;i++) {
                        stdlist.put(sc.next(), sc.nextInt());
                 }

                System.out.println(stdlist);

                Iterator<Entry<String, Integer>> it=stdlist.entrySet().iterator();
```

```
                while(it.hasNext())
                {
                        Entry<String, Integer> mp=it.next();
                        String key=mp.getKey();
                        System.out.println(key);
                        int val=mp.getValue();
                        System.out.println(val);

                        if(val>=passp)
                                grade="pass";
                        else
                                grade="fail";

                         gradelist.put(key, grade);

                }
            System.out.println(gradelist);
            sc.close();
        }
}
```

Output:

```
Enter Student ID and percetange rerpectively
101
40
102
39
103
78
104
90
105
65
{105=65, 104=90, 103=78, 102=39, 101=40}
105
65
104
90
103
78
102
39
101
40
{105=pass, 104=pass, 103=pass, 102=fail, 101=pass}
```

2. LinkedHashMap:

- follows the insertion order of keys
- implements hashtable which has linkedlist running through it.
- Not synchronized

Example:

```
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Map.Entry;

public class LinkedHashMapDemo {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Map<String, Integer> stdmap=new LinkedHashMap<String, Integer>();

		stdmap.put("A105",70);
		stdmap.put("A102",65);

		stdmap.put("A101",75);
		stdmap.put("A104",56);
		stdmap.put("A106",78);
		stdmap.put("A103",40);


		Iterator<Entry<String, Integer>> itstdmap=stdmap.entrySet().iterator();
		while(itstdmap.hasNext()) {
			Entry<String, Integer> entry=itstdmap.next();
			System.out.println(entry.getKey()+" "+entry.getValue());
		}
	}
}
```

Output:

```
A105 70
A102 65
A101 75
A104 56
A106 78
A103 40
```

15. SortedMap Interface and Classes:

1. TreeMap:

- orders the elements based on their keys (sorting)
- implements SortedMap interface
- not synchronized

Example:
Following program sorts the TreeMap elements on key

```
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;
import java.util.TreeMap;

public class TreeMapDemo {

	public static void main(String[] args) {

		Map<String, Integer> stdmap=new TreeMap<String, Integer>();

		stdmap.put("A105",70);
		stdmap.put("A102",65);
		stdmap.put("A101",75);
		stdmap.put("A104",56);
		stdmap.put("A106",78);
		stdmap.put("A103",40);


		Iterator<Entry<String, Integer>> itstdmap=stdmap.entrySet().iterator();
		while(itstdmap.hasNext()) {
			System.out.println(itstdmap.next());
		}
	}
}
```

Output:

```
A101=75
A102=65
A103=40
A104=56
A105=70
A106=78
```

16. Collection Interfaces used for Comparison:

16.1. Comparable Interface:

- resides in lang package
- used to sort the list of Comparable objects

- sorting in their natural ordering
- by using this interface, collection can be sorted only on single entity at a time.
- It must be implemented by the class on which we need sorting
- It has a method compareTo for sort
  int compareTo(Object obj) : used to compare current object with the specified object
- to sort the collection using Comparable, we need to pass one parameter i.e. object of collection to sort method. **Collections.sort(collectionobject);**

Syntax of Comparable:

```
public interface Comparable<E>
{
        int compareTo(Object obj);
}
```

Example:

Following program sorts the employee object on the basis of Salary of the employee.

Employee.java:

```
public class Employee  implements Comparable<Employee>{
        private int empid;
        private String empname;
        private int empsalary;
        private int empexperience;

        public Employee(int empid, String empname, int empsalary, int empexperience) {

                this.empid = empid;
                this.empname = empname;
                this.empsalary = empsalary;
                this.empexperience = empexperience;
        }
        public int getEmpid() {
                return empid;
        }
        public void setEmpid(int empid) {
                this.empid = empid;
        }
        public String getEmpname() {
                return empname;
        }
        public void setEmpname(String empname) {
                this.empname = empname;
        }
```

```
	public int getEmpsalary() {
		return empsalary;
	}
	public void setEmpsalary(int empsalary) {
		this.empsalary = empsalary;
	}
	public int getEmpexperience() {
		return empexperience;
	}
	public void setEmpexperience(int empexperience) {
		this.empexperience = empexperience;
	}

	@Override
	public String toString() {
		return "Employee [empid=" + empid + ", empname=" + empname
				+ ", empsalary=" + empsalary + ", empexperience="
				+ empexperience + "]";
	}

	@Override
	public int compareTo(Employee e) {
		return getEmpsalary()-e.getEmpsalary();
	}

}
```

ComparableDemo.java:

```
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class ComparableDemo {

	public static void main(String[] args) {

		List<Employee> emplist=new LinkedList<Employee>();
		emplist.add(new Employee(101,"Suhas Jagtap",45000,5));
		emplist.add(new Employee(102,"Amol Jagdale",55000,7));
		emplist.add(new Employee(103,"Kirti Mandel",42000,4));
		emplist.add(new Employee(104,"Johnson Tumer",35000,3));
		emplist.add(new Employee(105,"Somayya Khan",65000,8));
		emplist.add(new Employee(106,"Kishan Pandit",25000,2));
		emplist.add(new Employee(107,"Chanki Pande",38000,3));

		Collections.sort(emplist);
```

```
                System.out.println("After sorting on Salary basis");

                Iterator<Employee> empit=emplist.iterator();

                while(empit.hasNext())
                {
                        System.out.println(empit.next());
                }
        }
}
```

Output:

```
After sorting on Salary basis
Employee [empid=106, empname=Kishan Pandit, empsalary=25000, empexperience=2]
Employee [empid=104, empname=Johnson Tumer, empsalary=35000, empexperience=3]
Employee [empid=107, empname=Chanki Pande, empsalary=38000, empexperience=3]
Employee [empid=103, empname=Kirti Mandel, empsalary=42000, empexperience=4]
Employee [empid=101, empname=Suhas Jagtap, empsalary=45000, empexperience=5]
Employee [empid=102, empname=Amol Jagdale, empsalary=55000, empexperience=7]
Employee [empid=105, empname=Somayya Khan, empsalary=65000, empexperience=8]
```

16.2. Comparator Interface:

- Resides in util package
- To sort the collection on the basis of multiple entities.
- No need to change the existing class. New classes are created those implement Comparable interface
- It has the following method to compare two objects.
  public int compare(Object obj1,Object obj2): compares the first object with second object.
- To sort the collection we need to pass two parameters, i.e. collection object and class object implemening Comparator, to the sort method. **Collections.sort(collectionobject, comparatorobject)**

Syntax of Comparator:

```
public interface Comparator<E>
{
        int compare(Object obj1, Object obj2);
}
```

Example:

Employee.java : (existing class Employeee will not get modified)

```
public class Employee{
	private int empid;

	private String empname;
	private int empsalary;
	private int empexperience;

	public Employee(int empid, String empname, int empsalary, int empexperience) {

		this.empid = empid;
		this.empname = empname;
		this.empsalary = empsalary;
		this.empexperience = empexperience;
	}
	public int getEmpid() {
		return empid;
	}
	public void setEmpid(int empid) {
		this.empid = empid;
	}
	public String getEmpname() {
		return empname;
	}
	public void setEmpname(String empname) {
		this.empname = empname;
	}
	public int getEmpsalary() {
		return empsalary;
	}
	public void setEmpsalary(int empsalary) {
		this.empsalary = empsalary;
	}
	public int getEmpexperience() {
		return empexperience;
	}
	public void setEmpexperience(int empexperience) {
		this.empexperience = empexperience;
	}

	@Override
	public String toString() {
		return "Employee [empid=" + empid + ", empname=" + empname
				+ ", empsalary=" + empsalary + ", empexperience="
				+ empexperience + "]";
	}

}
```

EmployeeExperienceComparator.java:

```
import java.util.Comparator;

public class EmployeeExperienceComparator implements Comparator<Employee>{

	@Override
	public int compare(Employee e1, Employee e2) {
		return e1.getEmpexperience()-e2.getEmpexperience();

	}
}
```

EmployeeSalaryComparator.java:

```
public class EmployeeSalaryComparator implements Comparator<Employee>
{
	@Override
	public int compare(Employee e1, Employee e2) {
		// TODO Auto-generated method stub
		return e1.getEmpsalary()-e2.getEmpsalary();
	}
}
```

ComparatorDemo.java :

```
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class ComparatorDemo {

	public static void main(String[] args) {
		List<Employee> emplist=new LinkedList<Employee>();
		emplist.add(new Employee(101,"Suhas Jagtap",45000,5));
		emplist.add(new Employee(102,"Amol Jagdale",66000,7));
		emplist.add(new Employee(103,"Kirti Mandel",42000,4));
		emplist.add(new Employee(104,"Johnson Tumer",43000,3));
		emplist.add(new Employee(105,"Somayya Khan",65000,8));
		emplist.add(new Employee(106,"Kishan Pandit",25000,2));
		emplist.add(new Employee(107,"Chanki Pande",38000,3));

		Collections.sort(emplist,new EmployeeExperienceComparator());

		System.out.println("After sorting on Experience basis");
```

```
		Iterator<Employee> empit=emplist.iterator();


		while(empit.hasNext()) {
			System.out.println(empit.next());
		}

		Collections.sort(emplist,new EmployeeSalaryComparator());
		System.out.println("After sorting on Salary basis");

		empit=emplist.iterator();

		while(empit.hasNext()) {
			System.out.println(empit.next());
		}
	}
}
```

Difference between Comparable and Comparator:

Comparable	Comparator
Comparable provides single sorting sequence. In other words, we can sort the collection on the basis of single element such as id or name or price etc.	Comparator provides multiple sorting sequence. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc.
Comparable affects the original class i.e. actual class is modified.	Comparator doesn't affect the original class i.e. actual class is not modified.
Comparable provides compareTo() method to sort elements.	Comparator provides compare() method to sort elements.
Comparable is found in java.lang package.	Comparator is found in java.util package.
We can sort the list elements of Comparable type by Collections.sort(List) method.	We can sort the list elements of Comparator type by Collections.sort(List,Comparator) method.

17. Summary:

The core collection interfaces are the foundation of the Java Collections Framework.

The Java Collections Framework hierarchy consists of two distinct interface trees:

The first tree starts with the Collection interface, which provides for the basic functionality used by all collections, such as add and remove methods. Its subinterfaces — Set, List, and Queue — provide for more specialized collections.

- The Set interface does not allow duplicate elements. This can be useful for storing collections such as a deck of cards or student records. The Set interface has a subinterface, SortedSet, that provides for ordering of elements in the set.

- The List interface provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position.
- The Queue interface enables additional insertion, extraction, and inspection operations. Elements in a Queue are typically ordered in on a FIFO basis.
- The Deque interface enables insertion, deletion, and inspection operations at both the ends. Elements in a Deque can be used in both LIFO and FIFO.

The second tree starts with the Map interface, which maps keys and values similar to a Hashtable.

- Map's subinterface, SortedMap, maintains its key-value pairs in ascending order of key or in an order specified by a Comparator or Comparable.

These interfaces allow collections to be manipulated independently of the details of their representation.

ASSIGNMENTS

1. Implement the list of User accounts. Perform following actions by taking user input
   1. Adding new account
   2. Updating existing account
   3. Deleting existing account
   4. Searching particular account
   5. Displaying all accounts
   6. Depositing money in particular account
   7. Withdrawing money from particular account

   You can use ArrayList or LinkedList
2. Sort the above user accounts on account id and account name.
3. Collect the words and their meanings. Perform following actions by taking user input.
   1. Adding new word with its meaning
   2. Updating meaning of existing word
   3. Deleting existing word
   4. Searching meaning of particular word
   5. Displaying all word records.
   6. Sorting the words in ascending order

   Traverse the collection first on keys then on key-value pair
4. Perform the bulk operation on two ArrayLists which contain Integers.
5. Implement a program of Map which does not allow duplicate values.
6. Implement the program which extracts the keys from Map and stores into Set.