Module 2: The collection Frame work

1.Introduction to Colections

- A *colection* — sometimes caled a container — is simply an object that groups multiple elements into a single unit.
- Colections are used to store, retrieve, manipulate, and communicate aggregate data.
- Typicaly, they represent data items that form a natural group, such as a poker hand (a colection of cards), a mail folder (a colection of letters), or a telephone directory (a mapping of names to phone numbers).

2.What Is a Colections Framework?

A *colections framework* is a unified architecture for representing and manipulating colections. Al colections frameworks contain the folowing:

- **Interfaces:** These are abstract data types that represent colections. Interfaces alow colections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generaly form a hierarchy.
- **Implementations:** These are the concrete implementations of the colection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement colection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate colection interface. In essence, algorithms are reusable functionality.

Benefits of the Java Colections Framework

The Java Colections Framework provides the folowing benefits:

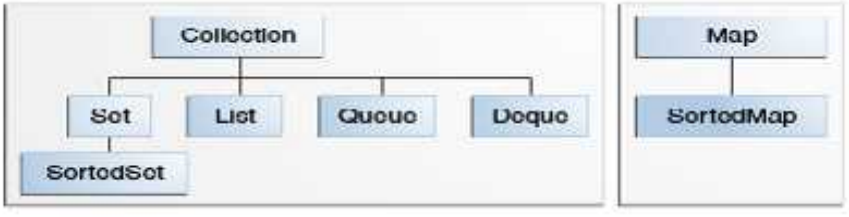
- **Reduces programming effort:** By providing useful data structures and algorithms, the Colections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Colections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality:** This Colections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily

tuned by switching colection implementations. Because you're freed from the drudgery of writing your own data structures, you'l have more time to devote to improving programs' quality and performance.

- **Alows interoperability among unrelated APIs:** The colection interfaces are the vernacular by which APIs pass colections back and forth. If my network administration API furnishes a colection of node names and if your GUI toolkit expects a colection of column headings, our APIs wil interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturaly take colections on input and furnish them as output. In the past, each such API had a smal sub-API devoted to manipulating its colections. There was little consistency among these ad hoc colections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard colection interfaces, the problem went away.
- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on colections; instead, they can use standard colection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard colection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces

2.colection interfaces:

- *The* core colection interfaces *encapsulate different types of colections, which are shown in the figure below.*
- *These interfaces alow colections to be manipulated independently of the details of their representation. Core colection interfaces are the foundation of the Java Colections Framework.*



- The core colection interfaces,The colections framework defines several interfaces. This section provides an overview of each interface −

Sr.No.	Interface & Description
1	The Colection Interface This enables you to work with groups of objects; it is at the top of the colections hierarchy.
2	The List Interface This extends **Colection** and an instance of List stores an ordered colection of elements.
3	The Set This extends Colection to handle sets, which must contain unique elements.
4	The SortedSet This extends Set to handle sorted sets.
5	The Map This maps unique keys to values.
6	The Map.Entry This describes an element (a key/value pair) in a map. This is an inner class of Map.
7	The SortedMap This extends Map so that the keys are maintained in an ascending order.
8	The Enumeration This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a colection of objects. This legacy interface has been superceded by Iterator.

3.1Colection Interface:

The Colection interface is the foundation upon which the colections framework is built.Colection is generic interface ,this is the declaration of the Colection interface.

public interface Colection<E>...

The <E> syntax tels you that the interface is generic.

Colection extends the iterable interface. This means that al colection can perform the perform the operation through For Each loop style. Different methods in colection interface are:

Sr.No.	Method & Description
1	**boolean add(Object obj)** Adds obj to the invoking colection. Returns true if obj was added to the colection. Returns false if obj is already a member of the colection, or if the colection does not alow duplicates.
2	**boolean addAl(Colection c)** Adds al the elements of c to the invoking colection. Returns true if the operation succeeds (i.e., the elements were added). Otherwise, returns false.
3	**void clear( )** Removes al elements from the invoking colection.
4	**boolean contains(Object obj)** Returns true if obj is an element of the invoking colection. Otherwise, returns false.
5	**boolean containsAl(Colection c)** Returns true if the invoking colection contains al elements of **c**. Otherwise, returns false.
6	**boolean equals(Object obj)** Returns true if the invoking colection and obj are equal. Otherwise, returns false.
7	**int hashCode( )** Returns the hash code for the invoking colection.
8	**boolean isEmpty( )** Returns true if the invoking colection is empty. Otherwise, returns false.
9	**Iterator iterator( )** Returns an iterator for the invoking colection.
10	**boolean remove(Object obj)** Removes one instance of obj from the invoking colection. Returns true if the element was removed. Otherwise, returns false.
11	**boolean removeAl(Colection c)**

	Removes al elements of c from the invoking colection. Returns true if the colection changed (i.e., elements were removed). Otherwise, returns false.
12	**boolean retainAl(Colection c)** Removes al elements from the invoking colection except those in c. Returns true if the colection changed (i.e., elements were removed). Otherwise, returns false.
13	**int size( )** Returns the number of elements held in the invoking colection.
14	**Object[ ] toArray( )** Returns an array that contains al the elements stored in the invoking colection. The array elements are copies of the colection elements.
15	**Object[ ] toArray(Object array[ ])** Returns an array containing only those colection elements whose type matches that of array

Several of these methods can throw an ***UnsupportedOperationException*** this occurs if a collection cannot be modified. A**ClassCastException** is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a colection. A **NulPointerException** is thrown if an attempt is made to store a **nul** object and **nul** elements are not alowed in the colection. An **IlegalArgumentException** is thrown if an invalid argument is used. An **IlegalStateException** is thrown if an attempt is made to add an element to a fixed-length colection that is ful.

Program:

```
import java.util.*;
public class A {
        public static void main(String ar[]){
                Colection<String> s=new HashSet<String>();
                s.add("a");
                s.add("d");
                s.add("c");
                System.out.println(" elements Are in colection="+s);
                System.out.println(" Size of colection is="+s.size());
```

```
                Colection<String> s1=new HashSet<String>();
                s1.addAl(s);
                System.out.println(" Afetr adding alelements in colection="+s1);
                s.remove("a");
                System.out.println("REmoved elements from colection="+s);

        }
}
```

output:

```
elements Are in colection=[d, c, a]
Size of colection is=3
Afetr adding alelements in colection=[d, c, a]
REmoved elements from colection=[d, c]
```

3.2. List interface:

The List interface extends **Colection** and declares the behavior of a colection 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 **Colection**, List defines some of its own, which are summarized in the folowing table.
- Several of the list methods wil throw an UnsupportedOperationException if the colection cannot be modified, and a ClassCastException is generated when one object is incompatible with another. this is the declaration of the Colection interface.

public interface List<E>...

- The <E> specifies the type of object that the list wil hold.

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 addAl(int index, Colection c)** Inserts al 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 colection.
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.

Program:

```
import java.util.*;
public class set {
        public static void main(String ar[]){
        List<String> s=new ArrayList<String>();
                s.add("a");
                s.add("d");
```

```
            s.add("c");
            System.out.println(" elements Are in colection="+s);
            System.out.println(" Size of colection is="+s.size());
            System.out.println(" Obtaining element form index="+s.get(1));
            System.out.println(" Obtaining index position of element="+s.indexOf("c"));
            List<String> s1=new ArrayList<String>();
            s1.addAl(s);
            System.out.println(" Afetr adding alelements in colection="+s1);
            s.remove("a");
            System.out.println("REmoved elements from colection="+s);

        }
}
```

output:

```
elements Are in colection=[a, d, c]
 Size of colection is=3
 Obtaining element form index=d
 Obtaining index position of element=2
 Afetr adding alelements in colection=[a, d, c]
REmoved elements from colection=[d, c]
```

3.3.Set Interface:

- A Set is a Colection that cannot contain duplicate elements. It models the mathematical set abstraction.
- The Set interface contains only methods inherited from Colection and adds the restriction that duplicate elements are prohibited.
- Set also adds a stronger contract on the behavior of the equals and hashCode operations, alowing Set instances to be compared meaningfuly even if their implementation types differ.
- this is the declaration of the Colection interface.

  public interface List<E>...
- The <E> specifies the type of object that the list wil hold.
- The methods declared by Set are summarized in the folowing table −

Sr.No.	Method & Description
1	**add( )** Adds an object to the colection.

2	**clear( )** Removes al objects from the colection.
3	**contains( )** Returns true if a specified object is an element within the colection.
4	**isEmpty( )** Returns true if the colection has no elements.
5	**iterator( )** Returns an Iterator object for the colection, which may be used to retrieve an object.
6	**remove( )** Removes a specified object from the colection.
7	**size( )** Returns the number of elements in the colection.

program:

```
import java.util.*;
public class set {
	public static void main(String ar[]){
		Set<String> s=new HashSet<String>();
		s.add("a");
		s.add("d");
		s.add("c");
		System.out.println(" elements Are in colection="+s);
		System.out.println(" Size of colection is="+s.size());
		Set<String> s1=new HashSet<String>();
		s1.addAl(s);
		System.out.println(" Afetr adding alelements in colection="+s1);
		s.remove("a");
		System.out.println("REmoved elements from colection="+s);
		System.out.println("Is element are present in Set="+s1.contains("c"));

	}

}
```

output:

```
elements Are in colection=[d, c, a]
 Size of colection is=3
 Afetr adding alelements in colection=[d, c, a]
REmoved elements from colection=[d, c]
Is element are present in Set=true
```

4.4.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, 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 NulPointerException is thrown if an attempt is made to use a nul object and nul is not alowed in the set.
- SortedSet is a generic interface that has this declaration:
    - **interface SortedSet<E>**
- <E> Specifies the type of object that the set wil hold.
- the SortedSet interface declares the methods summarized in the folowing table −

Sr.No.	Method & Description
1	**Comparator comparator( )** Returns the invoking sorted set's comparator. If the natural ordering is used for this set, nul 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 colection 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.

Program:

```
public class A {
        public static void main(String ar[]){
                SortedSet<String> s=new TreeSet<String>();
                s.add("a");
                s.add("d");
                s.add("c");
                System.out.println(" elements Are in colection="+s);
                System.out.println(" Size of colection is="+s.size());
                System.out.println("The first element in list="+s.first());
                System.out.println("The Last element in list="+s.last());
                System.out.println("The HeadSet elements in list="+s.headSet("c"));
                System.out.println("The TailSet elements in list="+s.tailSet("c"));
        }

}
```

output:

elements Are in colection=[a, c, d]
Size of colection is=3
The first element in list=a
The Last element in list=d
The HeadSet elements in list=[a]
The TailSet elements in list=[c, d]

4.5: **Queue Interface**

- A Queue is a colection for holding elements prior to processing. Besides basic Colection operations, queues provide additional insertion, removal, and inspection operations. The Queue interface folows.

public interface Queue<E>

<E> Specifies the type of object that the set wil hold.

the SortedSet interface declares the methods summarized in the folowing table:

Method	Description
E element( )	Returns the element at the head of the queue. The element is not removed. It throws NoSuchElementException if the queue is empty.
boolean offer(E obj)	Attempts to add obj to the queue. Returns true if obj was added and false otherwise.
E peek( )	Returns the element at the head of the queue. It returns nul if the queue is empty. The element is not removed.

E pol( ) Returns the element at the head of the queue, removing the element in the process. It returns nul if the queue is empty.
E remove( ) Removes the element at the head of the queue, returning the element in the process. It throws NoSuchElementException if the queue is empty.

Several methods throw a **ClassCastException** when an object is incompatible with the elements in the queue. A **NulPointerException** is thrown if an attempt is made to store a **nul** object and **nul** elements are not alowed in the queue. An **IlegalArgumentException** is thrown if an invalid argument is used. An **IlegalStateException** is thrown if an attempt is made to add an element to a fixed-length queue that is ful. A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty queue.

Note : class Program

4.6 Deque Interface

- The **Deque** interface was added by Java SE 6. It extends **Queue** and declares the behavior of a double-ended queue. Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks. **Deque** is a generic interface that has this declaration:

interface Deque<E>

- Here, **E** specifies the type of objects that the deque wil hold. In addition to the methods that it inherits from **Queue**, **Deque** adds those methods summarized in Table:

Method	**Description**
boolean add(object)	It is used to insert the specified element into this deque and return true upon success.
boolean offer(object)	It is used to insert the specified element into this deque.
Object remove()	It is used to retrieves and removes the head of this deque.
Object pol()	It is used to retrieves and removes the head of this deque, or returns nul if this deque is empty.

Object element()	It is used to retrieves, but does not remove, the head of this deque.
Object peek()	It is used to retrieves, but does not remove, the head of this deque, or returns nul if this deque is empty.

program:

```
import java.util.*;
public class A {
   public static void main(String[] args) {
   //Creating Deque and adding elements
   Deque<String> d = new ArrayDeque<String>();
   d.add("A");
   d.add("B");
   d.add("C");
   //Traversing elements
   for (String str : deque) {
   System.out.println(str);
   }
   }
}
```

output:

A B C

4.The Colection Classes

Java provides a set of standard colection classes that implement Colection interfaces. Some of the classes provide ful implementations that can be used as-is and others are abstract class

The standard colection classes are summarized in the folowing table –

Sr.No.	***Class & Description***
1	**AbstractColection** Implements most of the Colection interface.
2	**AbstractList** Extends AbstractColection and implements most of the List interface.

3	**AbstractSequentialist** Extends AbstractList for use by a colection that uses sequential rather than random access of its elements.
4	*LinkedList* Implements a linked list by extending AbstractSequentialist.
5	*ArrayList* Implements a dynamic array by extending AbstractList.
6	**AbstractSet** Extends AbstractColection and implements most of the Set interface.
7	*HashSet* Extends AbstractSet for use with a hash table.
8	*LinkedHashSet* Extends HashSet to alow 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	AbstractQueue Extends AbstractColection and implements parts of the queue interface

4.1ArrayList class:

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

class ArrayList<E>

<E> specifies the type of object .

constructors provided by the ArrayList class.

ArrayList( )

ArrayList(Colection<? extends E> *c*)

ArrayList(int *capacity*)

The first constructor builds an empty array list. The second constructor builds an array list that is initialized with the elements of the colection *c*. The third 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 automaticaly as elements are added to an array list.

Program:

```
public class A{
  public static void main(String args[])
{
     ArrayList<String> al = new ArrayList<String> ();
    System.out.println("Initial size of al: " + al.size());
    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());
    System.out.println("Contents of al: " + al);
    al.remove("F");
    al.remove(2);
    System.out.println("Size of al after deletions: " + al.size());
    System.out.println("Contents of al: " + al);
  }
}
```

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]

Obtaining an Arrya from An ArrayList:

When working with **ArrayList**, you wil sometimes want to obtain an actual array that contains the contents of the list. You can do this by caling **toArray( )**, which is defined by **Colection**. Several reasons exist why you might want to convert a colection into an array, such as:

- •To obtain faster processing times for certain operations
- •To pass an array to a method that is not overloaded to accept a colection
- •To integrate colection-based code with legacy code that does not understand collections

Object[ ] toArray( )

<T> T[ ] toArray(T *array*[ ])

program:

```
class A{
 public static void main(String args[]){
  ArrayList<Integre> al=new ArrayList<Integer>();
  al.add(10);
  al.add(20);
  al.add(30);
Integer i=new Integer[al.size()];
i=al.toArrya(i);
int sum=0,count=0;
  for(int a:i)  {
sum+=i;
count++;
}
int avg=sum/count;
   System.out.println("Avg="+avg);
 }
}
```

output:

Avg=20

5.2 :LinkedList class

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

class LinkedList<E>

<E> specifies the type of object .

- constructors provided by the ArrayList class.

LinkedList( )

LinkedList(Colection<? extends E> *c*)

The first constructor builds an empty linked list. The second constructor builds a linked list that is initialized with the elements of the colection *c.*

Program:

```
public class A {
  public static void main(String args[]) {
    LinkedList<String> l = new LinkedList<String>();
    l.add("F");
    l.add("B");
    l.add("D");
    l.add("E");
    l.add("C");
    l.addLast("Z");
    l.addFirst("A");
    l.add(1, "A2");
    System.out.println("Original contents of l: " + l);
    l.remove("F");
    l.remove(2);
    System.out.println("Contents of l after deletion: " + l);
    l.removeFirst();
    l.removeLast();
    System.out.println("l after deleting first and last: " + l);
    Object val = l.get(2);
    l.set(2, (String) val);
    System.out.println("l after change: " + l);
  }
}
```

output:

Original contents of list: A, A2, F, B, D, E, C, Z

Contents of list after deletion: A, A2, D, E, C, Z

list after deleting first and last: A2, D, E, C

list after change: A2, D, E C

4.5 HashSet:

- HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.
- A hash table stores information by using a mechanism called **hashing**.
- **HashSet** is a generic class that has this declaration:

class HashSet<E>

*Here, **E** specifies the type of objects that the set will hold.*

The following constructors are defined:

HashSet( )

HashSet(Collection<? extends E> *c*)

HashSet(int *capacity*)

HashSet(int* capacity, *float* fillRatio*)

The first form constructs a default hash set. The second form initializes the hash set by using the elements of *c.* The third form initializes the capacity of the hash set to *capacity.* The fourth form initializes both the capacity and the fill ratio (also called *load capacity*) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0.

program:

```
public class A {
  public static void main(String args[]) {
    HashSet<String>  hs = new HashSet<String> ();
    hs.add("B");
    hs.add("A");
    hs.add("D");
    hs.add("E");
    hs.add("C");
```

```
    hs.add("F");
    System.out.println("Elements in hashset "+hs);
  }
}
```

output:

Elements in hashset : A, B, C, D, E, F

4.6 LinkedHashSet:

- This class extends HashSet, but adds no members of its own.
- LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set.It is a generic class that has this declaration:

class LinkedHashSet<E>

Here, **E** specifies the type of objects that the set will hold. Its constructors parallel those in **HashSet**.

program:

```
public class A {
  public static void main(String args[]) {
    LinkedHashSet<String> hs = new LinkedHashSet<String>();
    hs.add("B");
    hs.add("A");
    hs.add("D");
    hs.add("E");
    hs.add("C");
    hs.add("F");
    System.out.println("Elements in Linkedhashset "+hs);
  }
}
```

output:

Elements in Linkedhashset : B, A, D, E, C, F

4.6 TreeSet:

- TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in a sorted and ascending order.
- Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.
- **TreeSet** is a generic class that has this declaration:

class TreeSet<E>

Here, **E** specifies the type of objects that the set will hold.

TreeSet has the following constructors:

TreeSet( )

TreeSet(Collection<? extends E> *c*)

TreeSet(Comparator<? super E> *comp*)

TreeSet(SortedSet<E>* ss*)

The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements. The second form builds a tree set that contains the elements of *c*. The third form constructs an empty tree set that will be sorted according to the comparator specified by *comp*. (Comparators are described later in this chapter.) The fourth form builds a tree set that contains the elements of *ss*.

program:

```
public class A {
  public static void main(String args[]) {
    TreeSet<String> ts = new TreeSet<String>();
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");
    System.out.println("Element in TreeSet="+ts);
 }}
```

output:*Element in TreeSet =A, B, C, D, E, F*

4.7 PriorityQueue:

- PriorityQueue extends AbstractQueue and implements the queue interface.
- The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.
- **PriorityQueue** is a generic class that has this declaration:

class PriorityQueue<E>

Here, **E** specifies the type of objects stored in the queue. **PriorityQueue**s are dynamic, growing as necessary. **PriorityQueue** defines the six constructors shown here:

PriorityQueue( )

PriorityQueue(int *capacity*)

PriorityQueue(int *capacity*, Comparator<? super E> *comp*)

PriorityQueue(Collection<? extends E> *c*)

PriorityQueue(PriorityQueue<? extends E> *c*)

PriorityQueue(SortedSet<? extends E> *c*)

The first constructor builds an empty queue. Its starting capacity is 11. The second constructor builds a queue that has the specified initial capacity. The third constructor builds a queue with the specified capacity and comparator. The last three constructors create queues that are initialized with the elements of the collection passed in *c.* In all cases, the capacity grows automatically as elements are added.

program:

```
public class A {
  public static void main(String args[]) {
    PriorityQueue<String> hs = new PriorityQueue <String>();
    hs.add("B");
    hs.add("A");
    hs.add("D");
    hs.add("E");
    hs.add("C");
    hs.add("F");
    System.out.println("Elements in Queue "+hs);
```

```
System.out.println("head:"+hs.element());
System.out.println("head:"+hs.peek());
  }
}
```

output:

Elements in Queue: B A D E C F

head:B

head:B

4.8 ArrayDeque.

- **ArrayDeque** class, which extends **AbstractCollection** and implements
- the **Deque** interface. It adds no methods of its own. **ArrayDeque** creates a dynamic array
- and has no capacity restrictions .**ArrayDeque** is a generic class that has this declaration:

class ArrayDeque<E>

Here, **E** specifies the type of objects stored in the collection.

ArrayDeque defines the following constructors:

ArrayDeque( )

ArrayDeque (int *size*)

ArrayDeque (Collection<? extends E> *c*)

The first constructor builds an empty deque. Its starting capacity is 16. The second constructor builds a deque that has the specified initial capacity. The third constructor creates a deque that is initialized with the elements of the collection passed in *c*. In all cases, the capacity grows as needed to handle the elements added to the deque.

program:

```
public class A {
  public static void main(String args[]) {
    ArrayDeque <String> hs = new ArrayDeque <String>();
    hs.add("B");
    hs.add("A");
    hs.push("D");
    hs.push("E");
```

```
    System.out.println("Elements in Queue "+hs);
System.out.println("Pop element:"+hs.pop());
  }
}
```

output:

Elements in Queue B A D E

Pop element E

5. Accessing A collection via an Iterator:

To access, modify or remove any element from any collection we need to first find the element, for which we have to cycle through the elements of the collection. There are three possible ways to cycle through the elements of any collection.

1. Using Iterator interface
2. Using ListIterator interface
3. Using for-each loop

1. Accessing elements using Iterator

Iterator Interface is used to traverse a list in forward direction, enabling you to remove or modify the elements of the collection. Each collection classes provide **iterator()** method to return an iterator. Which can be declared as

interface Iterator<E>

Steps to use an Iterator

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

Methods of Iterator:

Method	Description
boolean hasNext()	Returns true if there are more elements in the collection. Otherwise, returns false.
E next()	Returns the next element present in the collection. Throws NoSuchElementException if there is not a next element.

void remove()	Removes the current element. Throws IllegalStateException if an attempt is made to call remove() method that is not preceded by a call to next() method.

program:

```
class A
{
 public static void main(String[] args)
 {
  ArrayList< String> a = new ArrayList< String>();
  a.add("a");
  a.add("b");
  a.add("c");
  a.add("d");

 Iterator it = a.iterator();     //Declaring Iterator
  while(it.hasNext())
  {
   System.out.print(it.next());
  }
 }
}
output: a b c d
```

2.Accessing element using ListIterator

ListIterator Interface is used to traverse a list in both forward and backward direction. It is available to only those collections that implements the **List** Interface. Which can be declared as

interface ListIterator<E>

Methods of ListIterator:

Method	Description
void add(E obj)	Inserts obj into the list in front of the element that will be returned by the next call to next() method.
boolean hasNext()	Returns true if there is a next element. Otherwise, returns false.
boolean hasPrevious()	Returns true if there is a previous element. Otherwise, returns false.
E next()	Returns the next element. A NoSuchElementException is thrown if there is not a next element.
int nextIndex()	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous()	Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
int previousIndex()	Returns the index of the previous element. If there is not a previous element, returns -1.
void remove()	Removes the current element from the list. An IllegalStateException is

	thrown if remove() method is called before next() or previous() method is invoked.
void set(E obj)	Assigns obj to the current element. This is the element last returned by a call to either next() or previous() method.

program:

```
class A
{
 public static void main(String[] args)
 {
  ArrayList< String> a = new ArrayList< String>();
  a.add("a");
  a.add("b");
  a.add("c");
  a.add("d");

  ListIterator l = a.listIterator();

System.out.print("In forward direction");
 while(l.hasNext())                 //In forward direction
  {
   System.out.print(l.next());
  }
 System.out.print("In backward direction");
  while(l.hasPrevious())             //In backward direction
  {
   System.out.print(l.next());
  }
 }
}
```

output:

In forward direction

a b c d

In backward direction

d c b a

3.Using for-each loop

`for-each` version of for loop can also be used for traversing each element of a collection. But this can only be used if we don't want to modify the contents of a collection and we don't want any reverse access. `for-each` loop can cycle through any collection of object that implements Iterable interface.

Program:

```
class ForEachDemo
{
 public static void main(String[] args)
 {
  ArryaList< String> l = new ArryaList< String>();
```

```
  l.add("a");
  l.add("b");
  l.add("c");
  l.add("d");

  for(String str : l)
  {
   System.out.print(str);
  }
 }
}
```

output: a b c d

6. Storing User Defined Classes in Collections:

In collection collection class we can store not only the different collection class object but we can store any type of object, including object of class that create in collection class.

```
class A
{
String name;
String usn;
String Branch;
int p_No;
A(String name,String usn,String Branch,int p_no)
{
this.name=name;
this.usn=usn;
this.Branch=Branch;
p_No=p_no;
}
}
class LinkedListClass
{
public static void main(String ar[])
{
LinkedList<A> l=new LinkedList<A>();
l.add(new A("Amar","123","CSE",99999999));
l.add(new A("Annu","456","CSE",9900000));
l.add(new A("Raj","789","CSE",99999900));
System.out.println(l);
}
}
```

output:

```
Amar 123 CSE 99999999
Annu 456 CSE 9900000
Raj    789 CSE 99999900
```

7. Working with Maps:

A *map* is an object that stores associations between keys and values, or *key/value pairs*. Given a key, you can find its value. Both keys and values are objects. The keys must be unique, but the values may be duplicated. Some maps can accept a **null** key and **null** values, others cannot.

Interface	Description
Map	Maps unique key to value.
Map.Entry	Describe an element in key and value pair in a map. Entry is sub interface of Map.
NavigableMap	Extends SortedMap to handle the retrienal of entries based on closest match searches
SortedMap	Extends Map so that key are maintained in an ascending order.

7.1 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.
- **Map** is generic and is declared as shown here:

interface Map<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values

Several methods throw a **ClassCastException** 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. An **IllegalArgumentException** is thrown if an invalid argument is used.

Method	Description
Object put(Object key, Object value)	It is used to insert an entry in this map.

void putAll(Map map)	It is used to insert the specified map in this map.
Object remove(Object key)	It is used to delete an entry for the specified key.
Object get(Object key)	It is used to return the value for the specified key.
boolean containsKey(Object key)	It is used to search the specified key from this map.
Set keySet()	It is used to return the Set view containing all the keys.
Set entrySet()	It is used to return the Set view containing all the keys and values.
Void clear()	Removes all key/values pairs from the invoking map.
boolean containsvalues(Object key)	Returns true if the map contains value. otherwise false.
Boolean equals(Object obj)	Returns true if theobj is a map and contains the same entries. otherwise false
Int size()	Returns the number of key/value pairs in the map.
Collection<v> values()	Returns a collection containing the value in the map.

program:

```
import java.util.*;
class MA{
 public static void main(String args[]){
  Map<String,Integer> m=new HashMap< String, Integer>();
  m.put("a",new Integer(100));
  m.put("b",new Integer(200));
  m.put("c",new Integer(300));
  m.put("d",new Integer(400));
System.out.println(m);
System.out.println("get value="+m.get("a"));
System.out.println("empty="+m.isEmpty());
  }
 }output: {d=400, b=200, c=300, a=100}
```

get value=100

empty=false

7.2: The SortedMap:

- **The SortedMap** is the interface extends **Map interface**. It ensures that the entries are maintained in ascending order based on the keys.
- **SortedMap** is generic and is declared as shown here:

interface SortedMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

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. An **IllegalArgumentException** is thrown if an invalid argument is used.

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.

program:

```
import java.util.*;
class MA{
 public static void main(String args[]){
  SortedMap<String,Integer> m=new TreeMap< String, Integer>();
  m.put("a",new Integer(100));
  m.put("b",new Integer(200));
  m.put("c",new Integer(300));
  m.put("d",new Integer(400));
System.out.println(m);
System.out.println("First key="+m.firstKey());
System.out.println("Head map:"+m.headMap("c"));
System.out.println("Tail map:"+m.tailMap("b"));
System.out.println("Sub map:"+m.subMap("a", "d"));
  }
 }
```

output:

{a=100, b=200, c=300, d=400}

First key=a

Head map:{a=100, b=200}

Tail map:{b=200, c=300, d=400}

Sub map:{a=100, b=200, c=300}

7.3 NavigableMap Interface:

- It extends **SortedMap** and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys.
- **NavigableMap** is a generic interface that has this declaration:

interface NavigableMap<K,V>

Here, **K** specifies the type of the keys, and **V** specifies the type of the values associated with the keys

Method	Description
Map.Entry<K,V> ceilingEntry(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K ceilingKey(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> descendingKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map.
NavigableMap<K,V> descendingMap( )	Returns a **NavigableMap** that is the reverse of the invoking map. The resulting map is backed by the invoking map.
Map.Entry<K,V> firstEntry( )	Returns the first entry in the map. This is the entry with the least key.
Map.Entry<K,V> floorEntry(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K floorKey(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableMap<K,V> headMap(K *upperBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting map is backed by the invoking map.
Map.Entry<K,V> higherEntry(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K higherKey(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
Map.Entry<K,V> lastEntry( )	Returns the last entry in the map. This is the entry with the largest key.
Map.Entry<K,V> lowerEntry(K *obj*)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K lowerKey(K *obj*)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> navigableKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map. The resulting set is backed by the invoking map.
Map.Entry<K,V> pollFirstEntry( )	Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. **null** is returned if the map is empty.
Map.Entry<K,V> pollLastEntry( )	Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. **null** is returned if the map is empty.
NavigableMap<K,V> subMap(K *lowerBound*, boolean *lowIncl*, K *upperBound* boolean *highIncl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *highIncl* is included. The resulting map is backed by the invoking map.
NavigableMap<K,V> tailMap(K *lowerBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting map is backed by the invoking map.

Program:

```
import java.util.*;
class MA{
 public static void main(String args[]){
  NavigableMap<String,Integer> m=new TreeMap< String, Integer>();
  m.put("a",new Integer(100));
  m.put("b",new Integer(200));
  m.put("c",new Integer(300));
  m.put("d",new Integer(400));
System.out.println(m);
System.out.println("Floor key="+m.floorKey("c"));
System.out.println("Head map:"+m.headMap("b", true));

  }
 }
```

output:

{a=100, b=200, c=300, d=400}

floor key=c

Head map:{a=100, b=200}

7.4: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. **Map.Entry** is generic and is declared like this:

interface Map.Entry<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values

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.

program:

```
import java.util.*;
class MA{
 public static void main(String args[]){
  NavigableMap<String,Integer> m=new TreeMap< String, Integer>();
  m.put("a",new Integer(100));
  m.put("b",new Integer(200));
  m.put("c",new Integer(300));
  m.put("d",new Integer(400));
System.out.println(m);
Set set = m.entrySet();
Iterator i = set.iterator();

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

```
}
  }
 }
```

output:

{a=100, b=200, c=300, d=400}

a: 100

b: 200

c: 300

d: 400

8. Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:

Class	Description
AbstractMap	Implements most of the Map interface.
EnumMap	Extends AbstractMap for use with enum keys.
HashMap	Extends AbstractMap to use a hash table.
TreeMap	Extends AbstractMap to use a tree.
WeakHashMap	Extends AbstractMap to use a hash table with weak keys.
LinkedHashMap	Extends HashMap to allow insertion-order iterations.
IdentityHashMap	Extends AbstractMap and uses reference equality when comparing documents.

8.1 HashMap Class

- The **HashMap** class extends **AbstractMap** and implements the **Map** interface. It uses a hash table to store the map.
- This allows the execution time of **get( )** and **put( )** to remain constant even for large sets. **HashMap** is a generic class that has this declaration:

class HashMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following constructors are defined:

HashMap( )

HashMap(Map<? extends K, ? extends V> *m*)

HashMap(int *capacity*)

HashMap(int *capacity*, float *fillRatio*)

The first form constructs a default hash map. The second form initializes the hash map by using the elements of *m*. The third form initializes the capacity of the hash map to *capacity*. The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments. The meaning of capacity and fill ratio is the same as for **HashSet.**

program:

```
class MA{
 public static void main(String args[]){
  HashMap<String,Integer> m=new HashMap< String, Integer>();
  m.put("a",new Integer(100));
  m.put("b",new Integer(200));
  m.put("c",new Integer(300));
  m.put("d",new Integer(400));
System.out.println(m);
System.out.println("get value="+m.get("a"));
System.out.println("empty="+m.isEmpty());
  }
 }
```

output:

{d=400, b=200, c=300, a=100}

get value=100

empty=false

8.2 TreeMap Class:

- The **TreeMap** class extends **AbstractMap** and implements the **NavigableMap** interface.

- It creates maps stored in a tree structure. A **TreeMap** provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval.
- A tree map guarantees that its elements will be sorted in ascending key order.
- **TreeMap** is a generic class that has this declaration:

class TreeMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following **TreeMap** constructors are defined:

TreeMap( )

TreeMap(Comparator<? super K> *comp*)

TreeMap(Map<? extends K, ? extends V> *m*)

TreeMap(SortedMap<K, ? extends V> *sm*)

The first form constructs an empty tree map that will be sorted by using the natural order of its keys. The second form constructs an empty tree-based map that will be sorted by using the **Comparator** *comp.* (Comparators are discussed later in this chapter.) The third form initializes a tree map with the entries from *m*, which will be sorted by using the natural order of the keys. The fourth form initializes a tree map with the entries from *sm*, which will be sorted in the same order as *sm.*

program:

```
import java.util.*;
class MA{
 public static void main(String args[]){
  TreeMap<String,Integer> m=new TreeMap< String, Integer>();
  m.put("a",new Integer(100));
  m.put("b",new Integer(200));
  m.put("c",new Integer(300));
  m.put("d",new Integer(400));
System.out.println(m);
System.out.println("Floor key="+m.floorKey("c"));
System.out.println("Head map:"+m.headMap("b", true));
  }
 }
```

output:

{a=100, b=200, c=300, d=400}

floor key=c

Head map:{a=100, b=200}

8.3 LinkedHashMap Class

- **LinkedHashMap** extends **HashMap**. It maintains a linked list of the entries in the map, in the order in which they were inserted.
- This allows insertion-order iteration over the map. That is,when iterating through a collection-view of a **LinkedHashMap**, the elements will be returned in the order in which they were inserted.
- **LinkedHashMap** is a generic class that has this declaration:

class LinkedHashMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

LinkedHashMap defines the following constructors:

LinkedHashMap( )

LinkedHashMap(Map<? extends K, ? extends V> *m*)

LinkedHashMap(int *capacity*)

LinkedHashMap(int *capacity*, float *fillRatio*)

LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*)

The first form constructs a default **LinkedHashMap**. The second form initializes the **LinkedHashMap** with the elements from *m.* The third form initializes the capacity. The fourth form initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for **HashMap**. The default capactiy is 16. The default ratio is 0.75. The last form allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If *Order* is **true**, then access order is used. If *Order* is **false**, then insertion order is used.

program:

```
import java.util.*;
class MA{
 public static void main(String args[]){
  LinkedHashMap<Integer,String> m=new LinkedHashMap<Integer,String>();
  m.put(100,new String("A"));
  m.put(101,new String("B"));
  m.put(102,new String("C"));
  System.out.println("conatins any key:"+m.containsKey(100));
for(Map.Entry m1:m.entrySet()){
   System.out.println(m1.getKey()+" "+m1.getValue());
  }
 }
}
```

output:

conatins any key:true

100 A

101 B

102 C

8.4 IdentityHashMap Class

- **IdentityHashMap** extends **AbstractMap** and implements the **Map** interface. It is similar to **HashMap** except that it uses reference equality when comparing elements.
- **IdentityHashMap** is a generic class that has this declaration:

class IdentityHashMap<K, V>

Here, **K** specifies the type of key, and **V** specifies the type of value. The API documentation explicitly states that **IdentityHashMap** is not for general use.

Sr.No.	Constructor & Description
1	**IdentityHashMap()** This constructor constructs a new, empty identity hash map with a default expected maximum size (21).
2	**IdentityHashMap(int expectedMaxSize)** This constructor constructs a new, empty IdentityHashMap with the specified expected maximum size.
3	**IdentityHashMap(Map m)** This constructor constructs a new identity hash map containing the keys-value mappings in the specified map.

program:

```
import java.util.*;
public class A {
  public static void main(String args[]) {
    IdentityHashMap<String,Integer> m = new IdentityHashMap<String,Integer>();
    m.put("Ram", new Double(3434.34));
    m.put("Sham", new Double(123.22));
    m.put("Amar", new Double(1378.00));
    m.put("Annu", new Double(99.22));
    m.put("Priya", new Double(-19.08));
    Set set = m.entrySet();
    Iterator i = set.iterator();
    while(i.hasNext()) {
      Map.Entry me = (Map.Entry)i.next();
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }
```

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

output:

Sham: 123.22

Annu: 99.22

Priya: -19.08

Amar: 1378.0

Ram: 3434.34

RAM's new balance: 4434.34

8.5 The EnumMap Class

- **EnumMap** extends **AbstractMap** and implements **Map**. It is specifically for use with keys of an **enum** type.
- It is a generic class that has this declaration:

class EnumMap<K extends Enum<K>, V>

Here, **K** specifies the type of key, and **V** specifies the type of value. Notice that **K** must extend **Enum<K>**, which enforces the requirement that the keys must be of an **enum** type.

EnumMap defines the following constructors:

EnumMap(Class<K> *kType*)

EnumMap(Map<K, ? extends V> *m*)

EnumMap(EnumMap<K, ? extends V> *em*)

The first constructor creates an empty **EnumMap** of type *kType*. The second creates an **EnumMap** map that contains the same entries as *m*. The third creates an **EnumMap** initialized with the values in *em*. **EnumMap** defines no methods of its own

program:

```
import java.util.*;
   public enum Days {
   Monday, Tuesday, Wednesday, Thursday
   };

class A
{
   public static void main(String[] args) {
   //create and populate enum map
   EnumMap<Days, String> map = new EnumMap<Days, String>(Days.class);
   map.put(Days.Monday, "1");
   map.put(Days.Tuesday, "2");
   map.put(Days.Wednesday, "3");
   map.put(Days.Thursday, "4");
   // print the map
   for(Map.Entry m:map.entrySet()){
      System.out.println(m.getKey()+" "+m.getValue());
     }
   }
}
```

output:

Monday 1

Tuesday 2

Wednesday 3

Thursday 4

9. Comparators:

- **Comparator interface** is used to order the objects of user-defined class. Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.

- **Comparator** is a generic interface that has this declaration:

interface Comparator<T>

Here, **T** specifies the type of objects being compared.

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

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.

program:

```
import java.util.*;
class Comp implements Comparator<String>
{
        @Override
        public int compare(String a, String b) {

                if(a.compareTo(b)>0)
                        System.out.println("value of ="+a);
                else
                        System.out.println("value of ="+b);

                return 0;
        }

}
```

```
class MA{
 public static void main(String args[]){
  TreeSet<String> m=new TreeSet< String>(new Comp());
  m.add("a");
  m.add("b");
  m.add("c");
  m.add("d");
System.out.println(m);
  }
 }
```

output:

value of =b

value of =c

value of =d

[a]