Collection Framework

Collection framework provides architecture to store and manipulate the group of objects.

The **Java Collections Framework** is a collection of interfaces and classes which helps in storing and processing the data efficiently. This framework has several useful classes which have tons of useful functions which makes a programmer task super easy.

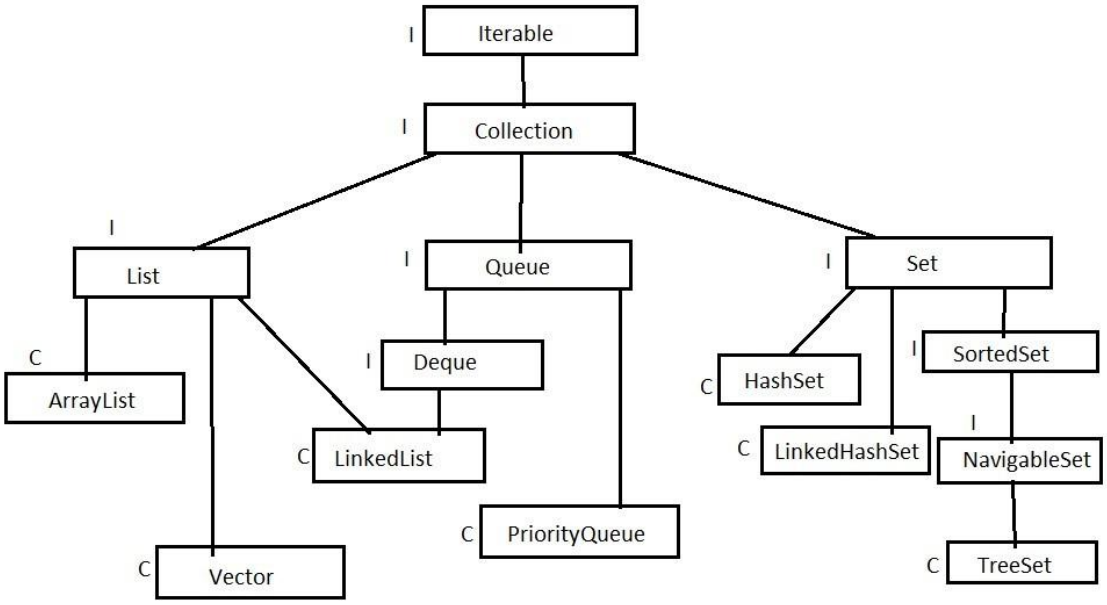

fig: Collection framework hierarchy

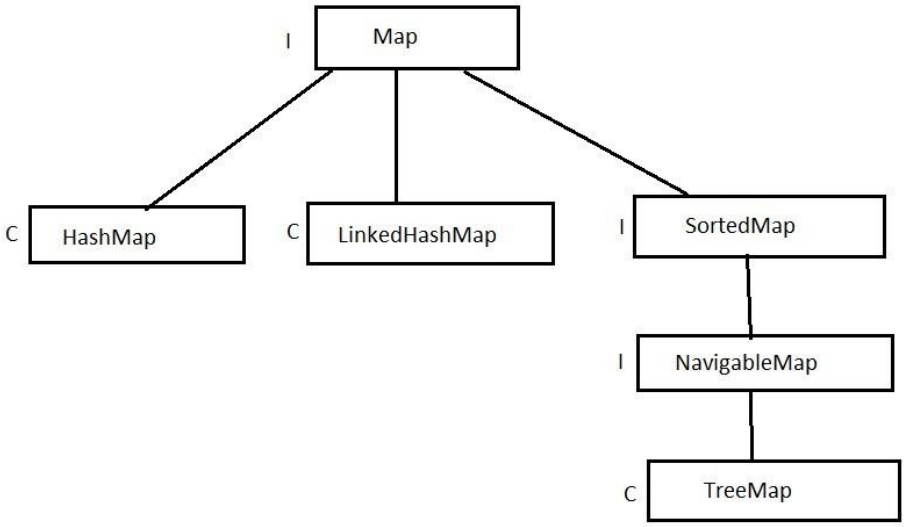

Fig: Map interface hierarchy

Collection Interface

Collection represents a single unit of objects i.e. a group.

Methods of Collection interface:

No.	Method	Description
1	public boolean add(Object element)	It is used to insert an element in this collection.
2	public boolean addAll(Collection c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from this collection.
4	public boolean removeAll(Collection c)	It is used to delete all the elements of specified collection from the invoking collection.
5	public boolean retainAll(Collection c)	It is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	Return the total number of elements in the collection.
7	public void clear()	Removes the total no of element from the collection.
8	public boolean contains(Object element)	It is used to search an element.

9	public boolean containsAll(Collection c)	It is used to search the specified collection in this collection.
10	public Iterator iterator()	Returns an iterator.
11	public Object[] toArray()	It converts collection into array.
12	public boolean isEmpty()	It checks if collection is empty.
13	public boolean equals(Object element)	It matches two collection.
14	public int hashCode()	It returns the hashcode number for collection.

List Interface

List interface is a sub interface of Collection interface. A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements. Elements can be inserted or accessed by their position in the list, using a zero-based index.

ArrayList Class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class is non synchronized.

- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

Methods of Java ArrayList

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll(Collection c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
void clear()	It is used to remove all of the elements from this list.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
Object[] toArray(Object[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean addAll(int index, Collection c)	It is used to insert all of the elements in the specified collection into this list, starting at the specified position.
Object clone()	It is used to return a shallow copy of an ArrayList.
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Example:

```
import java.util.*;
class ArrayListDemo{
 public static void main(String args[]){

  ArrayList<String> al=new ArrayList<String>();//creating arraylist
  al.add("BAAP");//adding object in arraylist
  al.add("TATA");
  al.add("BMC");
  al.add("Infosys");

  Iterator itr=al.iterator();//getting Iterator from arraylist to traverse elements
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

LinkedList Class

Java LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
- Java LinkedList class can be used as list, stack or queue.

Methods of Java LinkedList

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
void addFirst(Object o)	It is used to insert the given element at the beginning of a list.
void addLast(Object o)	It is used to append the given element to the end of a list.
int size()	It is used to return the number of elements in a list
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean contains(Object o)	It is used to return true if the list contains a specified element.

boolean remove(Object o)	It is used to remove the first occurence of the specified element in a list.
Object getFirst()	It is used to return the first element in a list.
Object getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.

Example:

```
class LinkedListDemo{
 public static void main(String args[]){
  LinkedList<String> al=new LinkedList<String>();//creating arraylist
  al.add("BAAP");//adding object in arraylist
  al.add("BAAP");
  al.add("BMC");
  al.add("Infosys");
  Iterator itr=al.iterator();//getting Iterator from arraylist to traverse elements
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }}
```

Difference between ArrayList and LinkedList

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

But there are many differences between ArrayList and LinkedList classes that are given below.

ArrayList	**LinkedList**
1) ArrayList internally uses **dynamic array** to store the elements.	LinkedList internally uses **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
3) ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

Vector Class

The Vector class implements List interface and uses a growable array of objects. Like an array, it contains components that 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.

It is similar to ArrayList, but with two differences −

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

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

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

Sr.No.	Method & Description
void add(int index, Object element)	Inserts the specified element at the specified position in this Vector.
boolean add(Object o)	Appends the specified element to the end of this Vector.
boolean addAll(Collection c)	Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.

boolean addAll(int index, Collection c)	Inserts all of the elements in in the specified Collection into this Vector at the specified position.
void addElement(Object obj)	Adds the specified component to the end of this vector, increasing its size by one.
int capacity()	Returns the current capacity of this vector.
void clear()	Removes all of the elements from this vector.
boolean contains(Object elem)	Tests if the specified object is a component in this vector.
boolean containsAll(Collection c)	Returns true if this vector contains all of the elements in the specified Collection.
void copyInto(Object[] anArray)	Copies the components of this vector into the specified array.
Object elementAt(int index)	Returns the component at the specified index.
Enumeration elements()	Returns an enumeration of the components of this vector.
Object firstElement()	Returns the first component (the item at index 0) of this vector.
Object get(int index)	Returns the element at the specified position in this vector.
int indexOf(Object elem)	Searches for the first occurence of the given argument, testing for equality using the equals method.
void insertElementAt(Object obj, int index)	Inserts the specified object as a component in this vector at the specified index.

boolean isEmpty()	Tests if this vector has no components.
Object lastElement()	Returns the last component of the vector.
int lastIndexOf(Object elem)	Returns the index of the last occurrence of the specified object in this vector.
int lastIndexOf(Object elem, int index)	Searches backwards for the specified object, starting from the specified index, and returns an index to it.
Object remove(int index)	Removes the element at the specified position in this vector.
boolean remove(Object o)	Removes the first occurrence of the specified element in this vector, If the vector does not contain the element, it is unchanged.
boolean removeAll(Collection c)	Removes from this vector all of its elements that are contained in the specified Collection.
boolean removeElement(Object obj)	Removes the first (lowest-indexed) occurrence of the argument from this vector.
void removeElementAt(int index)	removeElementAt(int index).
boolean retainAll(Collection c)	Retains only the elements in this vector that are contained in the specified Collection.
Object set(int index, Object element)	Replaces the element at the specified position in this vector with the specified element.
void setElementAt(Object obj, int index)	Sets the component at the specified index of this vector to be the specified object.
void setSize(int newSize)	Sets the size of this vector.

int size()	Returns the number of components in this vector.
Object[] toArray()	Returns an array containing all of the elements in this vector in the correct order.

Example:

```
import java.util.Vector;

public class BasicVectorOperations {


    public static void main(String a[]){

        Vector<String> vct = new Vector<String>();

        //adding elements to the end

        vct.add("First");

        vct.add("Second");

        vct.add("Third");

        System.out.println(vct);

        //adding element at specified index

        vct.add(2,"Random");

        System.out.println(vct);

        //getting elements by index

        System.out.println("Element at index 3 is: "+vct.get(3));

        //getting first element
```

```
        System.out.println("The first element of this vector is: "+vct.firstElement());
        //getting last element
        System.out.println("The last element of this vector is: "+vct.lastElement());
        //how to check vector is empty or not
        System.out.println("Is this vector empty? "+vct.isEmpty());
    }
}
```

Difference between ArrayList and Vector

ArrayList and Vector both implements List interface and maintains insertion order.

But there are many differences between ArrayList and Vector classes that are given below.

ArrayList	**Vector**
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity.	Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity.
3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2.	Vector is a **legacy** class.
4) 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 non-runnable state until current thread releases the lock of object.
5) ArrayList uses **Iterator** interface to traverse the elements.	Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also.

Stack Class

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

Methods of Stack Class

boolean empty()	Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
Object peek( )	Returns the element on the top of the stack, but does not remove it.
Object pop( )	Returns the element on the top of the stack, removing it in the process.
Object push(Object element)	Pushes the element onto the stack. Element is also returned.
int search(Object element)	Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned.

Example:

```
import java.util.*;

public class StackDemo {


   static void showpush(Stack st, int a) {
```

```
    st.push(new Integer(a));
    System.out.println("push(" + a + ")");
    System.out.println("stack: " + st);
  }

  static void showpop(Stack st) {
    System.out.print("pop -> ");
    Integer a = (Integer) st.pop();
    System.out.println(a);
    System.out.println("stack: " + st);
  }

  public static void main(String args[]) {
    Stack st = new Stack();
    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);
    try {
      showpop(st);
```

```
      }catch (EmptyStackException e) {
         System.out.println("empty stack");
      }
   }
}
```

Set Interface

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

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

LinkedHashSet.

HashSet Class

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order. Following are the important points about HashSet:

- HashSet doesn’t maintain any order, the elements would be returned in any random order.
- HashSet doesn’t allow duplicates. If you try to add a duplicate element in HashSet, the old value would be overwritten.

- HashSet allows null values however if you insert more than one nulls it would still return only one null value.
- HashSet is non-synchronized.
- 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.

Methods of Java HashSet class:

Method	Description
void clear()	It is used to remove all of the elements from this set.
boolean contains(Object o)	It is used to return true if this set contains the specified element.
boolean add(Object o)	It is used to adds the specified element to this set if it is not already present.
boolean isEmpty()	It is used to return true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
Object clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
Iterator iterator()	It is used to return an iterator over the elements in this set.
int size()	It is used to return the number of elements in this set.

Example:

```
import java.util.HashSet;

public class HashSetExample {

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

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

        // Adding elements to the HashSet
        hset.add("Apple");
        hset.add("Mango");
        hset.add("Grapes");
        hset.add("Orange");
        hset.add("Fig");
        //Addition of duplicate elements
        hset.add("Apple");
        hset.add("Mango");
        //Addition of null values
        hset.add(null);
        hset.add(null);
        //Displaying HashSet elements
        System.out.println(hset);
    }
}
```

LinkedHashSet Class

LinkedHashSet, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).

A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements. Use this class instead of HashSet when you care about the iteration order. When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

LinkedHashSet Methods

Method	**Purpose**
public boolean add(Object o)	Adds an object to a LinkedHashSet if already not present in HashSet.
public boolean remove(Object o)	Removes an object from LinkedHashSet if found in HashSet.
public boolean contains(Object o)	Returns true if object found else return false
public boolean isEmpty()	Returns true if LinkedHashSet is empty else return false
public int size()	Returns number of elements in the LinkedHashSet

Example:

```
import java.util.LinkedHashSet;

public class LinkedHashSetDemo {

	public static void main(String[] args) {

		LinkedHashSet<String> linkedset = new LinkedHashSet<String>();
```

```
        // Adding element to LinkedHashSet
        linkedset.add("Maruti");
        linkedset.add("BMW");
        linkedset.add("Honda");
        linkedset.add("Audi");
        linkedset.add("Maruti"); //This will not add new element as Maruti already exists
        linkedset.add("WalksWagon");
        System.out.println("Size of LinkedHashSet=" + linkedset.size());
        System.out.println("Original LinkedHashSet:" + linkedset);
        System.out.println("Removing Audi from LinkedHashSet: " + linkedset.remove("Audi"));
        System.out.println("Trying to Remove Z which is not present: "
                + linkedset.remove("Z"));
        System.out.println("Checking if Maruti is present=" + linkedset.contains("Maruti"));
        System.out.println("Updated LinkedHashSet: " + linkedset);
    }
}
```

TreeSet Class

TreeSet, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet. TreeSet is similar to HashSet except that it sorts the elements in the ascending order while HashSet doesn't maintain any order.

Methods of Java TreeSet class

Method	Description
boolean addAll(Collection c)	It is used to add all of the elements in the specified collection to this set.
boolean contains(Object o)	It is used to return true if this set contains the specified element.
boolean isEmpty()	It is used to return true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
void add(Object o)	It is used to add the specified element to this set if it is not already present.
void clear()	It is used to remove all of the elements from this set.
Object clone()	It is used to return a shallow copy of this TreeSet instance.
Object first()	It is used to return the first (lowest) element currently in this sorted set.
Object last()	It is used to return the last (highest) element currently in this sorted set.
int size()	It is used to return the number of elements in this set.

Example:

```
import java.util.TreeSet;
public class TreeSetExample {
    public static void main(String args[]) {
        // TreeSet of String Type
        TreeSet<String> tset = new TreeSet<String>();

        // Adding elements to TreeSet<String>
        tset.add("ABC");
        tset.add("String");
        tset.add("Test");
        tset.add("Pen");
        tset.add("Ink");
        tset.add("Jack");

        //Displaying TreeSet
        System.out.println(tset);
   }
}
```

Here all the elements in TreeSet will be maintained in the ascending order.

Queue Interface

The java.util.Queue interface is a subtype of the java.util.Collection interface. It represents an ordered list of objects just like a List, but its intended use is slightly different. A queue is designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue.

Methods of Java Queue Interface

Method	Description
boolean add(object)	It is used to insert the specified element into this queue and return true upon success.
boolean offer(object)	It is used to insert the specified element into this queue.
Object remove()	It is used to retrieves and removes the head of this queue.
Object poll()	It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.
Object element()	It is used to retrieves, but does not remove, the head of this queue.
Object peek()	It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

PriorityQueue Class

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.

Following are the important points about PriorityQueue:

- The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
- A priority queue does not permit null elements.
- A priority queue relying on natural ordering also does not permit insertion of non-comparable objects.

Methods of Java PriorityQueue Class

Method	**Description**
boolean add(object)	It is used to insert the specified element into this queue and return true upon success.
boolean offer(object)	It is used to insert the specified element into this queue.
Object remove()	It is used to retrieves and removes the head of this queue.
Object poll()	It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.
Object element()	It is used to retrieves, but does not remove, the head of this queue.
Object peek()	It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
Object [ ] toArray()	Returns an array containing all of the elements in this queue.

Example:

```
import java.util.*;

class PriorityQueueDemo{

public static void main(String args[]){


PriorityQueue<String> queue=new PriorityQueue<String>();

queue.add("element1");

queue.add("element2");

queue.add("element3");

queue.add("element4");

queue.add("element5");


System.out.println("head:"+queue.element());

System.out.println("head:"+queue.peek());  //peek()Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.


System.out.println("iterating the queue elements:");

Iterator itr=queue.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}
```

```
queue.remove();

queue.poll();  //Retrieves and removes the head of this queue, or returns null if this queue is empty.


System.out.println("after removing two elements:");

Iterator<String> itr2=queue.iterator();

while(itr2.hasNext()){

System.out.println(itr2.next());

}

}

}
```

Deque Interface

The java.util.Deque interface is a subtype of the java.util.Queue interface. It represents a queue where you can insert and remove elements from both ends of the queue. Thus, "Deque" is short for "Double Ended Queue".

Being a Queue subtype all methods in the Queue and Collection interfaces are also available in the Dequeinterface.

Since Deque is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following Deque implementations in the Java Collections API:

- java.util.ArrayDeque

- java.util.LinkedList

LinkedList is a pretty standard deque / queue implementation.

Methods of Deque interface

add(E e)	Inserts the specified element into the queue represented by this deque.
addFirst(E e)	Inserts the specified element at the front of this deque
addLast(E e)	Inserts the specified element at the end of this deque
contains(Object o)	Returns true if this deque contains the specified element.
element()	Retrieves, but does not remove, the head of the queue represented by this deque
getFirst()	Retrieves, but does not remove, the first element of this deque.
getLast()	Retrieves, but does not remove, the last element of this deque.
peek()	Retrieves, but does not remove, the head of the queue represented by this deque
poll()	Retrieves and removes the head of the queue represented by this deque
remove()	Retrieves and removes the head of the queue represented by this deque
size()	Returns the number of elements in this deque.

Example:

```
import java.util.*;

class DequeDemo

{

        public static void main(String[] args) {


Deque dequeA = new LinkedList();

dequeA.add      ("element 1"); //add element at tail

dequeA.addFirst("element 2"); //add element at head

dequeA.addLast ("element 3"); //add element at tail


System.out.println(dequeA);

Object firstElement1 = dequeA.element();

Object firstElement = dequeA.getFirst();

Object lastElement1  = dequeA.getLast();


System.out.println("First Ele: "+firstElement1+" First element "+firstElement+" last element"+ lastElement1);


dequeA.add("element 0");

dequeA.add("element 1");

dequeA.add("element 2");
```

```
System.out.println(dequeA);


//access via Iterator
Iterator iterator = dequeA.iterator();
while(iterator.hasNext()){
  String element = (String) iterator.next();
  System.out.println(element);
}


Object firstElementr1 = dequeA.remove();
Object firstElementr = dequeA.removeFirst();
Object lastElementr  = dequeA.removeLast();


System.out.println("First Ele: "+firstElementr1+" First element "+firstElementr+" last element"+ lastElementr);


System.out.println(dequeA);
        }
}
```

Map Interface

A map contains values on the basis of key i.e. key and value pair. Each key and value pair is known as an entry. Map contains only unique keys.

Map is useful if you have to search, update or delete elements on the basis of key.

The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit different from the rest of the collection types.

A map cannot contain duplicate keys. There are three main implementations of Map interfaces: HashMap, TreeMap, and LinkedHashMap.

Useful methods of Map interface

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.

SortedMap Interface

The java.util.SortedMap interface is a subtype of the **java.util.Map** interface, with the addition that the elements stored in the map are sorted internally.

By default the elements are iterated in ascending order, starting with the "smallest" and moving towards the "largest". But it is also possible to iterate the elements in descending order using the method TreeMap.descendingKeySet().

HashMap Class

HashMap: it makes no guarantees concerning the order of iteration.

The important points about Java HashMap class are:

- A HashMap contains values based on the key.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It maintains no order.

Methods of Java HashMap class

Method	**Description**
void clear()	It is used to remove all of the mappings from this map.
boolean containsKey(Object key)	It is used to return true if this map contains a mapping for the specified key.
boolean containsValue(Object value)	It is used to return true if this map maps one or more keys to the specified value.

boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
Object put(Object key, Object value)	It is used to associate the specified value with the specified key in this map.
int size()	It is used to return the number of key-value mappings in this map.
Collection values()	It is used to return a collection view of the values contained in this map.

Example:

```
import java.util.*;
class HashMapDemo{
 public static void main(String args[]){
  HashMap<Integer,String> hm=new HashMap<Integer,String>();
  hm.put(100,"Emp1");
  hm.put(101,"Emp2");
  hm.put(102,"Emp3");
  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
```

}

LinkedHashMap Class

LinkedHashMap: It orders its elements based on the order in which they were inserted into the set (insertion-order).

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

Methods of Java LinkedHashMap class

Method	**Description**
Object get(Object key)	It is used to return the value to which this map maps the specified key.
void clear()	It is used to remove all mappings from this map.
boolean containsKey(Object key)	It is used to return true if this map maps one or more keys to the specified value.

Example:

```
import java.util.LinkedHashMap;

import java.util.Set;

import java.util.Iterator;

import java.util.Map;

public class LinkedHashMapDemo {
```

```
public static void main(String args[]) {
    // HashMap Declaration
    LinkedHashMap<Integer, String> lhmap =
            new LinkedHashMap<Integer, String>();

    //Adding elements to LinkedHashMap
    lhmap.put(22, "Abey");
    lhmap.put(33, "Dawn");
    lhmap.put(1, "Sherry");
    lhmap.put(2, "Karon");
    lhmap.put(100, "Jim");

    // Generating a Set of entries
    Set set = lhmap.entrySet();

    // Displaying elements of LinkedHashMap
    Iterator iterator = set.iterator();
    while(iterator.hasNext()) {
      Map.Entry me = (Map.Entry)iterator.next();
      System.out.print("Key is: "+ me.getKey() +
              "& Value is: "+me.getValue()+"\n");
    }
}
```

}

TreeMap Class

TreeMap: It stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashMap.

TreeMap is Red-Black tree based NavigableMap implementation. It is sorted according to the natural ordering of its keys. **TreeMap class** implements Map interface similar to HashMap class. The main difference between them is that HashMap is an unordered collection while TreeMap is sorted in the ascending order of its keys. TreeMap is unsynchronized collection class which means it is not suitable for thread-safe operations until unless synchronized explicitly.

The important points about Java TreeMap class are:

- A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
- It contains only unique elements.
- It cannot have null key but can have multiple null values.
- It is same as HashMap instead maintains ascending order.

Methods of Java TreeMap class

Method	**Description**
boolean containsKey(Object key)	It is used to return true if this map contains a mapping for the specified key.

boolean containsValue(Object value)	It is used to return true if this map maps one or more keys to the specified value.
Object firstKey()	It is used to return the first (lowest) key currently in this sorted map.
Object get(Object key)	It is used to return the value to which this map maps the specified key.
Object lastKey()	It is used to return the last (highest) key currently in this sorted map.
Object remove(Object key)	It is used to remove the mapping for this key from this TreeMap if present.
void putAll(Map map)	It is used to copy all of the mappings from the specified map to this map.
Set entrySet()	It is used to return a set view of the mappings contained in this map.
int size()	It is used to return the number of key-value mappings in this map.
Collection values()	It is used to return a collection view of the values contained in this map.

Example:

```
import java.util.*;
class TestCollection15{
 public static void main(String args[]){
  TreeMap<Integer,String> hm=new TreeMap<Integer,String>();
  hm.put(100,"Emp1");
  hm.put(102,"Emp2");
```

```
  hm.put(101,"Emp3");
  hm.put(103,"Emp4");
  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```