Collection Framework-2022

- In order to handle group of objects we can use array of objects. If we have a class called Employ with members name and id, if we want to store details of 10 Employees, create an array of object to hold 10 Employ details.

Employ obj [] = new Employ [10];

- We cannot store different class objects into same array.
- Inserting element at the end of array is easy but at the middle is difficult.
- After retrieving the elements from the array, in order to process the elements we don’t have any methods

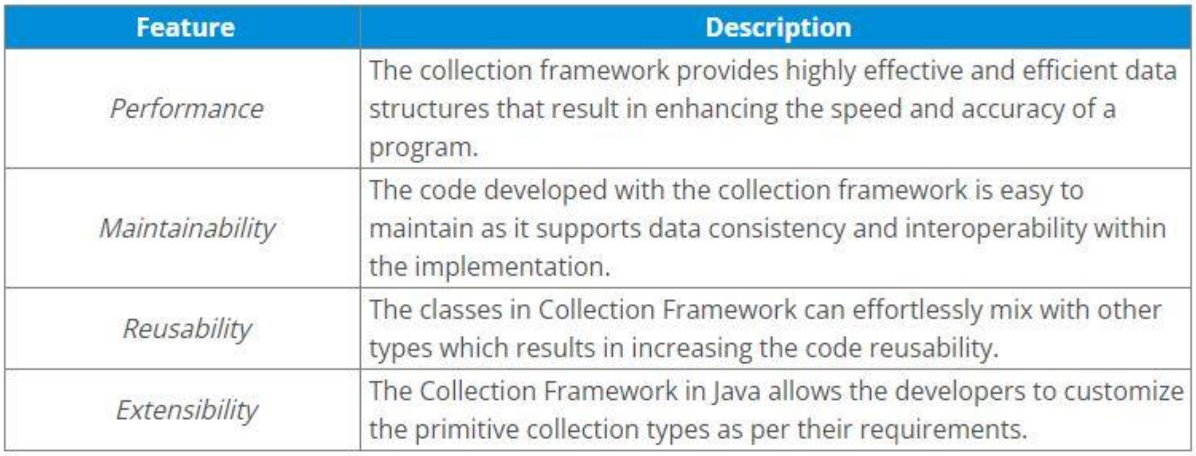
Collection Framework provides an architecture to store and manipulate the group of objects. All the operations that you perform on a data such as searching, sorting, insertion, deletion etc. can be performed by Java Collection Framework.

A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit.

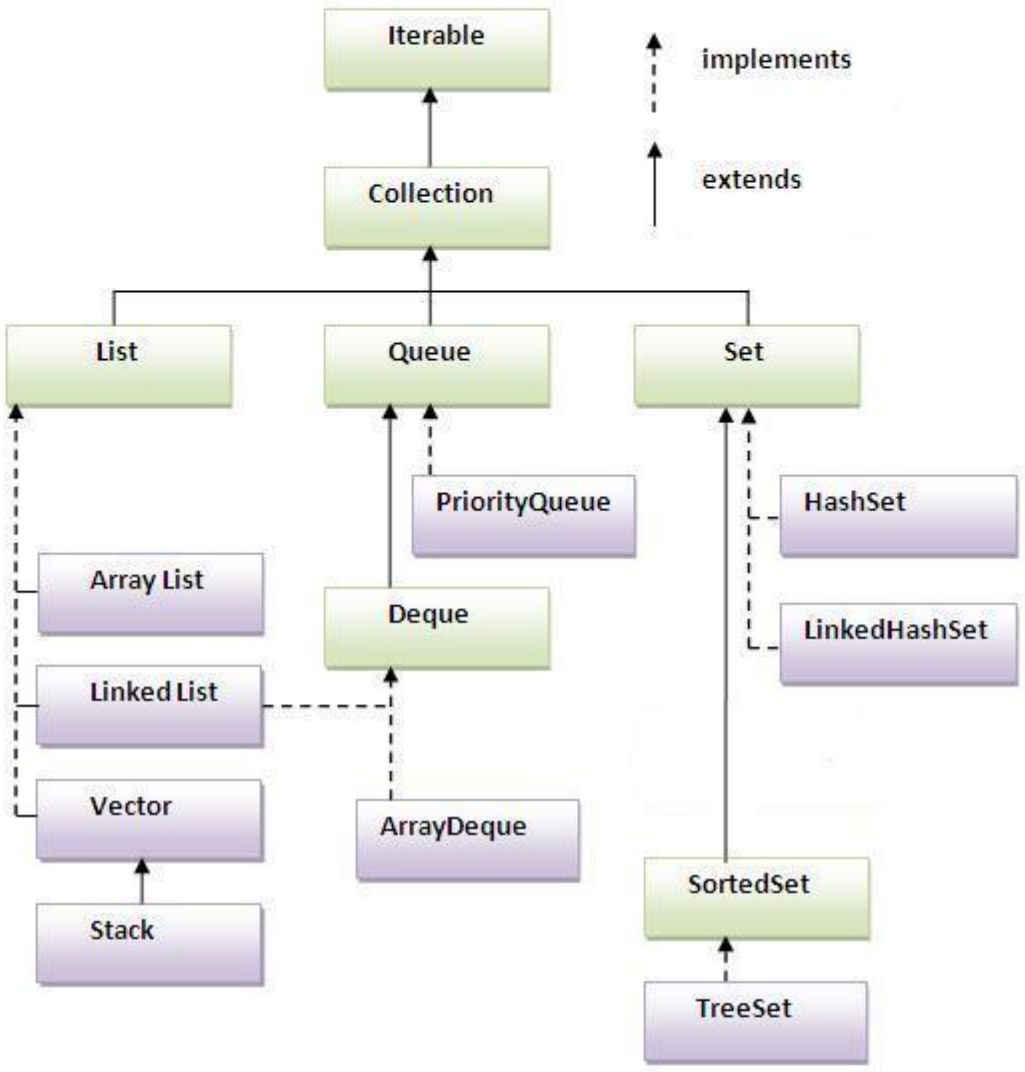
All collections frameworks contain the following:

- **Interfaces: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation.** In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations: These are the concrete implementations of the collection 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 collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.**

Advantages of Collection Framework:



Feature	Description
Performance	The collection framework provides highly effective and efficient data structures that result in enhancing the speed and accuracy of a program.
Maintainability	The code developed with the collection framework is easy to maintain as it supports data consistency and interoperability within the implementation.
Reusability	The classes in Collection Framework can effortlessly mix with other types which results in increasing the code reusability.
Extensibility	The Collection Framework in Java allows the developers to customize the primitive collection types as per their requirements.



Retrieving Elements from Collections: Following are the ways to retrieve any element from a collection object:

- **Using Iterator interface.**
- **Using ListIterator interface.**
- **Using Enumeration interface.**

Iterator Interface: Iterator is an interface that contains methods to retrieve the elements one by one from a collection object. It retrieves elements only in forward direction. It has three methods:

Method	Description
booleanhasNext()	This method returns true if the iterator has more elements.
element next()	This method returns the next element in the iterator.
void remove()	This method removes the last element from the collection returned by the iterator.

ListIterator Interface: ListIterator is an interface that contains methods to retrieve the elementsfrom a collection object, both in forward and reverse directions. It can retrieve the elements inforward and backward direction. It has the following important methods:

Method	Description
booleanhasNext()	This method returns true if the ListIterator has more elements when traversing the list in forward direction
element next()	This method returns the next element.
void remove()	This method removes the list last element that was returned by the next () or previous () methods.
booleanhasPrevious()	This method returns true if the ListIterator has more elements when traversing the list in reverse direction.
element previous()	This method returns the previous element in the list.

Enumeration Interface: This interface is useful to retrieve elements one by one like Iterator. Ithas two methods.

Method	Description
booleanhasMoreElements()	This method tests Enumeration has any more elements
elementnextElement()	This returns the next element that is available in Enumeration.

Iterator vs ListIterator:

1) Iterator is used for traversing `List` and `Set` both. Whereas, We can use ListIterator to traverse `List` only, we cannot traverse `Set` using ListIterator.

2) We can traverse in only forward direction using Iterator. Using ListIterator, we can traverse a List in both the directions (forward and Backward).

3) We cannot obtain indexes while using Iterator.We can obtain indexes at any point of time while traversing a list using ListIterator.The methods nextIndex() and previousIndex() are used for this purpose.

4) We cannot add element to collection while traversing it using Iterator, it throws ConcurrentModificationException when you try to do it.We can add element at any point of time while traversing a list using ListIterator.

5) We cannot replace the existing element value when using Iterator.By using set(E e) method of ListIterator we can replace the last element returned by next() or previous() methods

Enumeration:

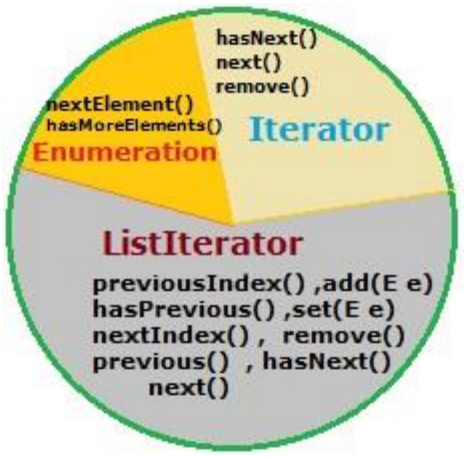
- ✓ Enumeration iterates Vector and Hashtable .
- ✓ Reverse iteration is not possible with enumeration.
- ✓ It cannot add or remove elements while iteration .

Iterator:

- ✓ Iterator iterates the implementations of List , Set .
- ✓ Reverse iteration is not possible with Iterator .
- ✓ Iterator cannot add elements , but it can remove elements while iteration .

ListIterator :

- ✓ ListIterator iterates List implementations (like ArrayList , LinkedList etc).
- ✓ Both forward and backward iterations are possible with ListIterator .
- ✓ Both elements addition and deletion can possible with ListIterator .



The Collection Interface:

- The Collection interface is the foundation upon which the collections framework is built.
- Enables you to work with groups of objects; it is at the top of the collections hierarchy.
- It declares the core methods that all collections will have. These methods are as follows

i) boolean add(Object *obj*)
Adds *obj* to the invoking collection. Returns true if *obj* was added to the collection. Returns false if *obj*is already a member of the collection, or if the collection does not allow duplicates.

ii)Boolean addAll(Collection *c*)
Adds all the elements of *c* to the invoking collection. Returns true if the operation succeeded (i.e., the elements were added). Otherwise, returns false.

iii)void clear( )
Removes all elements from the invoking collection.

iv)boolean contains(Object *obj*)
Returns true if *obj*is an element of the invoking collection. Otherwise, returns false.

v)boolean containsAll(Collection *c*)
Returns true if the invoking collection contains all elements of *c*. Otherwise, returns false.

vi)boolean equals(Object *obj*)
Returns true if the invoking collection and *obj* are equal. Otherwise, returns false.

The Set Interface

- The **Set** interface defines a set. It extends **Collection** and declares the behavior of a collection that does not allow duplicate elements.
- Therefore, the **add( )** method returns **false** if an attempt is made to add duplicate elements to a set. It does not define any additional methods of its own.

The SortedSet Interface

- The **SortedSet**interface extends **Set** and declares the behavior of a set sorted in ascending order.
- In addition to those methods defined by **Set**, the **SortedSet**interface declares the following extra methods

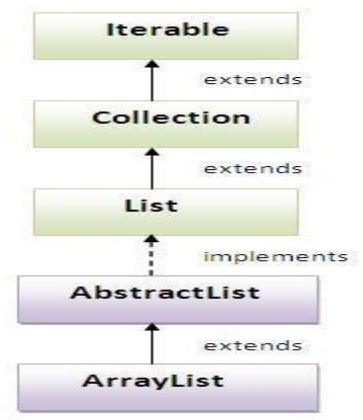
The List Interface

- The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index. A list may contain duplicate elements.
- In addition to the methods defined by **Collection**, **List** defines some of its own, which are shown below

The ArrayList Class:

- The **ArrayList** class extends **AbstractList** and implements the **List** interface &**RandomAccess** interface.
- **ArrayList** supports dynamic arrays that can grow as needed. It is **ordered** but **not sorted** collection.
- As it Implements **RandomAccess** interface, it gives fast iteration & fast random access.

Hierarchy of ArrayList class:



Constructors:--
ArrayList()
ArrayList(Collection c)
ArrayList(int capacity)

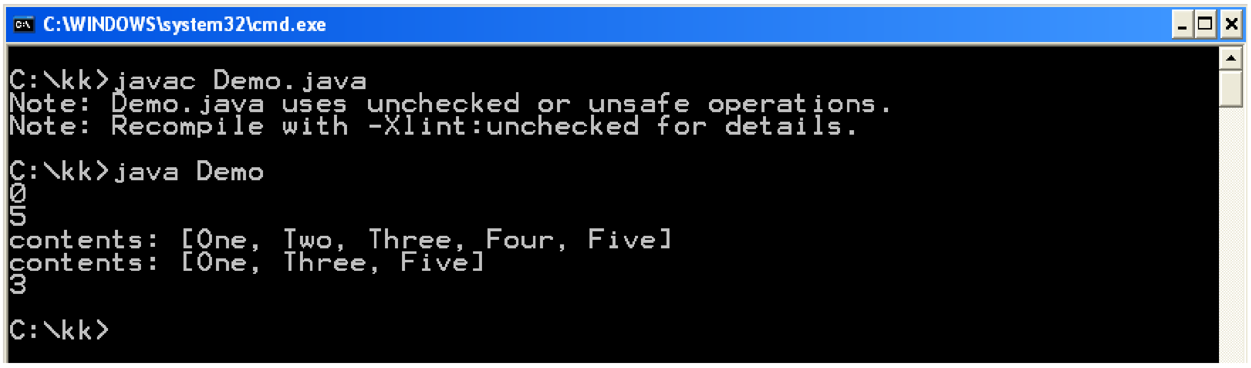
Methods of ArrayList Class:

- **boolean add (element obj)** : This method appends the specified element to the end of the ArrayList. If the element is added successfully then the method returns true.
- **void add(int position, element obj):** This method inserts the specified element at the specified position in the ArrayList.
- **element remove(int position) :**This method removes the element at the specified position in theArrayList and returns it.
- **boolean remove (Object**obj):This method removes the first occurrence of the specified element.
- **void clear ()** This method removes all the elements from the ArrayList.
- **element set(intposition,elementobj)**This method replaces an element at the specified position in theArrayList with the specified element obj.
- **boolean contains (Objectobj)**This method returns true if the ArrayList contains the specifiedelement obj.

- **element get (int position)**This method returns the element available at the specified position
  in the ArrayList.
- **int size ()** Returns number of elements in the ArrayList.
- **intindexOf (Object obj)**This method returns the index of the first occurrence of the specified element in the list, or -1 if the list does not contain theelement.
- **intlastIndexOf (Object obj)**This method returns the index of the last occurrence of the specifiedelement in the list, or -1 if the list does not contain the element.
- **Object[] toArray ()** This method converts the ArrayLlist into an array of Object classtype. All the elements of the ArrayList will be stored into the arrayin the same sequence.

Example:--

```
import java.util.*;

class Demo4
{
public static void main(String args[ ])
   {
     ArrayList al=new ArrayList();

System.out.println(al.size());

al.add("One");
al.add("Two");
al.add("Three");
al.add("Four");
al.add("Five");

System.out.println(al.size());

System.out.println("contents: " + al);

al.remove(1);
al.remove("Four");

System.out.println("contents: " + al);
System.out.println(al.size());
   }
}
```



```
C:\kk>javac Demo.java
Note: Demo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\kk>java Demo
0
5
contents: [One, Two, Three, Four, Five]
contents: [One, Three, Five]
3

C:\kk>
```

The Vector class:--

- **Vector is a class that reside in java.util package.**
- **An object of vector class represents a dynamic array of object type.**
- **Methods of vector are synchronized. So, Vector is synchronized**
- **A Vector is basically same as an ArrayList , but vector methods are synchronized for thread safety.**
- **Vector can grow and shrink its size dynamically.**
- **Vector is Ordered & Unsorted collection.**
- Instances of class java.util.Vector (that is, class Vector in package java.util) can be thought of as lists. **Java Vector objects are usually viewed as being implemented with arrays that expand whenever they fill up to always allow additional objects to be added to the list/array (as long as sufficient memory exists).**
- The objects in a Vector are instances of Java class Object, so they can be instances of any class (because all classes ultimately extend class Object). **Items can be added to the beginning, middle, or end of a vector. When an item is inserted at the beginning or in the middle, items already in the vector from the insertion index and higher are shifted up one position to make room. Items can be accessed from a vector by specifying the item's index, just as in an array (although the syntax is different). Any item can be removed from a vector, in which case the remaining elements with indices higher than the deleted item are moved down one index position.**

Constructor:

- **Vector(intinitialCapacity, intincr) - Construct a vector with the indicated initial capacity and add space for incr elements each time the vector is expanded.**

- **Vector(intinitialCapacity) - Construct a vector with the indicated initial capacity and double the capacity each time the vector is expanded.**

Some Vector Methods:

- **Void addElement(Object x) - Add x to the end of the list/vector.**
- **Void insertElementAt(Object x, int index) - Insert x at the given index.**
- **Void setElementAt(Object x, int index) - Replace the object at index with x. (Note: The given index must currently be in the list/vector.)**
- **VoidremoveElementAt(int index) - Remove the indicated element.**
- **Void removeAllElements() - Make the list empty.**
- **Object elementAt(int index) - Return the element at index.**
- **int size() - Return the number of elements currently in the vector.**
- **Boolean isEmpty() - Return whether the vector is empty.**
- **finalint capacity()-Returns the current capacity of this vector.**
- **final void copyInto(Object obj[ ])- Copies the components of this vector into the specified array.**

- **boolean add(element obj)**: This method appends the specified element to the end of the Vector.If the element is added successfully then the method returns true.
- **void add (int position, element obj):** This method inserts the specified element at the specified position in the Vector.
- **element remove (int position):** This method removes the element at the specified position in theVector and returns it.
- **boolean remove (Object obj):** This method removes the first occurrence of the specified elementobj from the Vector, if it is present.
- **void clear ()**:This method removes all the elements from the Vector.
- **element set (int position, element obj):** This method replaces an element at the specified position in theVector with the specified element obj.
- **boolean contains (Object obj):** This method returns true if the Vector contains the specifiedelement obj.
- **element get (int position)** This method returns the element available at the specified positionin the Vector.

Code1:

```
importjava.util.*;

class VectorDemo1
{
public static void main(String args[ ])
   {
        Vector v = new Vector();

        v.addElement("string 2");
        v.addElement("string 4");
        v.insertElementAt("string 1", 0);
        v.insertElementAt("string 3", 2);
        v.addElement("string 5");
        for (int i=0; i<v.size(); i++)
        {

                String s = (String) v.elementAt(i);
                System.out.println(s);
        }

        System.out.println("Now the element removed at 3 as:");
v.removeElementAt(3);

for(int i=0; i<v.size(); i++)
        {
                System.out.println((String)v.elementAt(i));
        }


    }
}
```

Code2:

```
importjava.util.*;
class First
{
void display()
  {
System.out.println("display in First class");
  }
}

class Demo
{
public static void main(String args[ ])
  {
   Vector v=new Vector(3,2);
System.out.println(v.capacity());
System.out.println(v.size());

v.addElement("One");
v.addElement("Two");
v.addElement("Three");

System.out.println(v.capacity());
System.out.println(v.size());

v.addElement("Four");

System.out.println(v.capacity());
System.out.println(v.size());

for(int i=0; i<v.size(); i++)
System.out.println(v.elementAt(i));

v.insertElementAt("Five",1);

for(int i=0; i<v.size(); i++)
System.out.println(v.elementAt(i));

v.removeElementAt(1) ;

for(int i=0; i<v.size(); i++)
System.out.println(v.elementAt(i));

    Object obj[ ]=new Object[v.size()];
v.copyInto(obj);

for(int i=0; i<obj.length; i++)
System.out.println((String)obj[i]);

    First f=new First();
v.addElement(f);

     First f1=(First)v.elementAt(4);
f1.display();
  }
}
```

No.	ArrayList	Vector
1)	ArrayList is not synchronized.	Vector is synchronized.
2)	ArrayList is not a legacy class.	Vector is a legacy class.
3)	ArrayList increases its size by 50% of the array size.	Vector increases its size by doubling the array size.

The List Interface

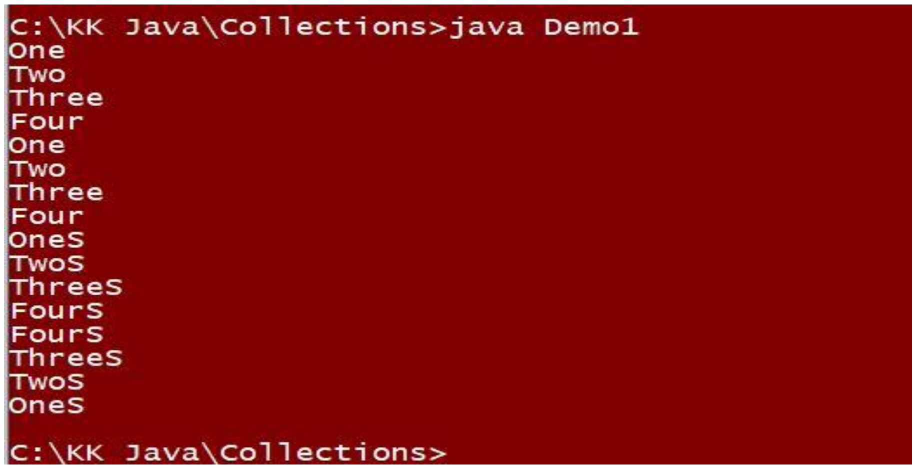
- The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index. A list may contain duplicate elements.
- In addition to the methods defined by **Collection**, **List** defines some of its own, which are shown below

Iterator and ListIterator:

- **An Iterator is an interface in Java and we can traverse the elements of a list in a forward direction whereas a ListIterator is an interface that extends the Iterator interface and we can traverse the elements in both forward and backward directions**

// Use of Iterator and ListIterator

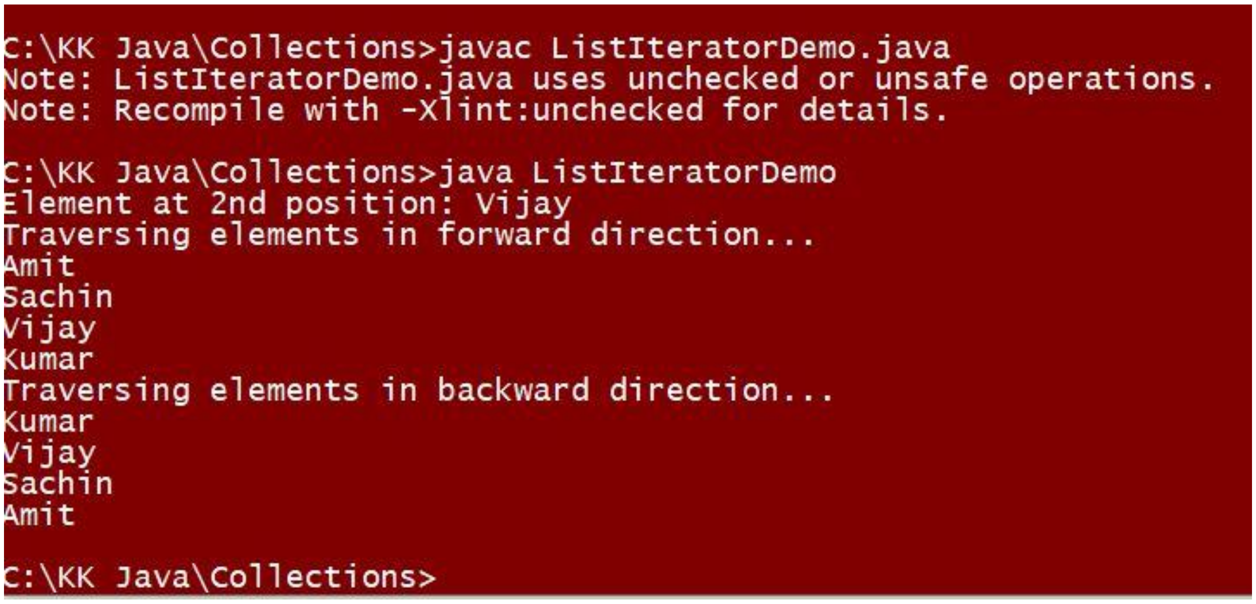
```
Import java.util.*;
class Demo
{
        public static void main(String args[ ])
        {
                ArrayList al=new ArrayList();
                al.add("One");
                al.add("Two");
                al.add("Three");
                al.add("Four");
                Iterator it=al.iterator();
                while(it.hasNext())
                {
                        System.out.println((String)it.next());
                }
                ListIterator lit=al.listIterator();
                //Modifying collection
                while(lit.hasNext())
                {
                        Object o=(String)lit.next();
                        System.out.println(o);
                        lit.set(o + "S");
                }
                lit=al.listIterator();
                while(lit.hasNext())
                {
                        Object o=(String)lit.next();
                        System.out.println(o);
                }
                //Iterating collection in reverse
                while(lit.hasPrevious())
                {
                     Object o=(String)lit.previous();
                    System.out.println(o);
                }

        }
}
```



```
C:\KK Java\Collections>java Demo1
One
Two
Three
Four
One
Two
Three
Four
OneS
TwoS
ThreeS
FourS
FourS
ThreeS
TwoS
OneS

C:\KK Java\Collections>
```

Code:

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

                ArrayList al=new ArrayList();
                al.add("Amit");
                al.add("Vijay");
                al.add("Kumar");
                al.add(1,"Sachin");
                System.out.println("Element at 2nd position: "+al.get(2));
                ListIteratoritr=al.listIterator();
                System.out.println("Traversing elements in forward direction...");
                while(itr.hasNext())
                {
                        System.out.println(itr.next());
                }
                System.out.println("Traversing elements in backward direction...");
                while(itr.hasPrevious())
                {
                        System.out.println(itr.previous());
                }
        }

}
```



```
C:\KK Java\Collections>javac ListIteratorDemo.java
Note: ListIteratorDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\KK Java\Collections>java ListIteratorDemo
Element at 2nd position: Vijay
Traversing elements in forward direction...
Amit
Sachin
Vijay
Kumar
Traversing elements in backward direction...
Kumar
Vijay
Sachin
Amit

C:\KK Java\Collections>
```

Difference between List and Set: List can contain duplicate elements whereas Set contains unique elements only.

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.

Methods:

- **Object put(object key,Object value):** is used to insert an entry in this map.
- **Void putAll(Map map):**is used to insert the specified map in this map.
- **Object remove(object key):**is used to delete an entry for the specified key.
- **Object get(Object key):**is used to return the value for the specified key.
- **Boolean containsKey(Object key):**is used to search the specified key from this map.
- **Boolean containsValue(Object value):**is used to search the specified value from this map.
- **Set keySet():**returns the Set view containing all the keys.
- **Set entrySet():**returns the Set view containing all the keys and values.

Entry: Entry is the sub interface of Map. So we will access it by Map. Entry name. It provides methods to get key and value.

Methods of Entry interface:

1. **Object getKey ():** is used to obtain key.
2. **Object getValue():**is used to obtain value.

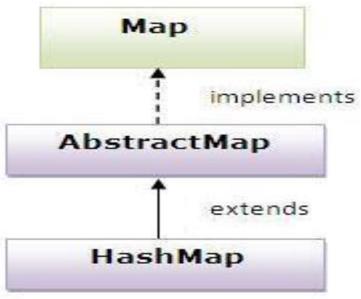
The Map Classes

- Several classes provide implementations of the map interfaces. Two classes that can be used for maps are shown below

Note: AbstractMap Implements most of the Map interface.

The 'HashMap' class:--

- **HashMap** implements **Map** and Extends **AbstractMap** to use a hash table. It is **unsorted, unordered** map.
- The position of key in the map is dependent on hashcode. HashMap allows one null key and multiple null values.
- This map provides faster iteration but slower insertion or deletion.
- It is used for random access.



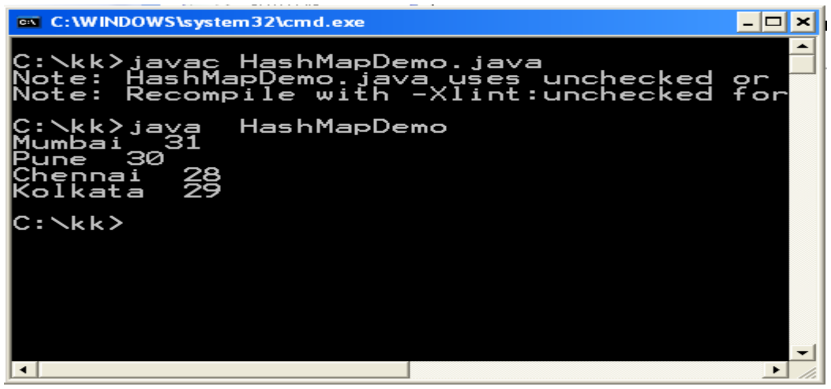
Methods:

- **value put (key, value)** This method stores key-value pair into the HashMap.
- **value get (Object key)** This method returns the corresponding value when key is given. If the key does not have a value associated with it, then it returns null.
- **Set keyset()** This method, when applied on a HashMap converts it into a set where only keys will be stored.
- **Collection values()**This method, when applied on a HashMap object returns all the values of the HashMap into a Collection object.
- **value remove (Object key)** This method removes the key and corresponding value from the HashMap.

- **void clear ()** This method removes all the key-value pairs from the map.
- **Boolean isEmpty ()** This method returns true if there are no key-value pairs in theHashMap.
- **int size ()** This method returns number of key-value pairs in the HashMap.
- **Object put(object key,Object value):** is used to insert an entry in this map.
- **Public Boolean containsKey(Object key):**is used to search the specified key from this map.
- **Boolean containsValue(Object value):**is used to search the specified value from this map.
- **public Set keySet():**returns the Set view containing all the keys.
- **public Set entrySet():**returns the Set view containing all the keys and values.

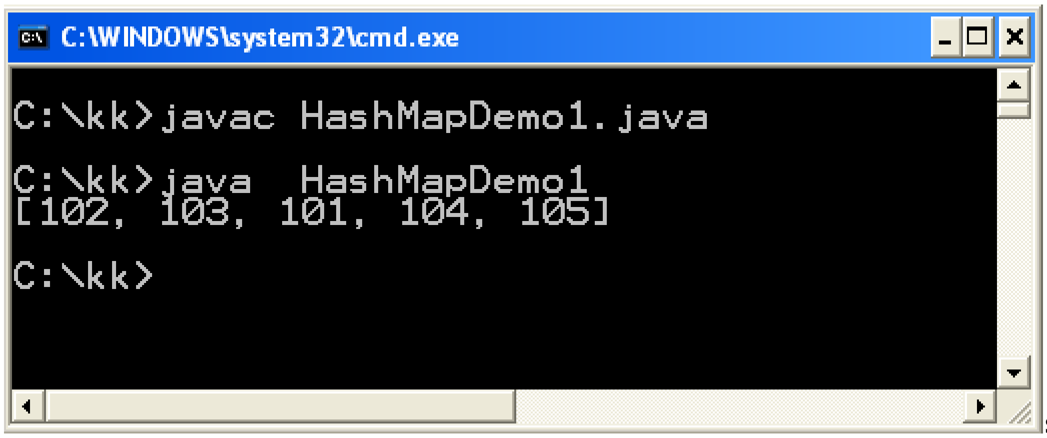
Example:--

```
import java.util.*;

class HashMapDemo
{
public static void main(String args[ ])
   {
HashMaphm=new HashMap();

hm.put("Mumbai", 31);
hm.put("Pune", 30);
hm.put("Chennai", 28);
hm.put("Kolkata", 29);

     //get set of entries
    Set set=hm.entrySet();

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

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

   }
}
```



```
C:\kk>javac HashMapDemo.java
Note: HashMapDemo.java uses unchecked or
Note: Recompile with -Xlint:unchecked for

C:\kk>java  HashMapDemo
Mumbai   31
Pune   30
Chennai   28
Kolkata   29

C:\kk>
```

```
import java.util.*;

class HashMapDemo1
{
        public static void main(String args[])
        {
                HashMap<Integer, String>hm = new HashMap<Integer, String> ();
                hm.put (new Integer (101),"Naresh");
                hm.put (new Integer (102),"Rajesh");
                hm.put (new Integer (103),"Suresh");
                hm.put (new Integer (104),"Mahesh");
                hm.put (new Integer (105),"Ramesh");
                Set<Integer> set = new HashSet<Integer>();
                set = hm.keySet();
                System.out.println (set);
        }
}
```



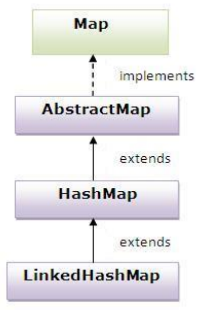
```
C:\kk>javac HashMapDemo1.java

C:\kk>java  HashMapDemo1
[102, 103, 101, 104, 105]

C:\kk>
```

s

The ‘LinkedHashMap’ class:--

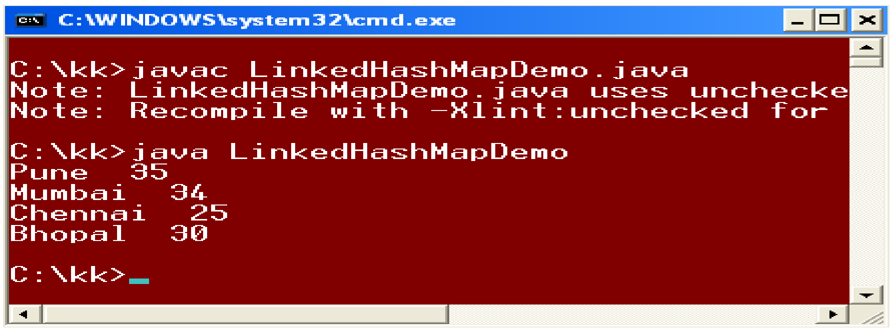
- This class extends HashMap class. It is ordered but non-sorted map. It maintains insertion order.
- This map provides faster insertion or deletion but slower iteration.
- A LinkedHashMap contains values based on the key.

Hierarchy of LinkedHashMap class



Example**:-**

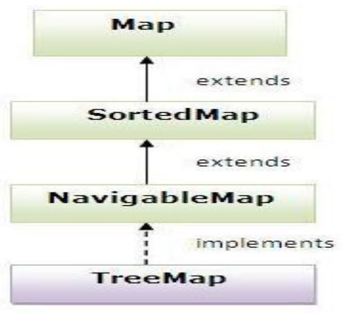
```
import java.util.*;
class Demo
{
public static void main(String args[ ])
   {
LinkedHashMap lhm=new LinkedHashMap();

lhm.put("Pune", 35);
lhm.put("Mumbai", 34);
lhm.put("Chennai", 25);
lhm.put("Bhopal", 30);
     Set s=lhm.entrySet();
     Iterator it=s.iterator();

while(it.hasNext())
      {
Map.Entry me=(Map.Entry)it.next() ;
System.out.println(me.getKey() + "  " + me.getValue());
      }

   }
}
```



```
C:\kk>javac LinkedHashMapDemo.java
Note: LinkedHashMapDemo.java uses unchecke
Note: Recompile with -Xlint:unchecked for

C:\kk>java LinkedHashMapDemo
Pune   35
Mumbai   34
Chennai   25
Bhopal   30

C:\kk>_
```

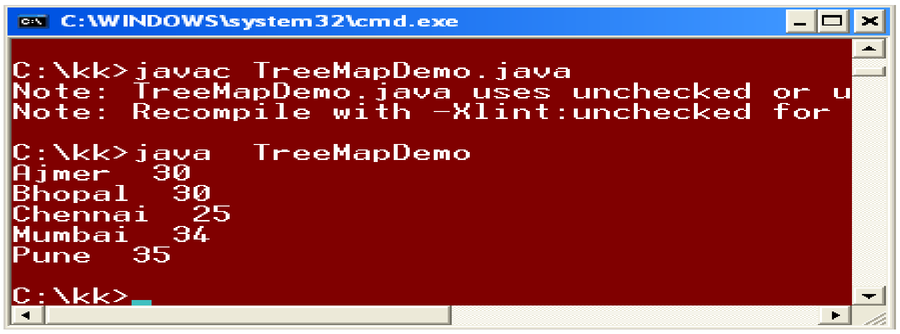
The 'TreeMap' class:--

- This class implements **NavigableMap** interface. It is **sorted** map. The elements are stored in ascending Key order.
- 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
- **TreeMap(class): Sorted, low performance.**



Example:-
```
import java.util.*;

class Demo
{
public static void main(String args[ ])
   {
TreeMap tm=new TreeMap();

tm.put("Pune", 35);
tm.put("Mumbai", 34);
tm.put("Chennai", 25);
tm.put("Bhopal", 30);
tm.put("Ajmer", 30);
```

```
    Set s=tm.entrySet();

    Iterator it=s.iterator();

while(it.hasNext())
     {
Map.Entry me=(Map.Entry)it.next() ;
System.out.println(me.getKey() + "  " + me.getValue());
     }

  }
}
```



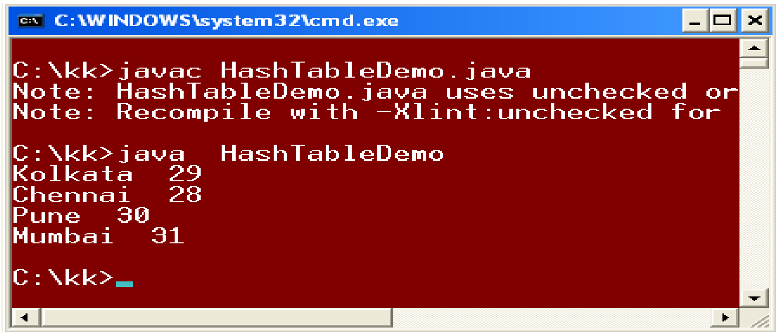
```
C:\kk>javac TreeMapDemo.java
Note: TreeMapDemo.java uses unchecked or u
Note: Recompile with -Xlint:unchecked for

C:\kk>java  TreeMapDemo
Ajmer   30
Bhopal   30
Chennai   25
Mumbai   34
Pune   35

C:\kk>
```

The ‘Hashtable’ class:--

- This class is same as HashMap but all methods are synchronized.
- HashMap allows null values as well as one null key.
- But Hashtable doesn’t allow anything that is null.
- **HashTable(class):**All methods are synchronized. It is same as that of Hash Map but all methods are synchronized. In other Maps , one key and multiple values can be null .

Example:
Same as above. Put Hashtable instead of HashMap. And check out sequence.

```
import java.util.*;

class HashTableDemo
{
public static void main(String args[ ])
  {
Hashtable hm=new Hashtable();

hm.put("Mumbai", 31);
hm.put("Pune", 30);
hm.put("Chennai", 28);
hm.put("Kolkata", 29);

    //get set of entries
    Set set=hm.entrySet();

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

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

   }
}
```



```
C:\kk>javac HashTableDemo.java
Note: HashTableDemo.java uses unchecked or
Note: Recompile with -Xlint:unchecked for

C:\kk>java  HashTableDemo
Kolkata  29
Chennai  28
Pune  30
Mumbai  31

C:\kk>_
```

Difference between HashMap and Hashtable?

HashMap	Hashtable
1) HashMap is **not synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread-safe and can be shared with many threads.
2) HashMap **allows one null key and multiple null values**.	Hashtable **doesn't allow any null key or value**.
3) HashMap is a **new class introduced in JDK 1.2**.	Hashtable is a **legacy class**.
4) HashMap is **fast**.	Hashtable is **slow**.
5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.

6) HashMap is **traversed by Iterator**.	Hashtable is **traversed by Enumerator and Iterator**.
7) Iterator in HashMap is **fail-fast**.	Enumerator in Hashtable is **not fail-fast**.
8) HashMap inherits **AbstractMap** class.	Hashtable inherits **Dictionary** class.