CHAPTER 16

COLLECTIONS

Introduction

Utility means usefulness. Utility class are those that help in creation of other classes. The *Java.util* package contains utility classes such as Date, Calender, Stack, LinketList, List, Set & Map etc. The classes in *Java.util* package use collections, i.e. a group of objects. Collection classes work on objects, i.e. they store/retrieve/manipulate objects.

A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers). If you have used the Java programming language — or just about any other programming language — you are already familiar with collections.

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

Note: Primitive data types like int, float, double, char and long cannot be directly store and retrieve in collections. Primitive data types may be converted into object types by using the wrapper classes stored in the *java.lang* package.

What Is a Collections Framework?

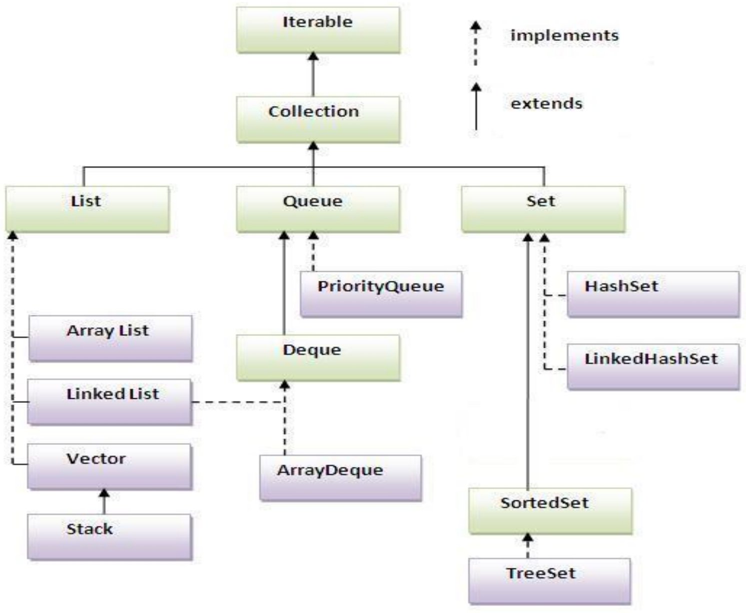
A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **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(classes):** These are the 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.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy. Historically, collections frameworks have been quite complex, which gave them a reputation for having a steep learning curve. We believe that the Java Collections Framework breaks with this tradition, as you will learn for yourself in this chapter.

Hierarchy of Collection Framework

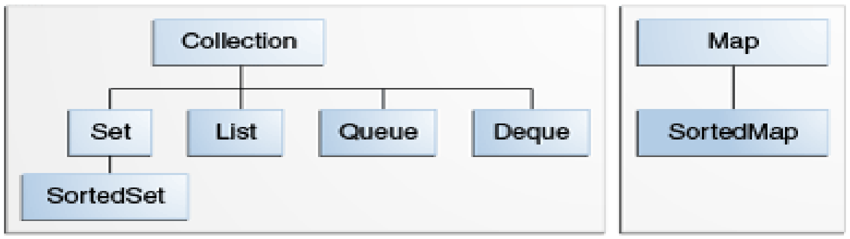
Let us see the hierarchy of collection framework.The **java.util** package contains all the classes and interfaces for Collection framework.



The core collection interfaces.

Interface	**Description**
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue. (Added by Java SE 6.)
List	Extends Collection to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches.
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted set

Interfaces



Collections are primarily defined through a set of interfaces.

The collections framework defines several interfaces. This section provides an overview of each interface:

SN	Interfaces with Description
1	The Collection Interface This enables you to work with groups of objects; it is at the top of the collections hierarchy.
2	The List Interface This extends **Collection** and an instance of List stores an ordered collection of elements.
3	The Set This extends Collection 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 SortedMap This extends Map so that the keys are maintained in ascending order.

Interface Type	Implemented by
Set	HashSet, LinkedHashSet, EnumSet
SortedSet	TreeSet
List	Vector, Stack, ArrayList, LinkedList
Queue	PriorityQueue, LinkedList
Map	Hashtable, HashMap, LinkedHashMap, WeakHashMap, IdentityHashMap
SortedMap	TreeMap

VECTORS

We have seen that Java supports the concept of variable length arguments to methods. This features can also be achieved in Java through the use of the **Vector** class contained in the *java.util* package. This class can be used to create a generic dynamic array known as vector that can hold object of any type and any number. The objects do not have to be homogeneous. Array can be easily implemented as vectors, Vector are created like array as follows:

Vector list = new Vector( ) ; // declaring without size

Vector list = new Vector(3); // declaring with size

Note that a vector can be declared without specifying any size explicitly. A vector without size can accommodate an unknown number of items. Even, when a size is specified , this can overlooked and a different number of items may be put into the vector. Remember, in contrast, an array must always have its size specified.

Vector possess a number of advantages over arrays.

1. **It is convenient to use vectors to store objects.**
2. **A vector can be used to store a list of objects that may vary in size.**
3. **We can add and delete objects from the list as and when required.**

A major constraint in using vectors is that we cannot directly store primitive or simple data type in a vector, we can only store object. Therefore, we need to convert simple type into object types by using wrapper classes.

Vector class supports a number of methods that can be used to manipulate the vectors created.

Method Call	**Task Performed**
list.addElement(<item>)	Add the item specified to the list at the end.
list.elementAt(5)	Gives the name of the 5th object.
list.size()	Gives the number of objects present
list.removeElement(<item>)	Remove the specified item from the list.
list.removeElementAt(<n>)	Remove the item stored in the nth position of the list
list.removeAllElements( )	Remove all the elements in the list.
list.copyInto(<array>)	Copies all items from list to array.
list.insertElementsAt(<item>, <n>)	Insert the items at nth position.

Below program illustrates the use of arrays, strings and vectors. This program converts a string vector into an array of strings and display the strings.

```
Import java.util.*;
class LanguageVector
{
   public static void main(String args[])
   {
       Vector list = new Vector();
        int l = args.length;
         for(int i = 0;i<l;i++)
        {
             list.addElement(args[i]);
         }
          list.insertElementAt("COBOL",2);
          int size = list.size();
          String listArray[] = new String[size];
          list.copyInto(listArray);
          System.out.println("List of Languages");
           for(int i =0; i< size; i++)
           {
                  System.out.println(listArray[i]);
            }
    }
}
```

The ArrayList Class

The **ArrayList** class extends **AbstractList** and implements the **List** interface. **ArrayList** is a generic class that has this declaration:

class ArrayList<E>

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

ArrayList supports dynamic arrays that can grow as needed. In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, the Collections Framework defines **ArrayList**. In essence, an **ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrink.

ArrayList has the constructors shown here:
ArrayList( )
ArrayList(int *capacity*)

The first constructor builds an empty array list. The second constructor builds an array list that has the specified initial *capacity*. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

The following program shows a simple use of **ArrayList**. An array list is created for objects of type **String**, and then several strings are added to it. (Recall that a quoted string is translated into a **String** object.) The list is then displayed. Some of the elements are removed and the list is displayed again.

```
// Demonstrate ArrayList.
import java.util.*;
class ArrayListDemo
{
   public static void main(String args[])
    {
       // Create an array list.
       ArrayList<String> al = new ArrayList<String>();
       System.out.println("Initial size of al: " +
       al.size());
       // Add elements to the array list.
       al.add("C");
       al.add("A");
       al.add("E");
       al.add("B");
       al.add("D");
       al.add("F");
       al.add(1, "A2");
       System.out.println("Size of al after additions: " +
       al.size());
       // Display the array list.
       System.out.println("Contents of al: " + al);
       // Remove elements from the array list.
       al.remove("F");
       al.remove(2);
       System.out.println("Size of al after deletions: " +
       al.size());
       System.out.println("Contents of al: " + al);
    }
}
```

The output from this program is shown here:

```
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]
```

Two ways to iterate the elements of collection:

1. By for-each loop.
2. By Iterator interface.

1. Iterating the elements of Collection by for-each loop:

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

  ArrayList al=new ArrayList();
```

```
    al.add("Ravi");
    al.add("Vijay");
    al.add("Ravi");
    al.add("Ajay");

    for(Object obj:al)
      System.out.println(obj);
   }
  }
```

```
Output:Ravi
       Vijay
       Ravi
       Ajay
```

2. Iterating the elements of Collection by Iterator interface:

```
import java.util.*;
class Simple
{
     public static void main(String args[])
     {
               ArrayList al=new ArrayList();
               al.add("Ravi");
               al.add("Vijay");
               al.add("Ravi");
               al.add("Ajay");
               Iterator itr=al.iterator();
               while(itr.hasNext())
               {
                  System.out.println(itr.next());
               }
          }
    }
```

```
Output:Ravi
       Vijay
       Ravi
       Ajay
```

```
ArrayList<String> stringList = new ArrayList<String>(); //Generic ArrayList to Store only
                                                         //String objects
```

2) Putting an Item into ArrayList

Second line will result in compilation error because **this Java ArrayList will only allow String elements**.

```
stringList.add("Item"); //no error because we are storing String
stringList.add(new Integer(2)); //compilation error
```

or

```
int n = 2
Integer  nobj = new Integer(n);
stringList.add(nobj);  //compilation error
```

Storing user-defined class objects:

```
class Student{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}
}
import java.util.*;
class Simple{
 public static void main(String args[]){

  Student s1=new Student(101,"Sonoo",23);
  Student s2=new Student(102,"Ravi",21);
  Student s2=new Student(103,"Hanumat",25);

  ArrayList al=new ArrayList();
  al.add(s1);
  al.add(s2);
  al.add(s3);

  Iterator itr=al.iterator();
  while(itr.hasNext())
{
    Student st=(Student) itr.next();
    System.out.println(st.rollno+" "+st.name+" "+st.age);
  }
 }
}
```

3) Checking size of ArrayList

Size of an ArrayList in Java is total number of elements currently stored in ArrayList.

```
int size = stringList.size();
```

4) Checking Index of an Item in Java ArrayList

You can use indexOf() method of ArrayList in Java to find out index of a particular object.

```
int index = stringList.indexOf("Item"); //location of Item object in List
```

5) Retrieving Item from ArrayList in a loop

Many a times we need to **traverse on Java ArrayList** and perform some operations on each retrieved item. Here are two ways of doing it without using Iterator. We will see use of Iterator in next section.

```
for (int i = 0; i < stringList.size(); i++)
  String item = stringList.get(i);
  System.out.println("Item " + i + " : " + item);
}

for(String item: stringList)
{
 System.out.println("retrieved element: " + item);
}
```

6) Checking ArrayList for an Item
Sometimes we need to *check whether an element exists in ArrayList in Java* or not for this purpose we can use contains() method of Java. contains() method takes type of object defined in ArrayList creation and returns true if this list contains the specified element.

7) Checking if ArrayList is Empty
We can use **isEmpty() method of Java ArrayList to check whether ArrayList is empty**. isEmpty() method returns true if this ArrayList contains no elements. You can also use size() method of List to check if List is empty

```
boolean result = stringList.isEmpty(); //isEmpty() will return true if List is empty

if(stringList.size() == 0){
  System.out.println("ArrayList is empty");
}
```

8) Removing an Item from ArrayList
There are two ways to **remove any elements from ArrayList in Java**. You can either remove an element based on its index or by providing object itself. Remove remove (int index) and remove (Object o) method is used to remove any element from ArrayList in Java. Since ArrayList allows duplicate its worth noting that remove (Object o) removes the first occurrence of the specified element from this list, if it is present. In below code first call will remove first element from ArrayList while second call will remove first occurrence of item from ArrayList in Java.

```
stringList.remove(0);
stringList.remove(item);
```

9) Copying data from one ArrayList to another ArrayList in Java
Many a times you need to **create a copy of ArrayList** for this purpose you can use addAll(Collection c) method of ArrayList in Java to copy all elements from on ArrayList to

another ArrayList in Java. Below code will add all elements of stringList to newly created copyOfStringList.

```
ArrayList<String> copyOfStringList = new ArrayList<String>();
copyOfStringList.addAll(stringList);
```

10) Replacing an element at a particular index
You can use set (int index, E element) method of java ArrayList to replace any element from a particular index. Below code will replace first element of stringList from "Item" to "Item2".

```
stringList.set(0,"Item2");
```

11) Clearing all data from ArrayList
ArrayList in Java provides clear () method which removes all of the elements from this list. Below code will remote all elements from our stringList and make the list empty. **You can reuse Java ArrayList** after clearing it.

```
stingList.clear();
```

12) Converting from ArrayList to Array in Java

Java ArrayList provides you facility to *get the array back from your ArrayList*. You can use toArray(T[] a) method returns an array containing all of the elements in this list in proper sequence (from first to last element). "a" is the array into which the elements of the list are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose.

here are two versions of **toArray( )**, which are shown again here for your convenience:

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

The first returns an array of **Object**. The second returns an array of elements that have the same type as **T**. Normally, the second form is more convenient because it returns the proper type of array. The following program demonstrates its use:

```
// Convert an ArrayList into an array.
import java.util.*;
class ArrayListToArray {
public static void main(String args[]) {
// Create an array list.
ArrayList<Integer> al = new ArrayList<Integer>();
// Add elements to the array list.
al.add(1);
al.add(2);
al.add(3);
al.add(4);
System.out.println("Contents of al: " + al);
// Get the array.
Integer ia[] = new Integer[al.size()];
ia = al.toArray(ia);
int sum = 0;
// Sum the array.
for(int i : ia) sum += i;
System.out.println("Sum is: " + sum);
}
}
```

The output from the program is shown here:

```
Contents of al: [1, 2, 3, 4]
Sum is: 10
```

The program begins by creating a collection of integers. Next, **toArray( )** is called and it obtains an array of **Integer**s. Then, the contents of that array are summed by use of a for-each style **for** loop.

There is something else of interest in this program. As you know, collections can store only references to, not values of, primitive types. However, autoboxing makes it possible to pass values of type **int** to **add( )** without having to manually wrap them within an **Integer**, as the program shows. Autoboxing causes them to be automatically wrapped. In this way, autoboxing significantly improves the ease with which collections can be used to store primitive values.

13) Creating Synchronized ArrayList
Some times you need to synchronize your ArrayList in java to make it shareable between multiple threads you can use Collections utility class for this purpose as shown below.

```
List list = Collections.synchronizedList(new ArrayList(...));
```

14) Creating ArrayList from Array in Java
ArrayList in Java is amazing you can create even an ArrayList full of your element from an already existing array. You need to use **Arrays.asList(T... a)** method for this purpose which returns a fixed-size list backed by the specified array.

```
ArrayList  stringList = Arrays.asList(new String[]{"One", "Two", "Three");
Or
String List[] = {[]{"One", "Two", "Three");
ArrayList  stringList = Arrays.asList(List);
//this is not read only List you can still update value of existing elements
```

Accessing a Collection via an Iterator

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. One way to do this is to employ an *iterator*, which is an object that implements either the **Iterator** or the **ListIterator** interface. **Iterator** enables you to cycle through a collection, obtaining or removing elements. **ListIterator** extends **Iterator** to allow bidirectional traversal of a list, and the modification of elements.

There are only three methods in the Iterator interface. They are:

Method	Description
boolean hasNext( )	Returns true if there are more elements. Otherwise, returns false.
E next( )	Returns the next element. 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( ) that is not preceded by a call to next( ).

Iterator and **ListIterator** are generic interfaces which are declared as shown here:

interface Iterator<E>
interface ListIterator<E>

Here, **E** specifies the type of objects being iterated. The **Iterator** interface declares the methods shown in above table1. The methods declared by **ListIterator** are shown in Table given below.. In both cases, operations that modify the underlying collection are optional. For example, **remove( )** will throw **UnsupportedOperationException** when used with a read-only collection. Various other exceptions are possible.

Using an Iterator

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an **iterator( )** method that returns an **iterator** to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:

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( )**. Have the loop iterate as long as **hasNext( )** returns **true**.
3. Within the loop, obtain each element by calling **next( )**.

Method	**Description**
boolean hasNext( )	Returns true if there are more elements.

	Otherwise, returns false.
E next( )	Returns the next element. 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( ) that is not preceded by a call to next( ).

TABLE The Methods Defined by Iterator

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( ).	
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, return -1.
void remove( )	Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) 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( ).

```
// Demonstrate iterators.
import java.util.*;
class IteratorDemo {
public static void main(String args[]) {
// Create an array list.
ArrayList<String> al = new ArrayList<String>();
// Add elements to the array list.
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
// Use iterator to display contents of al.
System.out.print("Original contents of al: ");
Iterator<String> itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
// Modify objects being iterated.
ListIterator<String> litr = al.listIterator();
while(litr.hasNext()) {
String element = litr.next();
litr.set(element + "+");
}
System.out.print("Modified contents of al: ");
itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
// Now, display the list backwards.
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
String element = litr.previous();
System.out.print(element + " ");
}
System.out.println();
}
}
```

The output is shown here:

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


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

Storing user-defined class objects:

class Student{

```
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}
}
```

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

  Student s1=new Student(101,"Sonoo",23);
  Student s2=new Student(102,"Ravi",21);
  Student s2=new Student(103,"Hanumat",25);

  ArrayList al=new ArrayList();
  al.add(s1);
  al.add(s2);
  al.add(s3);

  Iterator itr=al.iterator();
  while(itr.hasNext())
{
    Student st=(Student) itr.next();
    System.out.println(st.rollno+" "+st.name+" "+st.age);
  }
 }
}
```

```
Output:101 Sonoo 23
       102 Ravi 21
       103 Hanumat 25
```

Example of addAll(Collection c) method:

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

  ArrayList al=new ArrayList();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ajay");

  ArrayList al2=new ArrayList();
  al2.add("Sonoo");
  al2.add("Hanumat");

  al.addAll(al2);

  Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

```
Output:Ravi
       Vijay
       Ajay
       Sonoo
       Hanumat
```

Example of removeAll() method:

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

  ArrayList al=new ArrayList();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ajay");

  ArrayList al2=new ArrayList();
  al2.add("Ravi");
  al2.add("Hanumat");

  al.removeAll(al2);

  System.out.println("iterating the elements after removing the elements of al2...");
  Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }

  }
}
```

```
Output:iterating the elements after removing the elements of al2...
       Vijay
       Ajay
```

Example of retainAll() method:

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

  ArrayList al=new ArrayList();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ajay");

  ArrayList al2=new ArrayList();
  al2.add("Ravi");
  al2.add("Hanumat");

  al.retainAll(al2);

  System.out.println("iterating the elements after retaining the elements of al2...");
  Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

```
Output:iterating the elements after retaining the elements of al2...
       Ravi
```

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. **Set** is a generic interface that has this declaration:

interface Set<E>

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

The SortedSet Interface

The **SortedSet** interface extends **Set** and declares the behavior of a set sorted in ascending order. **SortedSet** is a generic interface that has this declaration:

interface SortedSet<E>

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

List Interface:

List Interface is the subinterface of Collection. It contains methods to insert and delete elements in index basis.

Commonly used methods of List Interface:

1. public void add(int index,Object element);
2. public boolean addAll(int index,Collection c);
3. public object get(int Index position);
4. public object set(int index,Object element);
5. public object remove(int index);
6. public ListIterator listIterator();
7. public ListIterator listIterator(int i);

ListIterator Interface:

ListIterator Interface is used to traverse the element in backward and forward direction.

Commonly used methods of ListIterator Interface:

1. public boolean hasNext();
2. public Object next();
3. public boolean hasPrevious();
4. public Object previous();

Example of ListIterator Interface:

```
import java.util.*;
class Simple5{
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));

ListIterator itr=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());
 }
}
}
```

```
Output:element at 2nd position: Vijay
       traversing elements in forward direction...
       Amit
       Sachin
       Vijay
       Kumar
       traversing elements in backward direction...
       Kumar
       Vijay
       Sachin
       Amit
```

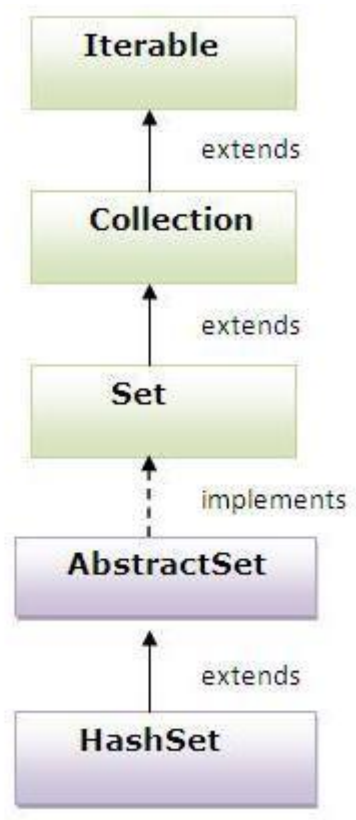
Difference between List and Set:

List can contain duplicate elements whereas Set contains unique elements only.

HashSet class:

- uses hashtable to store the elements. It extends AbstractSet class and implements Set interface.
- contains unique elements only.

Hierarchy of HashSet class:



Example of HashSet class:

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

  HashSet al=new HashSet();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");

  Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Output:Ajay

Vijay
Ravi

Map Interface

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

Commonly used methods of Map interface:

1. **public Object put(object key,Object value):** is used to insert an entry in this map.
2. **public void putAll(Map map):**is used to insert the specified map in this map.
3. **public Object remove(object key):**is used to delete an entry for the specified key.
4. **public Object get(Object key):**is used to return the value for the specified key.
5. **public boolean containsKey(Object key):**is used to search the specified key from this map.
6. **public boolean containsValue(Object value):**is used to search the specified value from this map.
7. **public Set keySet():**returns the Set view containing all the keys.
8. **public Set entrySet():**returns the Set view containing all the keys and values.

Entry

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

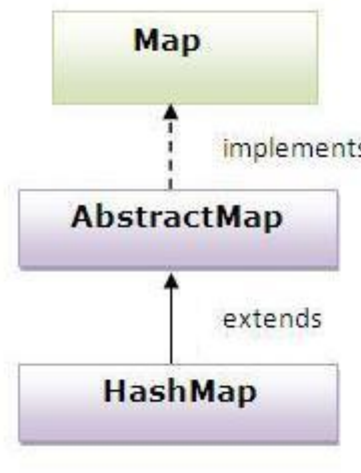
Methods of Entry interface:

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

HashMap class:

- A HashMap contains values based on the key. It implements the Map interface and extends AbstractMap class.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It maintains no order.

Hierarchy of HashMap class:



Example of HashMap class:

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

  HashMap hm=new HashMap();

  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");

  Set set=hm.entrySet();
  Iterator itr=set.iterator();

  while(itr.hasNext()){
   Map.Entry m=(Map.Entry)itr.next();
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

```
Output:102 Rahul
       100 Amit
       101 Vijay
```

What is difference between HashSet and HashMap?

HashSet contains only values whereas HashMap contains entry(key and value).

Basic collection types

The first part of the decision is choosing what *"basic category"* of organisation or functionality your data needs to have. The broad types are as follows:

Collection type	**Functionality**	**Typical uses**
List	• Essentially a variable-size array; • You can usually add/remove items at any arbitrary position; • The order of the items is well defined (i.e. you can say what position a given item goes in in the list).	Most cases where you just need to store or iterate through a "bunch of things" and later iterate through them.
Set	• Things can be "there or not"— when you add items to a set, there's no notion of *how many times* the item was added, and usually no notion of ordering.	• Remembering "which items you've already processed", e.g. when doing a web crawl; • Making other *yes-no decisions* about an item, e.g. "is the item a word of English", "is the item in the database?" , "is the item in this category?" etc.
Map	• Stores an *association* or mapping between "keys" and "values"	Used in cases where you need to say "for a given X, what is the Y"? It is often useful for implementing in-memory caches or indexes. For example: • For a given user ID, what is their cached name/User object? • For a given IP address, what is the cached country code? • For a given string, how many instances have I seen?
Queue	• Like a list, but where you **only ever access the *ends*** of the list (typically, you add to one end and remove from the other).	• Often used in **managing tasks** performed by different threads in an application (e.g. one thread receives incomming connections and puts them on a queue; other "worker" threads take connections off the queue for

		processing); • For **traversing hierarchical structures** such as a filing system, or in general where you need to remember **"what data to process next"**, whilst also adding to that list of data; • Related to the previous point, queues crop up in various algorithms, e.g. build the encoding tree for *Huffman compression*.

ENUMERATED TYPES

Java allows us to use the enumerated type in Java using the **enum** keyword. This keyword can be used similar to the static final constant in the earlier version of Java. For example consider the following code:

```
public class Days
{
        public static final int DAY_SUNDAY=0;
        public static final int DAY_MONDAY=1;
        public static final int DAY_TUESDAY=2;
        public static final int DAY_WEDNESDAY=3;
        public static final int DAY_THURSDAY=4;
        public static final int DAY_FRIDAY=5;
        public static final int DAY_SATURDAY=6;
}
```

Using enumerated type features provided by J2SE 5.0 the above code can be rewritten as:

```
public enum
Day{SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY}
```

The advantages of using the enumerated type are:

- Compile-time Safety
- We can use the enum keyword in switch statements.

```
public class WorkingDays
{
    enum Days
    {
            Sunday,
            Monday,
            Tuesday,
            Wednesday,
            Thursday,
            Friday,
            Saturday
       }
public static void main(String args[])
{
            for(Days d :  Days.values( ) )
               {
                        Weekend(d);
                }
}
private static void weekend(Days d)
{
    If(d.equals(Days.Sunday)
            System.out.println("Value = " + d + " is a Holiday");
    else
            System.out.println("Value = " + d + "is a working day");
 }
}
```