Collections in Java

A collection in java is a framework that provides architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Java Collection simply means a single unit of objects.

Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

Q. What is Collection in java

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

Q. What is framework in java

(i) provides readymade architecture.

(ii) represents set of classes and interface.

(iii) is optional.

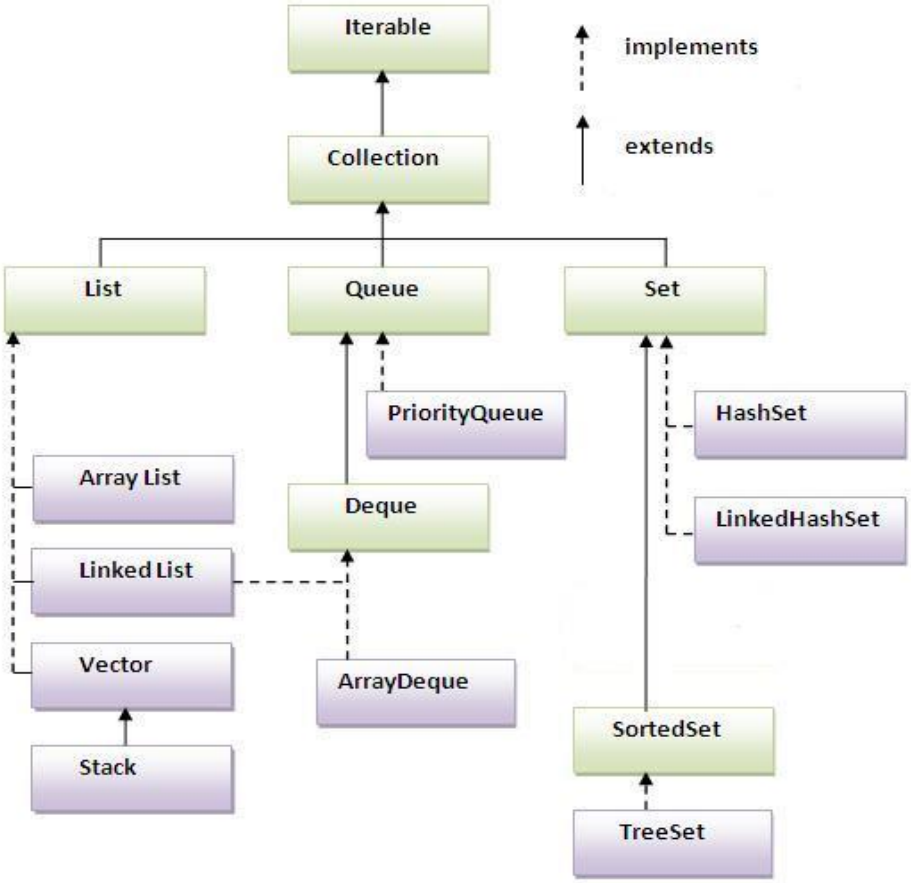
Q. What is Collection framework

Collection framework represents a unified architecture for storing and manipulating group of objects.

It has Interfaces and its implementations i.e. classes

Hierarchy of Collection Framework

The java.util package contains all the classes and interfaces for Collection framework.



Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(Object element)	is used to insert an element in this collection.
2	public boolean addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	is used to delete an element from this collection.
4	public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public boolean retainAll(Collection c)	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)	is used to search an element.
9	public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
10	public Iterator iterator()	returns an iterator.
11	public Object[] toArray()	converts collection into array.
12	public boolean isEmpty()	checks if collection is empty.
13	public boolean equals(Object element)	matches two collection.
14	public int hashCode()	returns the hashcode number for collection.

Iterator interface

Iterator interface provides the facility of iterating the elements in forward direction only.

Methods of Iterator interface

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

(i) public boolean hasNext() : It returns true if iterator has more elements.

(ii) public object next() : It returns the element and moves the cursor pointer to the next element.

(iii) public void remove() : It removes the last elements returned by the iterator. It is rarely used.

CLASS

I] ArrayList

(i) ArrayList class uses a dynamic array for storing the elements. It extends AbstractList class and implements List interface.

(ii) ArrayList class can contain duplicate elements.

(iii) ArrayList class maintains insertion order.

(iv) ArrayList class is non synchronized.

(v) ArrayList allows random access because array works at the index basis.

(vi) In ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

(vii) The default capacity of ArrayList is 10.

ArrayList has the following constructor:

1. ArrayList();
2. ArrayList(int capacity);
3. ArrayList(Collection c);

Methods

***Add element**

(i) public boolean add(Object e)

--Appends the specified object/element to the end of the list.

(ii) public void add(int index,Object element)

--Insert the specified element at the specified position in the list. Shift the element currently at that position to the right.

***Retrieving elements**

The **get(index)** method is used to retrieve an element from the list at a specified index.

Eg:

String x=al.get(2); //get an element at third position

int y=al.get(2);

***Updating elements**

Use the **set(index,element)** method to replace the element at specified index by the specified element.

Eg:

al.set(2,"Manjeet");

***Removing single element**

To remove an element from the list, use the **remove(index)** method which removes the element at the specified index.

Eg:

al.remove(2); //remove element at 3rd position

Note: if index is out of range java.lang.IndexOutOfBoundsException is thrown.

***Remove all elements**

To remove all elements in the list, use the **clear()** method.

Eg: public void clear();

al.clear();

*** public int size()**

The size() method return the total number of elements in the collection.

***Searching for an element in a list**

public boolean contains(Object element)

--is used to search an element.

Eg:

```
ArrayList<String> al=new ArrayList<String>();
al.add("Ajay");
al.add("Mantosh");
al.add("Rakesh");
al.add("Pradeep");
if(al.contains("Mantosh"))
   System.out.println("Found");
else
   System.out.println("Not Found");
```

*Sorting a list

The simplest way to sort out elements in a list is using the Collection.sort() static method which sorts the specified list into ascending order, based on natural ordering of its elements.

Eg:

Colloections.sort(al);

Collection.reverse(al);

Java Non-generic Vs Generic Collection

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Let's see the old non-generic example of creating java collection.

ArrayList al=new ArrayList();//creating old non-generic arraylist

Let's see the new generic example of creating java collection.

ArrayList<String> al=new ArrayList<String>();//creating new generic arraylist

In generic collection, we specify the data type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives compile time error.

Example of Java ArrayList class

```
import java.util.*;

class demoArrayList{

 public static void main(String args[]){

 ArrayList<String> al=new ArrayList<String>();//creating arraylist
```

```
 al.add("Santosh");//adding object in arraylist
 al.add("Rajiv");
 al.add("Pradeep");
 al.add("Raju");
 // retrieves element at 4th postion
  String  retval=a1.get(3);
  System.out.println("Retrieved element is = " + retval);
 Iterator itr=al.iterator();//getting Iterator from arraylist to traverse elements
 while(itr.hasNext()){
  System.out.println(itr.next());
 }
}
}
```

Two ways to iterate the elements of collection in java

(i) By Iterator interface.
(ii) By for-each loop.

In the above example, we have seen traversing ArrayList by Iterator. Let's see the example to traverse ArrayList elements using for-each loop.

```
import java.util.*;

class demoArrayList1{

 public static void main(String args[]){

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

    al.add("Santosh");//adding object in arraylist

  al.add("Rajiv");

  al.add("Pradeep");

  al.add("Raju");

  for(String obj:al)

    System.out.println(obj);

 }

}
```

User-defined class objects in Java ArrayList

(I)

```
class Student{

 int rollno;

 String name;

 int age;

 Student(int rollno,String name,int age){

  this.rollno=rollno;
```

```
        this.name=name;
        this.age=age;    }



}
```

(II)

```
import java.util.*;
public class TestCollection3{
 public static void main(String args[]){
  //Creating user-defined class objects
  Student s1=new Student(101,"Sonoo",23);
  Student s2=new Student(102,"Ravi",21);
  Student s2=new Student(103,"Hanumat",25);

  ArrayList<Student> al=new ArrayList<Student>();//creating arraylist
  al.add(s1);//adding Student class object
  al.add(s2);
  al.add(s3);



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

```
 }
}
```

*addAll(Collection c) method

This method is used to insert the specified collection elements in the invoking collection.

Eg:

```
import java.util.*;
class TestCollection4{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ajay");
  ArrayList<String> al2=new ArrayList<String>();
  al2.add("Sonoo");
  al2.add("Hanumat");

al.addAll(al2);
    Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
 }
```

*removeAll() method

This method is use ot delete all the elements of specified collection from the invoking collection.

Eg:

```
import java.util.*;
class TestCollection5{
 public static void main(String args[]){
   ArrayList<String> al=new ArrayList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ajay");
      ArrayList<String> al2=new ArrayList<String>();
  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());
  }    } }
```

O/P--iterating the elements after removing the elements of al2...

Vijay

Ajay

* retainAll() method

This method is used to delete all the elements of invoking collection except the specified collection.

Eg:

```
import java.util.*;

class TestCollection6{

 public static void main(String args[]){

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

  al.add("Ravi");

  al.add("Vijay");

  al.add("Ajay");

  ArrayList<String> al2=new ArrayList<String>();

  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());

}

}

}
```

o/p - Ravi

II] LinkedList

(i) LinkedList class uses doubly linked list to store the elements. It extends the AbstractList class and implements List and Deque interfaces.

(ii) LinkedList class can contain duplicate elements.

(iii) LinkedList class maintains insertion order.

(iv) LinkedList class is non synchronized.

(v) In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.

(vi) LinkedList class can be used as list, stack or queue.

(vii) The default capacity of LinkedList is zero.

Eg:-

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

  LinkedList<String> al=new LinkedList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");

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

}

***public void addFirst(Object element)**

--Insert the specified elements at the beginning of the list.

***public void addLast(Object element)**

--Append the specified element to the end of the list.

Difference between ArrayList and LinkedList

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

ArrayList

1) ArrayList internally uses dynamic array 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.

3) ArrayList class can act as a list only because it implements List only.

4) ArrayList is better for storing and accessing data.

LinkedList

1) LinkedList internally uses doubly linked list to store the elements.
2) Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
3) LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) LinkedList is better for manipulating data.

Example of ArrayList and LinkedList in Java

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

  List<String> al=new ArrayList<String>();//creating arraylist
  al.add("Mohit");//adding object in arraylist
  al.add("kuldeep");
  al.add("Rubi");
```

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

  List<String> al2=new LinkedList<String>();//creating linkedlist
  al2.add("sudesh");//adding object in linkedlist
  al2.add("manty");
  al2.add("Swati");
  al2.add("Juhi");
  System.out.println("arraylist: "+al);
  System.out.println("linkedlist: "+al2);
 }
}
```

O/P: arraylist: [Mohit, kuldeep, Rubi,Ajay]
linkedlist: [sudesh, manty,Swati, Juhi]

Java ListIterator Interface

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

Commonly used methods of ListIterator Interface:

- public boolean hasNext();
- public Object next();
- public boolean hasPrevious();
- public Object previous();

```
import java.util.*;

public class demoLI{

public static void main(String args[]){

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

al.add("Amit");

al.add("Vijay");

al.add("Kumar");

al.add(1,"Sachin");
```

```
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());
 }
}
}
```

Vector class

Vector implements a dynamic array to store elements. It is similar to ArrayList but having few differences:-

1. Vector is synchronized while ArrayList is not synchronized.
2. Vector is a legacy class while ArrayList is introduced in JDK 1.2.
3. Vector is slow because it is synchronized while ArrayList is fast because it is non-synchronized.
4. Vector uses Enumeration interface and Iterator interface to traverse the elements while ArrayList uses Iterator interface to traverse the elements.
5. Vector increments 100% means double the array size if total number of element exceed than its capacity while ArrayList increments 50% of current array size if the number of elements exceed from its capacity.

Eg:

```
import java.util.*;

public class DemoVector {

   public static void main(String[] args) {

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

      v.add("Ravi");//method of collection

      v.addElement("Manjeet");//method of Vector


      //Traversing elelements using Enumeration

      Enumeration e=v.elements();

      while(e.hasMoreElements())
```

```
        {
            System.out.println(e.nextElement());
        }
    }
}
```

HashSet

- HashSet class uses Hashtable to store the elements.
- It extends AbstractSet class and implements Set interface.
- It contains unique elements only.
- It maintains no order.

Difference between List and Set:

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

```
import java.util.*;
class demoHashSet{
 public static void main(String args[]){
    HashSet<String> al=new HashSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");

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

LinkedHashSet

It contains unique elements only like HashSet class.

It extends HashSet class and implements Set interface.

It maintains insertion order.

```
import java.util.*;
class TestCollection10{
 public static void main(String args[]){

  LinkedHashSet<String> al=new LinkedHashSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");

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

TreeSet

It contains unique elements only like HashSet.

The TreeSet class implements NavigableSet interface that extends the SortedSet interface.

It maintains ascending order.

```
import java.util.*;
class TestCollection11{
 public static void main(String args[]){

  TreeSet<String> al=new TreeSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");

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

Output:Ajay

Ravi

Vijay

Map Collection

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.

Why and When Use Maps:

Maps are perfectly for key-value association mapping such as dictionaries. Use Maps when you want to retrieve and update elements by keys, or perform lookups by keys. Some examples:

- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).

Useful methods of Map interface

Method	Description
public Object put(Object key, Object value)	is used to insert an entry in this map.
public void putAll(Map map)	is used to insert the specified map in this map.
public Object remove(Object key)	is used to delete an entry for the specified key.
public Object get(Object key)	is used to return the value for the specified key.
public boolean containsKey(Object key)	is used to search the specified key 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.

Entry Interface

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

Methods of Map.Entry interface

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

HashMap class

A HashMap class 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.

Example

```
import java.util.*;

class DemoHasMap{

 public static void main(String args[]){

  HashMap<Integer,String> hm=new HashMap<Integer,String>();

  hm.put(100,"Amit");

  hm.put(101,"Vijay");

  hm.put(102,"Rahul");

  for(Map.Entry m:hm.entrySet()){

   System.out.println(m.getKey()+" "+m.getValue());

  }

 }  }
```

Java LinkedHashMap class

- A LinkedHashMap contains values based on the key. It implements the Map interface and extends HashMap class.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It is same as HashMap instead maintains insertion order.

```
import java.util.*;

class DemoLinkedHashMap{

 public static void main(String args[]){

  LinkedHashMap<Integer,String> hm=new LinkedHashMap<Integer,String>();

  hm.put(100,"Amit");

  hm.put(101,"Vijay");

  hm.put(102,"Rahul");

for(Map.Entry m:hm.entrySet()){

   System.out.println(m.getKey()+" "+m.getValue());

  }

 }

}
```

Java TreeMap class

- 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.

```
import java.util.*;

class DemoTreeMap {

 public static void main(String args[]){

 TreeMap<Integer,String> hm=new TreeMap<Integer,String>();

  hm.put(100,"Amit");

  hm.put(102,"Ravi");

  hm.put(101,"Vijay");

  hm.put(103,"Rahul");


  for(Map.Entry m:hm.entrySet()){

   System.out.println(m.getKey()+" "+m.getValue());

  }

 }

}
```

What is difference between HashMap and TreeMap?

HashMap	TreeMap
1) HashMap can contain one null key.	TreeMap can not contain any null key.
2) HashMap maintains no order.	TreeMap maintains ascending order.