UNIT-IV-PART-II

UNIT-IV

Java String Handling: String Constructors, Special string operations, Character Extraction, String Comparisons, Modifying a string, String Buffer.

Collections Framework: Overview, Collection Interfaces, Collection Classes, Accessing a collection via Iterator, Working with Maps, Generics

Collections in Java

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.
Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
Java Collection means a single unit of objects.

What is Collection in Java

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

What is a framework in Java

Java collection framework represents a hierarchy of set of interfaces and classes that are used to manipulate group of objects.

Java collections framework is contained in java.util package.
It provides many important classes and interfaces to collect and organize group of objects.

Components of Collection Framework

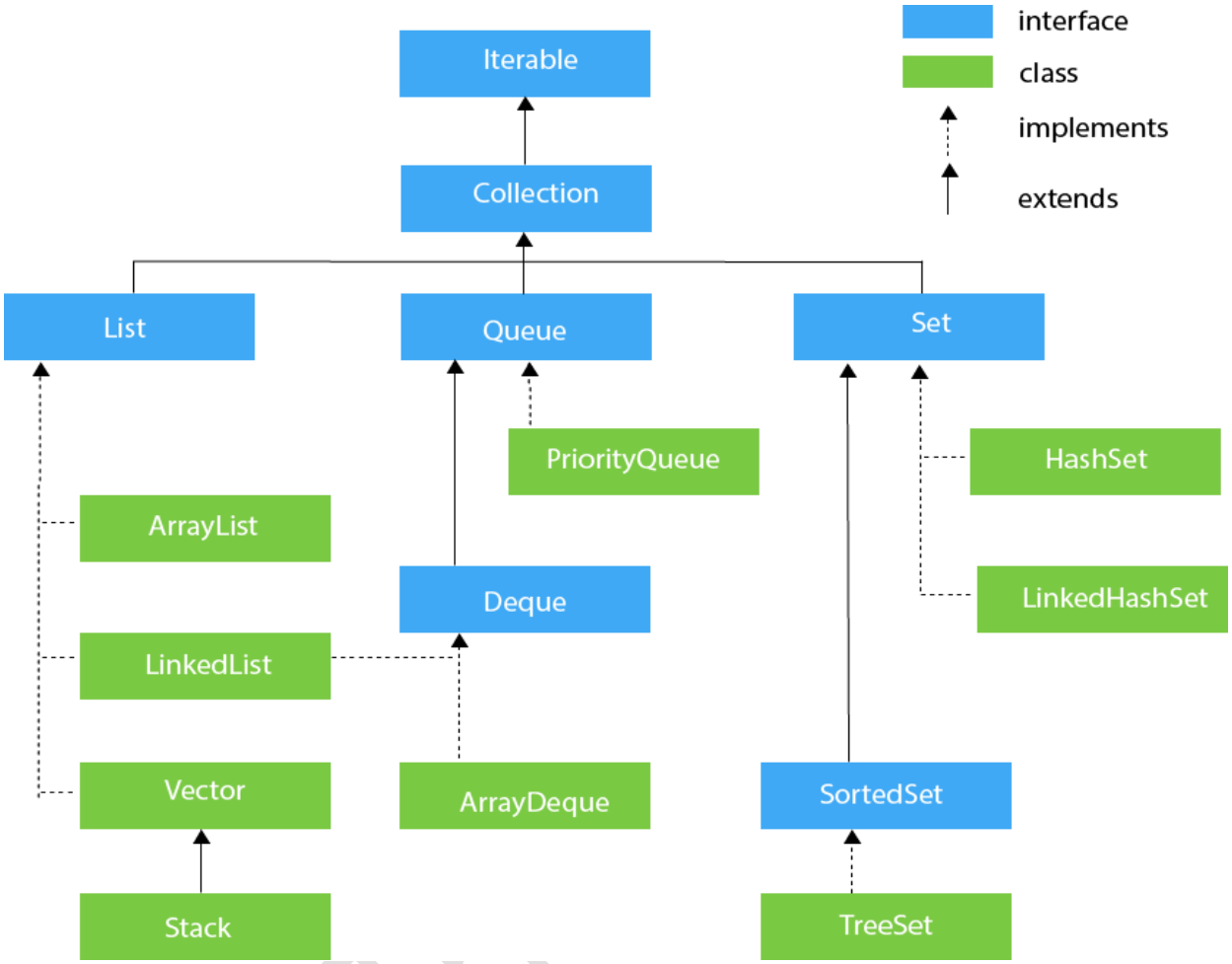
The collections framework consists of:
Collection interfaces such as sets, lists, and maps. These are used to collect different types of objects.
Collection classes such as ArrayList, HashSet etc that are implementations of collection interfaces.

Advantage of Collection Framework

It reduces programming effort by providing built-in set of data structures and algorithms.
Increases performance by providing high-performance implementations of data structures and algorithms.

Hierarchy of Collection Framework

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



Iterator interface

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

Iterator<T> iterator()

The Collection Interface

It is at the top of collection heirarchy and must be implemented by any class that defines a collection.

interface Collection <E>

The List Interface

It extends the Collection Interface, and defines storage as sequence of elements. Following is its general declaration,

interface List <E>

Allows random access and insertion, based on position.

It allows Duplicate elements.

The Set Interface

This interface defines a Set. It extends Collection interface and doesn't allow insertion of duplicate elements. It's general declaration is,

interface Set <E>

The Queue Interface

It extends collection interface and defines behaviour of queue, that is first-in, first-out. It's general declaration is,

interface Queue <E>

The Dequeue Interface

It extends Queue interface and implements behaviour of a double-ended queue. Its general declaration is,

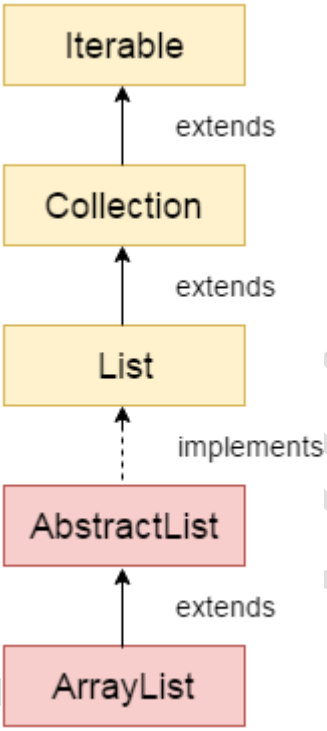
interface Dequeue <E>

Java Collection Interface

Collection is a group of objects, which are known as elements.

It is the root interface in the collection hierarchy.

ArrayList:-



Java **ArrayList** class uses a dynamic array for storing the elements.

It is like an array, but there is no size limit. We can add or remove elements anytime.

So, it is much more flexible than the traditional array.

It is found in the java.util package

The ArrayList in Java can have the duplicate elements also.

It implements the List interface so we can use all the methods of the List interface here.

The ArrayList maintains the insertion order internally.

Java ArrayList class can contain duplicate elements.

Java ArrayList class maintains insertion order.

Syntax:-

```
ArrayList<Integer> al = new ArrayList<Integer>(); // works fine
```

Creating an ArrayList

Lets create an ArrayList to store string elements. Here list is empty because we did not add elements to it.

```
import java.util.*;
class Demo
{
 public static void main(String[] args)
 {
  // Creating an ArrayList
  ArrayList< String> fruits = new ArrayList< String>();
     // Displaying Arraylist
  System.out.println(fruits);
```

```
  }
}
```
o/p:-
[]

Add Elements to ArrayList:-

To add elements into ArrayList, we are using add() method. It adds elements into the list in the insertion order.

```
    import java.util.*;
  class Demo
  {
   public static void main(String[] args)
   {
    // Creating an ArrayList
    ArrayList< String> fruits = new ArrayList< String>();
    // Adding elements to ArrayList
    fruits.add("Mango");
    fruits.add("Apple");
    fruits.add("Berry");
    // Displaying ArrayList
    System.out.println(fruits);
    }
   }
```
o/p:-
[Mango, Apple, Berry]

Removing Elements

To remove elements from the list, we are using remove method that remove the specified elements. We can also pass index value to remove the elements of it.

```
  import java.util.*;
 class Demo
 {
  public static void main(String[] args)
  {
    // Creating an ArrayList
    ArrayList< String> fruits = new ArrayList< String>();
    // Adding elements to ArrayList
    fruits.add("Mango");
    fruits.add("Apple");
    fruits.add("Berry");
    fruits.add("Orange");
    // Displaying ArrayList
    System.out.println(fruits);
    // Removing Elements
    fruits.remove("Apple");
    System.out.println("After Deleting Elements: \n"+fruits);
    // Removing Second Element
    fruits.remove(1);
    System.out.println("After Deleting Elements: \n"+fruits);
   }
}
```
o/p:-
[Mango, Apple, Berry, Orange]
After Deleting Elements: [Mango, Berry, Orange]

After Deleting Elements: [Mango, Orange]

Traversing Elements of ArrayList

Since ArrayList is a collection then we can use loop to iterate its elements. In this example we are traversing elements. See the below example.

```
import java.util.*;
class Demo
{
 public static void main(String[] args)
 {
  // Creating an ArrayList
  ArrayList< String> fruits = new ArrayList< String>();
  // Adding elements to ArrayList
  fruits.add("Mango");
  fruits.add("Apple");
  fruits.add("Berry");
  fruits.add("Orange");
  // Traversing ArrayList
  for(String element : fruits)
  {
   System.out.println(element);
  }
 }
}
```

o/p:-

Mango Apple Berry Orange

Get size of ArrayList

Sometimes we want to know number of elements an ArrayList holds. In that case we use size() then returns size of ArrayList which is equal to number of elements present in the list.

```
import java.util.*;
class Demo
{
 public static void main(String[] args)
 {
  // Creating an ArrayList
  ArrayList< String> fruits = new ArrayList< String>();
  // Adding elements to ArrayList
  fruits.add("Mango");
  fruits.add("Apple");
  fruits.add("Berry");
  fruits.add("Orange");
  // Traversing ArrayList
  for(String element : fruits)
  {
   System.out.println(element);
  }
  System.out.println("Total Elements: "+fruits.size());
 }
}
```

o/p:-

Mango Apple Berry Orange Total Elements: 4

Sorting ArrayList Elements

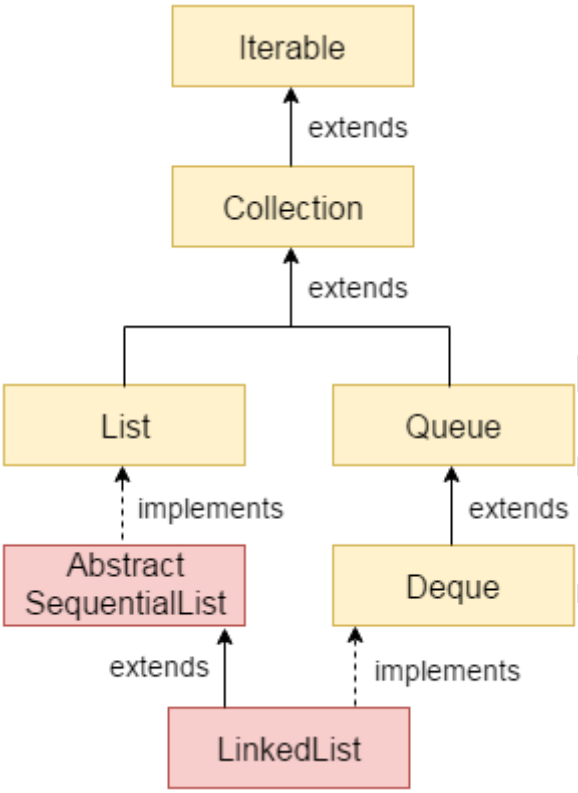
To sort elements of an ArrayList, Java provides a class Collections that includes a static method sort(). In this example, we are using sort method to sort the elements.

```
import java.util.*;
class Demo
{
 public static void main(String[] args)
 {
  // Creating an ArrayList
  ArrayList< String> fruits = new ArrayList< String>();
  // Adding elements to ArrayList
  fruits.add("Mango");
  fruits.add("Apple");
  fruits.add("Berry");
  fruits.add("Orange");
  // Sorting elements
  Collections.sort(fruits);
  // Traversing ArrayList
  for(String element : fruits)
  {
   System.out.println(element);
  }
 }
}
```

o/p:-

```
Apple Berry Mango Orange
Grapes
```

LinkedList class



Java LinkedList class uses a 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.
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 occur.
Java LinkedList class can be used as a list, stack or queue.

Program:-
Lets take an example to create a linked-list, no element is inserted so it creates an empty LinkedList. See the below example.

```
import java.util.*;
class Demo
{
public static void main(String[] args)
 {
// Creating LinkedList
  LinkedList< String> linkedList = new LinkedList< String>();
  System.out.println(linkedList);
 }
}
```

o/p:-
[]

Add Elements to LinkedList

To add elements into LinkedList, we are using add() method. It adds elements into the list in the insertion order.

```
 import java.util.*;
 class Demo
 {
 public static void main(String[] args)
 {
   // Creating LinkedList
   LinkedList< String> linkedList = new LinkedList< String>();
   linkedList.add("Delhi");
   linkedList.add("NewYork");
   linkedList.add("Moscow");
   linkedList.add("Dubai");
   // Displaying LinkedList
   System.out.println(linkedList);
 }
}
```

o/p:-

```
 [Delhi, NewYork, Moscow, Dubai]
```

Removing Elements

To remove elements from the list, we are using remove method that remove the specified elements. We can also pass index value to remove the elements of it.

```
 import java.util.*;
 class Demo
{
 public static void main(String[] args)
 {
   // Creating LinkedList
   LinkedList< String> linkedList = new LinkedList< String>();
   linkedList.add("Delhi");
   linkedList.add("NewYork");
   linkedList.add("Moscow");
   linkedList.add("Dubai");
   // Displaying LinkedList
   System.out.println(linkedList);
   // Removing Elements
   linkedList.remove("Moscow");
   System.out.println("After Deleting Elements: \n"+linkedList);
   // Removing Second Element
   linkedList.remove(1);
   System.out.println("After Deleting Elements: \n"+linkedList);
 }
}
```

o/p:-

```
[Delhi, NewYork, Moscow, Dubai]
After Deleting Elements: [Delhi, NewYork, Dubai]
After Deleting Elements: [Delhi, Dubai]
```

Traversing Elements of LinkedList

Since LinkedList is a collection then we can use loop to iterate its elements. In this example we are traversing elements. See the below example.

```
 import java.util.*;
class Demo
{
 public static void main(String[] args)
 {
  // Creating LinkedList
  LinkedList< String> linkedList = new LinkedList< String>();
  linkedList.add("Delhi");
  linkedList.add("NewYork");
  linkedList.add("Moscow");
  linkedList.add("Dubai");


  // Traversing ArrayList
    for(String element : linkedList)
    {
    System.out.println(element);
  }
 }
}
```

o/p:-

Delhi NewYork Moscow Dubai

Get size of LinkedList

Sometimes we want to know number of elements an LinkedList holds. In that case we use size() then returns size of LinkedList which is equal to number of elements present in the list.

```
import java.util.*;
class Demo
{
 public static void main(String[] args)
 {
  // Creating LinkedList
  LinkedList< String> linkedList = new LinkedList< String>();
  linkedList.add("Delhi");
  linkedList.add("NewYork");
  linkedList.add("Moscow");
  linkedList.add("Dubai");
  // Traversing ArrayList
  for(String element : linkedList)
  {
    System.out.println(element);
  }
System.out.println("Total Elements: "+linkedList.size());
 }
}
```

o/p:-

Delhi NewYork Moscow Dubai Total Elements: 4

Sorting LinkedList Elements

To sort elements of an LinkedList, Java provides a class Collections that includes a static method sort(). In this example, we are using sort method to sort the elements.

```
import java.util.*;
class Demo
{
  public static void main(String[] args)
  {
    // Creating LinkedList
    LinkedList< String> linkedList = new LinkedList< String>();
    linkedList.add("Delhi");
    linkedList.add("NewYork");
    linkedList.add("Moscow");
    linkedList.add("Dubai");
    // Sorting elements
    Collections.sort(linkedList);
    // Traversing ArrayList
    for(String element : linkedList) {
      System.out.println(element);
    }
  }
}
```

o/p:-

Delhi Dubai Moscow NewYork

List

List in Java provides the facility to maintain the ordered collection.
It contains the index-based methods to insert, update, delete and search the elements.
It can have the duplicate elements also.
We can also store the null elements in the list.
The List interface is found in the java.util package and inherits the Collection interface.

```
import java.util.*;
public class ListExample1
{
public static void main(String args[])
{
 //Creating a List
 List<String> list=new ArrayList<String>();
 //Adding elements in the List
 list.add("Mango");
 list.add("Apple");
 list.add("Banana");
 list.add("Grapes");
 //Iterating the List element using for-each loop
 for(String fruit:list)
  System.out.println(fruit);

}
}
```

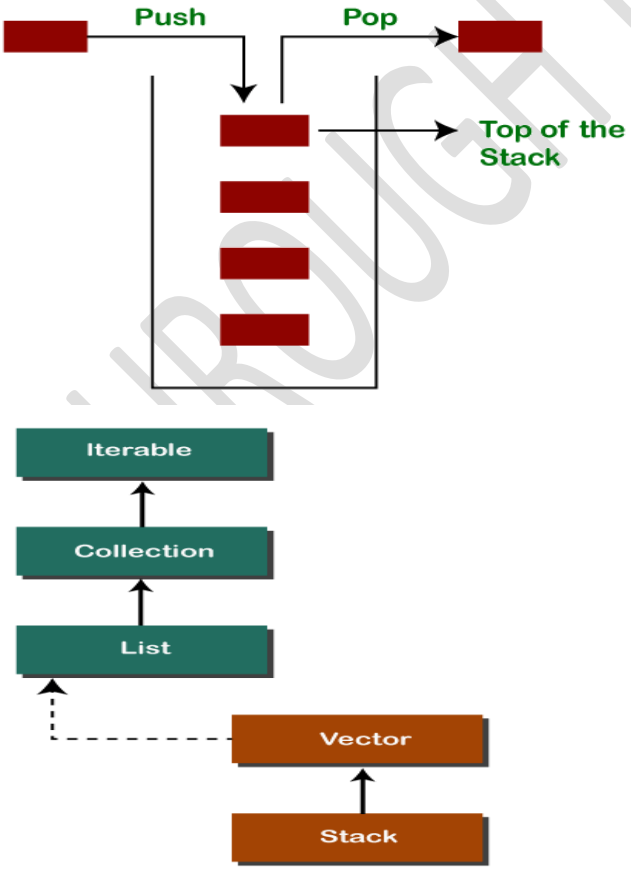
Output:
Mango
Apple
Banana
Grapes

Stack

The **stack** is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out** (LIFO).

Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the **Stack class** that provides different operations such as push, pop, search, etc. In this section, we will discuss the **Java Stack class**, its **methods,** and **implement** the stack data structure in a Java program.

The stack data structure has the two most important operations that are push and pop. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack. Let's see how they work on stack.



Creating a Stack

If we want to create a stack, first, import the java.util package and create an object of the Stack class.

```
import java.util.Stack;
public class StackSearchMethodExample
{
public static void main(String[] args)
{
Stack<String> stk= new Stack<>();
//pushing elements into Stack
stk.push("Mac Book");
stk.push("HP");
stk.push("DELL");
stk.push("Asus");
System.out.println("Stack: " + stk);
// Search an element
int location = stk.search("HP");
System.out.println("Location of Dell: " + location);
}
}
```

Vector

Vector is like the *dynamic array* which can grow or shrink its size.

Unlike array, we can store n-number of elements in it as there is no size limit.

It is recommended to use the Vector class in the thread-safe implementation only.

If you don't need to use the thread-safe implementation, you should use the ArrayList,

It is similar to the ArrayList, but with two differences-

Vector is synchronized.

Java Vector contains many legacy methods that are not the part of a collections framework.

Vector Example

```
import java.util.*;
public class VectorExample
 {
     public static void main(String args[])
 {
        //Create a vector
        Vector<String> vec = new Vector<String>();
        //Adding elements using add() method of List
        vec.add("Tiger");
        vec.add("Lion");
        vec.add("Dog");
        vec.add("Elephant");
        //Adding elements using addElement() method of Vector
        vec.addElement("Rat");
        vec.addElement("Cat");
        vec.addElement("Deer");

        System.out.println("Elements are: "+vec);
     }
}
```

Output:
Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]

Queue Interface

The interface Queue is available in the java.util package and does extend the Collection interface.
It is used to keep the elements that are processed in the First In First Out (FIFO) manner.
It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list.
FIFO concept is used for insertion and deletion of elements from a queue.
The Java Queue provides support for all of the methods of the Collection interface including deletion, insertion, etc.

PriorityQueue Class

PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority.
PriorityQueue Class Declaration
Let's see the declaration for java.util.PriorityQueue class.
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable

```
import java.util.*;
class Pqueue
{
public static void main(String args[])
{
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit");
queue.add("Vijay");
queue.add("Karan");
queue.add("Jai");
queue.add("Rahul");
System.out.println("head:"+queue);
}
}
```

Deque Interface

The interface called Deque is present in java.util package.
It is the subtype of the interface queue.
The Deque supports the addition as well as the removal of elements from both ends of the data structure.
Deque supports both, either of the mentioned operations can be performed on it.
Deque is an acronym for "double ended queue".

ArrayDeque class

We know that it is not possible to create an object of an interface in Java.
Therefore, for instantiation, we need a class that implements the Deque interface, and that class is ArrayDeque.
It grows and shrinks as per usage. It also inherits the AbstractCollection class.

- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.

- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

Example:-

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

o/p:-

```
Ravi
Vijay
Ajay
```

HashSet

Java HashSet class is used to store unique elements.
It uses hash table internally to store the elements.
It implements Set interface and extends the AbstractSet class.
Syntax:-

```
public class HashSet<E>extends AbstractSet<E>implements Set<E>, Cloneable, Serializable
```

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

  HashSet<String> hs = new HashSet<String>();
  // Displaying HashSet
  System.out.println(hs);
 }
}
```

o/p:-

```
[]
```

Add Elements to HashSet

In this example, we are creating a HashSet that store string values.
Since HashSet does not store duplicate elements, we tried to add a duplicate elements but the output contains only unique elements.

Example of HashSet

```
import java.util.*;
class Demo
{
  public static void main(String args[])
  {
    // Creating HashSet
    HashSet<String> hs = new HashSet<String>();
    // Adding elements
    hs.add("Mohan");
    hs.add("Rohan");
    hs.add("Sohan");
    hs.add("Mohan");
    // Displaying HashSet
    System.out.println(hs);
  }
}
```

O/p:-

[Mohan, Sohan, Rohan]

Remove Elements from HashSet

To remove elements from the hashset, we are using remove() method that remove the specified elements.

```
  import java.util.*;
class Demo
{
  public static void main(String args[])
  {
    // Creating HashSet
    HashSet<String> hs = new HashSet<String>();
    // Adding elements
    hs.add("Mohan");
    hs.add("Rohan");
    hs.add("Sohan");
    hs.add("Mohan");
    // Displaying HashSet
    System.out.println(hs);
    // Removing elements
    hs.remove("Mohan");
    System.out.println("After removing elements: \n"+hs);
  }
```

```
}
```

o/p:-

```
[Mohan, Sohan, Rohan]
After removing elements:
[Sohan, Rohan]
```

Traversing Elements of HashSet

Since HashSet is a collection then we can use loop to iterate its elements. In this example we are traversing elements using for loop. See the below example.

```
import java.util.*;
class Demo
{
  public static void main(String args[])
  {
    // Creating HashSet
    HashSet<String> hs = new HashSet<String>();
    // Adding elements
    hs.add("Mohan");
    hs.add("Rohan");
    hs.add("Sohan");
    hs.add("Mohan");
    // Traversing ArrayList
    for(String element : hs)
    {
      System.out.println(element);
    }
  }
}
```

O/p:-

```
Mohan
Sohan
Rohan
```

Get size of HashSet

Sometimes we want to know number of elements an HashSet holds.
In that case we use size() then returns size of HashSet which is equal to number of elements present in the list.

```
 import java.util.*;
 class Demo
 {
  public static void main(String args[])
  {
    // Creating HashSet
    HashSet<String> hs = new HashSet<String>();
    // Adding elements
    hs.add("Mohan");
    hs.add("Rohan");
```

```
    hs.add("Sohan");
    hs.add("Mohan");
    // Traversing ArrayList
    for(String element : hs)
   {
     System.out.println(element);
   }
   System.out.println("Total elements : "+hs.size());
  }
}
```

o/p:-

```
Mohan
Sohan
Rohan
Total elements : 3
```

Treeset

TreeSet class used to store unique elements in ascending order.
It is similar to HashSet except that it sorts the elements in the ascending order while HashSet doesn't maintain any order.
Java TreeSet class implements the Set interface and use tree based data structure storage.
Important Points

- It stores the elements in ascending order.
- It uses a Tree structure to store elements.
- It contains unique elements only like HashSet.
- It's access and retrieval times are quite fast.
- It doesn't allow null element.
- It is non synchronized.

Creating an TreeSet

Lets create a TreeSet to store string elements. Here set is empty because we did not add elements to it.

```
import java.util.*;
class Demo
{
  public static void main(String[] args)
  {
    TreeSet< String> fruits = new TreeSet< String>();
     System.out.println(fruits);
  }
}
```

O/p:-

```
[]
```

Add Elements To TreeSet

Lets take an example to create a TreeSet that contains duplicate elements.
But you can notice that it prints unique elements that means it does not allow duplicate elements.

```
import java.util.*;
class Demo
{
  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 itr=al.iterator();
    while(itr.hasNext())
   {
      System.out.println(itr.next());
    }
  }
}
```

O/p:-

```
Ajay Ravi Vijay
```

Removing Elements From the TreeSet

We can use remove() method of this class to remove the elements

```
import java.util.*;
class Demo
{
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 itr=al.iterator();
    while(itr.hasNext()){
      System.out.println(itr.next());
    }
    al.remove("Ravi");
    System.out.println("After Removing: "+al);
  }
}
```

o/p:-

```
Ajay
Ravi
Vijay
After Removing: [Ajay, Vijay]
```

Search an Element in TreeSet

TreeSet provides contains() method that true if elements is present in the set.

```
import java.util.*;
class Demo
{
  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 itr=al.iterator();
    while(itr.hasNext())
    {
      System.out.println(itr.next());
    }
     boolean iscontain = al.contains("Ravi");
    System.out.println("Is contain Ravi: "+iscontain);
  }
}
```

O/P:-

```
Ajay
Ravi
Vijay
Is contain Ravi: true
```

```
//Ascending &descending order
  import java.util.*;
  class Demo
 {
  public static void main(String args[])
  {
    TreeSet<String> al = new TreeSet<String>();
    al.add("Ravi");
    al.add("Vijay");
    al.add("Ravi");
    al.add("Ajay");
    System.out.println("Ascending...");
    Iterator itr=al.iterator();
    while(itr.hasNext())
   {
      System.out.println(itr.next());
    }
    System.out.println("Descending...");
    Iterator itr2=al.descendingIterator();
    while(itr2.hasNext()){
      System.out.println(itr2.next());
    }
```

```
  }
}
```
O/P:-

```
Ascending...
Ajay
Ravi
Vijay
Descending...
Vijay
Ravi
Ajay
```

Hashmap

Java HashMap class is an implementation of Map interface based on hash table.
It stores elements in key & value pairs which is denoted as HashMap<Key, Value> or HashMap<K, V>.
It extends AbstractMap class, and implements Map interface and can be accessed by importing java.util package.
Important Points:

- It is member of the Java Collection Framework.
- It uses a hashtable to store the map. This allows the execution time of get() and put() to remain same.
- HashMap does not maintain order of its element.
- It contains values based on the key.
- It allows only unique keys.
- It is unsynchronized.
- Its initial default capacity is 16.
- It permits null values and the null key
- It is unsynchronized.

Example: **Creating a HashMap**

Lets take an example to create a hashmap that can store integer type key and string values. Initially it is empty because we did not add elements to it. Its elements are enclosed into curly braces.

```
import java.util.*;
class Demo
{
  public static void main(String args[])
  {
    HashMap<Integer,String> hashMap = new HashMap<Integer,String>();
    System.out.println(hashMap);
  }
}
```
o/p:-{}

Adding Elements To HashMap

After creating a hashmap, now lets add elements to it.
HashMap provides put() method that takes two arguments first is key and second is value. See the below example.

```
import java.util.*;
class Demo
{
 public static void main(String args[])
 {
 // Creating HashMap
        HashMap<Integer,String> hashMap = new      HashMap<Integer,String>();
   // Adding elements
   hashMap.put(1, "One");
   hashMap.put(2, "Two");
   hashMap.put(3, "Three");
   hashMap.put(4, "Four");
   // Displaying HashMap
   System.out.println(hashMap);
 }
}
```

o/p:-

```
{1=One, 2=Two, 3=Three, 4=Four}
```

Removing Elements From HashMap

In case, we need to remove any element from the hashmap. We can use remove() method that takes key as an argument. it has one overloaded remove() method that takes two arguments first is key and second is value.

```
 import java.util.*;
class Demo
{
 public static void main(String args[])
 {
 // Creating HashMap
   HashMap<Integer,String> hashMap = new HashMap<Integer,String>();
   // Adding elements
   hashMap.put(1, "One");
   hashMap.put(2, "Two");
   hashMap.put(3, "Three");
   hashMap.put(4, "Four");
   // Displaying HashMap
   System.out.println(hashMap);
   // Remove element by key
   hashMap.remove(2);
   System.out.println("After Removing 2 :\n"+hashMap);
   // Remove by key and value
   hashMap.remove(3, "Three");
```

```
   System.out.println("After Removing 3 :\n"+hashMap);

  }
}
```
o/p:-

```
{1=One, 2=Two, 3=Three, 4=Four}
After Removing 2 :
{1=One, 3=Three, 4=Four}
After Removing 3 :
{1=One, 4=Four}
```

Traversing Elements

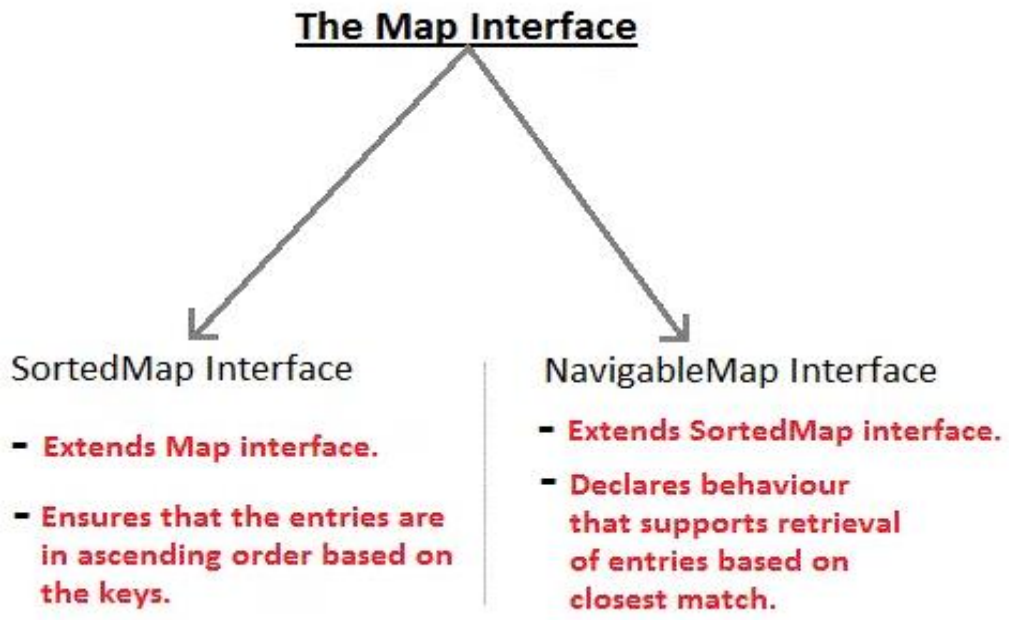
To access elements of the hashmap, we can traverse them using the loop. In this example, we are using for loop to iterate the elements.

```
 import java.util.*;
class Demo
{
 public static void main(String args[])
 {
 // Creating HashMap
   HashMap<Integer,String> hashMap = new HashMap<Integer,String>();
   // Adding elements
   hashMap.put(1, "One");
   hashMap.put(2, "Two");
   hashMap.put(3, "Three");
   hashMap.put(4, "Four");
   // Traversing HashMap
   for(Map.Entry<Integer, String> entry : hashMap.entrySet()) {
     System.out.println(entry.getKey()+" : "+entry.getValue());
   }
 }
}
```
o/p:-

```
1 : One
2 : Two
3 : Three
4 : Four
```

Map Interface - Java Collections

A Map stores data in key and value association. Both key and values are objects.
The key must be unique but the values can be duplicate.
It provides various classes: HashMap, TreeMap, LinkedHashMap for map implementation.
All these classes implements Map interface to provide Map properties to the collection.



HashMap Example

Lets take an example to create hashmap and store values in key and value pair.
Notice to insert elements, we used put() method because map uses put to insert element, not add() method that we used in list interface.

```
import java.util.*;

class Demo
{
  public static void main(String args[])
  {
    HashMap< String,Integer> hm = new HashMap< String,Integer>();
    hm.put("a",100);
    hm.put("b",200);
    hm.put("c",300);
    hm.put("d",400);

    Set<Map.Entry<String,Integer>> st = hm.entrySet();  //returns Set view
    for(Map.Entry<String,Integer> me:st)
    {
      System.out.print(me.getKey()+":");
      System.out.println(me.getValue());
    }
  }
}
```

a:100 b:200 c:300 d:400

TreeMap class

TreeMap class extends AbstractMap and implements NavigableMap interface.
It creates Map, stored in a tree structure.
A TreeMap provides an efficient means of storing key/value pair in efficient order.
It provides key/value pairs in sorted order and allows rapid retrieval.
Example:
In this example, we are creating treemap to store data. It uses tree to store data and data is always in sorted order. See the below example.

```
import java.util.*;
class Demo
{
 public static void main(String args[])
 {
   TreeMap<String,Integer> tm = new TreeMap<String,Integer>();
   tm.put("a",100);
   tm.put("b",200);
   tm.put("c",300);
   tm.put("d",400);

   Set<Map.Entry<String,Integer>> st = tm.entrySet();
   for(Map.Entry<String,Integer> me:st)
   {
     System.out.print(me.getKey()+":");
     System.out.println(me.getValue());
   }
 }
}
```

a:100 b:200 c:300 d:400

LinkedHashMap class

LinkedHashMap extends HashMap class.
It maintains a linked list of entries in map in order in which they are inserted.
Example:
Here we are using linkedhashmap to store data. It stores data into insertion order and use linked-list internally. See the below example.

```
 import java.util.*;
class Demo
{
 public static void main(String args[])
 {
   LinkedHashMap<String,Integer> tm = new LinkedHashMap<String,Integer>();
   tm.put("a",100);
   tm.put("b",200);
   tm.put("c",300);
   tm.put("d",400);
   Set<Map.Entry<String,Integer>> st = tm.entrySet();
   for(Map.Entry<String,Integer> me:st)
   {
     System.out.print(me.getKey()+":");
```

```
      System.out.println(me.getValue());
    }
  }
}
```

o/p:-

a:100 b:200 c:300 d:400

Adding Elements To HashMap

After creating a hashmap, now lets add elements to it. HashMap provides put() method that takes two arguments first is key and second is value. See the below example.

```
import java.util.*;
class Demo
{
  public static void main(String args[])
  {
  // Creating HashMap
    HashMap<Integer,String> hashMap = new HashMap<Integer,String>();
    // Adding elements
    hashMap.put(1, "One");
    hashMap.put(2, "Two");
    hashMap.put(3, "Three");
    hashMap.put(4, "Four");
    // Displaying HashMap
    System.out.println(hashMap);
  }
}
```

{1=One, 2=Two, 3=Three, 4=Four}

Removing Elements From HashMap

In case, we need to remove any element from the hashmap. We can use remove() method that takes key as an argument. it has one overloaded remove() method that takes two arguments first is key and second is value.

```
  import java.util.*;
  class Demo
 {
  public static void main(String args[])
  {
  // Creating HashMap
    HashMap<Integer,String> hashMap = new HashMap<Integer,String>();
    // Adding elements
    hashMap.put(1, "One");
    hashMap.put(2, "Two");
    hashMap.put(3, "Three");
    hashMap.put(4, "Four");
    // Displaying HashMap
    System.out.println(hashMap);
    // Remove element by key
    hashMap.remove(2);
    System.out.println("After Removing 2 :\n"+hashMap);
    // Remove by key and value
```

```
    hashMap.remove(3, "Three");
    System.out.println("After Removing 3 :\n"+hashMap);

  }
}
```

o/p:-

{1=One, 2=Two, 3=Three, 4=Four} After Removing 2 : {1=One, 3=Three, 4=Four} After Removing 3 : {1=One, 4=Four}

Traversing Elements

To access elements of the hashmap, we can traverse them using the loop. In this example, we are using for loop to iterate the elements.

```
   import java.util.*;
  class Demo
  {
    public static void main(String args[])
    {
  // Creating HashMap
    HashMap<Integer,String> hashMap = new HashMap<Integer,String>();
    // Adding elements
    hashMap.put(1, "One");
    hashMap.put(2, "Two");
    hashMap.put(3, "Three");
    hashMap.put(4, "Four");
    // Traversing HashMap
    for(Map.Entry<Integer, String> entry : hashMap.entrySet())
    {
      System.out.println(entry.getKey()+" : "+entry.getValue());
    }
  }
}
```

o/p:-

1 : One 2 : Two 3 : Three 4 : Four

Iterators

To access elements of a collection, either we can use index if collection is list based or we need to traverse the element. There are three possible ways to traverse through the elements of any collection.

Using Iterator interface

Using ListIterator interface

Using for-each loop

Accessing elements using Iterator

Iterator is an interface that is used to iterate the collection elements. It is part of java collection framework. It provides some methods that are used to check and access elements of a collection. Iterator Interface is used to traverse a list in forward direction,

Iterator Example

In this example, we are using iterator() method of collection interface that returns an instance of Iterator interface. After that we are using hasNext() method that returns true of collection contains an elements and within the loop, obtain each element by calling next() method.

```
import java.util.*;
class Demo
{
```

```
  public static void main(String[] args)
  {
    ArrayList< String> ar = new ArrayList< String>();
    ar.add("ab");
    ar.add("bc");
    ar.add("cd");
    ar.add("de");
    Iterator it = ar.iterator();    //Declaring Iterator
    while(it.hasNext())
    {
      System.out.print(it.next()+" ");
    }
  }
}
```
ab bc cd de

ListIterator Example

Lets create an example to traverse the elements of ArrayList. ListIterator works only with list collection.

```
import java.util.*;
class Demo
{
  public static void main(String[] args)
  {
    ArrayList< String> ar = new ArrayList< String>();
    ar.add("ab");
    ar.add("bc");
    ar.add("cd");
    ar.add("de");
    ListIterator litr = ar.listIterator();
    while(litr.hasNext())   //In forward direction
    {
      System.out.print(litr.next()+" ");
    }
    while(litr.hasPrevious())   //In backward direction
    {
      System.out.print(litr.previous()+" ");
    }
  }
}
```
o/p:-

ab bc cd de de cd bc ab

<u>for-each loop</u>

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

Exmaple:

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

```
  {
    LinkedList< String> ls = new LinkedList< String>();
    ls.add("a");
    ls.add("b");
    ls.add("c");
    ls.add("d");
    for(String str : ls)
    {
      System.out.print(str+" ");
    }
  }
}
```

a b c d

Traversing using for loop

we can use for loop to traverse the collection elements but only index-based collection can be accessed. For example, list is index-based collection that allows to access its elements using the index value.

```
  import java.util.*;
class Demo
{
  public static void main(String[] args)
  {
    LinkedList<String> ls = new LinkedList<String>();
    ls.add("a");
    ls.add("b");
    ls.add("c");
    ls.add("d");
    for(int i = 0; i<ls.size(); i++)
    {
      System.out.print(ls.get(i)+" ");
    }
  }
}
```

o/p:-

a b c d

Comparable Interface

Java Comparable interface is a member of collection framework which is used to compare objects and sort them according to the natural order.

The natural ordering refers to the behavior of compareTo() method which is defined into Comparable interface.

Sorting list

Lets take an example to sort an ArrayList that stores integer values. We are using sort() method of Collections class that sort those object which implements Compares interface. Since integer wrapper class implements Comparable so we are able to get sorted objects. See the below example.

```
import java.util.*;
public class Demo
{
   public static void main(String a[])
  {
     // Creating List
       ArrayList<Integer> list = new ArrayList<>();
       // Adding elements
       list.add(100);
       list.add(2);
       list.add(66);
       list.add(22);
       list.add(10);
       // Displaying list
       System.out.println(list);
       // Sorting list
       Collections.sort(list);
       // Displaying sorted list
       System.out.println("Sorted List : "+list);
   }
}
```

o/p:-

[100, 2, 66, 22, 10]
Sorted List : [2, 10, 22, 66, 100]

Comparator Interface

In Java, Comparator interface is used to order(sort) the objects in the collection in your own way. It gives you the ability to decide how elements will be sorted and stored within collection and map.

Comparator Interface defines compare() method. This method has two parameters. This method compares the two objects passed in the parameter. It returns 0 if two objects are equal. It returns a positive value if object1 is greater than object2. Otherwise a negative value is returned.

MyComparator class:

This class defines the comparison logic for Student class based on their roll. Student object will be sorted in ascending order of their roll.

```
class MyComparator implements Comparator<Student>
{
  public int compare(Student s1,Student s2)
    {
       if(s1.roll == s2.roll) return 0;
       else if(s1.roll > s2.roll) return 1;
       else return -1;
    }
}
```

```
public class Test
{

  public static void main(String[] args)
  {
     TreeSet< Student> ts = new TreeSet< Student>(new MyComparator());
     ts.add(new Student(45, "Rahul"));
     ts.add(new Student(11, "Adam"));
     ts.add(new Student(19, "Alex"));
     System.out.println(ts);
  }

}
```

o/p:-

[11 Adam, 19 Alex, 45 Rahul]

Generics in Java

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

Type-safety: We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

```
import java.util.*;
class TestGenerics1
{
public static void main(String args[])
{
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error
String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);
Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
element is: jai
rahul
jai
```

Syntax for creating an Object of a Generic type

Class_name <data type> reference_name = new Class_name<data type> ();
OR
Class_name <data type> reference_name = new Class_name<>();

Example of Generic class

//<> brackets indicates that the class is of generic type

```
class Gen <T>
{
 T ob;    //an object of type T is declared<
 Gen(T o)  //constructor
 {
  ob = o;
 }
 public T getOb()
 {
  return ob;
 }
}
```

```
class Demo
{
  public static void main (String[] args)
  {
    Gen < Integer> iob = new Gen<>(100);   //instance of Integer type Gen Class
    int  x = iob.getOb();
    System.out.println(x);
    Gen < String> sob = new Gen<>("Hello");  //instance of String type Gen Class
    String str = sob.getOb();
    System.out.println(str);
  }
}
```

o/p:-

```
100
 Hello
```