Collection

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

What is Collection in Java

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

What is a framework in Java

- ❍It provides readymade architecture.
- ❍It represents a set of classes and interfaces.
- ❍It is optional.

What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

Do you know

0 What are the two ways to iterate the elements of a collection?

❍What is the difference between ArrayList and LinkedList classes in collection framework?

❍What is the difference between ArrayList and Vector classes in collection framework?

❍What is the difference between HashSet and HashMap classes in collection framework?

❍What is the difference between HashMap and Hashtable class?

❍What is the difference between Iterator and Enumeration interface in collection framework?

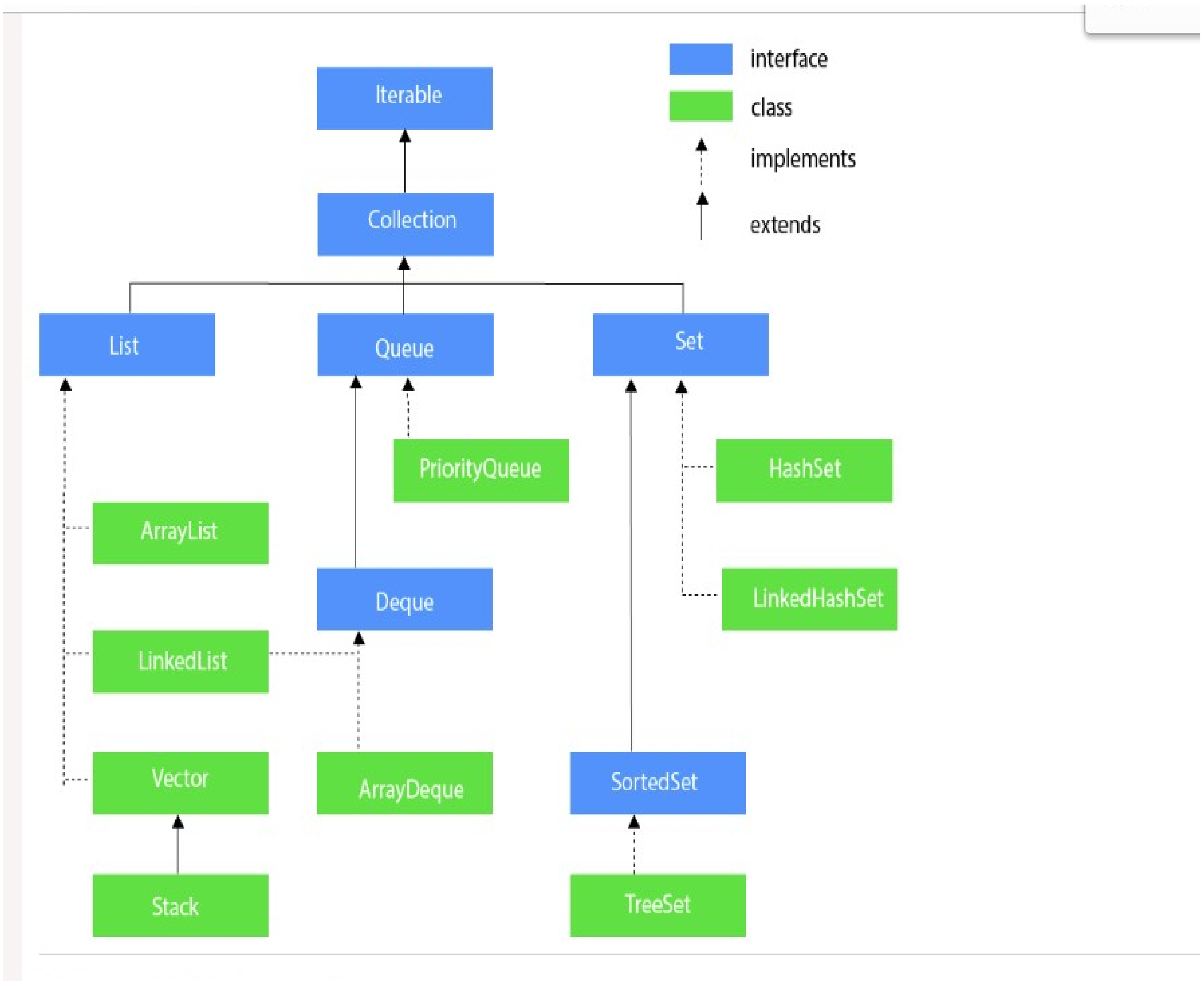
❍How can we sort the elements of an object? What is the difference between Comparable and Comparator interfaces?

❍What does the hashcode() method?

❍What is the difference between Java collection and Java collections?

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



Methods of Collection interface

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

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
1	public Object[] toArray()	It converts collection into array.

2		
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

Iterator interface

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

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

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterable Interface

The Iterable interface is the root interface for all the collection classes.

The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

1. Iterator<T> iterator()
   It returns the iterator over the elements of type T.

Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which **we can store the ordered collection of objects. It can have duplicate values.It contains the index-based methods to insert, update, delete and search the elements. I**

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

```
//Creating a List of type String using ArrayList

1.List<String> list=new ArrayList<String>();
2.
3.//Creating a List of type Integer using ArrayList
4.List<Integer> list=new ArrayList<Integer>();
5.
```

```java
//Creating a List of type Book using ArrayList
List<Book> list=new ArrayList<Book>();

//Creating a List of type String using LinkedList
List<String> list=new LinkedList<String>();
```

```java
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

How to convert Array to List

We can convert the Array to List by traversing the array and adding the element in list one by one using list.add() method. Let's see a simple example to convert array elements into List.

```java
import java.util.*;
public class ArrayToListExample{
public static void main(String args[]){
//Creating Array
String[] array={"Java","Python","PHP","C++"};
System.out.println("Printing Array: "+Arrays.toString(array));
//Converting Array to List
List<String> list=new ArrayList<String>();
for(String lang:array){
list.add(lang);
}
System.out.println("Printing List: "+list);

}
}
```

Output:

```
Printing Array: [Java, Python, PHP, C++]
Printing List: [Java, Python, PHP, C++]
```

How to convert List to Array

We can convert the List to Array by calling the list.toArray() method. Let's see a simple example to convert list elements into array.

```
import java.util.*;
public class ListToArrayExample{
public static void main(String args[]){
 List<String> fruitList = new ArrayList<>();
 fruitList.add("Mango");
 fruitList.add("Banana");
 fruitList.add("Apple");
 fruitList.add("Strawberry");
 //Converting ArrayList to Array
String[] array = fruitList.toArray(new String[fruitList.size()]);
System.out.println("Printing Array: "+Arrays.toString(array));
System.out.println("Printing List: "+fruitList);
}
}
```

Test it Now

Output:

```
Printing Array: [Mango, Banana, Apple, Strawberry]
Printing List: [Mango, Banana, Apple, Strawberry]
```

How to Sort List

There are various ways to sort the List, here we are going to use Collections.sort() method to sort the list element. The *java.util* package provides a utility class **Collections** which has the static method sort(). Using the **Collections.sort()** method, we can easily sort any List.

```
import java.util.*;
class SortArrayList{
 public static void main(String args[]){
  //Creating a list of fruits
  List<String> list1=new ArrayList<String>();
  list1.add("Mango");
  list1.add("Apple");
  list1.add("Banana");
  list1.add("Grapes");
  //Sorting the list
  Collections.sort(list1);
   //Traversing list through the for-each loop
for(String fruit:list1)
  System.out.println(fruit);

System.out.println("Sorting numbers...");
//Creating a list of numbers
List<Integer> list2=new ArrayList<Integer>();
list2.add(21);
list2.add(11);
list2.add(51);
list2.add(1);
//Sorting the list
Collections.sort(list2);
 //Traversing list through the for-each loop
for(Integer number:list2)
  System.out.println(number);
}

}
```
Output:

```
Apple
Banana
Grapes
Mango
Sorting numbers...
1
11
21
51
```

****

ArrayList

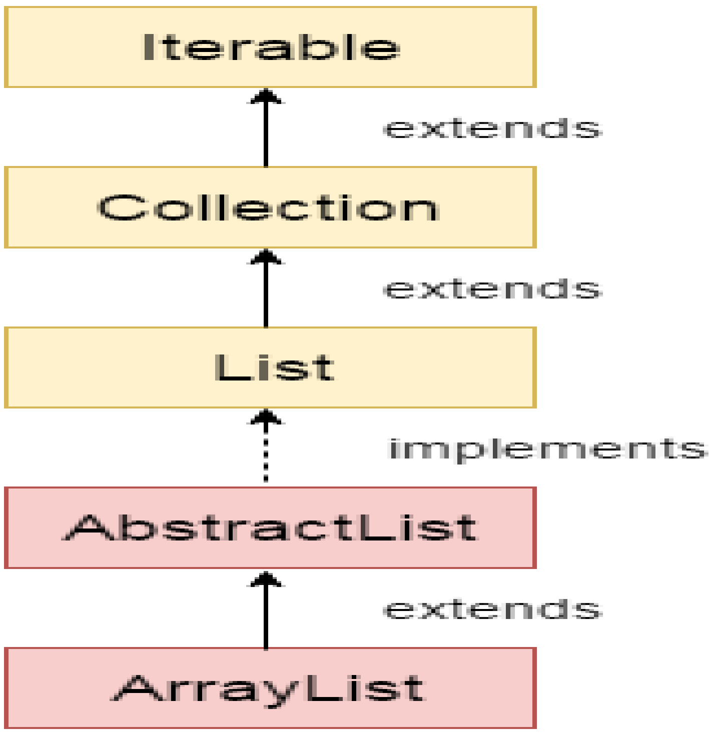
The ArrayList class implements the List interface. **It uses a dynamic array to store the duplicate element of different data types.** The ArrayList class maintains the insertion order and is **non-synchronized**. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```java
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

Java 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 important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

Methods of ArrayList

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.

boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
void ensureCapacity(int requiredCapacity)	It is used to enhance the capacity of an ArrayList instance.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
Iterator()	
listIterator()	
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It returns true if the list contains the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the

	specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
boolean removeIf(Predicate<? super E> filter)	It is used to remove all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	It is used to remove all the elements lies within the given range.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Spliterator<E> spliterator()	It is used to create spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

ArrayList	**LinkedList**
1) ArrayList internally uses a dynamic array to store the elements.	LinkedList internally uses a doubly linked list to store the elements.
2) Manipulation with ArrayList	Manipulation with LinkedList

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

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 a collection. Now it is type safe so typecasting is not required at runtime.

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

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

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

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

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

```
import java.util.*;

public class ArrayListExample2{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
  list.add("Mango");//Adding object in arraylist
  list.add("Apple");
  list.add("Banana");
  list.add("Grapes");
  //Traversing list through Iterator
  Iterator itr=list.iterator();//getting the Iterator
  while(itr.hasNext()){//check if iterator has the elements
   System.out.println(itr.next());//printing the element and move to next
  }
 }
}
```

Get and Set ArrayList

The get() method returns the element at the specified index, whereas the set() method changes the element.

```
import java.util.*;
public class ArrayListExample4{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Mango");
  al.add("Apple");
  al.add("Banana");
  al.add("Grapes");
  //accessing the element
 System.out.println("Returning element: "+al.get(1));//it will return the 2nd element, because index starts from 0
 //changing the element
 al.set(1,"Dates");
 //Traversing list
 for(String fruit:al)
   System.out.println(fruit);

}
}
```

Test it Now

Output:

```
Returning element: Apple
Mango
Dates
Banana
Grapes
```

Ways to iterate the elements of the collection in Java

There are various ways to traverse the collection elements:

1.By Iterator interface.

2.By for-each loop.

3.By ListIterator interface.

4.By for loop.

5.By forEach() method.

6.By forEachRemaining() method.

Iterating Collection through remaining ways

Let's see an example to traverse the ArrayList elements through other ways

```
import java.util.*;
class ArrayList4{
 public static void main(String args[]){
   ArrayList<String> list=new ArrayList<String>();//Creating arraylist
        list.add("Ravi");//Adding object in arraylist
        list.add("Vijay");
        list.add("Ravi");
        list.add("Ajay");

        System.out.println("Traversing list through List Iterator:");
        //Here, element iterates in reverse order
          ListIterator<String> list1=list.listIterator(list.size());
          while(list1.hasPrevious())
          {
              String str=list1.previous();
              System.out.println(str);
          }
      System.out.println("Traversing list through for loop:");
        for(int i=0;i<list.size();i++)
        {
         System.out.println(list.get(i));
        }

      System.out.println("Traversing list through forEach() method:");
      //The forEach() method is a new feature, introduced in Java 8.
         list.forEach(a->{ //Here, we are using lambda expression
             System.out.println(a);
          });

         System.out.println("Traversing list through forEachRemaining() method:");
          Iterator<String> itr=list.iterator();
        itr.forEachRemaining(a-> //Here,  we  are  using  lambda  expression
        {
       System.out.println(a);
        });
}
}
```

Output:

```
Traversing list through List Iterator:
Ajay
Ravi
Vijay
Ravi
Traversing list through for loop:
Ravi
Vijay
Ravi
Ajay
Traversing list through forEach() method:
Ravi
Vijay
Ravi
Ajay
Traversing list through forEachRemaining() method:
Ravi
Vijay
Ravi
Ajay
```

User-defined class objects in Java ArrayList

Let's see an example where we are storing Student class object in an array list.

```
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 ArrayList5{
 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);
  //creating arraylist
  ArrayList<Student> al=new ArrayList<Student>();
  al.add(s1);//adding Student class object
  al.add(s2);
  al.add(s3);
  //Getting Iterator
  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);
  }
 }
}
```

Output:

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

Java ArrayList Serialization and Deserialization Example

Let's see an example to serialize an ArrayList object and then deserialize it.

```
import java.io.*;
import java.util.*;
 class ArrayList6 {

     public static void main(String [] args)
     {
      ArrayList<String> al=new ArrayList<String>();
      al.add("Ravi");
      al.add("Vijay");
      al.add("Ajay");

      try
      {
         //Serialization
         FileOutputStream fos=new FileOutputStream("file");
         ObjectOutputStream oos=new ObjectOutputStream(fos);
         oos.writeObject(al);
         fos.close();
         oos.close();
         //Deserialization
         FileInputStream fis=new FileInputStream("file");
         ObjectInputStream ois=new ObjectInputStream(fis);
       ArrayList  list=(ArrayList)ois.readObject();
       System.out.println(list);
      }catch(Exception e)
      {
         System.out.println(e);
      }
     }
   }
```

Output:

```
[Ravi, Vijay, Ajay]
```

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. **It can store the duplicate elements**. It maintains the insertion order and is not **synchronized**. In LinkedList, the manipulation is fast because no shifting is required.

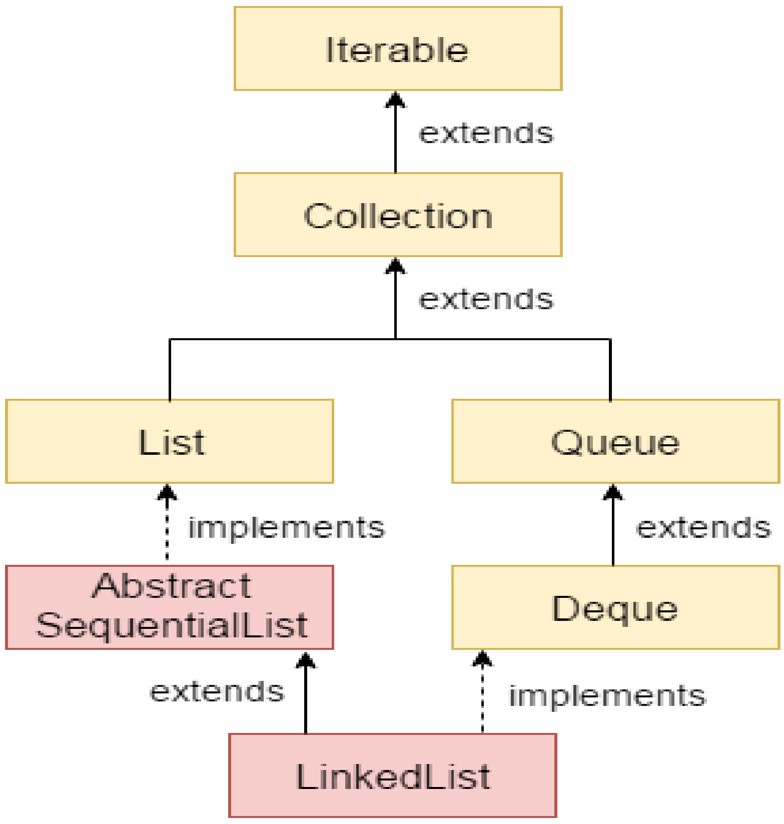
Consider the following example.

```java
import java.util.*;
public class TestJavaCollection2{
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<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

Java 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.

The important points about Java LinkedList are:

❍Java LinkedList class can contain duplicate elements.

❍Java LinkedList class maintains insertion order.

❍Java LinkedList class is non synchronized.

❍In Java LinkedList class, manipulation is fast because no shifting needs to occur.

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

Hierarchy of LinkedList class

As shown in the above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

void clear()	It is used to remove all the elements from a list.

boolean add(E e)	It is used to append the specified element to the end of a list.

boolean contains(Object o)	It is used to return true if a list contains a specified element

Java LinkedList example to add elements

Here, we see different ways to add elements.

```
import java.util.*;
public class LinkedList2{
public static void main(String args[]){
LinkedList<String> ll=new LinkedList<String>();
        System.out.println("Initial list of elements: "+ll);
        ll.add("Ravi");
        ll.add("Vijay");
        ll.add("Ajay");
        System.out.println("After invoking add(E e) method: "+ll);
```

```java
        //Adding an element at the specific position
        ll.add(1, "Gaurav");
        System.out.println("After invoking add(int index, E element) method: "+ll);
        LinkedList<String> ll2=new LinkedList<String>();
        ll2.add("Sonoo");
        ll2.add("Hanumat");
        //Adding second list elements to the first list
        ll.addAll(ll2);
        System.out.println("After invoking addAll(Collection<? extends E> c) method: "+ll);
        LinkedList<String> ll3=new LinkedList<String>();
        ll3.add("John");
        ll3.add("Rahul");
        //Adding second list elements to the first list at specific position
        ll.addAll(1, ll3);
        System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+ll);
        //Adding an element at the first position
        ll.addFirst("Lokesh");
        System.out.println("After invoking addFirst(E e) method: "+ll);
        //Adding an element at the last position
        ll.addLast("Harsh");
        System.out.println("After invoking addLast(E e) method: "+ll);

 }
}
```

```
Initial list of elements: []
After invoking add(E e) method: [Ravi, Vijay, Ajay]
After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]
After invoking addAll(Collection<? extends E> c) method:
[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addAll(int index, Collection<? extends E> c) method:
[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addFirst(E e) method:
[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
```

```
After invoking addLast(E e) method:
[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat, Harsh]
Initial list of elements: []
After invoking add(E e) method: [Ravi, Vijay, Ajay]
After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]
After
```

Java LinkedList example to remove elements

Here, we see different ways to remove an element.

```java
import java.util.*;
public class LinkedList3 {

    public static void main(String [] args)
    {
      LinkedList<String> ll=new LinkedList<String>();
      ll.add("Ravi");
      ll.add("Vijay");
      ll.add("Ajay");
      ll.add("Anuj");
      ll.add("Gaurav");
      ll.add("Harsh");
      ll.add("Virat");
      ll.add("Gaurav");
      ll.add("Harsh");
      ll.add("Amit");
      System.out.println("Initial list of elements: "+ll);
     //Removing specific element from arraylist
        ll.remove("Vijay");
        System.out.println("After invoking remove(object) method: "+ll);
     //Removing element on the basis of specific position
        ll.remove(0);
        System.out.println("After invoking remove(index) method: "+ll);
        LinkedList<String> ll2=new LinkedList<String>();
        ll2.add("Ravi");
        ll2.add("Hanumat");
     // Adding new elements to arraylist
        ll.addAll(ll2);
        System.out.println("Updated list : "+ll);
     //Removing all the new elements from arraylist
        ll.removeAll(ll2);
        System.out.println("After invoking removeAll() method: "+ll);
     //Removing first element from the list
        ll.removeFirst();
        System.out.println("After invoking removeFirst() method: "+ll);
     //Removing first element from the list
        ll.removeLast();
        System.out.println("After invoking removeLast() method: "+ll);
     //Removing first occurrence of element from the list
        ll.removeFirstOccurrence("Gaurav");
        System.out.println("After invoking removeFirstOccurrence() method: "+ll);
     //Removing last occurrence of element from the list
        ll.removeLastOccurrence("Harsh");
        System.out.println("After invoking removeLastOccurrence() method: "+ll);

        //Removing all the elements available in the list
        ll.clear();
        System.out.println("After invoking clear() method: "+ll);
    }
  }
```

```
Initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav,
Harsh, Amit]
After invoking remove(index) method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh,
Amit]
Updated list : [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit, Ravi, Hanumat]
After invoking removeAll() method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
After invoking removeFirst() method: [Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
After invoking removeLast() method: [Gaurav, Harsh, Virat, Gaurav, Harsh]
After invoking removeFirstOccurrence() method: [Harsh, Virat, Gaurav, Harsh]
After invoking removeLastOccurrence() method: [Harsh, Virat, Gaurav]
After invoking clear() method: []
```

Java LinkedList Example to reverse a list of elements

```java
import java.util.*;
public class LinkedList4{
 public static void main(String args[]){

  LinkedList<String> ll=new LinkedList<String>();
```

```
        ll.add("Ravi");
        ll.add("Vijay");
        ll.add("Ajay");
        //Traversing the list of elements in reverse order
        Iterator i=ll.descendingIterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }

}
}
```

```
Output: Ajay
Vijay
Ravi
```

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is **synchronized** and contains many methods that are not the part of Collection framework.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ayush
```

```
Amit
Ashish
Garima
```

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like **boolean push(), boolean peek(), boolean push(object o),** which defines its properties.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ayush
Garvit
Amit
Ashish
```

****

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

```java
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

HashSet

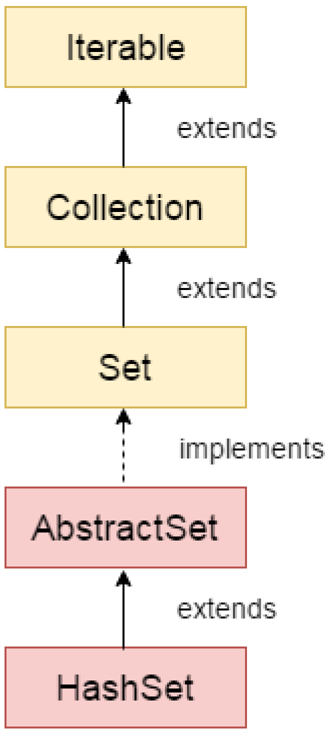
HashSet class implements Set Interface. It represents the collection that uses a **hash table for storage.** Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

```java
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Vijay
Ravi
Ajay
```



Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

❍HashSet stores the elements by using a mechanism called hashing.

❍HashSet contains unique elements only.

❍**HashSet allows null value.**

❍HashSet class is non synchronized.

❍**HashSet doesn't maintain the insertion order**. Here, elements are inserted on the basis of their hashcode.

❍HashSet is the **best approach for search operations.**

❍The initial default capacity of HashSet is 16, and the load factor is 0.75.

Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.

SN	Modifier & Type	Method	Description
1)	boolean	add(E e)	It is used to add the specified element to this set if it is not already present.

2)	void	clear()	It is used to remove all of the elements from the set.
3)	object	clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
4)	boolean	contains(Object o)	It is used to return true if this set contains the specified element.
5)	boolean	isEmpty()	It is used to return true if this set contains no elements.
6)	Iterator<E>	iterator()	It is used to return an iterator over the elements in this set.
7)	boolean	remove(Object o)	It is used to remove the specified element from this set if it is present.
8)	int	size()	It is used to return the number of elements in the set.
9)	Spliterator<E>	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

Java HashSet example to remove elements

Here, we see different ways to remove an element.

```
import java.util.*;
class HashSet3{
 public static void main(String args[]){
  HashSet<String> set=new HashSet<String>();
           set.add("Ravi");
           set.add("Vijay");
           set.add("Arun");
           set.add("Sumit");
           System.out.println("An initial list of elements: "+set);
           //Removing specific element from HashSet
           set.remove("Ravi");
           System.out.println("After invoking remove(object) method: "+set);
           HashSet<String> set1=new HashSet<String>();
```

```
        set1.add("Ajay");
        set1.add("Gaurav");
        set.addAll(set1);
        System.out.println("Updated List: "+set);
        //Removing all the new elements from HashSet
        set.removeAll(set1);
        System.out.println("After invoking removeAll() method: "+set);
        //Removing elements on the basis of specified condition
        set.removeIf(str->str.contains("Vijay"));
        System.out.println("After invoking removeIf() method: "+set);
        //Removing all the elements available in the set
        set.clear();
        System.out.println("After invoking clear() method: "+set);
 }
}
```

```
An initial list of elements: [Vijay, Ravi, Arun, Sumit]
After invoking remove(object) method: [Vijay, Arun, Sumit]
Updated List: [Vijay, Arun, Gaurav, Sumit, Ajay]
After invoking removeAll() method: [Vijay, Arun, Sumit]
After invoking removeIf() method: [Arun, Sumit]
After invoking clear() method: []
```

Java HashSet from another Collection

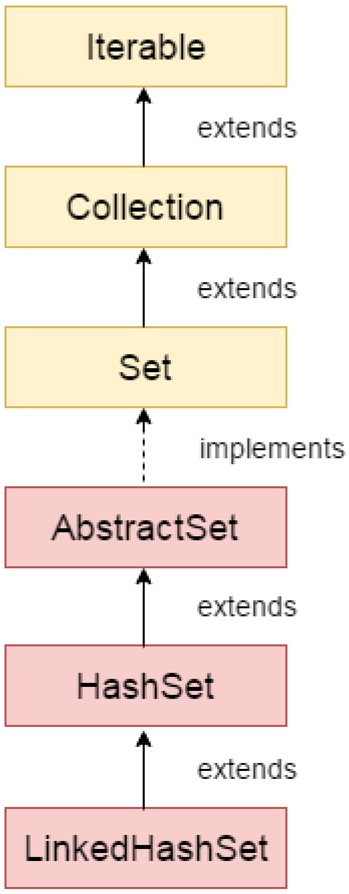
```
import java.util.*;
class HashSet4{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();
        list.add("Ravi");
        list.add("Vijay");
        list.add("Ajay");

        HashSet<String> set=new HashSet(list);
        set.add("Gaurav");
        Iterator<String> i=set.iterator();
        while(i.hasNext())
        {
        System.out.println(i.next());
        }
    }
}
```

```
Vijay
Ravi
Gaurav
Ajay
```

LinkedHashSet

Java LinkedHashSet class



Java LinkedHashSet class is a Hashtable and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

- Java LinkedHashSet class contains unique elements only like HashSet.

❍Java LinkedHashSet class provides all optional set operation and **permits null elements.**

❍Java LinkedHashSet class is non synchronized.

❍**Java LinkedHashSet class maintains insertion order.**

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection8{
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ajay
```

SortedSet Interface

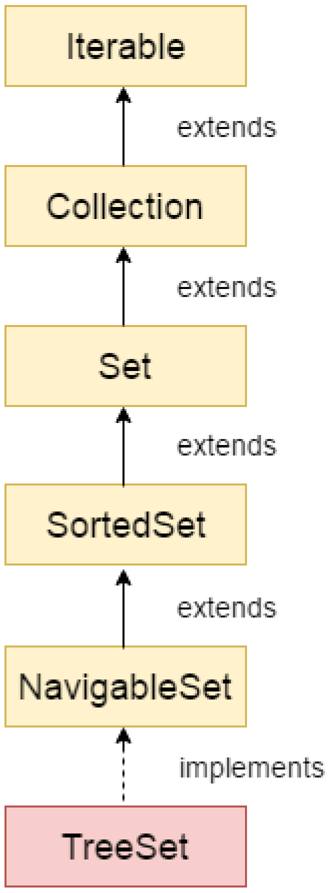
SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet

Java TreeSet class



Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

❍Java TreeSet class contains unique elements only like HashSet.

❍Java TreeSet class access and retrieval times are quiet fast.

❍Java TreeSet class doesn't allow null element.

❍Java TreeSet class is non synchronized.

❍Java TreeSet class **maintains ascending order.**

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```java
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ajay
Ravi
Vijay
```

```java
import java.util.*;

class TreeSet2{
 public static void main(String args[]){
 TreeSet<String> set=new TreeSet<String>();
        set.add("Ravi");
        set.add("Vijay");
        set.add("Ajay");
        System.out.println("Traversing element through Iterator in descending order");
        Iterator i=set.descendingIterator();
        while(i.hasNext())
        {
           System.out.println(i.next());
        }

 }
}
```

Test it Now

Output:

```
Traversing element through Iterator in descending order
Vijay
Ravi
Ajay
Traversing element through NavigableSet in descending order
Vijay
Ravi
Ajay
```

Java TreeSet Example 3:

Let's see an example to retrieve and remove the highest and lowest Value.

```java
import java.util.*;
class TreeSet3{
 public static void main(String args[]){
 TreeSet<Integer> set=new TreeSet<Integer>();
        set.add(24);
        set.add(66);
        set.add(12);
        set.add(15);
        System.out.println("Highest Value: "+set.pollFirst());
        System.out.println("Lowest Value: "+set.pollLast());
 }
}
```

Output:

```
Highest Value: 12
Lowest Value: 66
```

Java TreeSet Example 4:

In this example, we perform various NavigableSet operations.

```java
import java.util.*;
class TreeSet4{
 public static void main(String args[]){
```

```
TreeSet<String> set=new TreeSet<String>();
      set.add("A");
      set.add("B");
      set.add("C");
      set.add("D");
      set.add("E");
      System.out.println("Initial Set: "+set);

      System.out.println("Reverse Set: "+set.descendingSet());

      System.out.println("Head Set: "+set.headSet("C", true));

      System.out.println("SubSet: "+set.subSet("A", false, "E", true));

      System.out.println("TailSet: "+set.tailSet("C", false));
}
}
```

```
Initial Set: [A, B, C, D, E]
Reverse Set: [E, D, C, B, A]
Head Set: [A, B, C]
SubSet: [B, C, D, E]
TailSet: [D, E]
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like **PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.**

Queue interface can be instantiated as:

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

There are various classes that implement the Queue interface, some of them are given below.

---

ethods of Java Queue Interface

Method	Description

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

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. **PriorityQueue doesn't allow null values to be stored in the queue.**

Consider the following example.

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit Sharma");
queue.add("Vijay Raj");
queue.add("JaiShankar");
queue.add("Raj");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
}
```

Output:

```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj
```

Java PriorityQueue Example: Book

Let's see a PriorityQueue example where we are adding books to queue and printing all the books. The elements in PriorityQueue must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in PriorityQueue, you need to implement Comparable interface.

```
import java.util.*;
class Book implements Comparable<Book>{
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
   this.id = id;
   this.name = name;
   this.author = author;
   this.publisher = publisher;
   this.quantity = quantity;
}
```

```java
public int compareTo(Book b) {
  if(id>b.id){
    return 1;
  }else if(id<b.id){
    return -1;
  }else{
  return 0;
  }
}
}
public class LinkedListExample {
public static void main(String[] args) {
  Queue<Book> queue=new PriorityQueue<Book>();
  //Creating Books
  Book b1=new Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
  Book b2=new Book(233,"Operating System","Galvin","Wiley",6);
  Book b3=new Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
  //Adding Books to the queue
  queue.add(b1);
  queue.add(b2);
  queue.add(b3);
  System.out.println("Traversing the queue elements:");
  //Traversing queue elements
  for(Book b:queue){
  System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
  }
  queue.remove();
  System.out.println("After removing one book record:");
  for(Book b:queue){
    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

```
Traversing the queue elements:
101 Data Communications & Networking Forouzan Mc Graw Hill 4
```

```
233 Operating System Galvin Wiley 6
121 Let us C Yashwant Kanetkar BPB 8
After removing one book record:
121 Let us C Yashwant Kanetkar BPB 8
233 Operating System Galvin Wiley 6
```

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

```java
Deque d = new ArrayDeque();
```

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```java
import java.util.*;
public class TestJavaCollection6{
public static void main(String[] args) {
//Creating Deque and adding elements
Deque<String> deque = new ArrayDeque<String>();
deque.add("Gautam");
deque.add("Karan");
deque.add("Ajay");
//Traversing elements
for (String str : deque) {
System.out.println(str);
}
}
}
```

Output:

```
Gautam
Karan
Ajay
```