Unit-4

JAVA COLLECTION FRAMEWORK

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. 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.

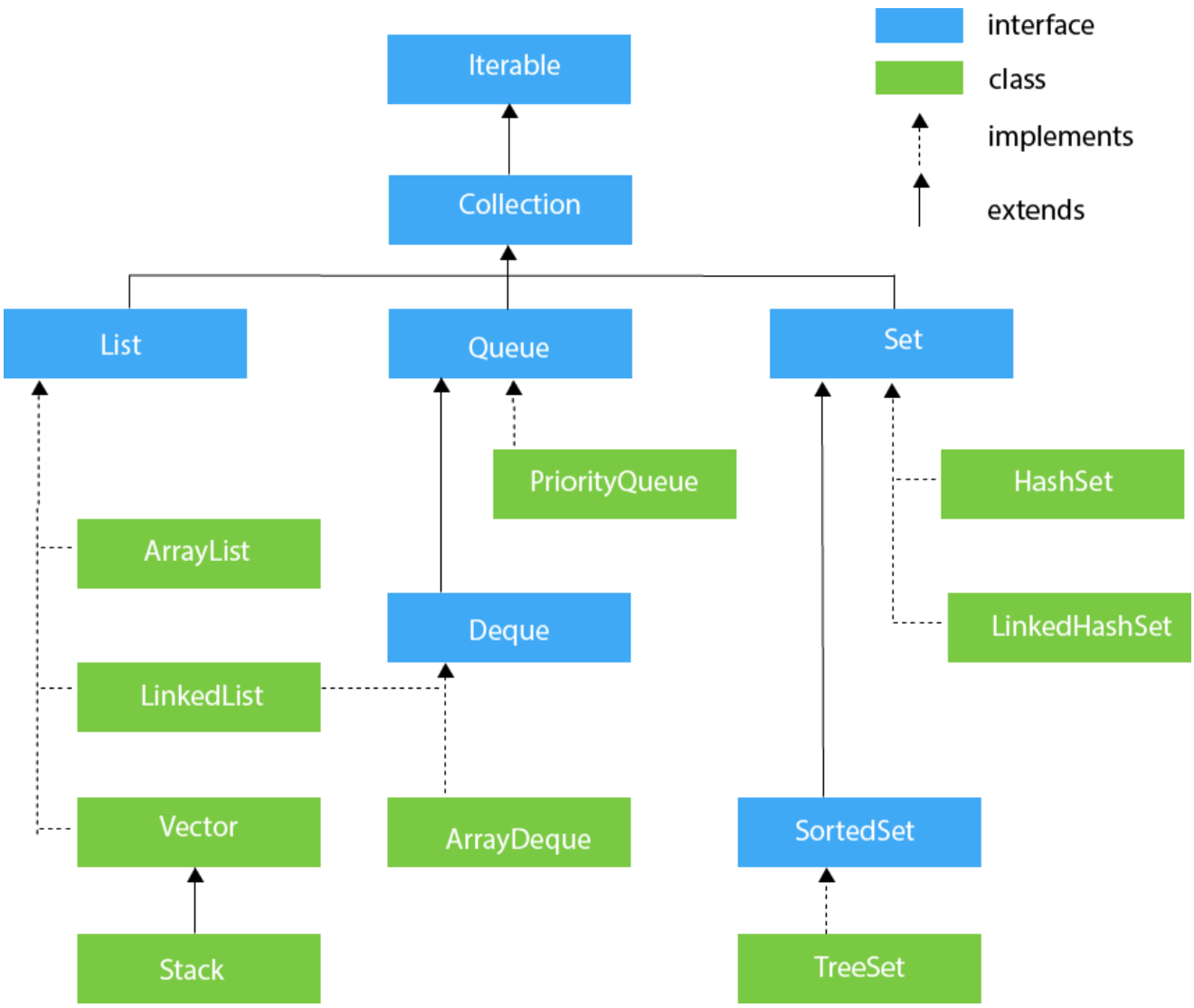
What is Collection framework

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

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.



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.

Iterate an Iterable using Iterator

We can iterate the elements of Java Iterable by obtaining the Iterator from it using the **iterator() method**.

The methods used while traversing the collections using Iterator to perform the operations are:

- **hasNext()**: It returns false if we have reached the end of the collection, otherwise returns true.
- **next()**: Returns the next element in a collection.
- **remove()**: Removes the last element returned by the iterator from the collection.
- **forEachRemaining()**: Performs the given action for each remaining element in a collection, in sequential order.

```
// Java Program to demonstrate iterate

// an Iterable using an Iterator



import java.io.*;

import java.util.*;



class IterateUsingIterator {

    public static void main(String[] args)

    {

        List<String> list = new ArrayList<>();



        list.add("Geeks");

        list.add("for");
```

```
        list.add("Geeks");



        Iterator<String> iterator =
list.iterator();



        while (iterator.hasNext()) {

            String element = iterator.next();

            System.out.println(element);

        }

    }

}
```

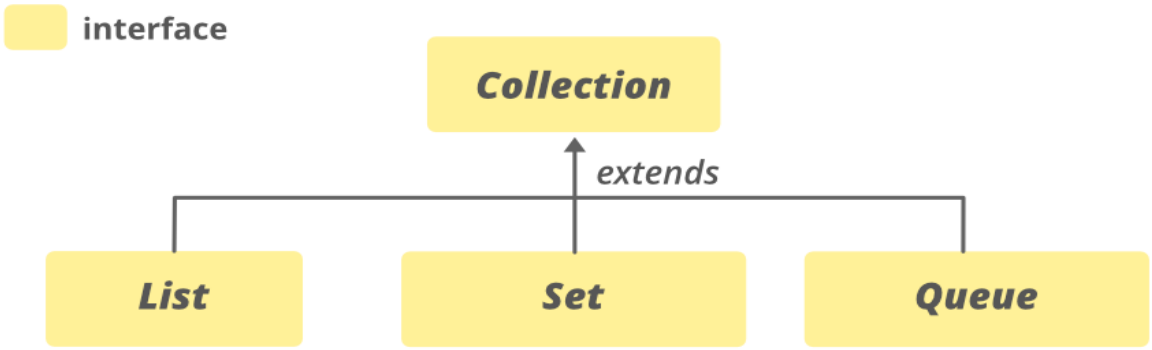
Collection Interface

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.

The **Collection** interface is a member of the Java Collections Framework. It is a part of **java.util** package. It is one of the root interfaces of the Collection Hierarchy. The Collection interface is not directly implemented by any class. However, it is implemented indirectly via its subtypes or subinterfaces like List, Queue, and Set.
For Example, the HashSet class implements the Set interface which is a subinterface of the Collection interface. If a collection implementation doesn't implement a particular operation, it should define the corresponding method to throw **UnsupportedOperationException**.

The Hierarchy of Collection



List: This is a child interface of the collection interface. This interface is dedicated to the data of the list type in which we can store all the ordered collection of the objects. This also allows duplicate data to be present in it. This list interface is implemented by various classes like ArrayList, Vector, Stack, etc. Since all the subclasses implement the list, we can instantiate a list object with any of these classes. For example,
List <T> al = new ArrayList<> ();

List <T> ll = new LinkedList<> ();

List <T> v = new Vector<> ();

Where T is the type of the object

Set: A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects. This set interface is implemented by various classes like HashSet, TreeSet, LinkedHashSet, etc. Since all the subclasses implement the set, we can instantiate a set object with any of these classes. For example,
Set<T> hs = new HashSet<> ();

Set<T> lhs = new LinkedHashSet<> ();

Set<T> ts = new TreeSet<> ();

Where T is the type of the object.

SortedSet: This interface is very similar to the set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements. The sorted set interface extends the set interface and is used to handle the data which needs to be sorted. The class which implements this interface is TreeSet. Since this class implements the SortedSet, we can instantiate a SortedSet object with this class. For example,
SortedSet<T> ts = new TreeSet<> ();

Where T is the type of the object.

Queue: As the name suggests, a queue interface maintains the FIFO(First In First Out) order similar to a real-world queue line. This interface is dedicated to storing all the elements where the order of

the elements matter. For example, whenever we try to book a ticket, the tickets are sold at the first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket. There are various classes like PriorityQueue, Deque, ArrayDeque, etc. Since all these subclasses implement the queue, we can instantiate a queue object with any of these classes. For example,
Queue <T> pq = new PriorityQueue<> ();

Queue <T> ad = new ArrayDeque<> ();

Where T is the type of the object.

Deque: This is a very slight variation of the queue data structure. Deque, also known as a double-ended queue, is a data structure where we can add and remove the elements from both the ends of the queue. This interface extends the queue interface. The class which implements this interface is ArrayDeque. Since this class implements the deque, we can instantiate a deque object with this class. For example,
Deque<T> ad = new ArrayDeque<> ();

Where T is the type of the object.

Declaration:

```
public interface Collection<E> extends Iterable<E>
```

Here, **E** is the type of elements stored in the collection.
Example:

- Java

```
// Java program to illustrate Collection interface

import java.io.*;

import java.util.*;

public class CollectionDemo {

    public static void main(String args[])

    {



        // creating an empty LinkedList

        Collection<String> list = new LinkedList<String>();

         // use add() method to add elements in the list
```

```
        list.add("Geeks");

        list.add("for");

        list.add("Geeks");

         // Output the present list

        System.out.println("The list is: " + list);

         // Adding new elements to the end

        list.add("Last");

        list.add("Element");

         // printing the new list

        System.out.println("The new List is: " + list);

    }

}
```

List Interface

List Interface in Java

The List interface is found in java.util package and inherits the Collection interface. It is a factory of the ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of the List interface are ArrayList, LinkedList, Stack, and Vector. ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

List Interface in Java

The List interface is found in java.util package and inherits the Collection interface. It is a factory of the ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of the List interface are ArrayList, LinkedList, Stack, and Vector. ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

Declaration of Java List Interface

```
public interface List<E> extends Collection<E> ;
```

Let us elaborate on creating objects or instances in a List class. Since **List** is an interface, objects cannot be created of the type list. We always need a class that implements this **List** in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the List. Just like several other user-defined 'interfaces' implemented by user-defined 'classes', **List** is an 'interface', implemented by the **ArrayList** class, pre-defined in **java.util** package.

Syntax of Java List

This type of safelist can be defined as:

```
List<Obj> list = new ArrayList<Obj> ();
```

Note: *Obj is the type of the object to be stored in List*

Example of Java List

- Java

```
// Java program to Demonstrate List Interface

 // Importing all utility classes

import java.util.*;

 // Main class

// ListDemo class

class GFG {

     // Main driver method

    public static void main(String[] args)

    {

         // Creating an object of List interface

        // implemented by the ArrayList class

        List<Integer> l1 = new ArrayList<Integer>();

        // Adding elements to object of List interface

        // Custom inputs
```

```
 l1.add(0, 1);

l1.add(1, 2);

 // Print the elements inside the object

System.out.println(l1);

 // Now creating another object of the List

// interface implemented ArrayList class

// Declaring object of integer type

List<Integer> l2 = new ArrayList<Integer>();



// Again adding elements to object of List interface

// Custom inputs

l2.add(1);

l2.add(2);

l2.add(3);

 // Will add list l2 from 1 index

l1.addAll(1, l2);

 System.out.println(l1);

 // Removes element from index 1

l1.remove(1);

// Printing the updated List 1

System.out.println(l1);
```

```
          // Prints element at index 3 in list 1

        // using get() method

        System.out.println(l1.get(3));

          // Replace 0th element with 5

        // in List 1

        l1.set(0, 5);



        // Again printing the updated List 1

        System.out.println(l1);

    }

}
```

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.

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

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. 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.

Operations in a Java List Interface

Since List is an interface, it can be used only with a class that implements this interface. Now, let's see how to perform a few frequently used operations on the List.

- **Operation 1:** Adding elements to List class using add() method

- **Operation 2:** Updating elements in List class using set() method
- **Operation 3:** Searching for elements using indexOf(), lastIndexOf methods
- **Operation 4:** Removing elements using remove() method
- **Operation 5:** Accessing Elements in List class using get() method
- **Operation 6:** Checking if an element is present in the List class using contains() method

Now let us discuss the operations individually and implement the same in the

1. Adding elements to List class using add() method

In order to add an element to the list, we can use the **add()** method. This method is overloaded to perform multiple operations based on different parameters.

Parameters: It takes 2 parameters, namely:

- **add(Object):** This method is used to add an element at the end of the List.
- **add(int index, Object):** This method is used to add an element at a specific index in the List

Example:

- Java

```
// Java Program to Add Elements to a List



// Importing all utility classes

import java.util.*;



// Main class

class GFG {



    // Main driver method

    public static void main(String args[])

    {

        // Creating an object of List interface,

        // implemented by ArrayList class
```

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



        // Adding elements to object of List interface

        // Custom elements

        al.add("Geeks");

        al.add("Geeks");

        al.add(1, "For");



        // Print all the elements inside the

        // List interface object

        System.out.println(al);

    }

}
```

Output

```
[Geeks, For, Geeks]
```

2. Updating elements

After adding the elements, if we wish to change the element, it can be done using the **set()** method. Since List is indexed, the element which we wish to change is referenced by the index of the element. Therefore, this method takes an index and the updated element which needs to be inserted at that index.

Example:

- Java

```
// Java Program to Update Elements in a List
```

```
// Importing utility classes
import java.util.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String args[])
    {
        // Creating an object of List interface
        List<String> al = new ArrayList<>();

        // Adding elements to object of List class
        al.add("Geeks");
        al.add("Geeks");
        al.add(1, "Geeks");

        // Display theinitial elements in List
        System.out.println("Initial ArrayList " + al);
```

```
        // Setting (updating) element at 1st index

        // using set() method

        al.set(1, "For");



        // Print and display the updated List

        System.out.println("Updated ArrayList " + al);

    }

}
```

Output

```
Initial ArrayList [Geeks, Geeks, Geeks]
Updated ArrayList [Geeks, For, Geeks]
```

3. Searching for elements

Searching for elements in the List interface is a common operation in Java programming. The List interface provides several methods to search for elements, such as the **indexOf()**, **lastIndexOf()** methods.

The indexOf() method returns the index of the first occurrence of a specified element in the list, while the lastIndexOf() method returns the index of the last occurrence of a specified element.

Parameters:

- **indexOf(element):** Returns the index of the first occurrence of the specified element in the list, or -1 if the element is not found
- **lastIndexOf(element):** Returns the index of the last occurrence of the specified element in the list, or -1 if the element is not found

Example:

- Java

```
import java.util.ArrayList;

import java.util.List;
```

```
public class ListExample {

    public static void main(String[] args)

    {

        // create a list of integers

        List<Integer> numbers = new ArrayList<>();



        // add some integers to the list

        numbers.add(1);

        numbers.add(2);

        numbers.add(3);

        numbers.add(2);



        // use indexOf() to find the first occurrence of an

        // element in the list

        int index = numbers.indexOf(2);

        System.out.println(

            "The first occurrence of 2 is at index "

            + index);



        // use lastIndexOf() to find the last occurrence of
```

```
        // an element in the list

        int lastIndex = numbers.lastIndexOf(2);

        System.out.println(

            "The last occurrence of 2 is at index "

            + lastIndex);

    }

}
```

Output

```
The first occurrence of 2 is at index 1
The last occurrence of 2 is at index 3
```

4. Removing Elements

In order to remove an element from a list, we can use the **remove()** method. This method is overloaded to perform multiple operations based on different parameters. They are:

Parameters:

- **remove(Object):** This method is used to simply remove an object from the List. If there are multiple such objects, then the first occurrence of the object is removed.
- **remove(int index):** Since a List is indexed, this method takes an integer value which simply removes the element present at that specific index in the List. After removing the element, all the elements are moved to the left to fill the space and the indices of the objects are updated.

Example:

- Java

```
// Java Program to Remove Elements from a List



// Importing List and ArrayList classes

// from java.util package
```

```
import java.util.ArrayList;

import java.util.List;



// Main class

class GFG {



    // Main driver method

    public static void main(String args[])

    {



        // Creating List class object

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



        // Adding elements to the object

        // Custom inputs

        al.add("Geeks");

        al.add("Geeks");



        // Adding For at 1st indexes

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

```
        // Print the initialArrayList
        System.out.println("Initial ArrayList " + al);

        // Now remove element from the above list
        // present at 1st index
        al.remove(1);

        // Print the List after removal of element
        System.out.println("After the Index Removal " + al);

        // Now remove the current object from the updated
        // List
        al.remove("Geeks");

        // Finally print the updated List now
        System.out.println("After the Object Removal "
                           + al);
    }
}
```

Output

```
Initial ArrayList [Geeks, For, Geeks]
After the Index Removal [Geeks, Geeks]
After the Object Removal [Geeks]
```

5. Accessing Elements

In order to access an element in the list, we can use the **get()** method, which returns the element at the specified index

Parameters:

get(int index): This method returns the element at the specified index in the list.

Example:

- Java

```
// Java Program to Access Elements of a List



// Importing all utility classes

import java.util.*;



// Main class

class GFG {

    // Main driver method

    public static void main(String args[])

    {

        // Creating an object of List interface,

        // implemented by ArrayList class

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



        // Adding elements to object of List interface

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

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

        al.add("Geeks");



        // Accessing elements using get() method

        String first = al.get(0);

        String second = al.get(1);

        String third = al.get(2);



        // Printing all the elements inside the

        // List interface object

        System.out.println(first);

        System.out.println(second);

        System.out.println(third);

        System.out.println(al);

    }

}
```

Output

```
Geeks
For
Geeks
[Geeks, For, Geeks]
```

6. Checking if an element is present in the List

In order to check if an element is present in the list, we can use the **contains()** method. This method returns true if the specified element is present in the list, otherwise, it returns false.

Parameters:

contains(Object): This method takes a single parameter, the object to be checked if it is present in the list.

Example:

- Java

```
// Java Program to Check if an Element is Present in a List



// Importing all utility classes

import java.util.*;



// Main class

class GFG {

    // Main driver method

    public static void main(String args[])

    {

        // Creating an object of List interface,

        // implemented by ArrayList class

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



        // Adding elements to object of List interface

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

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

        al.add("Geeks");



        // Checking if element is present using contains()

        // method

        boolean isPresent = al.contains("Geeks");



        // Printing the result

        System.out.println("Is Geeks present in the list? "

                           + isPresent);

    }

}
```

Output

```
Is Geeks present in the list? true
```

ArrayList

ArrayList in Java

Last Updated : 05 Apr, 2024

Java ArrayList is a part of the Java **collections framework** and it is a class of java.util package. It provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. This class is found in **java.util** package. The main **advantage of ArrayList in Java** is, that if we declare an array then we need to mention the size, but in ArrayList, it is not needed to mention the size of ArrayList. If you want to mention the size then you can do it.

ArrayList is a Java class implemented using the List interface. Java ArrayList, as the name suggests, provides the functionality of a dynamic array where the size is not fixed as an array. Also, as a part of the Collections framework, it has many features not available with arrays.

```java
// Java program to demonstrate the
// working of ArrayList
import java.io.*;
import java.util.*;

class ArrayListExample {
    public static void main(String[] args)
    {
        // Size of the
        // ArrayList
        int n = 5;

        // Declaring the ArrayList with
        // initial size n
        ArrayList<Integer> arr1 = new ArrayList<Integer>(n);

        // Declaring the ArrayList
        ArrayList<Integer> arr2 = new ArrayList<Integer>();

        // Printing the ArrayList
        System.out.println("Array 1:" + arr1);
        System.out.println("Array 2:" + arr2);

        // Appending new elements at
        // the end of the list
        for (int i = 1; i <= n; i++) {
            arr1.add(i);
            arr2.add(i);
        }

        // Printing the ArrayList
        System.out.println("Array 1:" + arr1);
        System.out.println("Array 2:" + arr2);
    }
}
```

Output

```
Array 1:[]
Array 2:[]
Array 1:[1, 2, 3, 4, 5]
Array 2:[1, 2, 3, 4, 5]
```

Explanation of the above Program:

ArrayList is a dynamic array and we do not have to specify the size while creating it, the size of the array automatically increases when we dynamically add and remove items. Though the actual library implementation may be more complex, the following is a very

basic idea explaining the working of the array when the array becomes full and if we try to add an item:

- Creates a bigger-sized memory on heap memory (for example memory of double size).
- Copies the current memory elements to the new memory.
- The new item is added now as there is bigger memory available now.
- Delete the old memory.

Important Features of ArrayList in Java

- ArrayList inherits AbstractList class and implements the List interface.
- ArrayList is initialized by size. However, the size is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.
- ArrayList in Java can be seen as a vector in C++.
- ArrayList is not Synchronized. Its equivalent synchronized class in Java is Vector

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.

```
1. import java.util.*;
2. class TestJavaCollection1{
3. public static void main(String args[]){
4. ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. while(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

Some Key Points of ArrayList in Java

1. ArrayList is Underlined data Structure Resizable Array or Growable Array.
2. ArrayList Duplicates Are Allowed.
3. Insertion Order is Preserved.
4. Heterogeneous objects are allowed.
5. Null insertion is possible.

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.

Linked List is a part of the Collection framework present in java.util package. This class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node.
Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. It also has a few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach a node we wish to access.

How Does LinkedList work Internally?

Since a LinkedList acts as a dynamic array and we do not have to specify the size while creating it, the size of the list automatically increases when we dynamically add and remove items. And also, the elements are not stored in a continuous fashion. Therefore, there is no need to increase the size. Internally, the LinkedList is implemented using the doubly linked list data structure.
The main difference between a normal linked list and a doubly LinkedList is that a doubly linked list contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

Methods for Java LinkedList:

Method	Description
add(int index, E element)	This method Inserts the specified element at the specified position in this list.
add(E e)	This method Appends the specified element to the end of this

Method	Description
	list.
addAll(int index, Collection<E> c)	This method Inserts all of the elements in the specified collection into this list, starting at the specified position.
addAll(Collection<E> c)	This method Appends 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.
addFirst(E e)	This method Inserts the specified element at the beginning of this list.
addLast(E e)	This method Appends the specified element to the end of this list.
clear()	This method removes all of the elements from this list.
clone()	This method returns a shallow copy of this LinkedList.
contains(Object o)	This method returns true if this list contains the specified element.
descendingIterator()	This method returns an iterator over the elements in this deque in reverse sequential order.
element()	This method retrieves but does not remove, the head (first element) of this list.
get(int index)	This method returns the element at the specified position in this list.

Method	Description
getFirst()	This method returns the first element in this list.
getLast()	This method returns the last element in this list.
indexOf(Object o)	This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
lastIndexOf(Object o)	This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
listIterator(int index)	This method returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
offer(E e)	This method Adds the specified element as the tail (last element) of this list.
offerFirst(E e)	This method Inserts the specified element at the front of this list.
offerLast(E e)	This method Inserts the specified element at the end of this list.
peek()	This method retrieves but does not remove, the head (first element) of this list.
peekFirst()	This method retrieves, but does not remove, the first element of this list, or returns null if this list is empty.

Method	Description
peekLast()	This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty.
poll()	This method retrieves and removes the head (first element) of this list.
pollFirst()	This method retrieves and removes the first element of this list, or returns null if this list is empty.
pollLast()	This method retrieves and removes the last element of this list, or returns null if this list is empty.
pop()	This method Pops an element from the stack represented by this list.
push(E e)	This method pushes an element onto the stack represented by this list.
remove()	This method retrieves and removes the head (first element) of this list.
remove(int index)	This method removes the element at the specified position in this list.
remove(Object o)	This method removes the first occurrence of the specified element from this list if it is present.
removeFirst()	This method removes and returns the first element from this list.
removeFirstOccurrence(Object	This method removes the first occurrence of the specified

Method	Description
o)	element in this list (when traversing the list from head to tail).
removeLast()	This method removes and returns the last element from this list.
removeLastOccurrence(Object o)	This method removes the last occurrence of the specified element in this list (when traversing the list from head to tail).
set(int index, E element)	This method replaces the element at the specified position in this list with the specified element.
size()	This method returns the number of elements in this list.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection2{
3. public static void main(String args[]){
4. LinkedList<String> al=new LinkedList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

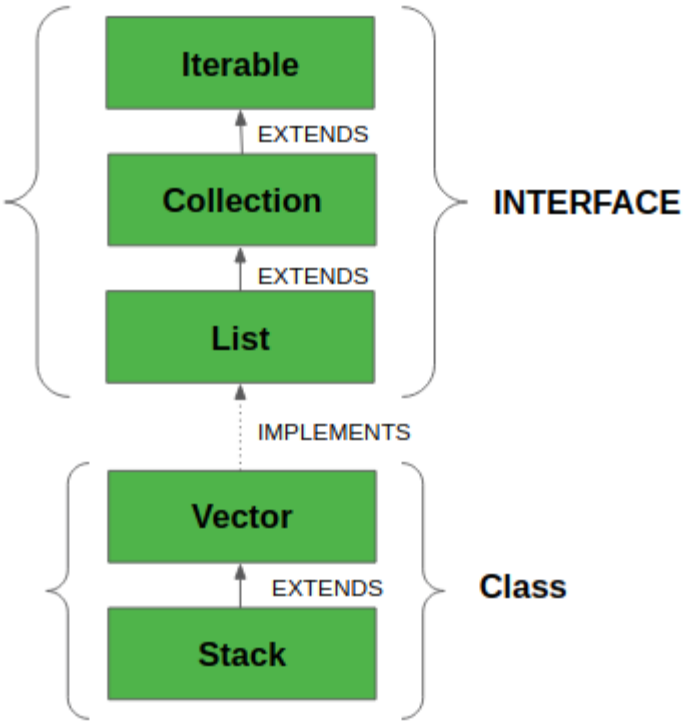
```
Ravi
```

```
Vijay
Ravi
Ajay
```

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.

The Vector class implements a growable array of objects. Vectors fall in legacy classes, but now it is fully compatible with collections. It is found in java.util package and implement the List interface, so we can use all the methods of the List interface as shown below as follows:



- Vector implements a dynamic array which means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index.
- They are very similar to ArrayList, but Vector is synchronized and has some legacy methods that the collection framework does not contain.
- It also maintains an insertion order like an ArrayList. Still, it is rarely used in a non-thread environment as it is **synchronized**, and due to this, it gives a poor performance in adding, searching, deleting, and updating its elements.
- The Iterators returned by the Vector class are fail-fast. In the case of concurrent modification, it fails and throws the **ConcurrentModificationException.**

Syntax:

```
public class Vector<E> extends AbstractList<E> implements List<E>, RandomAccess,
Cloneable, Serializable
```

Here, **E** is the type of element.

- It extends AbstractList and implements List interfaces.
- It implements Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess interfaces.
- The directly known subclass is Stack.

Important points regarding the Increment of vector capacity are as follows:

If the increment is specified, Vector will expand according to it in each allocation cycle. Still, if the increment is not specified, then the vector's capacity gets doubled in each allocation cycle. Vector defines three protected data members:

- ***int capacityIncrement:*** Contains the increment value.
- ***int elementCount:*** Number of elements currently in vector stored in it.
- ***Object elementData[]:*** Array that holds the vector is stored in it.

Common Errors in the declaration of Vectors are as follows**:**

- Vector throws an **IllegalArgumentException** if the InitialSize of the vector defined is negative.
- If the specified collection is null, It throws **NullPointerException**.

Constructors

1. Vector(): Creates a default vector of the initial capacity is 10.

```
Vector<E> v = new Vector<E>();
```

2. Vector(int size): Creates a vector whose initial capacity is specified by size.

```
Vector<E> v = new Vector<E>(int size);
```

3. Vector(int size, int incr): Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time a vector is resized upward.

```
Vector<E> v = new Vector<E>(int size, int incr);
```

4. Vector(Collection c): Creates a vector that contains the elements of collection c.

```
Vector<E> v = new Vector<E>(Collection c);
```

Methods in Vector Class

METHOD	DESCRIPTION
add(E e)	Appends the specified element to the end of this Vector.
add(int index, E element)	Inserts the specified element at the specified position in this Vector.
addAll(Collection<? extends E> c)	Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
addAll(int index,	Insert all of the elements in the specified

METHOD	DESCRIPTION
Collection<? extends E> c)	Collection into this Vector at the specified position.
addElement(E obj)	Adds the specified component to the end of this vector, increasing its size by one.
capacity()	Returns the current capacity of this vector.
clear()	Removes all of the elements from this Vector.
clone()	Returns a clone of this vector.
contains(Object o)	Returns true if this vector contains the specified element.
containsAll(Collection<?> c)	Returns true if this Vector contains all of the elements in the specified Collection.
copyInto(Object[] anArray)	Copies the components of this vector into the specified array.
elementAt(int index)	Returns the component at the specified index.
elements()	Returns an enumeration of the components of this vector.
ensureCapacity(int minCapacity)	Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified

METHOD	DESCRIPTION
	by the minimum capacity argument.
equals(Object o)	Compares the specified Object with this Vector for equality.
firstElement()	Returns the first component (the item at index 0) of this vector.
forEach(Consumer<? super E> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
get(int index)	Returns the element at the specified position in this Vector.
hashCode()	Returns the hash code value for this Vector.
indexOf(Object o)	Returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
indexOf(Object o, int index)	Returns the index of the first occurrence of the specified element in this vector, searching forwards from the index, or returns -1 if the element is not found.
insertElementAt(E obj, int index)	Inserts the specified object as a component in this vector at the specified index.
isEmpty()	Tests if this vector has no components.

METHOD	DESCRIPTION
iterator()	Returns an iterator over the elements in this list in a proper sequence.
lastElement()	Returns the last component of the vector.
lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
lastIndexOf(Object o, int index)	Returns the index of the last occurrence of the specified element in this vector, searching backward from the index, or returns -1 if the element is not found.
listIterator()	Returns a list iterator over the elements in this list (in proper sequence).
listIterator(int index)	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
remove(int index)	Removes the element at the specified position in this Vector.
remove(Object o)	Removes the first occurrence of the specified element in this Vector. If the Vector does not contain the element, it is unchanged.
removeAll(Collection<?> c)	Removes from this Vector all of its elements contained in the specified Collection.

METHOD	DESCRIPTION
removeAllElements()	Removes all components from this vector and sets its size to zero.
removeElement(Object obj)	Removes the first (lowest-indexed) occurrence of the argument from this vector.
removeElementAt(int index)	Deletes the component at the specified index.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection3{
3. public static void main(String args[]){
4. Vector<String> v=new Vector<String>();
5. v.add("Ayush");
6. v.add("Amit");
7. v.add("Ashish");
8. v.add("Garima");
9. Iterator<String> itr=v.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

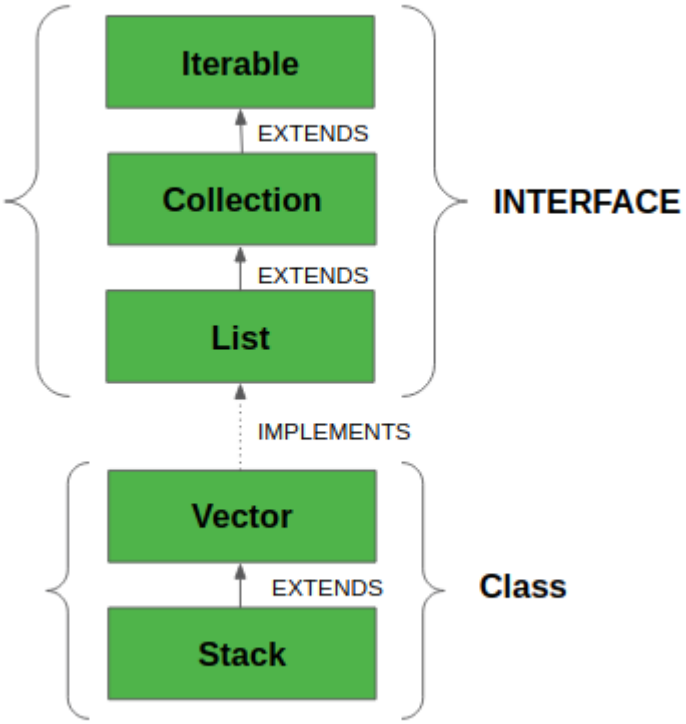
Output:

```
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.

Java Collection framework provides a Stack class that models and implements a **Stack data structure**. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search, and peek. The class can also be said to extend Vector and treats the class as a stack with the five mentioned functions. The class can also be referred to as the subclass of Vector.



All Implemented Interfaces:

- **Serializable:** It is a marker interface that classes must implement if they are to be serialized and deserialized.
- **Cloneable:** This is an interface in Java which needs to be implemented by a class to allow its objects to be cloned.
- **Iterable<E>:** This interface represents a collection of objects which is iterable — meaning which can be iterated.
- **Collection<E>:** A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired.
- **List<E>:** The List interface provides a way to store the ordered collection. It is a child interface of Collection.

- **RandomAccess:** This is a marker interface used by List implementations to indicate that they support fast (generally constant time) random access.

How to Create a Stack?

In order to create a stack, we must import **java.util.stack** package and use the Stack() constructor of this class. The below example creates an empty Stack.
Stack<E> stack = new Stack<E>();
Here E is the type of Object.

Example:

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection4{
3. public static void main(String args[]){
4. Stack<String> stack = new Stack<String>();
5. stack.push("Ayush");
6. stack.push("Garvit");
7. stack.push("Amit");
8. stack.push("Ashish");
9. stack.push("Garima");
10. stack.pop();
11. Iterator<String> itr=stack.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Ayush
Garvit
Amit
Ashish
```

Performing various operations on Stack class

1. Adding Elements: In order to add an element to the stack, we can use the *push() method*. This **push()** operation place the element at the top of the stack.

- Java

```
// Java program to add the

// elements in the stack

import java.io.*;

import java.util.*;



class StackDemo {



      // Main Method

    public static void main(String[] args)

    {



        // Default initialization of Stack

        Stack stack1 = new Stack();



        // Initialization of Stack

        // using Generics

        Stack<String> stack2 = new Stack<String>();



        // pushing the elements

        stack1.push("4");

        stack1.push("All");
```

```
        stack1.push("Geeks");



        stack2.push("Geeks");

        stack2.push("For");

        stack2.push("Geeks");



          // Printing the Stack Elements

        System.out.println(stack1);

        System.out.println(stack2);

    }

}
```

Output:

```
[4, All, Geeks]
[Geeks, For, Geeks]
```

2. Accessing the Element: To retrieve or fetch the first element of the Stack or the element present at the top of the Stack, we can use **peek()** method. The element retrieved does not get deleted or removed from the Stack.

- Java

```
// Java program to demonstrate the accessing

// of the elements from the stack

import java.util.*;

import java.io.*;
```

```
public class StackDemo {



      // Main Method

    public static void main(String args[])

    {

        // Creating an empty Stack

        Stack<String> stack = new Stack<String>();



        // Use push() to add elements into the Stack

        stack.push("Welcome");

        stack.push("To");

        stack.push("Geeks");

        stack.push("For");

        stack.push("Geeks");



        // Displaying the Stack

        System.out.println("Initial Stack: " + stack);



        // Fetching the element at the head of the Stack

        System.out.println("The element at the top of the"

                           + " stack is: " + stack.peek());
```

```
        // Displaying the Stack after the Operation
        System.out.println("Final Stack: " + stack);
    }
}
```

Output:

```
Initial Stack: [Welcome, To, Geeks, For, Geeks]
The element at the top of the stack is: Geeks
Final Stack: [Welcome, To, Geeks, For, Geeks]
```

3. Removing Elements: To pop an element from the stack, we can use the **pop()** method. The element is popped from the top of the stack and is removed from the same.

- Java

```
// Java program to demonstrate the removing
// of the elements from the stack
import java.util.*;
import java.io.*;

public class StackDemo {
    public static void main(String args[])
    {
        // Creating an empty Stack
        Stack<Integer> stack = new Stack<Integer>();
```

```
        // Use add() method to add elements
        stack.push(10);
        stack.push(15);
        stack.push(30);
        stack.push(20);
        stack.push(5);

        // Displaying the Stack
        System.out.println("Initial Stack: " + stack);

        // Removing elements using pop() method
        System.out.println("Popped element: "
                           + stack.pop());
        System.out.println("Popped element: "
                           + stack.pop());

        // Displaying the Stack after pop operation
        System.out.println("Stack after pop operation "
                           + stack);
    }
}
```

Output:

```
Initial Stack: [10, 15, 30, 20, 5]
Popped element: 5
Popped element: 20
Stack after pop operation [10, 15, 30]
```

Example

In Java, the Stack class is a subclass of the Vector class and represents a last-in-first-out (LIFO) stack of objects. It extends the Vector class to allow for easy implementation of the stack data structure.
Here's an example of how you can use the Stack class in Java:

- Java

```
import java.util.Stack;



public class StackExample {

    public static void main(String[] args) {

        // Create a new stack

        Stack<Integer> stack = new Stack<>();



        // Push elements onto the stack

        stack.push(1);

        stack.push(2);

        stack.push(3);

        stack.push(4);



        // Pop elements from the stack

        while(!stack.isEmpty()) {
```

```
            System.out.println(stack.pop());

        }

    }

}
```

Output

```
4
3
2
1
```

In this example, we first import the Stack class from the java.util package. We then create a new Stack object called stack using the default constructor. We push four integers onto the stack using the push() method. We then pop the elements from the stack using the pop() method inside a while loop. The isEmpty() method is used to check if the stack is empty before attempting to pop an element.

This code creates a stack of integers and pushes 4 integers onto the stack in the order 1 -> 2 -> 3 -> 4. We then pop elements from the stack one by one using the pop() method, which removes and returns the top element of the stack. Since the stack follows a last-in-first-out (LIFO) order, the elements are popped in the reverse order of insertion, resulting in the output shown above.

The Stack class provides several other methods for manipulating the stack, such as peek() to retrieve the top element without removing it, search() to search for an element in the stack and return its position, and size() to return the current size of the stack. The Stack class also provides several constructors for creating a stack with a specified initial capacity or by copying an existing stack.

Methods in Stack Class

METHOD	DESCRIPTION
empty()	It returns true if nothing is on the top of the stack. Else, returns false.
peek()	Returns the element on the top of the stack, but does not remove it.
pop()	Removes and returns the top element of the stack. An 'EmptyStackException' An exception is thrown if we call pop() when the invoking stack is empty.

METHOD	DESCRIPTION
push(Object element)	Pushes an element on the top of the stack.
search(Object element)	It determines whether an object exists in the stack. If the element is found, It returns the position of the element from the top of the stack. Else, it returns -1.

Methods inherited from class java.util.Vector

METHOD	DESCRIPTION
add(Object obj)	Appends the specified element to the end of this Vector.
add(int index, Object obj)	Inserts the specified element at the specified position in this Vector.
addAll(Collection c)	Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
addAll(int index, Collection c)	Inserts all the elements in the specified Collection into this Vector at the specified position.
addElement(Object o)	Adds the specified component to the end of this vector, increasing its size by one.
capacity()	Returns the current capacity of this vector.
clear()	Removes all the elements from this Vector.
clone()	Returns a clone of this vector.
contains(Object o)	Returns true if this vector contains the specified element.

METHOD	DESCRIPTION
containsAll(Collection c)	Returns true if this Vector contains all the elements in the specified Collection.
copyInto(Object []array)	Copies the components of this vector into the specified array.
elementAt(int index)	Returns the component at the specified index.
elements()	Returns an enumeration of the components of this vector.
ensureCapacity(int minCapacity)	Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
equals()	Compares the specified Object with this Vector for equality.
firstElement()	Returns the first component (the item at index 0) of this vector.
get(int index)	Returns the element at the specified position in this Vector.
hashCode()	Returns the hash code value for this Vector.
indexOf(Object o)	Returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
indexOf(Object o, int index)	Returns the index of the first occurrence of the specified element in this vector, searching forwards from the index, or returns -1 if the element is not found.
insertElementAt(Object o, int index)	Inserts the specified object as a component in this vector at the specified index.

METHOD	DESCRIPTION
isEmpty()	Tests if this vector has no components.
iterator()	Returns an iterator over the elements in this list in proper sequence.
lastElement()	Returns the last component of the vector.
lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element in this vector, or -1 If this vector does not contain the element.
lastIndexOf(Object o, int index)	Returns the index of the last occurrence of the specified element in this vector, searching backward from the index, or returns -1 if the element is not found.
listIterator()	Returns a list iterator over the elements in this list (in proper sequence).
listIterator(int index)	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
remove(int index)	Removes the element at the specified position in this Vector.
remove(Object o)	Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
removeAll(Collection c)	Removes from this Vector all of its elements that are contained in the specified Collection.
removeAllElements()	Removes all components from this vector and sets its size to zero.

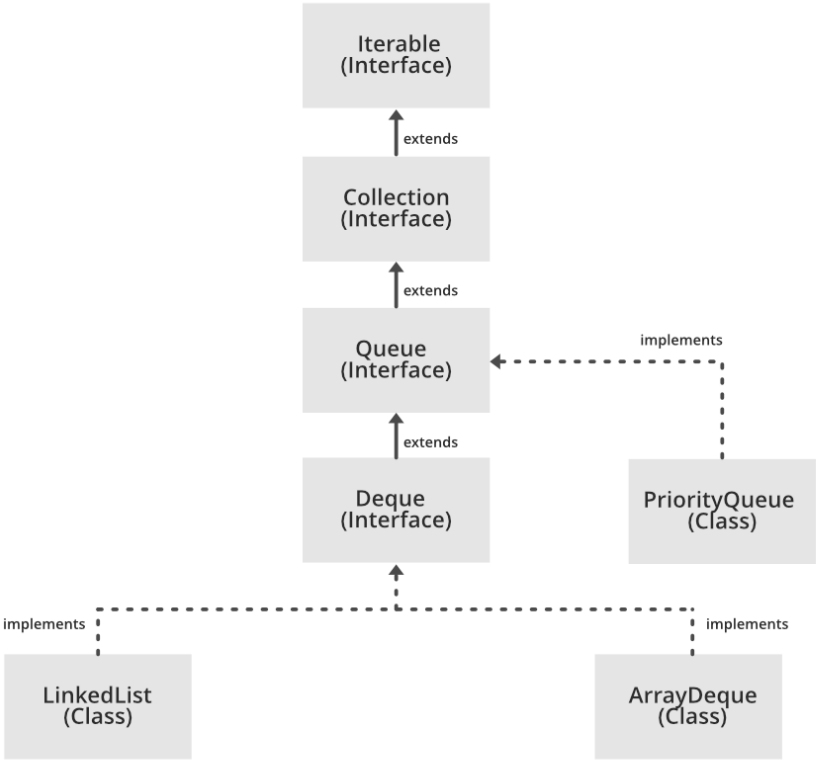
METHOD	DESCRIPTION
removeElement(Object o)	Removes the first (lowest-indexed) occurrence of the argument from this vector.
removeElementAt(int index)	Deletes the component at the specified index.
removeRange(int fromIndex, int toIndex)	Removes from this list all the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
retainAll(Collection c)	Retains only the elements in this Vector that are contained in the specified Collection.
set(int index, Object o)	Replaces the element at the specified position in this Vector with the specified element.
setElementAt(Object o, int index)	Sets the component at the specified index of this vector to be the specified object.
setSize(int newSize)	Sets the size of this vector.
size()	Returns the number of components in this vector.
subList(int fromIndex, int toIndex)	Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
toArray()	Returns an array containing all of the elements in this Vector in the correct order.
toArray(Object []array)	Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.

METHOD	DESCRIPTION
toString()	Returns a string representation of this Vector, containing the String representation of each element.
trimToSize()	Trims the capacity of this vector to be the vector's current size.

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.

The Queue interface is present in java.util package and extends the Collection interface is used to hold the elements about to be processed in FIFO(First In First Out) order. It is an ordered list of objects with its use limited to inserting elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the **FIFO** or the First-In-First-Out principle.



Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java. Note that neither of these implementations is thread-safe. PriorityBlockingQueue is one alternative implementation if the thread-safe implementation is needed.

Declaration: The Queue interface is declared as:

```
public interface Queue extends Collection
```

Creating Queue Objects: Since *Queue* is an interface, objects cannot be created of the type queue. We always need a class which extends this list in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Queue. This type-safe queue can be defined as:

```
// Obj is the type of the object to be stored in Queue
Queue<Obj> queue = new PriorityQueue<Obj> ();
```

In Java, the Queue interface is a subtype of the Collection interface and represents a collection of elements in a specific order. It follows the first-in, first-out (FIFO) principle, which means that the elements are retrieved in the order in which they were added to the queue.

The Queue interface provides several methods for adding, removing, and inspecting elements in the queue. Here are some of the most commonly used methods:

add(element): Adds an element to the rear of the queue. If the queue is full, it throws an exception.

offer(element): Adds an element to the rear of the queue. If the queue is full, it returns false.

remove(): Removes and returns the element at the front of the queue. If the queue is empty, it throws an exception.

poll(): Removes and returns the element at the front of the queue. If the queue is empty, it returns null.

element(): Returns the element at the front of the queue without removing it. If the queue is empty, it throws an exception.

peek(): Returns the element at the front of the queue without removing it. If the queue is empty, it returns null.

The Queue interface is implemented by several classes in Java, including LinkedList, ArrayDeque, and PriorityQueue. Each of these classes provides different implementations of the queue interface, with different performance characteristics and features.

Overall, the Queue interface is a useful tool for managing collections of elements in a specific order, and is widely used in many different applications and industries.

Example:

- Java

```
import java.util.LinkedList;

import java.util.Queue;
```

```
public class QueueExample {

    public static void main(String[] args) {

        Queue<String> queue = new LinkedList<>();



        // add elements to the queue

        queue.add("apple");

        queue.add("banana");

        queue.add("cherry");



        // print the queue

        System.out.println("Queue: " + queue);



        // remove the element at the front of the queue

        String front = queue.remove();

        System.out.println("Removed element: " + front);



        // print the updated queue

        System.out.println("Queue after removal: " + queue);



        // add another element to the queue
```

```
        queue.add("date");



        // peek at the element at the front of the queue

        String peeked = queue.peek();

        System.out.println("Peeked element: " + peeked);



        // print the updated queue

        System.out.println("Queue after peek: " + queue);

    }

}
```

Output

```
Queue: [apple, banana, cherry]
Removed element: apple
Queue after removal: [banana, cherry]
Peeked element: banana
Queue after peek: [banana, cherry, date]
```

Queue interface can be instantiated as:

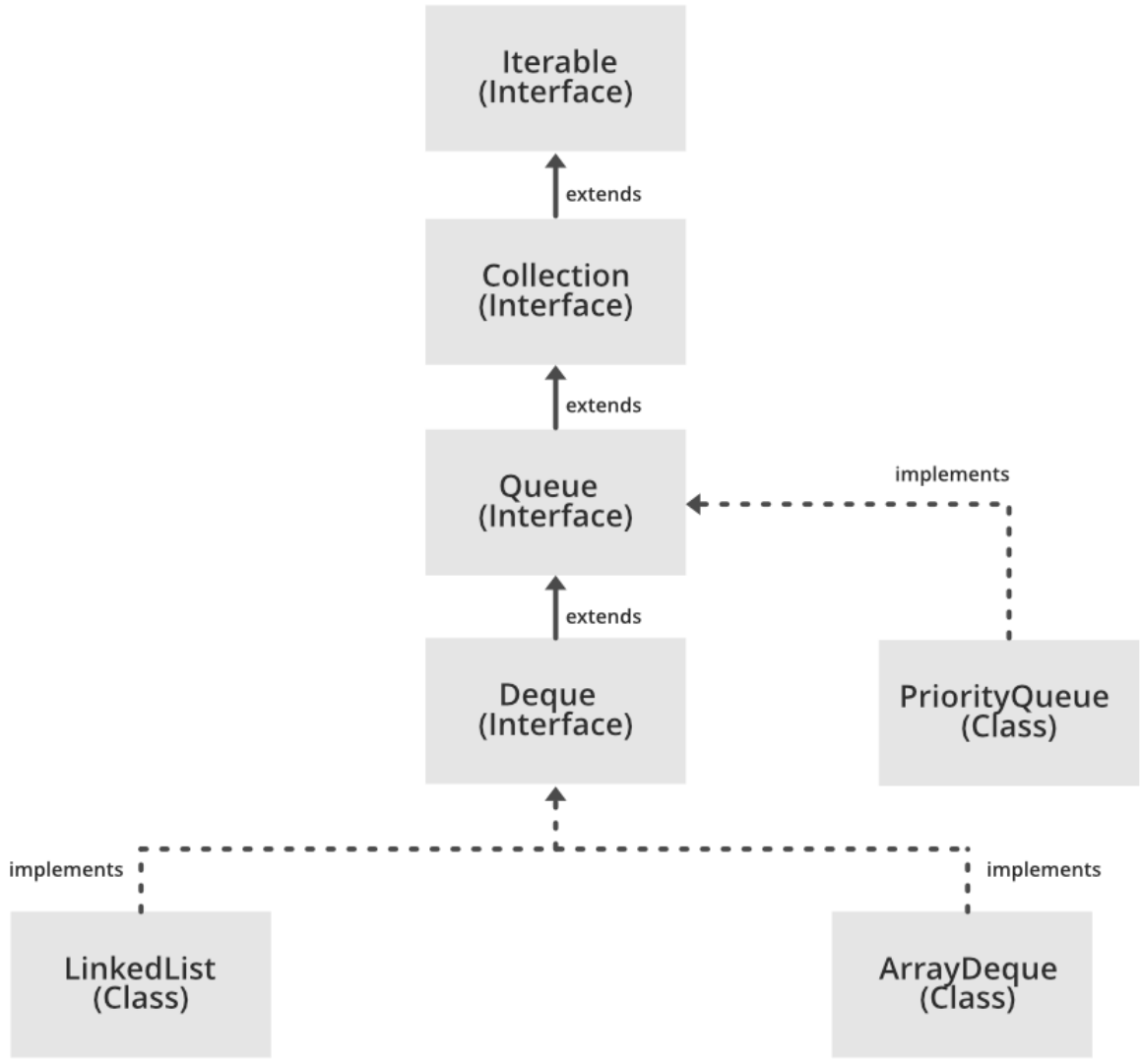
1. Queue<String> q1 = **new** PriorityQueue();
2. Queue<String> q2 = **new** ArrayDeque();

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

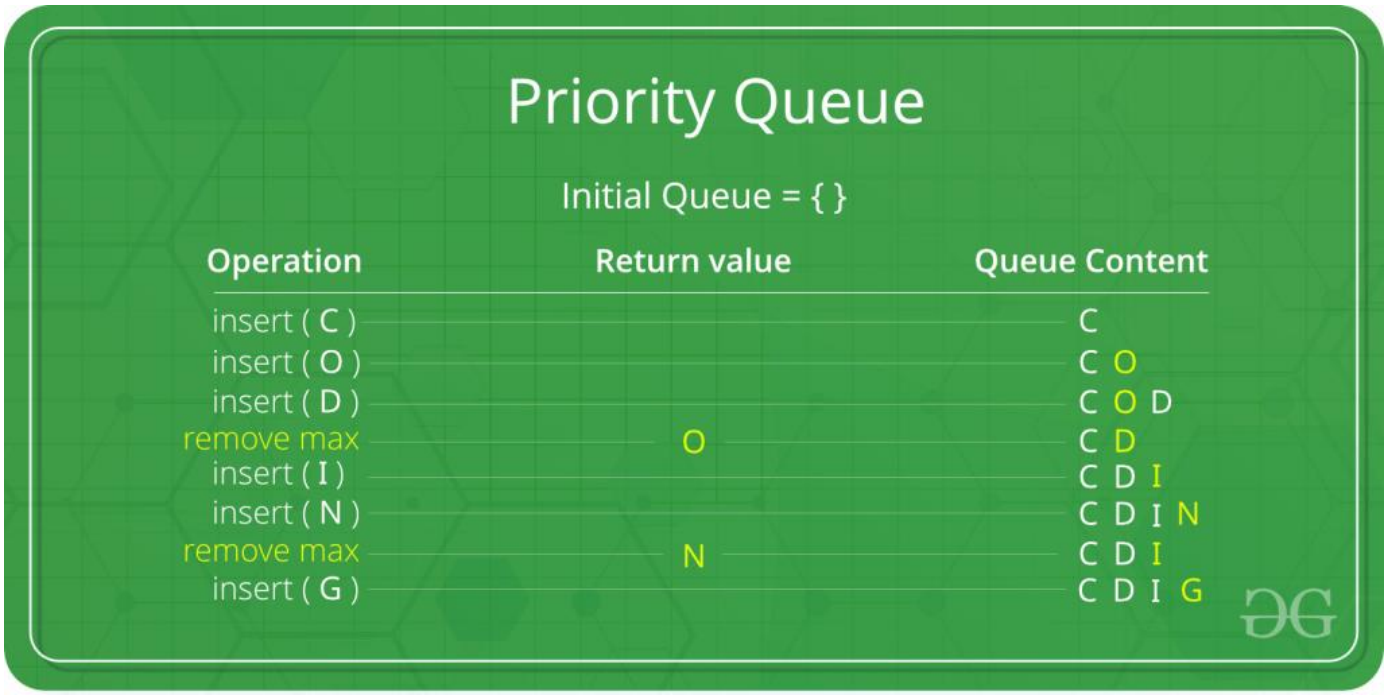
PriorityQueue

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.

A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a Queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority, that's when the PriorityQueue comes into play.
The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.



In the below priority queue, an element with a maximum ASCII value will have the highest priority.



Declaration:

```
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable


where E is the type of elements held in this queue
```

The class implements **Serializable**, **Iterable<E>**, **Collection<E>**, Queue<E> interfaces.
A few **important points on Priority Queue** are as follows:

- PriorityQueue doesn't permit null.
- We can't create a PriorityQueue of Objects that are non-comparable
- PriorityQueue are unbound queues.
- The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for the least value, the head is one of those elements — ties are broken arbitrarily.
- Since PriorityQueue is not thread-safe, java provides PriorityBlockingQueue class that implements the BlockingQueue interface to use in a java multithreading environment.
- The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.
- It provides O(log(n)) time for add and poll methods.
- It inherits methods from **AbstractQueue**, **AbstractCollection**, **Collection,** and **Object** class.

Constructors:

1. PriorityQueue(): Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
PriorityQueue<E> pq = new PriorityQueue<E>();

2. PriorityQueue(Collection<E> c): Creates a PriorityQueue containing the elements in the specified collection.
PriorityQueue<E> pq = new PriorityQueue<E>(Collection<E> c);

3. PriorityQueue(int initialCapacity): Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.

PriorityQueue<E> pq = new PriorityQueue<E>(int initialCapacity);

4. PriorityQueue(int initialCapacity, Comparator<E> comparator): Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
PriorityQueue<E> pq = new PriorityQueue(int initialCapacity, Comparator<E> comparator);

5. PriorityQueue(PriorityQueue<E> c): Creates a PriorityQueue containing the elements in the specified priority queue.
PriorityQueue<E> pq = new PriorityQueue(PriorityQueue<E> c);

6. PriorityQueue(SortedSet<E> c): Creates a PriorityQueue containing the elements in the specified sorted set.
PriorityQueue<E> pq = new PriorityQueue<E>(SortedSet<E> c);

7. PriorityQueue(Comparator<E> comparator): Creates a PriorityQueue with the default initial capacity and whose elements are ordered according to the specified comparator.
PriorityQueue<E> pq = new PriorityQueue<E>(Comparator<E> c);

Example:
The example below explains the following basic operations of the priority queue.

- boolean add(E element): This method inserts the specified element into this priority queue.
- public peek(): This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
- public poll(): This method retrieves and removes the head of this queue, or returns null if this queue is empty.

- Java

```
// Java program to demonstrate the

// working of PriorityQueue

import java.util.*;



class PriorityQueueDemo {



      // Main Method

    public static void main(String args[])
```

```
    {

        // Creating empty priority queue

        PriorityQueue<Integer> pQueue = new PriorityQueue<Integer>();



        // Adding items to the pQueue using add()

        pQueue.add(10);

        pQueue.add(20);

        pQueue.add(15);



        // Printing the top element of PriorityQueue

        System.out.println(pQueue.peek());



        // Printing the top element and removing it

        // from the PriorityQueue container

        System.out.println(pQueue.poll());



        // Printing the top element again

        System.out.println(pQueue.peek());

    }

}
```

Output

```
10
```

```
10
15
```

Operations on PriorityQueue

Let's see how to perform a few frequently used operations on the Priority Queue class.

1. Adding Elements: In order to add an element in a priority queue, we can use the add() method. The insertion order is not retained in the PriorityQueue. The elements are stored based on the priority order which is ascending by default.

- Java

```
/*package whatever //do not write package name here */



import java.util.*;

import java.io.*;



public class PriorityQueueDemo {



    public static void main(String args[])

    {

        PriorityQueue<Integer> pq = new PriorityQueue<>();

        for(int i=0;i<3;i++){

            pq.add(i);

            pq.add(1);

        }

        System.out.println(pq);

    }
```

```
}
```

Output

```
[0, 1, 1, 1, 2, 1]
```

We will not get sorted elements by printing PriorityQueue.

- Java

```
/*package whatever //do not write package name here */



import java.util.*;

import java.io.*;



public class PriorityQueueDemo {



    public static void main(String args[])

    {

        PriorityQueue<String> pq = new PriorityQueue<>();



        pq.add("Geeks");

        pq.add("For");

        pq.add("Geeks");



        System.out.println(pq);
```

```
    }

}
```

Output

```
[For, Geeks, Geeks]
```

2. Removing Elements: In order to remove an element from a priority queue, we can use the remove() method. If there are multiple such objects, then the first occurrence of the object is removed. Apart from that, the poll() method is also used to remove the head and return it.

- Java

```
// Java program to remove elements

// from a PriorityQueue



import java.util.*;

import java.io.*;



public class PriorityQueueDemo {



    public static void main(String args[])

    {

        PriorityQueue<String> pq = new PriorityQueue<>();



        pq.add("Geeks");

        pq.add("For");
```

```
        pq.add("Geeks");



        System.out.println("Initial PriorityQueue " + pq);



          // using the method

        pq.remove("Geeks");



        System.out.println("After Remove - " + pq);



        System.out.println("Poll Method - " + pq.poll());



        System.out.println("Final PriorityQueue - " + pq);

    }

}
```

Output

```
Initial PriorityQueue [For, Geeks, Geeks]
After Remove - [For, Geeks]
Poll Method - For
Final PriorityQueue - [Geeks]
```

3. Accessing the elements: Since Queue follows the First In First Out principle, we can access only the head of the queue. To access elements from a priority queue, we can use the peek() method.

- Java

```
// Java program to access elements
// from a PriorityQueue
import java.util.*;

class PriorityQueueDemo {

      // Main Method
    public static void main(String[] args)
    {

        // Creating a priority queue
        PriorityQueue<String> pq = new PriorityQueue<>();
        pq.add("Geeks");
        pq.add("For");
        pq.add("Geeks");
        System.out.println("PriorityQueue: " + pq);

        // Using the peek() method
        String element = pq.peek();
        System.out.println("Accessed Element: " + element);
    }
```

```
}
```

Output

```
PriorityQueue: [For, Geeks, Geeks]
Accessed Element: For
```

4. Iterating the PriorityQueue: There are multiple ways to iterate through the PriorityQueue. The most famous way is converting the queue to the array and traversing using the for loop. However, the queue also has an inbuilt iterator which can be used to iterate through the queue.

- Java

```
// Java program to iterate elements
// to a PriorityQueue

import java.util.*;

public class PriorityQueueDemo {

      // Main Method
    public static void main(String args[])
    {
        PriorityQueue<String> pq = new PriorityQueue<>();

        pq.add("Geeks");
        pq.add("For");
```

```
        pq.add("Geeks");



        Iterator iterator = pq.iterator();



        while (iterator.hasNext()) {

            System.out.print(iterator.next() + " ");

        }

    }

}
```

Output

```
For Geeks Geeks
```

Example:

- Java

```
import java.util.PriorityQueue;



public class PriorityQueueExample {



    public static void main(String[] args) {



        // Create a priority queue with initial capacity 10
```

```
        PriorityQueue<Integer> pq = new PriorityQueue<>(10);



        // Add elements to the queue

        pq.add(3);

        pq.add(1);

        pq.add(2);

        pq.add(5);

        pq.add(4);



        // Print the queue

        System.out.println("Priority queue: " + pq);



        // Peek at the top element of the queue

        System.out.println("Peek: " + pq.peek());



        // Remove the top element of the queue

        pq.poll();



        // Print the queue again

        System.out.println("Priority queue after removing top element: "
+ pq);
```

```
        // Check if the queue contains a specific element

        System.out.println("Does the queue contain 3? " +
pq.contains(3));




        // Get the size of the queue

        System.out.println("Size of queue: " + pq.size());




        // Remove all elements from the queue

        pq.clear();




        // Check if the queue is empty

        System.out.println("Is the queue empty? " + pq.isEmpty());

    }

}
```

Output

```
Priority queue: [1, 3, 2, 5, 4]
Peek: 1
Priority queue after removing top element: [2, 3, 4, 5]
Does the queue contain 3? true
Size of queue: 4
Is the queue empty? true
```

Real Time Examples:

Priority Queue is a data structure in which elements are ordered by priority, with the highest-priority elements appearing at the front of the queue. Here are some real-world examples of where priority queues can be used:

- **Task Scheduling**: In operating systems, priority queues are used to schedule tasks based on their priority levels. For example, a high-priority task like a critical system update may be scheduled ahead of a lower-priority task like a background backup process.
- **Emergency Room:** In a hospital emergency room, patients are triaged based on the severity of their condition, with those in critical condition being treated first. A priority queue can be used to manage the order in which patients are seen by doctors and nurses.
- **Network Routing**: In computer networks, priority queues are used to manage the flow of data packets. High-priority packets like voice and video data may be given priority over lower-priority data like email and file transfers.
- **Transportation**: In traffic management systems, priority queues can be used to manage traffic flow. For example, emergency vehicles like ambulances may be given priority over other vehicles to ensure that they can reach their destination quickly.
- **Job Scheduling**: In job scheduling systems, priority queues can be used to manage the order in which jobs are executed. High-priority jobs like critical system updates may be scheduled ahead of lower-priority jobs like data backups.
- **Online Marketplaces**: In online marketplaces, priority queues can be used to manage the delivery of products to customers. High-priority orders like express shipping may be given priority over standard shipping orders.

Overall, priority queues are a useful data structure for managing tasks and resources based on their priority levels in various real-world scenarios.

Methods in PriorityQueue class

METHOD	DESCRIPTION
add(E e)	Inserts the specified element into this priority queue.
clear()	Removes all of the elements from this priority queue.
comparator()	Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements.
contains?(Object o)	Returns true if this queue contains the specified element.
forEach?(Consumer<? super E> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.

METHOD	DESCRIPTION
iterator()	Returns an iterator over the elements in this queue.
offer?(E e)	Inserts the specified element into this priority queue.
remove?(Object o)	Removes a single instance of the specified element from this queue, if it is present.
removeAll?(Collection<?> c)	Removes all of this collection's elements that are also contained in the specified collection (optional operation).
removeIf?(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.
retainAll?(Collection<?> c)	Retains only the elements in this collection that are contained in the specified collection (optional operation).
spliterator()	Creates a late-binding and fail-fast Spliterator over the elements in this queue.
toArray()	Returns an array containing all of the elements in this queue.
toArray?(T[] a)	Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array.

Methods Declared in class java.util.AbstractQueue

METHOD	DESCRIPTION
addAll(Collection<? extends E> c)	Adds all of the elements in the specified collection to this queue.

METHOD	DESCRIPTION
element()	Retrieves, but does not remove, the head of this queue.
remove()	Retrieves and removes the head of this queue.

Methods Declared in class java.util.AbstractCollection

METHOD	DESCRIPTION
containsAll(Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.
isEmpty()	Returns true if this collection contains no elements.
toString()	Returns a string representation of this collection.

Methods Declared in interface java.util.Collection

METHOD	DESCRIPTION
containsAll(Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.
equals(Object o)	Compares the specified object with this collection for equality.
hashCode()	Returns the hash code value for this collection.
isEmpty()	Returns true if this collection contains no elements.
parallelStream()	Returns a possibly parallel Stream with this collection as its source.

METHOD	DESCRIPTION
size()	Returns the number of elements in this collection.
stream()	Returns a sequential Stream with this collection as its source.
toArray(IntFunction<T[]> generator)	Returns an array containing all of the elements in this collection, using the provided generator function to allocate the returned array.

Methods Declared in interface java.util.Queue

METHOD	DESCRIPTION
peek()	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
poll()	Retrieves and removes the head of this queue, or returns null if this queue is empty.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection5{
3. public static void main(String args[]){
4. PriorityQueue<String> queue=new PriorityQueue<String>();
5. queue.add("Amit Sharma");
6. queue.add("Vijay Raj");
7. queue.add("JaiShankar");
8. queue.add("Raj");
9. System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. while(itr.hasNext()){
14. System.out.println(itr.next());
15. }
```

```
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
20. while(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }
```

Output:

A

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

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:

1. Deque d = **new** ArrayDeque();

Deque interface present in java.util package is a subtype of the queue interface. The Deque is related to the double-ended queue that supports the addition or removal of elements from either end of the data structure. It can either be used as a queue(first-in-first-out/FIFO) or as a stack(last-in-first-out/LIFO). Deque is the acronym for double-ended queue.

The Deque (double-ended queue) interface in Java is a subinterface of the Queue interface and extends it to provide a double-ended queue, which is a queue that allows elements to be added and removed from both ends. The Deque interface is part of the Java Collections Framework and is used to provide a generic and flexible data structure that can be used to implement a variety of algorithms and data structures.

Here’s an example of how you might use a Deque in Java:

- Java

```
import java.util.ArrayDeque;

import java.util.Deque;



public class Example {

  public static void main(String[] args) {

    Deque<Integer> deque = new ArrayDeque<>();

    deque.addFirst(1);

    deque.addLast(2);

    int first = deque.removeFirst();

    int last = deque.removeLast();

    System.out.println("First: " + first + ", Last: " + last);

  }

}
```

Output

```
First: 1, Last: 2
```

Advantages of using Deque:

1. Double-Ended: The main advantage of the Deque interface is that it provides a double-ended queue, which allows elements to be added and removed from both ends of the queue. This makes it a good choice for scenarios where you need to insert or remove elements at both the front and end of the queue.
2. Flexibility: The Deque interface provides a number of methods for adding, removing, and retrieving elements from both ends of the queue, giving you a great deal of flexibility in how you use it.

3. Blocking Operations: The Deque interface provides blocking methods, such as takeFirst and takeLast, that allow you to wait for elements to become available or for space to become available in the queue. This makes it a good choice for concurrent and multithreaded applications.

Disadvantages of using Deque:

1. Performance: The performance of a Deque can be slower than other data structures, such as a linked list or an array, because it provides more functionality.
2. Implementation Dependent: The behavior of a Deque can depend on the implementation you use. For example, some implementations may provide thread-safe operations, while others may not. It's important to choose an appropriate implementation and understand its behavior before using a Deque.

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.

The ArrayDeque in Java provides a way to apply resizable-array in addition to the implementation of the Deque interface. It is also known as ***Array Double Ended Queue*** or ***Array Deck***. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue.
The ArrayDeque class in Java is an implementation of the Deque interface that uses a resizable array to store its elements. This class provides a more efficient alternative to the traditional Stack class, which was previously used for double-ended operations. The ArrayDeque class provides constant-time performance for inserting and removing elements from both ends of the queue, making it a good choice for scenarios where you need to perform many add and remove operations.

Here's an example of how you might use an ArrayDeque in Java:

- Java

```
import java.util.ArrayDeque;

import java.util.Deque;
```

```
public class Example {

  public static void main(String[] args) {

    Deque<Integer> deque = new ArrayDeque<>();

    deque.addFirst(1);

    deque.addLast(2);

    int first = deque.removeFirst();

    int last = deque.removeLast();

    System.out.println("First: " + first + ", Last: " + last);

  }

}
```

Output

```
First: 1, Last: 2
```

Advantages of using ArrayDeque:

1. Efficient: The ArrayDeque class provides constant-time performance for inserting and removing elements from both ends of the queue, making it a good choice for scenarios where you need to perform many add and remove operations.
2. Resizable: The ArrayDeque class uses a resizable array to store its elements, which means that it can grow and shrink dynamically to accommodate the number of elements in the queue.
3. Lightweight: The ArrayDeque class is a lightweight data structure that does not require additional overhead, such as linked list nodes, making it a good choice for scenarios where memory is limited.
4. Thread-safe: The ArrayDeque class is not thread-safe, but you can use the Collections.synchronizedDeque method to create a thread-safe version of the ArrayDeque class.

Disadvantages of using ArrayDeque:

1. Not synchronized: By default, the ArrayDeque class is not synchronized, which means that multiple threads can access it simultaneously, leading to potential data corruption.

2. Limited capacity: Although the ArrayDeque class uses a resizable array to store its elements, it still has a limited capacity, which means that you may need to create a new ArrayDeque when the old one reaches its maximum size.

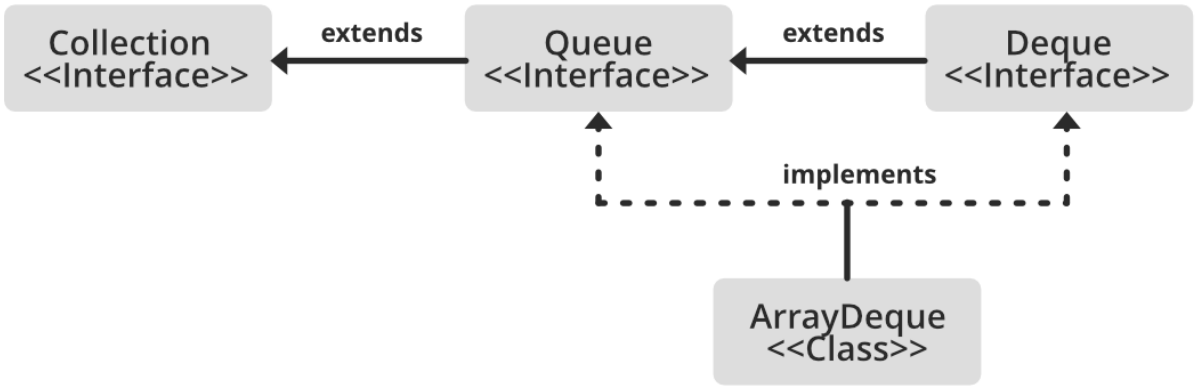
Few ***important features*** of ArrayDeque are as follows:

- Array deques have no capacity restrictions and they grow as necessary to support usage.
- They are not thread-safe which means that in the absence of external synchronization, ArrayDeque does not support concurrent access by multiple threads.
- Null elements are prohibited in the ArrayDeque.
- ArrayDeque class is likely to be faster than Stack when used as a stack.
- ArrayDeque class is likely to be faster than LinkedList when used as a queue.

Interfaces implemented by ArrayDeque:
The ArrayDeque class implements these two interfaces:

- **Queue Interface:** It is an Interface that is a FirstIn – FirstOut Data Structure where the elements are added from the back.
- **Deque Interface:** It is a Doubly Ended Queue in which you can insert the elements from both sides. It is an interface that implements the Queue.

ArrayDeque implements both Queue and Deque. It is dynamically resizable from both sides. All implemented interfaces of ArrayDeque in the hierarchy are **Serializable**, **Cloneable**, **Iterable<E>**, **Collection<E>**, Deque<E>, Queue<E>



Syntax: Declaration
public class ArrayDeque<E>

extends *AbstractCollection<E>*
implements *Deque<E>, Cloneable, Serializable*
*Here, **E** refers to the element which can refer to any class, such as Integer or String class.*
Now we are done with syntax now let us come up with constructors been defined for it prior before implementing to grasp it better and perceiving the output better.

- **ArrayDeque():** This constructor is used to create an empty ArrayDeque and by default holds an initial capacity to hold 16 elements.

```
ArrayDeque<E> dq = new ArrayDeque<E>();
```

- **ArrayDeque(Collection<? extends E> c):** This constructor is used to create an ArrayDeque containing all the elements the same as that of the specified collection.

```
ArrayDeque<E> dq = new ArrayDeque<E>(Collection col);
```

- **ArrayDeque(int numofElements):** This constructor is used to create an empty ArrayDeque and holds the capacity to contain a specified number of elements.

```
ArrayDeque<E> dq = new ArrayDeque<E>(int numofElements);
```

Methods in ArrayDeque are as follows:

Note: *Here,* ***Element*** *is the type of elements stored by ArrayDeque.*

METHOD	DESCRIPTION
add(Element e)	The method inserts a particular element at the end of the deque.
addAll(Collection<? extends E> c)	Adds all of the elements in the specified collection at the end of this deque, as if by calling addLast(E) on each one, in the order that they are returned by the collection's iterator.
addFirst(Element e)	The method inserts particular element at the start of the deque.
addLast(Element e)	The method inserts a particular element at the end of the deque. It is similar to the add() method
clear()	The method removes all deque elements.
clone()	The method copies the deque.
contains(Obj)	The method checks whether a deque contains the element or not
element()	The method returns element at the head of the deque
forEach(Consumer<? super E> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.

METHOD	DESCRIPTION
getFirst()	The method returns first element of the deque
getLast()	The method returns last element of the deque
isEmpty()	The method checks whether the deque is empty or not.
iterator()	Returns an iterator over the elements in this deque.
offer(Element e)	The method inserts element at the end of deque.
offerFirst(Element e)	The method inserts element at the front of deque.
offerLast(Element e)	The method inserts element at the end of the deque.
peek()	The method returns head element without removing it.
poll()	The method returns head element and also removes it
pop()	The method pops out an element for stack represented by deque
push(Element e)	The method pushes an element onto stack represented by deque
remove()	The method returns head element and also removes it
remove(Object o)	Removes a single instance of the specified element from this deque.
removeAll(Collection<?> c)	Removes all of this collection’s elements that are also contained in the

METHOD	DESCRIPTION
	specified collection (optional operation).
removeFirst()	The method returns the first element and also removes it
removeFirstOccurrence (Object o)	Removes the first occurrence of the specified element in this deque (when traversing the deque from head to tail).
removeIf(Predicate<? super Element> filter)	Removes all of the elements of this collection that satisfy the given predicate.
removeLast()	The method returns the last element and also removes it
removeLastOccurrence(Object o)	Removes the last occurrence of the specified

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection6{
3. public static void main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = new ArrayDeque<String>();
6. deque.add("Gautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. //Traversing elements
10. for (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }
```

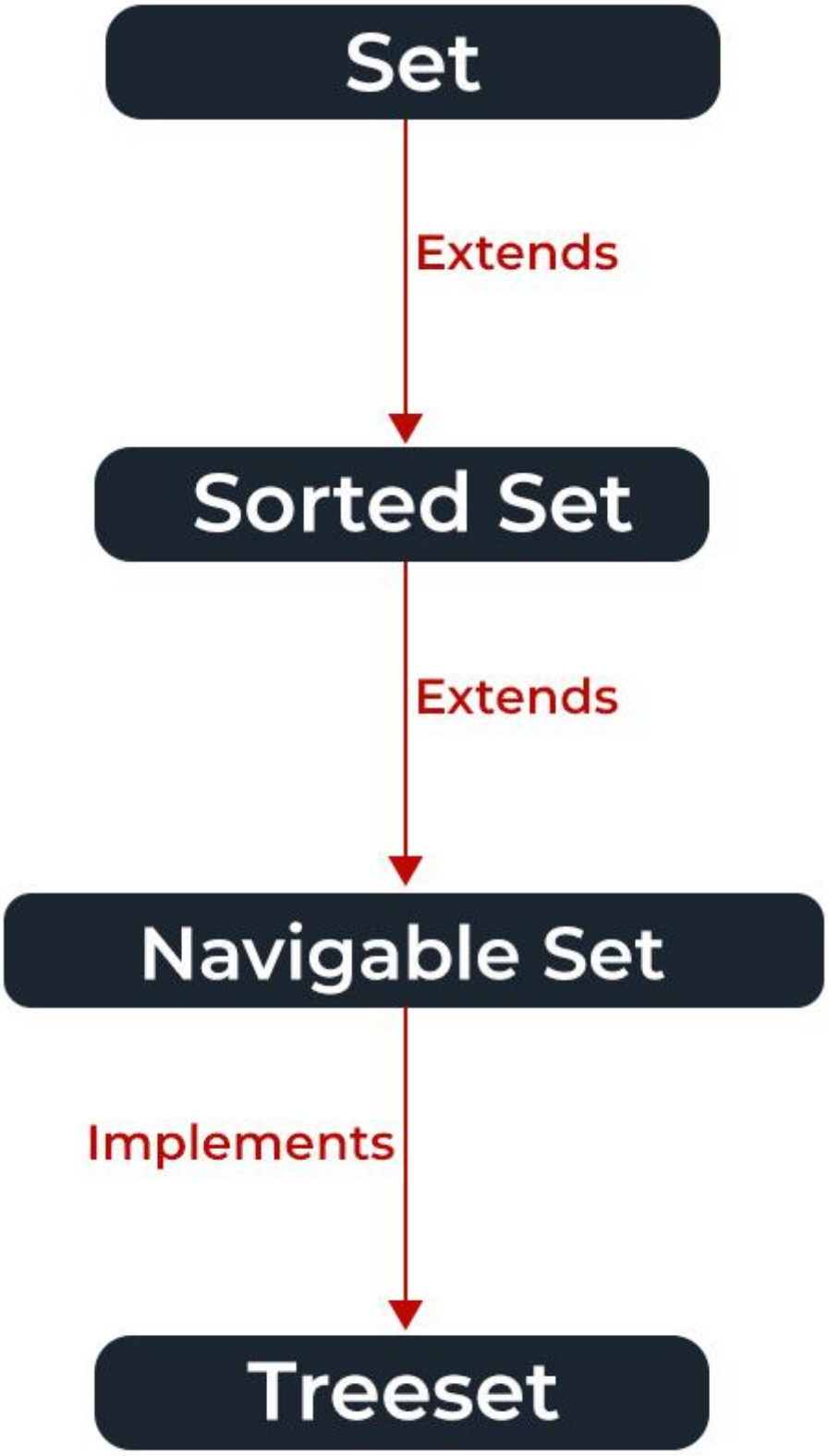
Output:

```
Gautam
Karan
Ajay
```

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.

The set interface is present in java.util package and extends the Collection interface. It is an unordered collection of objects in which duplicate values cannot be stored. It is an interface that implements the mathematical set. This interface contains the methods inherited from the Collection interface and adds a feature that restricts the insertion of the duplicate elements. There are two interfaces that extend the set implementation namely SortedSet and NavigableSet.


Set
Extends
Sorted Set
Extends
Navigable Set
Implements
Treeset

In the above image, the navigable set extends the sorted set interface. Since a set doesn't retain the insertion order, the navigable set interface provides the implementation to navigate through the Set. The class which implements the navigable set is a TreeSet which is an implementation of a self-balancing tree. Therefore, this interface provides us with a way to navigate through this tree.

Declaration: The Set interface is declared as:

```
public interface Set extends Collection
```

Creating Set Objects

Since Set is an interface, objects cannot be created of the typeset. We always need a class that extends this list in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Set. This type-safe set can be defined as:

```
// Obj is the type of the object to be stored in Set

Set<Obj> set = new HashSet<Obj> ();
```

Let us discuss methods present in the Set interface provided below in a tabular format below as follows:

Method	Description
add(element)	This method is used to add a specific element to the set. The function adds the element only if the specified element is not already present in the set else the function returns False if the element is already present in the Set.
addAll(collection)	This method is used to append all of the elements from the mentioned collection to the existing set. The elements are added randomly without following any specific order.
clear()	This method is used to remove all the elements from the set but not delete the set. The reference for the set still exists.
contains(element)	This method is used to check whether a specific element is present in the Set or not.
containsAll(collection)	This method is used to check whether the set contains all the elements present in the given collection or not. This method returns true if the set contains all the elements and returns false if any of the elements are missing.

Method	Description
hashCode()	This method is used to get the hashCode value for this instance of the Set. It returns an integer value which is the hashCode value for this instance of the Set.
isEmpty()	This method is used to check whether the set is empty or not.
iterator()	This method is used to return the iterator of the set. The elements from the set are returned in a random order.
remove(element)	This method is used to remove the given element from the set. This method returns True if the specified element is present in the Set otherwise it returns False.
removeAll(collection)	This method is used to remove all the elements from the collection which are present in the set. This method returns true if this set changed as a result of the call.
retainAll(collection)	This method is used to retain all the elements from the set which are mentioned in the given collection. This method returns true if this set changed as a result of the call.
size()	This method is used to get the size of the set. This returns an integer value which signifies the number of elements.
toArray()	This method is used to form an array of the same elements as that of the Set.

Illustration: Sample Program to Illustrate Set interface

- Java

```
// Java program Illustrating Set Interface
```

```
// Importing utility classes
import java.util.*;

// Main class
public class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Demonstrating Set using HashSet
        // Declaring object of type String
        Set<String> hash_Set = new HashSet<String>();

        // Adding elements to the Set
        // using add() method
        hash_Set.add("Geeks");
        hash_Set.add("For");
        hash_Set.add("Geeks");
        hash_Set.add("Example");
        hash_Set.add("Set");
```

```
        // Printing elements of HashSet object
        System.out.println(hash_Set);
    }
}
```

Output

```
[Set, Example, Geeks, For]
```

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. 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.

Java HashSet class implements the Set interface, backed by a hash table which is actually a HashMap instance. No guarantee is made as to the iteration order of the hash sets which means that the class does not guarantee the constant order of elements over time. This class permits the null element. The class also offers constant time performance for the basic operations like add, remove, contains, and size assuming the hash function disperses the elements properly among the buckets, which we shall see further in the article.

Java HashSet Features

A few important features of HashSet are mentioned below:

- Implements Set Interface.
- The underlying data structure for HashSet is Hashtable.
- As it implements the Set Interface, duplicate values are not allowed.
- Objects that you insert in HashSet are not guaranteed to be inserted in the same order. Objects are inserted based on their hash code.
- NULL elements are allowed in HashSet.
- HashSet also implements **Serializable** and **Cloneable** interfaces.

Declaration of HashSet

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

where **E** is the type of elements stored in a HashSet.

```
import java.util.*;



// Main class

// HashSetDemo

class GFG {



    // Main driver method

    public static void main(String[] args)

    {



        // Creating an empty HashSet

        HashSet<String> h = new HashSet<String>();



        // Adding elements into HashSet

        // using add() method

        h.add("India");

        h.add("Australia");

        h.add("South Africa");
```

```
// Adding duplicate elements
h.add("India");

// Displaying the HashSet
System.out.println(h);
System.out.println("List contains India or not:"
                   + h.contains("India"));

// Removing items from HashSet
// using remove() method
h.remove("Australia");
System.out.println("List after removing Australia:"
                   + h);

// Display message
System.out.println("Iterating over list:");

// Iterating over hashSet items
Iterator<String> i = h.iterator();
```

```
        // Holds true till there is single element remaining

        while (i.hasNext())



            // Iterating over elements

            // using next() method

            System.out.println(i.next());

    }

}
```

Output:

```
[South Africa, Australia, India]
List contains India or not:true
List after removing Australia:[South Africa, India]
Iterating over list:
South Africa
India
```

Methods in HashSet

METHOD	DESCRIPTION
add(E e)	Used to add the specified element if it is not present, if it is present then return false.
clear()	Used to remove all the elements from the set.
contains(Object o)	Used to return true if an element is present in a set.
remove(Object o)	Used to remove the element if it is present in set.

METHOD	DESCRIPTION
iterator()	Used to return an iterator over the element in the set.
isEmpty()	Used to check whether the set is empty or not. Returns true for empty and false for a non-empty condition for set.
size()	Used to return the size of the set.
clone()	Used to create a shallow copy of the set.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection7{
3. public static void main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=new HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Vijay
Ravi
Ajay
```

LinkedHashSet

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.

The **LinkedHashSet** is an ordered version of HashSet that maintains a doubly-linked List across all elements. When the iteration order is needed to be maintained this class is used. When iterating through a HashSet the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted. When cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection8{
3. public static void main(String args[]){
4. LinkedHashSet<String> set=new LinkedHashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

```
Ravi
Vijay
Ajay
```

Performing Various Operations on the LinkedHashSet Class

Let's see how to perform a few frequently used operations on the LinkedHashSet.

Operation 1: Adding Elements

In order to add an element to the LinkedHashSet, we can use the add() method. This is different from HashSet because in HashSet, the insertion order is not retained but is retained in the LinkedHashSet.

Example:

- Java

```
// Java Program to Add Elements to LinkedHashSet



// Importing required classes

import java.io.*;

import java.util.*;



// Main class

// AddingElementsToLinkedHashSet

class GFG {



    // Main driver method

    public static void main(String[] args)

    {



        // Creating an empty LinkedHashSet

        LinkedHashSet<String> hs = new LinkedHashSet<String>();



        // Adding elements to above Set
```

```
        // using add() method



        // Note: Insertion order is maintained

        hs.add("Geek");

        hs.add("For");

        hs.add("Geeks");



        // Printing elements of Set

        System.out.println("LinkedHashSet : " + hs);

    }

}
```

Output:

```
LinkedHashSet : [Geek, For, Geeks]
```

Operation 2: Removing Elements
The values can be removed from the LinkedHashSet using the remove() method.
Example:

- Java

```
// Java program to Remove Elements from LinkedHashSet



// Importing required classes

import java.io.*;

import java.util.*;
```

```
// Main class
// RemoveElementsFromLinkedHashSet
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating an empty LinekdhashSet of string type
        LinkedHashSet<String> hs
            = new LinkedHashSet<String>();

        // Adding elements to above Set
        // using add() method
        hs.add("Geek");
        hs.add("For");
        hs.add("Geeks");
        hs.add("A");
        hs.add("B");
        hs.add("Z");
```

```
        // Printing all above elements to the console
        System.out.println("Initial HashSet " + hs);

        // Removing the element from above Set
        hs.remove("B");

        // Again removing the element
        System.out.println("After removing element " + hs);

        // Returning false if the element is not present
        System.out.println(hs.remove("AC"));
    }
}
```

Output:

```
Initial HashSet [Geek, For, Geeks, A, B, Z]
After removing element [Geek, For, Geeks, A, Z]
false
```

Operation 3: Iterating through LinkedHashSet

Iterate through the elements of LinkedHashSet using the iterator() method. The most famous one is to use the enhanced for loop.

Example:

- Java

```
// Java Program to Illustrate Iterating over LinkedHashSet



// Importing required classes

import java.io.*;

import java.util.*;



// Main class

// IteratingLinkedHashSet

class GFG {



    // Main driver method

    public static void main(String[] args)

    {



        // Instantiate an object of Set

        // Since LinkedHashSet implements Set

        // Set points to LinkedHashSet

        Set<String> hs = new LinkedHashSet<String>();



        // Adding elements to above Set

        // using add() method
```

```
        hs.add("Geek");

        hs.add("For");

        hs.add("Geeks");

        hs.add("A");

        hs.add("B");

        hs.add("Z");



        // Iterating though the LinkedHashSet

        // using iterators

        Iterator itr = hs.iterator();



        while (itr.hasNext())

            System.out.print(itr.next() + ", ");



        // New line

        System.out.println();



        // Using enhanced for loop for iteration

        for (String s : hs)

            System.out.print(s + ", ");

        System.out.println();
```

```
    }

}
```

Output:

```
Geek, For, Geeks, A, B, Z,
Geek, For, Geeks, A, B, Z,
```

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:

1. SortedSet<data-type> set = **new** TreeSet();

The SortedSet interface present in java.util package extends the Set interface present in the collection framework. It is an interface that implements the mathematical set. This interface contains the methods inherited from the Set interface and adds a feature that stores all the elements in this interface to be stored in a sorted manner.

Example of a Sorted Set:

```
// Java program to demonstrate the

// Sorted Set

import java.util.*;



class SortedSetExample{



    public static void main(String[] args)

    {
```

```
SortedSet<String> ts

    = new TreeSet<String>();



// Adding elements into the TreeSet

// using add()

ts.add("India");

ts.add("Australia");

ts.add("South Africa");



// Adding the duplicate

// element

ts.add("India");



// Displaying the TreeSet

System.out.println(ts);



// Removing items from TreeSet

// using remove()

ts.remove("Australia");

System.out.println("Set after removing "

                   + "Australia:" + ts);



// Iterating over Tree set items
```

```
        System.out.println("Iterating over set:");

        Iterator<String> i = ts.iterator();

        while (i.hasNext())

            System.out.println(i.next());

    }

}
```

Output:

```
[Australia, India, South Africa]
Set after removing Australia:[India, South Africa]
Iterating over set:
India
South Africa
```

Performing Various Operations on SortedSet

Since SortedSet is an interface, it can be used only with a class which implements this interface. TreeSet is the class which implements the SortedSet interface. Now, let's see how to perform a few frequently used operations on the TreeSet.

1. Adding Elements: In order to add an element to the SortedSet, we can use the add() method. However, the insertion order is not retained in the TreeSet. Internally, for every element, the values are compared and sorted in the ascending order. We need to keep a note that duplicate elements are not allowed and all the duplicate elements are ignored. And also, Null values are not accepted by the SortedSet.

```
// Java code to demonstrate

// the working of SortedSet

import java.util.*;



class GFG {
```

```
    public static void main(String[] args)
    {
        SortedSet<String> ts
            = new TreeSet<String>();

        // Elements are added using add() method
        ts.add("A");
        ts.add("B");
        ts.add("C");
        ts.add("A");

        System.out.println(ts);
    }
}
```

Output:

```
[A, B, C]
```

2. Accessing the Elements: After adding the elements, if we wish to access the elements, we can use inbuilt methods like contains(), first(), last(), etc.

```
// Java code to demonstrate
// the working of SortedSet
```

```
import java.util.*;

class GFG {



    public static void main(String[] args)

    {

        SortedSet<String> ts

            = new TreeSet<String>();



        // Elements are added using add() method

        ts.add("A");

        ts.add("B");

        ts.add("C");

        ts.add("A");



        System.out.println("Sorted Set is " + ts);



        String check = "D";



        // Check if the above string exists in

        // the SortedSet or not

        System.out.println("Contains " + check
```

```
                           + " " + ts.contains(check));



        // Print the first element in

        // the SortedSet

        System.out.println("First Value " + ts.first());



        // Print the last element in

        // the SortedSet

        System.out.println("Last Value " + ts.last());

    }

}
```

Output:

```
Sorted Set is [A, B, C]
Contains D false
First Value A
Last Value C
```

3. Removing the Values: The values can be removed from the SortedSet using the remove() method.

```
// Java code to demonstrate

// the working of SortedSet



import java.util.*;

class GFG{
```

```
public static void main(String[] args)
{
    SortedSet<String> ts
        = new TreeSet<String>();

    // Elements are added using add() method
    ts.add("A");
    ts.add("B");
    ts.add("C");
    ts.add("B");
    ts.add("D");
    ts.add("E");

    System.out.println("Initial TreeSet " + ts);

    // Removing the element b
    ts.remove("B");

    System.out.println("After removing element " + ts);
}
```

```
}
```

Output:

```
Initial TreeSet [A, B, C, D, E]
After removing element [A, C, D, E]
```

4. Iterating through the SortedSet: There are various ways to iterate through the SortedSet. The most famous one is to use the enhanced for loop.

```
// Java code to demonstrate
// the working of SortedSet


import java.util.*;
class GFG
 {
    public static void main(String[] args)
    {
        SortedSet<String> ts
            = new TreeSet<String>();


        // Elements are added using add() method
        ts.add("C");
        ts.add("D");
        ts.add("E");
        ts.add("A");
```

```
        ts.add("B");

        ts.add("Z");



        // Iterating though the SortedSet

        for (String value : ts)

            System.out.print(value

                             + ", ");

        System.out.println();

    }

}
```

Output:

```
A, B, C, D, E, Z,
```

TreeSet

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.

TreeSet is one of the most important implementations of the SortedSet interface in Java that uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface.
It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used. The TreeSet implements a NavigableSet interface by inheriting AbstractSet class.

Consider the following example:

```
1. import java.util.*;
2. public class TestJavaCollection9{
3. public static void main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=new TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Ajay
Ravi
Vijay
```

Implementation:
Here we will be performing various operations over the TreeSet object to get familiar with the methods and concepts of TreeSet in java. Let's see how to perform a few frequently used operations on the TreeSet. They are listed as follows:

- Adding elements
- Accessing elements
- Removing elements
- Iterating through elements

Now let us discuss each operation individually one by one later alongside grasping with the help of a clean java program.

Operation 1: Adding Elements
In order to add an element to the TreeSet, we can use the add() method. However, the insertion order is not retained in the TreeSet. Internally, for every element, the values are compared and sorted in ascending order. We need to keep a note that duplicate elements are not allowed and all the duplicate elements are ignored. And also, Null values are not accepted by the TreeSet.
Example

- Java

```
// Java code to Illustrate Addition of Elements to TreeSet



// Importing utility classes

import java.util.*;



// Main class

class GFG {



    // Main driver method

    public static void main(String[] args)

    {

        // Creating a Set interface with

        // reference to TreeSet class

        // Declaring object of string type

        Set<String> ts = new TreeSet<>();



        // Elements are added using add() method

        ts.add("Geek");

        ts.add("For");

        ts.add("Geeks");
```

```
        // Print all elements inside object

        System.out.println(ts);

    }

}
```

Output:

```
[For, Geek, Geeks]
```

Operation 2: Accessing the Elements
After adding the elements, if we wish to access the elements, we can use inbuilt methods like contains(), first(), last(), etc.
Example

- Java

```
// Java code to Illustrate Working of TreeSet by

// Accessing the Element of TreeSet



// Importing utility classes

import java.util.*;



// Main class

class GFG {



    // Main driver method

    public static void main(String[] args)
```

```
{

    // Creating a NavigableSet object  with

  // reference to TreeSet class

    NavigableSet<String> ts = new TreeSet<>();



    // Elements are added using add() method

    ts.add("Geek");

    ts.add("For");

    ts.add("Geeks");



     // Printing the elements inside the TreeSet object

    System.out.println("Tree Set is " + ts);



    String check = "Geeks";



    // Check if the above string exists in

    // the treeset or not

    System.out.println("Contains " + check + " "

                       + ts.contains(check));



    // Print the first element in
```

```
        // the TreeSet
        System.out.println("First Value " + ts.first());

        // Print the last element in
        // the TreeSet
        System.out.println("Last Value " + ts.last());

        String val = "Geek";

        // Find the values just greater
        // and smaller than the above string
        System.out.println("Higher " + ts.higher(val));
        System.out.println("Lower " + ts.lower(val));
    }
}
```

Output:

```
Tree Set is [For, Geek, Geeks]
Contains Geeks true
First Value For
Last Value Geeks
Higher Geeks
Lower For
```

Operation 3: Removing the Values

The values can be removed from the TreeSet using the remove() method. There are various other methods that are used to remove the first value or the last value.

Example

- Java

```
// Java Program to Illustrate Removal of Elements
// in a TreeSet



// Importing utility classes
import java.util.*;



// Main class
class GFG {



    // Main driver method
    public static void main(String[] args)
    {
        // Creating an object of NavigableSet
        // with reference to TreeSet class
        // Declaring object of string type
        NavigableSet<String> ts = new TreeSet<>();



        // Elements are added
        // using add() method
```

```
        ts.add("Geek");

        ts.add("For");

        ts.add("Geeks");

        ts.add("A");

        ts.add("B");

        ts.add("Z");



        // Print and display initial elements of TreeSet

        System.out.println("Initial TreeSet " + ts);



        // Removing a specific existing element inserted

        // above

        ts.remove("B");



        // Printing the updated TreeSet

        System.out.println("After removing element " + ts);



        // Now removing the first element

        // using pollFirst() method

        ts.pollFirst();
```

```
        // Again printing the updated TreeSet
        System.out.println("After removing first " + ts);


        // Removing the last element
        // using pollLast() method
        ts.pollLast();


        // Lastly printing the elements of TreeSet remaining
        // to figure out pollLast() method
        System.out.println("After removing last " + ts);
    }
}
```

Output:

```
Initial TreeSet [A, B, For, Geek, Geeks, Z]
After removing element [A, For, Geek, Geeks, Z]
After removing first [For, Geek, Geeks, Z]
After removing last [For, Geek, Geeks]
```

Operation 4: **Iterating through the TreeSet**

There are various ways to iterate through the TreeSet. The most famous one is to use the enhanced for loop. and geeks mostly you would be iterating the elements with this approach while practicing questions over TreeSet as this is most frequently used when it comes to tree, maps, and graphs problems.

Example

- Java

```
// Java Program to Illustrate Working of TreeSet



// Importing utility classes

import java.util.*;



// Main class

class GFG {



    // Main driver method

    public static void main(String[] args)

    {

        // Creating an object of Set with reference to

        // TreeSet class



        // Note: You can refer above media if geek

        // is confused in programs why we are not

        // directly creating TreeSet object

        Set<String> ts = new TreeSet<>();



        // Adding elements in above object

        // using add() method
```

```
        ts.add("Geek");

        ts.add("For");

        ts.add("Geeks");

        ts.add("A");

        ts.add("B");

        ts.add("Z");



        // Now we will be using for each loop in order

        // to iterate through the TreeSet

        for (String value : ts)



            // Printing the values inside the object

            System.out.print(value + ", ");



        System.out.println();

    }

}
```

Output:

```
A, B, For, Geek, Geeks, Z,
```

Features of a TreeSet:

1. TreeSet implements the SortedSet interface. So, duplicate values are not allowed.
2. Objects in a TreeSet are stored in a sorted and ascending order.

3. TreeSet does not preserve the insertion order of elements but elements are sorted by keys.
4. If we are depending on the default natural sorting order, the objects that are being inserted into the tree should be homogeneous and comparable. TreeSet does not allow the insertion of heterogeneous objects. It will throw a classCastException at Runtime if we try to add heterogeneous objects.
5. The TreeSet can only accept generic types which are comparable.
   For example, the StringBuffer class implements the Comparable interface

Java Map Interface

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. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

In Java, Map Interface is present in java.util package represents a mapping between a key and a value. Java Map interface is not a subtype of the Collection interface. Therefore it behaves a bit differently from the rest of the collection types. A map contains unique keys.

Creating Map Objects

Since Map is an interface, objects cannot be created of the type map. We always need a class that extends this map in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Map.

Syntax: Defining Type-safe Map

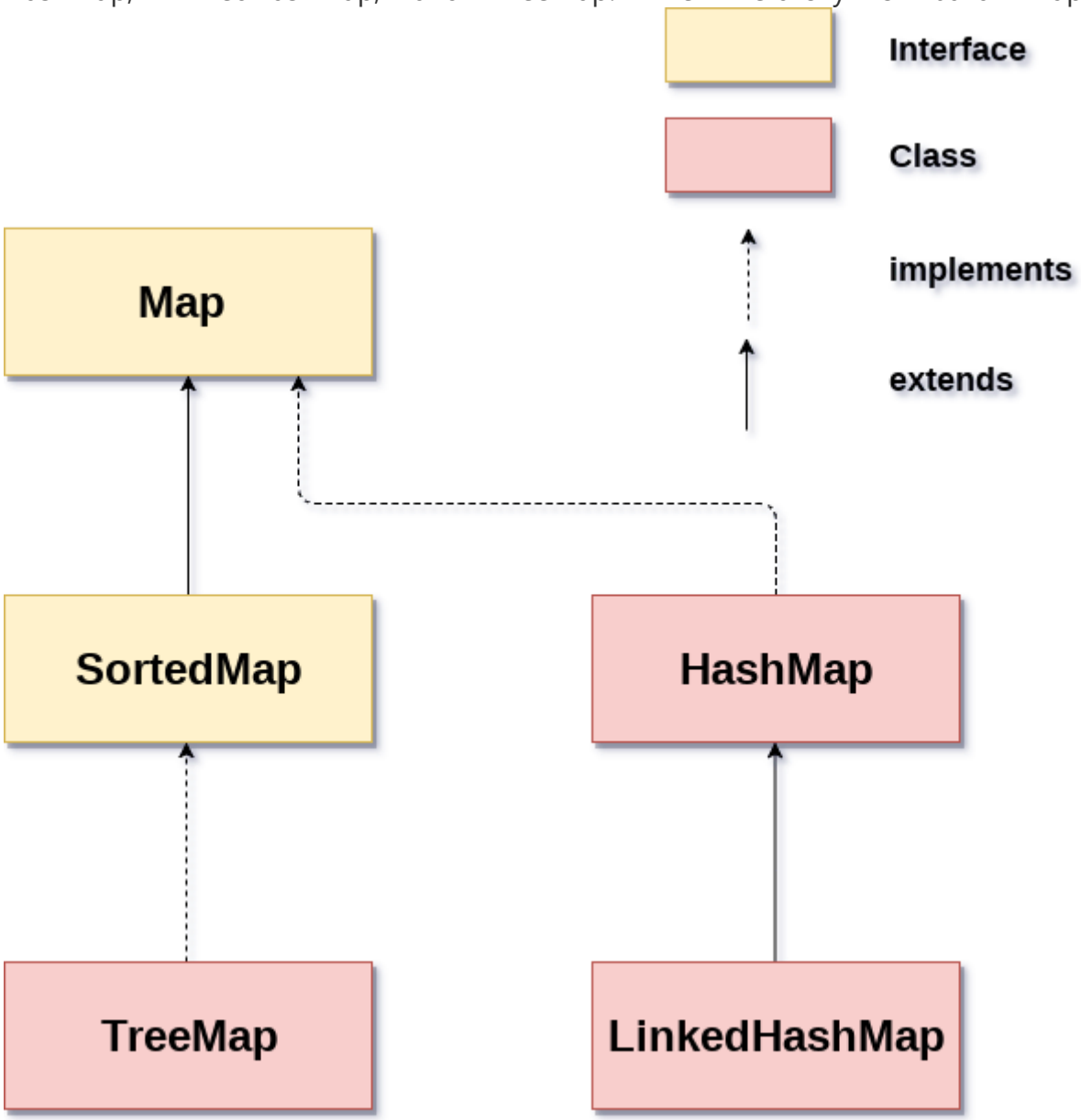
```
Map hm = new HashMap();

// Obj is the type of the object to be stored in Map
```

Characteristics of a Map Interface

1. A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null values like the HashMap and LinkedHashMap, but some do not like the TreeMap.
2. The order of a map depends on the specific implementations. For example, TreeMap and LinkedHashMap have predictable orders, while HashMap does not.
3. There are two interfaces for implementing Map in Java. They are Map and SortedMap, and three classes: HashMap, TreeMap, and LinkedHashMap.

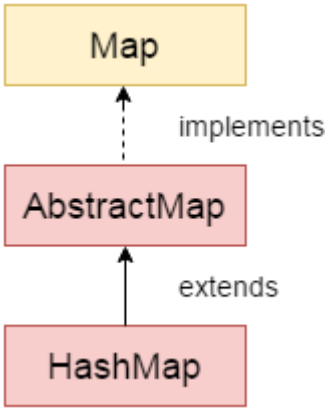
Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:



A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

Java HashMap



Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

Points to remember

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

- **In Java, HashMap** is a part of Java's collection since Java 1.2. This class is found in **java.util** package. It provides the basic implementation of the Map interface of Java. HashMap in Java stores the data in (Key, Value) pairs, and you can access them by an index of another type (e.g. an Integer). One object is used as a key (index) to another object (value). If you try to insert the duplicate key in HashMap, it will replace the element of the corresponding key.

- **What is HashMap?**
- **Java HashMap** is similar to HashTable, but it is unsynchronized. It allows to store the null keys as well, but there should be only one null key object and there can be any number of null values. This class makes no guarantees as to the order of the

map. To use this class and its methods, you need to import **java.util.HashMap** package or its superclass.

Characteristics of Java HashMap

A HashMap is a data structure that is used to store and retrieve values based on keys. Some of the key characteristics of a hashmap include:

- **Fast access time**: HashMaps provide constant time access to elements, which means that retrieval and insertion of elements are very fast, usually O(1) time complexity.
- **Uses hashing function**: HashMaps uses a hash function to map keys to indices in an array. This allows for a quick lookup of values based on keys.
- **Stores key-value pairs:** Each element in a HashMap consists of a key-value pair. The key is used to look up the associated value.
- **Supports null keys and values**: HashMaps allow for null values and keys. This means that a null key can be used to store a value, and a null value can be associated with a key.
- **Not ordered:** HashMaps are not ordered, which means that the order in which elements are added to the map is not preserved. However, LinkedHashMap is a variation of HashMap that preserves the insertion order.
- **Allows duplicates**: HashMaps allow for duplicate values, but not duplicate keys. If a duplicate key is added, the previous value associated with the key is overwritten.
- **Thread-unsafe**: HashMaps are not thread-safe, which means that if multiple threads access the same hashmap simultaneously, it can lead to data inconsistencies. If thread safety is required, ConcurrentHashMap can be used.
- **Capacity and load factor**: HashMaps have a capacity, which is the number of elements that it can hold, and a load factor, which is the measure of how full the hashmap can be before it is resized.

Example-1

```
// Java Program to Create

// HashMap in Java

import java.util.HashMap;



// Driver Class

public class ExampleHashMap {
```

```
// main function
public static void main(String[] args) {

// Create a HashMap
HashMap<String, Integer> hashMap = new HashMap<>();

// Add elements to the HashMap
hashMap.put("John", 25);
hashMap.put("Jane", 30);
hashMap.put("Jim", 35);

// Access elements in the HashMap
System.out.println(hashMap.get("John"));
// Output: 25

// Remove an element from the HashMap
hashMap.remove("Jim");

// Check if an element is present in the HashMap
System.out.println(hashMap.containsKey("Jim"));
// Output: false

// Get the size of the HashMap
```

```
        System.out.println(hashMap.size());

        // Output: 2

    }

}
```

Output

```
25
false
2
```

Example-2

```
import java.util.*;
public class Map1 {
public static void main(String[] args) {
   Map map=new HashMap();
   //Adding elements to map
   map.put(1,"Amit");
   map.put(5,"Rahul");
   map.put(2,"Jai");
   map.put(6,"Amit");
   //Traversing Map
   Set set=map.entrySet();
   Iterator itr=set.iterator();
   while(itr.hasNext()){
    Map.Entry entry=(Map.Entry)itr.next();
    System.out.println(entry.getKey()+" "+entry.getValue());
```

```
}

}

}
```

LinkedHashMap in Java

The **LinkedHashMap Class** is just like HashMap with an additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion, which the LinkedHashMap provides where the elements can be accessed in their insertion order.

Important Features of a LinkedHashMap are listed as follows:

- A LinkedHashMap contains values based on the key. It implements the Map interface and extends the HashMap class.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It is non-synchronized.
- It is the same as HashMap with an additional feature that it maintains insertion order. For example, when we run the code with a HashMap, we get a different order of elements.

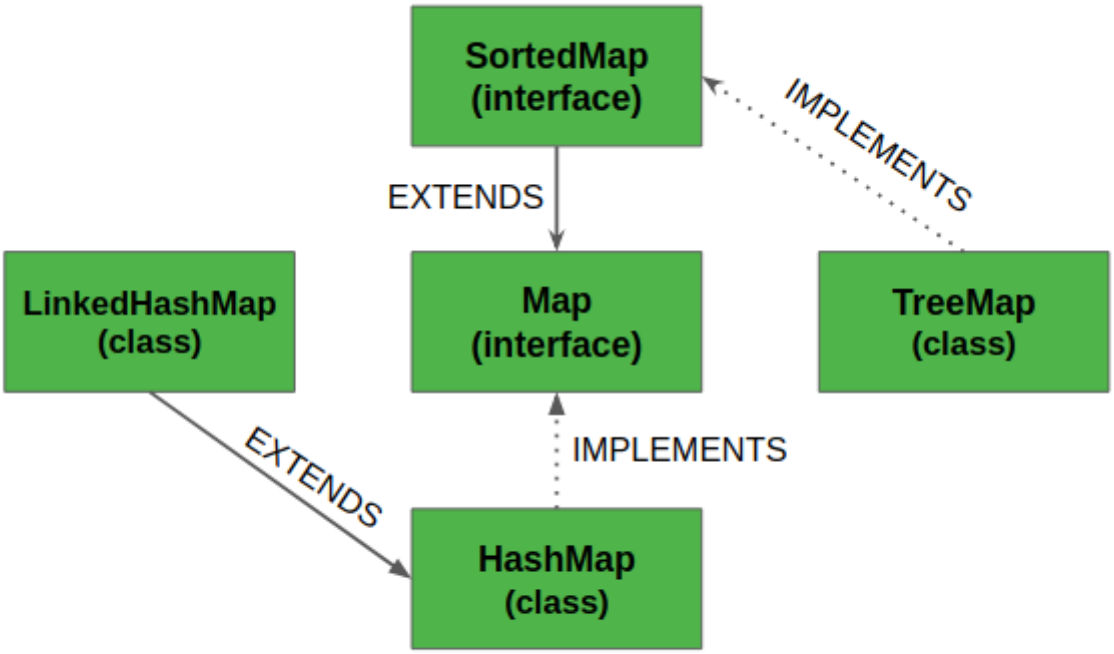
Declaration:

```
public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>
```

Here, **K** is the key Object type and **V** is the value Object type

- **K** – The type of the keys in the map.
- **V** – The type of values mapped in the map.

It implements Map<K, V> interface, and extends HashMap<K, V> class. Though the Hierarchy of LinkedHashMap is as depicted in below media as follows**:**



MAP Hierarchy in Java

How LinkedHashMap Work Internally?

A LinkedHashMap is an extension of the **HashMap** class and it implements the **Map** interface. Therefore, the class is declared as:

```
public class LinkedHashMap

extends HashMap

implements Map
```

In this class, the data is stored in the form of nodes. The implementation of the LinkedHashMap is very similar to a doubly-linked list. Therefore, each node of the LinkedHashMap is represented as:

- **Hash:** All the input keys are converted into a hash which is a shorter form of the key so that the search and insertion are faster.
- **Key:** Since this class extends HashMap, the data is stored in the form of a key-value pair. Therefore, this parameter is the key to the data.

- **Value:** For every key, there is a value associated with it. This parameter stores the value of the keys. Due to generics, this value can be of any form.
- **Next:** Since the LinkedHashMap stores the insertion order, this contains the address to the next node of the LinkedHashMap.
- **Previous:** This parameter contains the address to the previous node of the LinkedHashMap.

Example-1

```
import java.util.*;

  // Main class

// IteratingOverLinkedHashMap

class LinkedMap2 {


   // Main driver method

   public static void main(String args[])

   {

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

        hm.put(3, "Geeks");

       hm.put(2, "For");

       hm.put(1, "Geeks");


       // For-each loop for traversal over Map
```

```
        for (Map.Entry<Integer, String> mapElement:hm.entrySet()) {

            Integer key = mapElement.getKey();

            // Finding the value
            // using getValue() method
            String value = mapElement.getValue();

            // Printing the key-value pairs
            System.out.println(key + " : " + value);
        }
    }
}
```

TreeMap in Java

The TreeMap in Java is used to implement Map interface and NavigableMap along with the AbstractMap Class. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. This proves to be an efficient way of sorting and storing the key-value pairs. The storing order maintained by the treemap must be consistent with equals just like any other sorted map, irrespective of the explicit comparators. The treemap implementation is not synchronized in the sense that if a map is accessed by multiple threads, concurrently and at least one of the threads modifies the map structurally, it must be synchronized externally.
The TreeMap in Java is a concrete implementation of the java.util.SortedMap interface. It provides an ordered collection of key-value pairs, where the keys are ordered based on their natural order or a custom Comparator passed to the constructor.

Features of a TreeMap

Some important features of the treemap are as follows:

1. This class is a member of the Java Collections Framework.
2. The class implements Map interfaces including NavigableMap, SortedMap, and extends AbstractMap class.
3. TreeMap in Java does not allow null keys (like Map) and thus a NullPointerException is thrown. However, multiple null values can be associated with different keys.
4. Entry pairs returned by the methods in this class and its views represent snapshots of mappings at the time they were produced. They do not support the Entry.setValue method.

Example-1

```
import java.io.*;

import java.util.*;


class TreeMap1 {


    // Main Method

   public static void main(String args[])

   {

      // Initialization of a SortedMap

      // using Generics

      SortedMap<Integer, String> tm

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

```
        // Inserting the Elements
        tm.put(3, "Geeks");
        tm.put(2, "Geeks");
        tm.put(1, "Geeks");
        tm.put(4, "For");

        System.out.println(tm);

        tm.remove(4);

        System.out.println(tm);
    }
}
```

Example-2

```
import java.util.*;

class TreeMap2 {
```

```
  // Main Method
public static void main(String args[])
{
    // Initialization of a SortedMap
    // using Generics
    SortedMap<Integer, String> tm
        = new TreeMap<Integer, String>();

    // Inserting the Elements
    tm.put(3, "Geeks");
    tm.put(2, "For");
    tm.put(1, "Geeks");

    for (Map.Entry mapElement : tm.entrySet()) {
        int key = (int)mapElement.getKey();

        // Finding the value
        String value = (String)mapElement.getValue();
```

```
            System.out.println(key + " : " + value);
        }
    }
}
```

Hashtable in Java

The **Hashtable** class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.
The java.util.Hashtable class is a class in Java that provides a key-value data structure, similar to the Map interface. It was part of the original Java Collections framework and was introduced in Java 1.0.

However, the Hashtable class has since been considered obsolete and its use is generally discouraged. This is because it was designed prior to the introduction of the Collections framework and does not implement the Map interface, which makes it difficult to use in conjunction with other parts of the framework. In addition, the Hashtable class is synchronized, which can result in slower performance compared to other implementations of the Map interface.

In general, it's recommended to use the Map interface or one of its implementations (such as HashMap or ConcurrentHashMap) instead of the Hashtable class.

Features of Hashtable

- It is similar to HashMap, but is synchronized.
- Hashtable stores key/value pair in hash table.
- In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.
- HashMap doesn't provide any Enumeration, while Hashtable provides not fail-fast Enumeration.

Example-1

```
import java.util.Hashtable;

import java.util.Map;


public class Hashtable {

   public static void main(String[] args)

   {

        // Create an instance of Hashtable

      Hashtable<String, Integer> ht = new Hashtable<>();


        // Adding elements using put method

      ht.put("vishal", 10);

      ht.put("sachin", 30);

      ht.put("vaibhav", 20);

            // Iterating using enhanced for loop

      for (Map.Entry<String, Integer> e : ht.entrySet())

         System.out.println(e.getKey() + " "

                        + e.getValue());

   }

}
```

Sorting in Java

Whenever we do hear sorting algorithms come into play such as selection sort, bubble sort, insertion sort, radix sort, bucket sort, etc but if we look closer here we are not asked to use any kind of algorithms. It is as simple sorting with the help of linear and non-linear data structures present within java. So there is sorting done with the help of brute force in java with the help of loops and there are two in-built methods to sort in Java.

Ways of sorting in Java

1. Using loops
2. Using sort() method of Arrays class
3. Using sort method of Collections class

Way 1: Using loops

```
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Custom input array
        int arr[] = { 4, 3, 2, 1 };

        // Outer loop
        for (int i = 0; i < arr.length; i++) {

            // Inner nested loop pointing 1 index ahead
            for (int j = i + 1; j < arr.length; j++) {

                // Checking elements
                int temp = 0;
                if (arr[j] < arr[i]) {

                    // Swapping
                    temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }

            // Printing sorted array elements
            System.out.print(arr[i] + " ");
        }
    }
}
```

Output

```
1 2 3 4
```

Way 2: Using sort() method of Arrays class
Arrays.sort() works for arrays which can be of primitive data type also which in turn by default sorts in ascending order.

Example-1

```
import java.util.Arrays;

// Main class
public class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Custom input array
        int[] arr = { 13, 7, 6, 45, 21, 9, 101, 102 };

        // Calling the sort() method present
        // inside Arrays class
        Arrays.sort(arr);

        // Printing and display sorted array
        System.out.printf("Modified arr[] : %s",
                          Arrays.toString(arr));
    }
}
```

Output

```
Modified arr[] : [6, 7, 9, 13, 21, 45, 101, 102]
```

Example-2

```
import java.util.Arrays;
import java.util.Collections;

public class GFG {
    public static void main(String[] args)
    {
        // Note that we have Integer here instead of
        // int[] as Collections.reverseOrder doesn't
        // work for primitive types.
        Integer[] arr = { 13, 7, 6, 45, 21, 9, 2, 100 };

        // Sorts arr[] in descending order
        Arrays.sort(arr, Collections.reverseOrder());

        System.out.printf("Modified arr[] : %s",
                          Arrays.toString(arr));
    }
}
```

Output

```
Modified arr[] : [100, 45, 21, 13, 9, 7, 6, 2]
```

Way 3: Using sort() method of Collections class
Collections.sort() works for objects Collections like ArrayList and LinkedList.

Example

JAVA

```
// Java program to demonstrate working of Collections.sort()
import java.util.*;

public class GFG {
    public static void main(String[] args)
    {
        // Create a list of strings
        ArrayList<String> al = new ArrayList<String>();
        al.add("Geeks For Geeks");
        al.add("Friends");
        al.add("Dear");
        al.add("Is");
        al.add("Superb");

        /* Collections.sort method is sorting the
        elements of ArrayList in ascending order. */
        Collections.sort(al);

        // Let us print the sorted list
        System.out.println("List after the use of"
                           + " Collection.sort() :\n" + al);
    }
}
```

Output

```
List after the use of Collection.sort() :

[Dear, Friends, Geeks For Geeks, Is, Superb]
```

Comparable Interface in Java with Examples

The Comparable interface is used to compare an object of the same class with an instance of that class, it provides ordering of data for objects of the user-defined class. The class has to implement the **java.lang.Comparable** interface to compare its instance, it provides the compareTo method that takes a parameter of the object of that class. In this article, we will see how we can sort an array of pairs of different data types on the different parameters of comparison.

Using Comparable Interface

- In this method, we are going to implement the Comparable interface from **java.lang** Package in the Pair class.
- The Comparable interface contains the method **compareTo** to decide the order of the elements.
- Override the **compareTo** method in the Pair class.
- Create an array of Pairs and populate the array.
- Use the **Arrays.sort**() function to sort the array.

Example-1

```
import java.util.*;

import java.io.*;

class Student implements Comparable<Student>{

int rollno;

String name;

int age;

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

this.rollno=rollno;

this.name=name;

this.age=age;

}

public int compareTo(Student st){

if(age==st.age)
```

```
return 0;
else if(age>st.age)
return 1;
else
return -1;
}
}
//Creating a test class to sort the elements
public class TestSort3{
public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

Collections.sort(al);
for(Student st:al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

Comparator Interface in Java with Examples

A comparator interface is used to order the objects of user-defined classes. A comparator object is capable of comparing two objects of the same class**.** Following function compare obj1 with obj2.

Syntax:

```
public int compare(Object obj1, Object obj2):
```

Java Comparator interface is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Java Comparator interface is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Methods of Java Comparator Interface

Method	Description
public int compare(Object obj1, Object obj2)	It compares the first object with the second object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.

Example

```
import java.util.*;
import java.util.Comparator;
class Comp
{
public static void main(String args[])
{
Comparator<Integer> com=new Comparator<Integer>()
{
public int compare(Integer i,Integer j)
{
if(i%10>j%10)

return 1;
else
return -1;
}
};
```

```
List<Integer> nums=new ArrayList<>();
nums.add(3);
nums.add(31);
nums.add(36);
nums.add(76);
nums.add(56);
nums.add(34);
nums.add(67);
Collections.sort(nums,com);
System.out.println(nums);
}
}
```

there are many differences between Comparable and Comparator interfaces that are given below.

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.

4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

Properties Class in Java

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. It belongs to **java.util** package. **Properties** define the following instance variable. This variable holds a default property list associated with a **Properties** object.
Properties defaults: *This variable holds a default property list associated with a Properties object.*

The **properties** object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.

It can be used to get property value based on the property key. The Properties class provides methods to get data from the properties file and store data into the properties file. Moreover, it can be used to get the properties of a system.

Features of Properties class:

- **Properties** is a subclass of Hashtable.
- It is used to maintain a list of values in which the key is a string and the value is also a string i.e; it can be used to store and retrieve string type data from the properties file.
- Properties class can specify other properties list as it's the default. If a particular key property is not present in the original Properties list, the default properties will be searched.
- Properties object does not require external synchronization and Multiple threads can share a single Properties object.
- Also, it can be used to retrieve the properties of the system.

Advantage of a Properties file
In the event that any data is changed from the properties record, you don't have to recompile the java class. It is utilized to store data that is to be changed habitually.

Note: The Properties class does not inherit the concept of a load factor from its superclass, **Hashtable**.

Declaration

```
public class Properties extends Hashtable<Object,Object>
```

Example:-

```
import java.util.*;
import java.io.*;
public class Prop {
   public static void main(String[] args) throws Exception
   {
      // create a reader object on the properties file
      FileReader reader = new FileReader("C:/Users/vinita
sharma/Desktop/db.properties.txt");
       // create properties object
      Properties p = new Properties();
       // Add a wrapper around reader object
      p.load(reader);
       // access properties data
      System.out.println(p.getProperty("username"));
      System.out.println(p.getProperty("password"));
   }
```

```
}
```