

## Unit II

1D and 2D Collections: 1D Collection: 1D Collection Interfaces: Collection, Set, List, NavigableSet, SortedSet, Queue, Deque. 1D Collection Classes-Hash Set, Linked HashSet, TreeSet, ArrayList, LinkedList.

2D Collection: 2D Collection Interfaces-Map, NavigableMap, SortedMap, 2D Collection Classes-HashMap, LinkedHashMap, TreeMap.

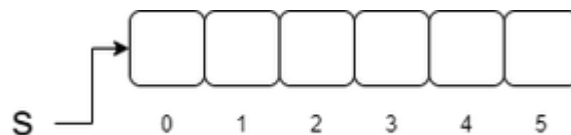
### Introduction to Collections

Collection Framework is a powerful framework in java. This framework defines the most common methods that can be used for any collection of objects. But the question arises that we have an array concept in java then why we need collection framework in java? Now let's see that why we need collection framework in java with some valid points of difference between array and collection.

**Syntax:** Declaring variables

int x = 10 or int y = 30

We declare the variables in our program as shown above which are initialized to custom random integer values. But like this how many elements we are going to declare? What if I want to declare 100 and 1000 elements in the code then the single variable declaration method is not suitable for declaration. Here array concept came into the picture. For the declaration of 1000 variables or elements, we can declare an array with some size. Array concept is very efficient and suitable for various operations.



**Syntax:**

```
Student[] s = new Student[5];
```

Array is very efficient for some operations But there are some limitations while we are using arrays like:

Arrays are fixed in size i.e. once we created an array with some size then there is no increasing or decreasing its size based on the requirements.

Arrays can hold only homogeneous data elements.

Array concept is not implemented based on some standard data structure. Hence ready-made methods are not available for the requirement.

**Illustration:**

```
Student s = new Student[1000];
```

we can declare like this : s[0] = new Student

but we cannot declare like this : s[1] = new customer

- To overcome these drawbacks or limitations of the array we need collection framework in java. Collection framework is used with various operations and having various in-build methods. They are as follows:
  - As collection framework is growable in nature some need not worry about the size.
  - Collection framework can hold both homogeneous and heterogeneous objects.
  - Collection framework is implemented based on some standard data structure. Hence, ready-made methods are available to use as per the requirement.

### **Differences Between Arrays and Collection**

| <b>Sno</b> | <b>Arrays</b>  | <b>Collection</b>  |
|------------|--|--|
| 1          | Arrays are fixed in size that is once we create an array we can not increased or decreased based on our requirement. | Collection are growable in nature that is based on our requirement. We can increase or decrease of size. |
| 2          | Write in memory Arrays are not recommended to use.   | Write in memory collection are recommended to use.   |
| 3          | With respect to performance Arrays are recommended to use.   | With respect to performance collection are not recommended to use.                                       |
| 4          | Arrays can hold only homogeneous data types elements.  | Collection can hold both homogeneous and heterogeneous elements.   |

| Sno | Arrays   | Collection  |
|-----|--|---|
| 5   | There is no underlying data structure for arrays and hence ready made method support is not available. | Every collection class is implemented based on some standard data structure, and hence, ready-made method support is available for every requirement. As performance is crucial, we can use these methods directly, and we are not responsible for implementing them. |
| 6   | Arrays can hold both object and primitive data type .  | Collection can hold only object types but not primitive datatypes such as int, long, short, etc.  |

### **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**

**Def:**A group of individual objects is called “Collection”.

### **What is a framework in Java**

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

## Collection framework

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

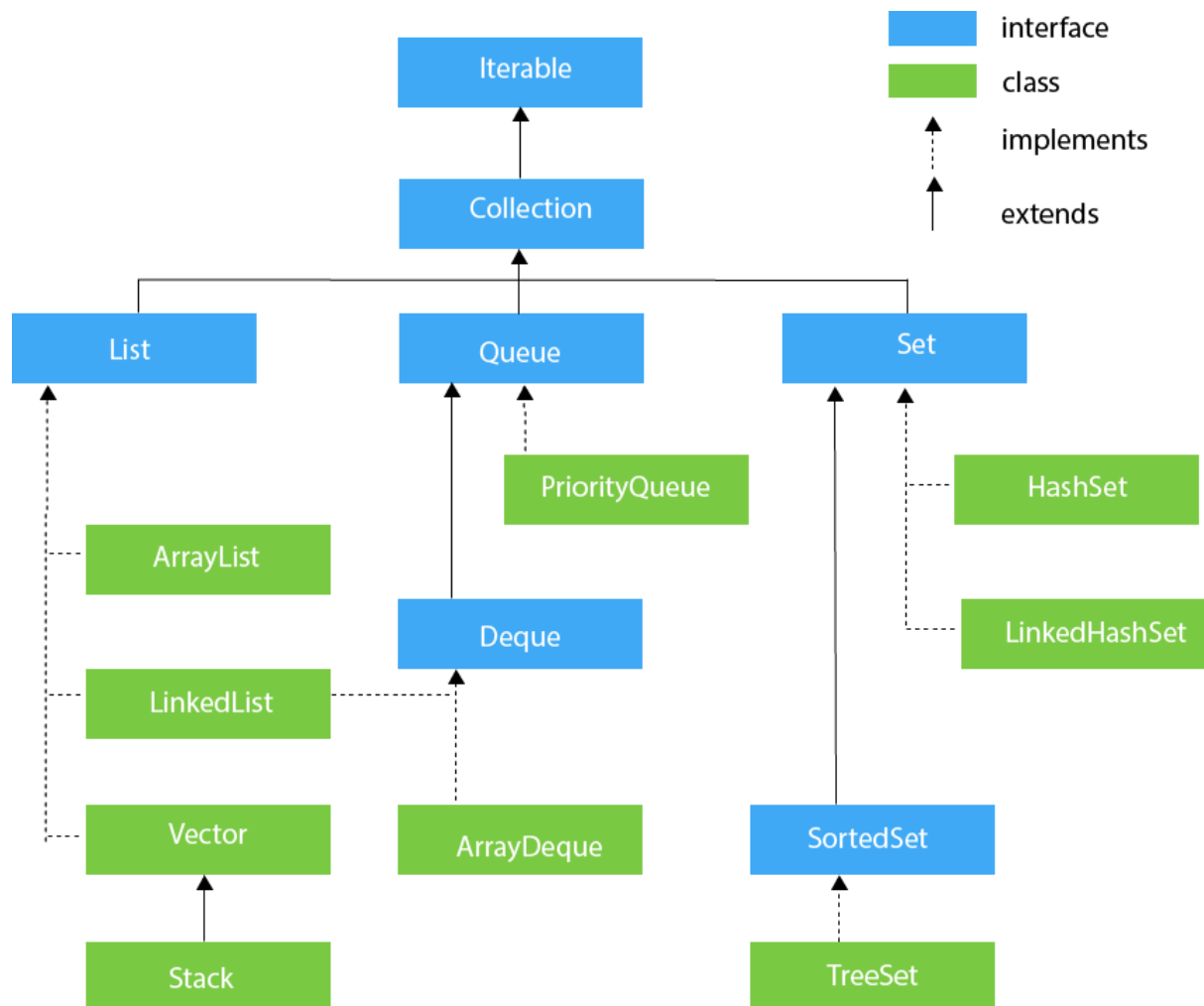
**Def:** It defines several classes and interfaces which can be used a group of objects as single entity.

### Note:

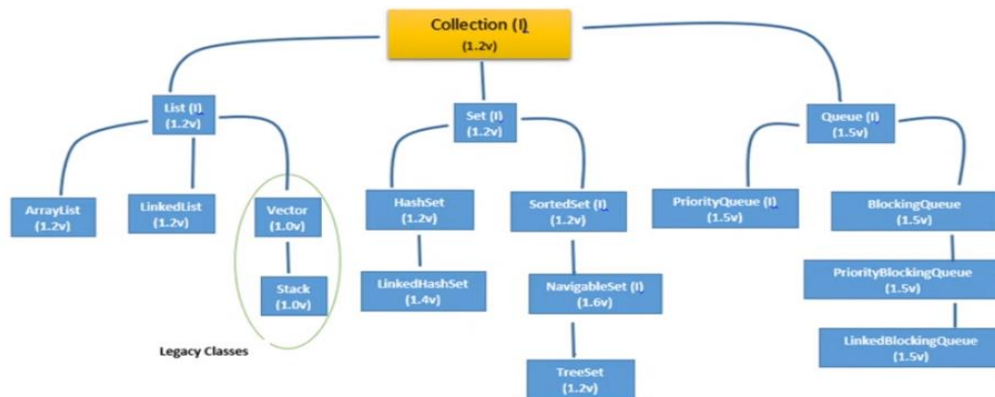
Collection and Collection Framework are not new in java, they are already existed in C++ as Container and STL(Standard Template Library).

### 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.



## Collection Framework with Versions



### Collection Interface

- 1.If you want to represent group of individual objects as an entity then we can go for using Collection Interface.
2. Collection interface defines most common methods which are applicable for any Collection object.
3. Generally Collection interface is considered as root interface of Collection Framework.

**Note:** There is no class which implements Collection interface directly.

### Methods of Collection interface

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

| No. | Method   | Description  |
|-----|--|--|
| 1   | public boolean add(E e)                          | It is used to insert an element in this collection.                                |
| 2   | Public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3   | public boolean remove(Object element)            | It is used to delete an element from the collection.                               |

|    |  |   |
|----|--|---|
| 4  | public boolean<br>removeAll(Collection<?> c)             | It is used to delete all the elements of the specified collection from the invoking collection.                 |
| 5  | default boolean<br>removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate.                   |
| 6  | public boolean<br>retainAll(Collection<?> c)             | It is used to delete all the elements of invoking collection except the specified collection.                   |
| 7  | public int size()  | It returns the total number of elements in the collection.  |
| 8  | public void clear()                                      | It removes the total number of elements from the collection.  |
| 9  | public boolean<br>contains(Object element)               | It is used to search an element.  |
| 10 | public boolean<br>containsAll(Collection<?> c)           | It is used to search the specified collection in the collection.  |
| 11 | public Iterator iterator()                               | It returns an iterator.   |
| 12 | public Object[] toArray()                                | It converts collection into array.  |
| 13 | public <T> T[] toArray(T[] a)                            | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty()                                 | It checks if collection is empty.   |
| 15 | default Stream<E><br>parallelStream()                    | It returns a possibly parallel Stream with the collection as its source.  |

|    |                                       |   |
|----|---------------------------------------|---|
| 16 | default Stream<E> stream()            | It returns a sequential Stream with the collection as its source.         |
| 17 | default Spliterator<E> spliterator()  | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections.   |
| 19 | public int hashCode()                 | It returns the hash code number of the collection.                        |

### **Differences Between Collection and Collections**

**Collection:** It is an interface used to represent a group of objects as a single entity.

**Collections:** It is an utility class present in java.util package to define several utility methods like Sorting ,Searching for Collection objects.

### **Iterator interface**

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

### **Methods of Iterator interface**

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

| No. | Method                   | Description   |
|-----|--------------------------|---|
| 1   | public boolean hasNext() | It returns true if the iterator has more elements otherwise it returns false. |
| 2   | public Object next()     | It returns the element and moves the cursor pointer to the next element.      |

|   |                    |      |   |
|---|--------------------|------|---|
| 3 | public<br>remove() | void | It removes the last elements returned by the iterator. It is less used. |
|---|--------------------|------|---|

## **Iterable Interface**

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

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

```
Iterator<T> iterator()
```

It returns the iterator over the elements of type T.

## **Iterable Interface**

```
Public interface Iterator{
    abstract Boolean hasNext();
    abstract E next();
}
```

## **List Interface**

List interface is the child interface of Collection interface.

It inhibits a list type data structure in which we can store the ordered collection of objects.

**If we want to represent the group of objects as a single entity where duplicates are allowed and insertion order preserved then we should go for List Interface.**

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

To instantiate the List interface, we must use :

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
```



```
List <data-type> list3 = new Vector();
```

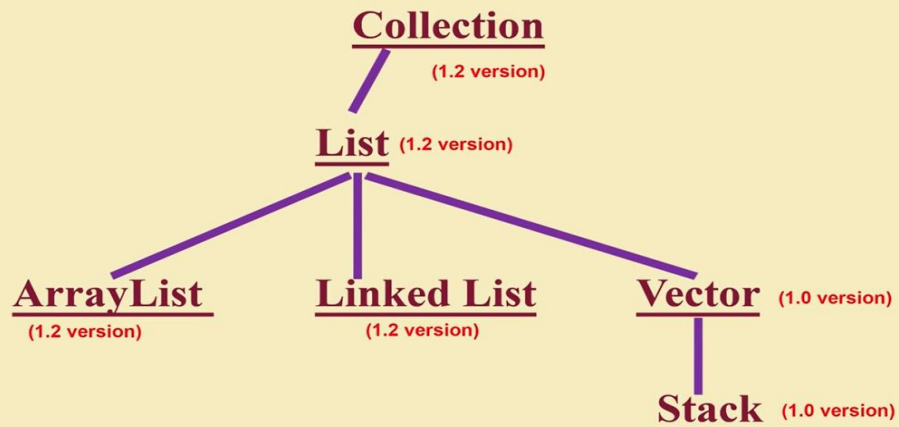
```
List <data-type> list4 = new Stack();
```

We can differentiate the duplicates by using Index.

We can preserve insertion order by using index, hence index play very important role in the list interface.

## 9 key interfaces of Collection Framework

### ii. List :



**Note:** Vector and Stack classes are re-engineered in 1.2v to implement list interface 

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

List interface specific methods:

```
void add(int index, Object o)
```

```
boolean addAll(int index, Collection c)
```

```
Object get(int index)
```

```
Object remove(int index)
```

```
Object set(int index, Object new)
```

```
int indexOf(Object o)
```

```
int lastIndexOf(Object o)
```

```
ListIterator listiterator()
```

The classes that implement the List interface are given below.

## ArrayList

The ArrayList class implements the List interface.

It uses a dynamic array or Resizable array or Growable array to store the elements.

Duplicate elements are allowed.

Insertion order is preserved.

Heterogeneous objects are allowed except TreeSet and MapSet.

Null insertion is possible.

## ArrayList Constructors

**ArrayList al=new ArrayList();**

It creates an empty ArrayList with default initial capacity 10.

Once ArrayList reaches to its maximum capacity a new ArrayList will be created with

```
new_capacity=(CurrentCapacity*3/2)+1.
```

```
ArrayList al=new ArrayList(int initialCapacity);
```

```
ArrayList al=new ArrayList(Collection c);
```

**Example:**

[illegible]

```
}  
}  
}
```

### **Output:**

```
C:\Windows\System32\cmd.exe  
E:\Collection Framework>javac ArrayListDemo.java  
E:\Collection Framework>java ArrayListDemo  
[Rama, 10, null, Rama]  
[Rama, 10, Rama]  
[Rama, 10, Krishna, Rama, Lakshmana, Lakshmana, Lakshmana, Lakshmana, Lakshmana, Lakshmana, Lakshmana]  
E:\Collection Framework>
```

Usually, we can use Collections to hold and transfer Objects from one place to another place, to provide support for this requirement every Collection already implements Serializable and Cloneable interfaces.

ArrayList and Vector classes implement the RandomAccess interface so that we can access any Random element with the same speed.

Hence, if our frequent operation is retrieval operation then ArrayList is the best choice.

The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

### **Example2**

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

```

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

```

Output:

```

Ravi
Vijay
Ravi
Ajay

```

### **Example3:**

```

//Write a java Program to describe ArrayList methods
import java.util.*;
class ArrayListDemo
{
public static void main(String args[])
{
    List<Integer> intlist=new ArrayList<Integer>();
    intlist.add(10);    intlist.add(20);    intlist.add(30);
    intlist.add(40);    intlist.add(50); intlist.add(10);
    System.out.println(intlist.add(null));
    System.out.println(intlist.isEmpty());
    Iterator<Integer> iterator=intlist.iterator();
    System.out.println("\nDisplay Elements using Iterator:");
    while(iterator.hasNext())
    {
        System.out.print(iterator.next()+" ");
    }
    System.out.println(intlist.remove(null));
    Object[] arr=intlist.toArray();

```

```

        System.out.println("\nDisplay Elements in the form of an
        Array:");
        for(int i=0;i<arr.length;i++)
        {
            System.out.print(arr[i]+" ");
        }
        System.out.println("\n"+intlist.hashCode());
    }
}

```

### **Note:**

ArrayList is the best choice if our frequent operation is retrieval operation(Because ArrayList implements RandomAccess interface).

ArrayList is the worst choice if our frequent operation is insertion or deletion in the middle of the list(Because several shift operations are required).

### **RandomAccess Interface**

It will present in the java.util package.

It does not contain any methods hence it is called as “Marker Interface”.

### **LinkedList**

LinkedList implements the Collection interface.

It uses a data structure 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.

LinkedList implements Serializable and Cloneable interfaces but not RandomAccess interface.

LinkedList is the best choice if our operation is insertion and deletion at the middle of the list.

LinkedList is the worst operation if our frequent operation is retrieval operation.

Usually we can use LinkedList to implement Stack and Queues to provide support for this requirement, LinkedList provides the following specific methods.

Void addFirst()

Void addLast()

Object getFirst()

Object getLast()

Object removeFirst()

Object removeLast()

### **LinkedList Constructors:**

1. LinkedList l=new LinkedList()→Creates an empty linkedlist.

2. LinkedList l=new LinkedList(Collection c)→creates an equivalent linkedlist object for the given linkedlist.

### **Example.**

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

### **Output:**

```
Ravi
Vijay
```

## **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.

### **Vector Specific Methods**

#### **For adding Objects:**

add(Object o) //From Collection Interface-List(I)

add(int index, Object o) //From List

addElement(Object o) //From Vector

#### **For Removing Objects:**

remove(Object o) [From Collection]

removeElement(Object o) [From Vector]

remove(int index) [From List]

removeElementAt(int index) [From Vector]

clear() [From Collection]

removeAllElements() [From Vector]

#### **For Accessing Elements:**

Object get(int index) [From List]

Object elementAt(int index) [From Vector]

Object firstElement() [From Vector]

Object lastElement() [From Vector]

#### **Other Methods:**

int size()

int capacity()

Enumeration elements()

### **Constructors of Vector Class:**

Vector v=new Vector();

It creates an empty vector object with default initial capacity 10. Once vector reaches it's max capacity a new vector object will be created with **new capacity=2\*current capacity**.

(ii) Vector v=new Vector(int initialCapacity);

It creates an empty vector object with specified initial capacity.

(iii) Vector v=new Vector(int initialcapacity,int incrementcapacity)

(iv) Vector v=new Vector(Collection c)

->It creates an equivalent vector Object for the given Collection.

### **Example:**

```
import java.util.*;

class VectorDemo

{

public static void main(String args[])

{

    Vector v=new Vector();

    System.out.println(v.capacity());

    for(int i=0;i<10;i++)

    {

        v.addElement(i);

    }

    System.out.println(v.capacity());

    v.addElement("A");

    System.out.println(v.capacity());

}
```



```

        System.out.println(v);
    }

}

```

### **Output:**

```

C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.2006]
(c) Microsoft Corporation. All rights reserved.

E:\Collection Framework>javac VectorDemo.java
Note: VectorDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

E:\Collection Framework>java VectorDemo
10
10
20
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A]

E:\Collection Framework>_

```

### **Example2.**

```

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

```

Output:

```
Ayush
Amit
Ashish
Garima
```

## **Stack**

The stack is the subclass of Vector.

It implements the last-in-first-out data structure, i.e., Stack.

The stack contains all of the methods of Vector class and also provides its methods like

void push(Object obj) → For inserting an object to the Stack.

Object pop() → To remove and returns the top of the Stack.

Object peek() → It returns the top element of the stack without removing.

int search(Object obj):

→ It returns Offset from the top of the stack, if the  
specified object is available.

It returns -1, if the specified object is not available

## **Example 1:**

```
import java.util.*;

class StackDemo

{

public static void main(String args[])

{

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

    s.push("A");

    s.push("B");

    s.push("C");
```

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

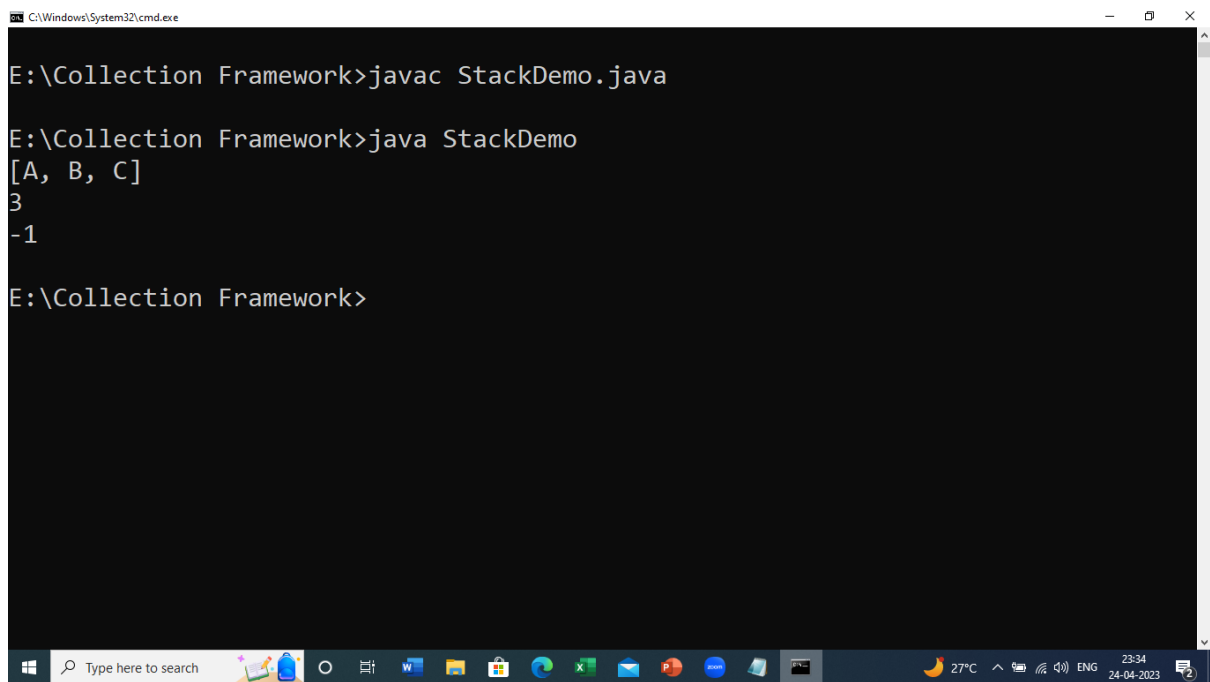
        System.out.println(s.search("A"));

        System.out.println(s.search("z"));

    }

}
```

### **Output:**



```
C:\Windows\System32\cmd.exe

E:\Collection Framework>javac StackDemo.java

E:\Collection Framework>java StackDemo
[A, B, C]
3
-1

E:\Collection Framework>
```

### **Example2: Java Program to store set of elements on stack and iterate using Iterator interface.**

```
import java.util.*;

public class TestJavaCollection4{
    public static void main(String args[]){
        Stack<String> stack = new Stack<String>();
        stack.push("Ayush");
        stack.push("Garvit");
        stack.push("Amit");
        stack.push("Ashish");
        stack.push("Garima");
        stack.pop();
    }
}
```

```

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

```

### **Output:**

```

Ayush
Garvit
Amit
Ashish

```

### **Queue Interface**

A collection designed for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.

Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation).

The latter form of the insert operation is designed specifically for use with capacity-restricted Queue implementations; in most implementations, insert operations cannot fail.

| Summary of Queue methods |                         |                              |
|--------------------------|-------------------------|------------------------------|
|                          | <i>Throws exception</i> | <i>Returns special value</i> |
| <b>Insert</b>            | add(e)                  | offer(e)                     |
| <b>Remove</b>            | remove()                | poll()                       |
| <b>Examine</b>           | element()               | peek()                       |

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner.

Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out).

Whatever the ordering used, the *head* of the queue is that element which would be removed by a call to remove() or poll().

In a FIFO queue, all new elements are inserted at the *tail* of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

Some of the commonly used methods of the Queue interface are:

`add()` - Inserts the specified element into the queue. If the task is successful, `add()` returns true, if not it throws an exception.

`offer()` - Inserts the specified element into the queue. If the task is successful, `offer()` returns true, if not it returns false.

`element()` - Returns the head of the queue. Throws an exception if the queue is empty.

`peek()` - Returns the head of the queue. Returns null if the queue is empty.

`remove()` - Returns and removes the head of the queue. Throws an exception if the queue is empty.

`poll()` - Returns and removes the head of the queue. Returns null if the queue is empty.

## **Implementing the LinkedList Class**

### **Code:**

```
import java.util.Queue;
import java.util.LinkedList;

class Main {

    public static void main(String[] args) {

        // Creating Queue using the LinkedList class
        Queue<Integer> numbers = new LinkedList<>();

        // offer elements to the Queue
        numbers.offer(1);
        numbers.offer(2);
```

```
numbers.offer(3);
System.out.println("Queue: " + numbers);
// Access elements of the Queue
int accessedNumber = numbers.peek();
System.out.println("Accessed Element: " + accessedNumber);
// Remove elements from the Queue
int removedNumber = numbers.poll();
System.out.println("Removed Element: " + removedNumber);
System.out.println("Updated Queue: " + numbers);
}
}
```

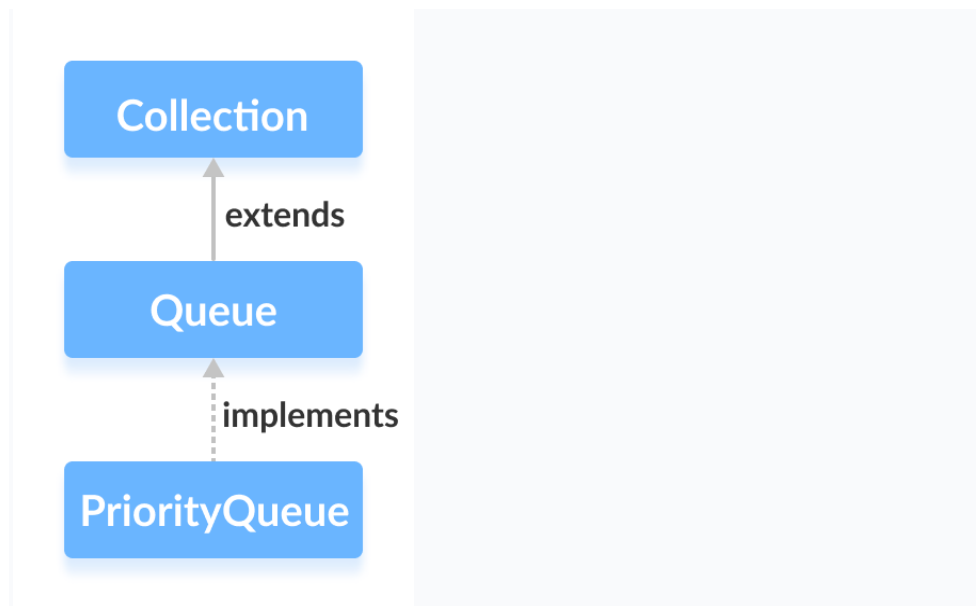
### Output

```
Queue: [1, 2, 3]
Accessed Element: 1
Removed Element: 1
Updated Queue: [2, 3]
```

### PriorityQueue

The `PriorityQueue` class provides the functionality of the **heap data structure**.

It implements the **Queue interface**.



Unlike normal queues, priority queue elements are retrieved in sorted order.

Suppose, we want to retrieve elements in the ascending order. In this case, the head of the priority queue will be the smallest element. Once this element is retrieved, the next smallest element will be the head of the queue.

It is important to note that the elements of a priority queue may not be sorted. However, elements are always retrieved in sorted order.

### **Creating PriorityQueue**

In order to create a priority queue, we must import the `java.util.PriorityQueue` package. Once we import the package, here is how we can create a priority queue in Java.

```
PriorityQueue<Integer> numbers = new PriorityQueue<>();
```

Here, we have created a priority queue without any arguments. In this case, the head of the priority queue is the smallest element of the queue. And elements are removed in ascending order from the queue.

However, we can customize the ordering of elements with the help of the `Comparator` interface.

## **Methods of PriorityQueue**

The `PriorityQueue` class provides the implementation of all the methods present in the `Queue` interface.

Insert Elements to PriorityQueue

`add()` - Inserts the specified element to the queue. If the queue is full, it throws an exception.

`offer()` - Inserts the specified element to the queue. If the queue is full, it returns `false`.

### **Example**

```
import java.util.PriorityQueue;

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

        // Creating a priority queue
        PriorityQueue<Integer> numbers = new PriorityQueue<>();

        // Using the add() method
        numbers.add(4);
        numbers.add(2);
        System.out.println("PriorityQueue: " + numbers);

        // Using the offer() method
        numbers.offer(1);
        System.out.println("Updated PriorityQueue: " + numbers);
    }
}
```

### **Output**

```
PriorityQueue: [2, 4]
Updated PriorityQueue: [1, 4, 2]
```

Here, we have created a priority queue named `numbers`. We have inserted 4 and 2 to the queue.

Although 4 is inserted before 2, the head of the queue is 2. It is because the head of the priority queue is the smallest element of the queue.



## **Deque Interface**

The `Deque` interface of the Java collections framework provides the functionality of a double-ended queue. It extends the `Queue` interface.

## **Working of Deque**

In a regular queue, elements are added from the rear and removed from the front. However, in a deque, we can **insert and remove elements from both front and rear**.

## **Classes that implement Deque**

In order to use the functionalities of the `Deque` interface, we need to use classes that implement it:

`ArrayDeque`

`LinkedList`

How to use Deque?

In Java, we must import the `java.util.Deque` package to use `Deque`.

```
// Array implementation of Deque
Deque<String> animal1 = new ArrayDeque<>();

// LinkedList implementation of Deque
Deque<String> animal2 = new LinkedList<>();
```

## **Methods of Deque**

Since `Deque` extends the `Queue` interface, it inherits all the methods of the Queue interface.

Besides methods available in the `Queue` interface, the `Deque` interface also includes the following methods:

**addFirst()** - Adds the specified element at the beginning of the deque.

Throws an exception if the deque is full.

**addLast()** - Adds the specified element at the end of the deque. Throws an exception if the deque is full.

**offerFirst()** - Adds the specified element at the beginning of the deque. Returns `false` if the deque is full.

**offerLast()** - Adds the specified element at the end of the deque. Returns `false` if the deque is full.

**getFirst()** - Returns the first element of the deque. Throws an exception if the deque is empty.

**getLast()** - Returns the last element of the deque. Throws an exception if the deque is empty.

**peekFirst()** - Returns the first element of the deque. Returns `null` if the deque is empty.

**peekLast()** - Returns the last element of the deque. Returns `null` if the deque is empty.

**removeFirst()** - Returns and removes the first element of the deque. Throws an exception if the deque is empty.

**removeLast()** - Returns and removes the last element of the deque. Throws an exception if the deque is empty.

**pollFirst()** - Returns and removes the first element of the deque. Returns `null` if the deque is empty.

**pollLast()** - Returns and removes the last element of the deque. Returns `null` if the deque is empty.

**push()** - adds an element at the beginning of deque

**pop()** - removes an element from the beginning of deque

**peek()** - returns an element from the beginning of deque

Implementation of Deque in ArrayDeque Class

```
import java.util.Deque;
import java.util.ArrayDeque;

class Main {

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

```
// Creating Deque using the ArrayDeque class
Deque<Integer> numbers = new ArrayDeque<>();
```

```
// add elements to the Deque
numbers.offer(1);
numbers.offerLast(2);
numbers.offerFirst(3);
System.out.println("Deque: " + numbers);
```

```
// Access elements of the Deque
int firstElement = numbers.peekFirst();
System.out.println("First Element: " + firstElement);
```

```
int lastElement = numbers.peekLast();
System.out.println("Last Element: " + lastElement);
```

```
// Remove elements from the Deque
int removedNumber1 = numbers.pollFirst();
System.out.println("Removed First Element: " + removedNumber1);
```

```
int removedNumber2 = numbers.pollLast();
System.out.println("Removed Last Element: " + removedNumber2);
```

```
System.out.println("Updated Deque: " + numbers);
```

```
}
```

## Output

```
Deque: [3, 1, 2]
First Element: 3
Last Element: 2
Removed First Element: 3
Removed Last Element: 2
Updated Deque: [1]
```

## **ArrayDeque**

In Java, we can use the ArrayDeque class to implement queue and deque data structures using arrays.

### **Creating ArrayDeque**

In order to create an array deque, we must import the java.util.ArrayDeque package.

Here is how we can create an Array deque in Java:

```
ArrayDeque<Type> obj = new ArrayDeque<>();
```

Here, Type indicates the type of the array deque. For example,

```
// Creating String type ArrayDeque
```

```
ArrayDeque<String> obj = new ArrayDeque<>();
```

```
// Creating Integer type ArrayDeque
```

```
ArrayDeque<Integer> obj = new ArrayDeque<>();
```

### **Methods of ArrayDeque**

The ArrayDeque class provides implementations for all the methods present in Queue and Deque interface.

#### **1.Insert Elements to Deque**

add() - inserts the specified element at the end of the array deque

addFirst() - inserts the specified element at the beginning of the array deque

addLast() - inserts the specified at the end of the array deque (equivalent to add())

Note: If the array deque is full, all these methods add(), addFirst() and addLast() throws IllegalStateException.

### **Example**

```
import java.util.ArrayDeque;
```

```
class Main {
```

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

```
    ArrayDeque<String> animals= new ArrayDeque<>();
```

```
    // Using add()
```

```
    animals.add("Dog");
```

```

// Using addFirst()
animals.addFirst("Cat");

// Using addLast()
animals.addLast("Horse");
System.out.println("ArrayDeque: " + animals);
}
}

```

### **Output**

ArrayDeque: [Cat, Dog, Horse]

## **2.Insert elements using offer(), offerFirst() and offerLast()**

offer() - inserts the specified element at the end of the array  
deque

offerFirst() - inserts the specified element at the beginning of  
the array deque

offerLast() - inserts the specified element at the end of the  
array deque

Note: offer(), offerFirst() and offerLast() returns true if the element is successfully inserted; if the array deque is full, these methods return false.

### **Example**

```

import java.util.ArrayDeque;
class Main {
public static void main(String[] args) {
    ArrayDeque<String> animals= new ArrayDeque<>();
    // Using offer()
    animals.offer("Dog");
    // Using offerFirst()
    animals.offerFirst("Cat");
    // Using offerLast()
    animals.offerLast("Horse");
    System.out.println("ArrayDeque: " + animals);
}
}

```

### **Output**

ArrayDeque: [Cat, Dog, Horse]

**Note:** If the array deque is full  
the add() method will throw an exception  
the offer() method returns false

### **3.Access ArrayDeque Elements**

getFirst() - returns the first element of the array deque

getLast() - returns the last element of the array deque

**Note:** If the array deque is empty, getFirst() and getLast() throws NoSuchElementException.

### **Example**

```
import java.util.ArrayDeque;

class Main {
public static void main(String[] args) {
    ArrayDeque<String> animals= new ArrayDeque<>();
    animals.add("Dog");
    animals.add("Cat");
    animals.add("Horse");
    System.out.println("ArrayDeque: " + animals);

    // Get the first element
    String firstElement = animals.getFirst();
    System.out.println("First Element: " + firstElement);

    // Get the last element
    String lastElement = animals.getLast();
    System.out.println("Last Element: " + lastElement);
}
}
```

### **Output**

```
ArrayDeque: [Dog, Cat, Horse]
First Element: Dog
Last Element: Horse
```

#### **4. Access elements using peek(), peekFirst() and peekLast() method**

peek() - returns the first element of the array deque

peekFirst() - returns the first element of the array deque (equivalent to peek())

peekLast() - returns the last element of the array deque

#### **Example:**

```
import java.util.ArrayDeque;

class Main {

    public static void main(String[] args) {

        ArrayDeque<String> animals= new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("ArrayDeque: " + animals);
        // Using peek()
        String element = animals.peek();
        System.out.println("Head Element: " + element);
        // Using peekFirst()
        String firstElement = animals.peekFirst();
        System.out.println("First Element: " + firstElement);
        // Using peekLast
        String lastElement = animals.peekLast();
        System.out.println("Last Element: " + lastElement);
    }

}
```

#### **Output**

ArrayDeque: [Dog, Cat, Horse]

Head Element: Dog

First Element: Dog

Last Element: Horse

**Note:** If the array deque is empty, peek(), peekFirst() and getLast() throws NoSuchElementException.

### **Remove ArrayDeque Elements**

remove() - returns and removes an element from the first element of the array deque

remove(element) - returns and removes the specified element from the head of the array deque

removeFirst() - returns and removes the first element from the array deque (equivalent to remove())

removeLast() - returns and removes the last element from the array deque

**Note:** If the array deque is empty, remove(), removeFirst() and removeLast() method throws an exception. Also, remove(element) throws an exception if the element is not found.

### **Example:**

```
import java.util.ArrayDeque;

class Main {

public static void main(String[] args) {

    ArrayDeque<String> animals= new ArrayDeque<>();

    animals.add("Dog");
    animals.add("Cat");
    animals.add("Cow");
    animals.add("Horse");

    System.out.println("ArrayDeque: " + animals);

    // Using remove()

    String element = animals.remove();

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

    System.out.println("New ArrayDeque: " + animals);

    // Using removeFirst()

    String firstElement = animals.removeFirst();

    System.out.println("Removed First Element: " + firstElement);
```



```

// Using removeLast()
String lastElement = animals.removeLast();
System.out.println("Removed Last Element: " + lastElement);
}
}

```

### **Output**

ArrayDeque: [Dog, Cat, Cow, Horse]

Removed Element: Dog

New ArrayDeque: [Cat, Cow, Horse]

Removed First Element: Cat

Removed Last Element: Horse

### **5. Remove elements using the poll(), pollFirst() and pollLast() method**

poll() - returns and removes the first element of the array deque

pollFirst() - returns and removes the first element of the array deque  
(equivalent to poll())

pollLast() - returns and removes the last element of the array deque

**Note:** If the array deque is empty, poll(), pollFirst() and pollLast() returns null if the element is not found.

### **Example**

```

import java.util.ArrayDeque;

class Main {

public static void main(String[] args) {

    ArrayDeque<String> animals= new ArrayDeque<>();
    animals.add("Dog");
    animals.add("Cat");
    animals.add("Cow");
    animals.add("Horse");

    System.out.println("ArrayDeque: " + animals);

    // Using poll()

    String element = animals.poll();

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

```

```

        System.out.println("New ArrayDeque: " + animals);
        // Using pollFirst()
        String firstElement = animals.pollFirst();
        System.out.println("Removed First Element: " + firstElement);
        // Using pollLast()
        String lastElement = animals.pollLast();
        System.out.println("Removed Last Element: " + lastElement);
    }
}

```

### **Output**

```

ArrayDeque: [Dog, Cat, Cow, Horse]
Removed Element: Dog
New ArrayDeque: [Cat, Cow, Horse]
Removed First Element: Cat
Removed Last Element: Horse

```

### **6. Remove Element: using the clear() method**

**To remove all the elements from the array deque, we use the clear() method. For example,**

```

import java.util.ArrayDeque;
class Main {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("ArrayDeque: " + animals);
        // Using clear()
        animals.clear();
    }
}

```

```
        System.out.println("New ArrayDeque: " + animals);
    }
}
```

### **Output**

ArrayDeque: [Dog, Cat, Horse]

New ArrayDeque: []

### **Iterating the ArrayDeque**

iterator() - returns an iterator that can be used to iterate over the array deque

descendingIterator() - returns an iterator that can be used to iterate over the array deque in reverse order

In order to use these methods, we must import the java.util.Iterator package.

### **For Example:**

```
import java.util.ArrayDeque;
import java.util.Iterator;
class Main {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.print("ArrayDeque: ");
        // Using iterator()
        Iterator<String> iterate = animals.iterator();
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
        System.out.print("\nArrayDeque in reverse order: ");
    }
}
```

```

// Using descendingIterator()
Iterator<String> desIterate = animals.descendingIterator();
while(desIterate.hasNext()) {
    System.out.print(desIterate.next());
    System.out.print(", ");
}
}
}

```

### **Output**

ArrayDeque: [Dog, Cat, Horse]

ArrayDeque in reverse order: [Horse, Cat, Dog]

### **Other Methods**

element(): Returns an element from the head of the array deque.

contains(element): Searches the array deque for the specified element. If the element is found, it returns true, if not it returns false.

size() : Returns the length of the array deque.

toArray(): Converts array deque to array and returns it.

clone() : Creates a copy of the array deque and returns it.

### **ArrayDeque as a Stack**

To implement a LIFO (Last-In-First-Out) stacks in Java, it is recommended to use a deque over the Stack class.

The ArrayDeque class is likely to be faster than the Stack class.

ArrayDeque provides the following methods that can be used for implementing a stack.

push() - adds an element to the top of the stack

peek() - returns an element from the top of the stack

pop() - returns and removes an element from the top of the stack

### **Example:**

```

import java.util.ArrayDeque;

class Main {

    public static void main(String[] args) {

```

```

ArrayDeque<String> stack = new ArrayDeque<>();
// Add elements to stack
stack.push("Dog");
stack.push("Cat");
stack.push("Horse");
System.out.println("Stack: " + stack);
// Access element from top of stack
String element = stack.peek();
System.out.println("Accessed Element: " + element);
// Remove elements from top of stack
String remElement = stack.pop();
System.out.println("Removed element: " + remElement);
}
}

```

### **Output**

Stack: [Horse, Cat, Dog]

Accessed Element: Horse

Removed Element: Horse

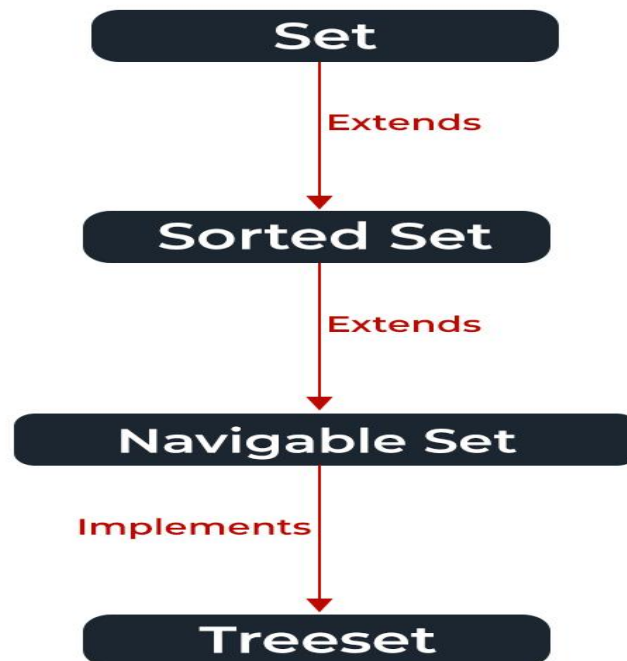
### **ArrayDeque Vs. LinkedList Class**

- Both ArrayDeque and Java LinkedList implements the Deque interface.
- However, there exist some differences between them.
  - LinkedList supports null elements, whereas ArrayDeque doesn't.
  - Each node in a linked list includes links to other nodes. That's why LinkedList requires more storage than ArrayDeque.
  - If you are implementing the queue or the deque data structure, an ArrayDeque is likely to be faster than a LinkedList.

### **Set Interface**

- 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.



- 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   |
|--------------------------------|---|
| <u>add(element)</u>            | 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. |
| <u>addAll(collection)</u>      | 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.  |
| <u>clear()</u>                 | This method is used to remove all the elements from the set but not delete the set. The reference for the set still exists.   |
| <u>contains(element)</u>       | This method is used to check whether a specific element is present in the Set or not.   |
| <u>containsAll(collection)</u> | 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.    |
| <u>hashCode()</u>              | 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.   |
| <u>isEmpty()</u>               | This method is used to check whether the set is empty or not.   |
| <u>iterator()</u>              | This method is used to return the <u>iterator</u> of the set. The elements from the set are returned in a   |

| Method                                       | Description  |
|--|--|
|  | random order.  |
| <u><a href="#">remove(element)</a></u>       | 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.              |
| <u><a href="#">removeAll(collection)</a></u> | 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.         |
| <u><a href="#">retainAll(collection)</a></u> | 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. |
| <u><a href="#">size()</a></u>                | This method is used to get the size of the set. This returns an integer value which signifies the number of elements.  |
| <u><a href="#">toArray()</a></u>             | 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 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]

## Operations on the Set Interface

The set interface allows the users to perform the basic mathematical operation on the set. Let's take two arrays to understand these basic operations. Let set1 = [1, 3, 2, 4, 8, 9, 0] and set2 = [1, 3, 7, 5, 4, 0, 7, 5]. Then the possible operations on the sets are:

**1. Intersection:** This operation returns all the common elements from the given two sets. For the above two sets, the intersection would be:  
Intersection = [0, 1, 3, 4]

**2. Union:** This operation adds all the elements in one set with the other. For the above two sets, the union would be:  
Union = [0, 1, 2, 3, 4, 5, 7, 8, 9]

**3. Difference:** This operation removes all the values present in one set from the other set. For the above two sets, the difference would be:  
Difference = [2, 8, 9]

Now let us implement the following operations as defined above as follows:

### **Example:**

```

// Java Program Demonstrating Operations on the Set
// such as Union, Intersection and Difference operations

```

```

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

// Main class
public class SetExample {

// Main driver method
public static void main(String args[])
{
// Creating an object of Set class Declaring object of Integer
//type
Set<Integer> a = new HashSet<Integer>();

// Adding all elements to List
a.addAll(Arrays.asList( new Integer[] { 1, 3, 2, 4, 8, 9, 0 }));

// Again declaring object of Set class with reference to HashSet
Set<Integer> b = new HashSet<Integer>();

b.addAll(Arrays.asList(new Integer[] { 1, 3, 7, 5, 4, 0, 7, 5 }));


// To find union
Set<Integer> union = new HashSet<Integer>(a);
union.addAll(b);
System.out.print("Union of the two Set");
System.out.println(union);

// To find intersection
Set<Integer> intersection = new HashSet<Integer>(a);
intersection.retainAll(b);
System.out.print("Intersection of the two Set");
System.out.println(intersection);

// To find the symmetric difference
Set<Integer> difference = new HashSet<Integer>(a);
difference.removeAll(b);
System.out.print("Difference of the two Set");
System.out.println(difference);
}
}

```

### **Output**

Union of the two Set: [0, 1, 2, 3, 4, 5, 7, 8, 9]

Intersection of the two Set: [0, 1, 3, 4]

Difference of the two Set: [2, 8, 9]

## **Performing Various Operations on Set**

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.

Since Set is an interface, it can be used only with a class that implements this interface.

HashSet is one of the widely used classes which implements the Set interface. Now, let's see how to perform a few frequently used operations on the HashSet.

We are going to perform the following operations as follows:

1. Adding elements
2. Accessing elements
3. Removing elements
4. Iterating elements
5. Iterating through Set

Now let us discuss these operations individually as follows:

### **Operations 1: Adding Elements**

In order to add an element to the Set, we can use the add() method.

However, the insertion order is not retained in the Set. Internally, for every element, a hash is generated and the values are stored with respect to the generated hash.

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 accepted by the Set.

#### **Example**

```
import java.util.*;

//Java Program to describe Set Operations

class SetOperationsDemo {

public static void main(String[] args)

{

/ Creating an object of Set and declaring object of type String

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

    //1.Adding elements to above object using add() method

    hs.add("A");

    hs.add("B");

    hs.add("C");

    hs.add("B");

    hs.add("D");

    hs.add("E");

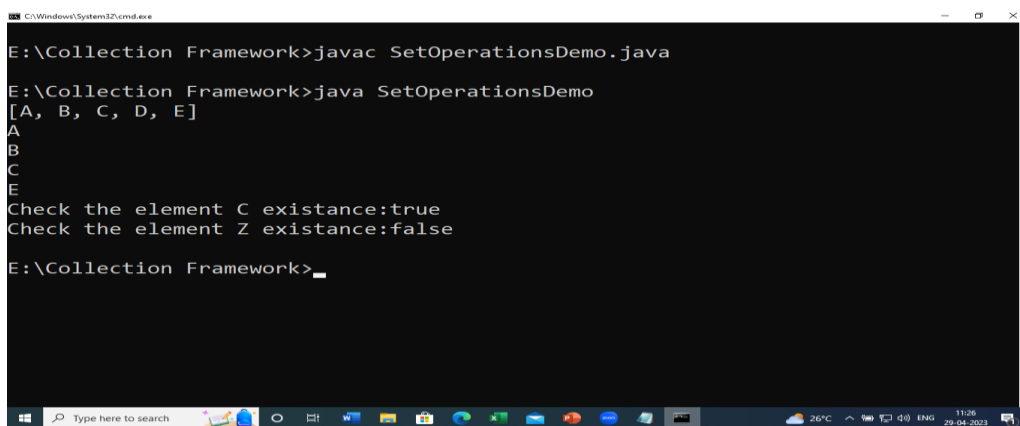
    System.out.println(hs);
```

```
//2.Removing the elements from the set
hs.remove("D");

//3. Accessing the elements using Set
for(String str:hs)
    System.out.println(str);

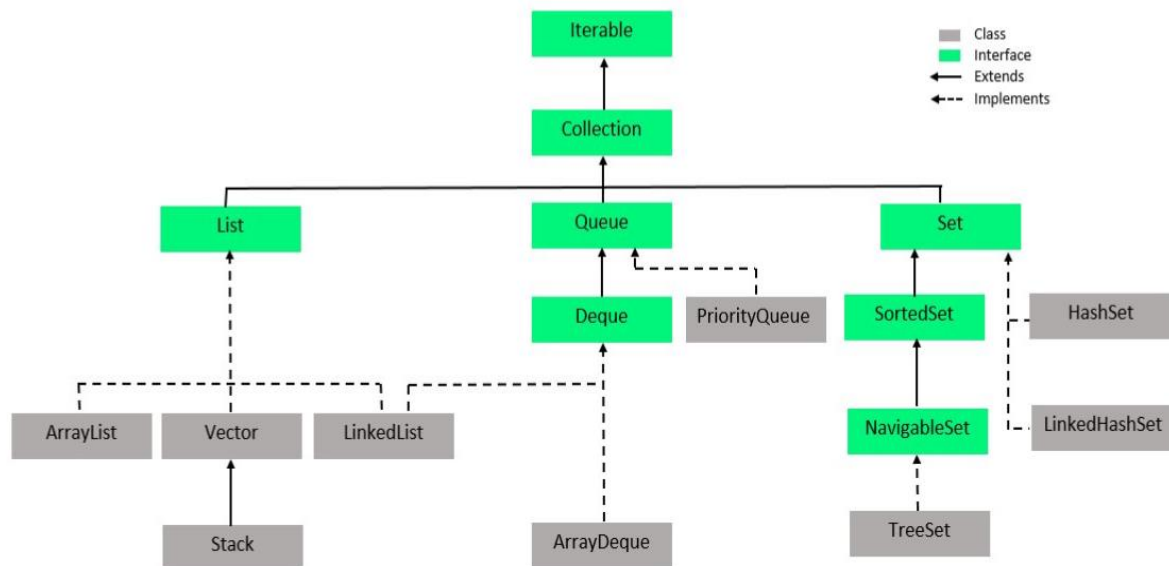
System.out.println("Check the element C existence:"+hs.contains("C"));
    System.out.println("Check the element Z existence:"+hs.contains("Z"));
}
}
```

### **Output:**



```
C:\Windows\System32\cmd.exe
E:\Collection Framework>javac SetOperationsDemo.java
E:\Collection Framework>java SetOperationsDemo
[A, B, C, D, E]
A
B
C
E
Check the element C existence:true
Check the element Z existence:false
E:\Collection Framework>_
```

- Classes that implement the Set interface in Java Collections can be easily perceived from the image below as follows and are listed as follows:
  - HashSet
  - LinkedHashSet
  - TreeSet



## 1. **HashSet**

HashSet class which is implemented in the collection framework is an inherent implementation of the hash table data structure.

The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashCode.

This class also allows the insertion of NULL elements. Let's see how to create a set object using this class.

### **Example**

```
// Java program Demonstrating Creation of Set object

// Using the Hashset class

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

class HashSetDemo {

public static void main(String[] args)
{

    // Creating object of Set of type String
    Set<String> h = new HashSet<String>();

    // Adding elements into the HashSet using add() method
```

```

h.add("India");
h.add("Australia");
h.add("South Africa");
h.add("Kenada");
// Adding the duplicate element
h.add("India");
// Displaying the HashSet
System.out.println(h);
// Removing items from HashSet using remove() method
h.remove("Australia");
System.out.println("Set after removing "+ "Australia:" + h);
// Iterating over hash set items
System.out.println("Iterating over set:");
// Iterating through iterators
Iterator<String> i = h.iterator();
// It holds true till there is a single element remaining in the object
while (i.hasNext())
    System.out.println(i.next());
}
}

```

**Output:**

```
C:\Windows\System32\cmd.exe
E:\Collection Framework>javac HashSetDemo.java

E:\Collection Framework>java HashSetDemo
[Kenada, South Africa, Australia, India]
Set after removing Australia:[Kenada, South Africa, India]
Iterating over set:
Kenada
South Africa
India

E:\Collection Framework>
```

## **2. LinkedHashSet**

LinkedHashSet class which is implemented in the collections framework 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.

### **Example**

// Java program Demonstrating Creation of Set object

// Using the LinkedHashset class

```
import java.util.*;

class LinkedHashSetDemo {

    public static void main(String[] args)

    {

        // Creating object of Set of type String

        Set<String> lh = new LinkedHashSet<String>();
```

```
// Adding elements into the HashSet using add() method
lh.add("B");
lh.add("C");
lh.add("A");
lh.add("D");

// Adding the duplicate element
lh.add("A");

// Displaying the HashSet
System.out.println(lh);

// Removing items from HashSet using remove() method
lh.remove("A");

System.out.println("Set after removing A:" + lh);

// Iterating over hash set items
System.out.println("Iterating over set:");

// Iterating through iterators
Iterator<String> i = lh.iterator();

// It holds true till there is a single element
// remaining in the object
while (i.hasNext())

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

**Output:**



```
C:\Windows\System32\cmd.exe

E:\Collection Framework>javac LinkedListDemo.java

E:\Collection Framework>java LinkedListDemo
[B, C, A, D]
Set after removing A:[B, C, D]
Iterating over set:
B
C
D

E:\Collection Framework>
```

## SortedSet Interface

The `SortedSet` interface of the Java Collections framework is used to store elements with some order in a set.

It extends the `Set` interface.

To use `SortedSet`, we must import the `java.util.SortedSet` package first.

```
// SortedSet implementation by TreeSet class
SortedSet<String> animals = new TreeSet<>();
```

We have created a sorted set called `animals` using the `TreeSet` class.

## Methods of SortedSet

The `SortedSet` interface includes all the methods of the `Set` interface. It's because `Set` is a super interface of `SortedSet`.

Besides methods included in the `Set` interface, the `SortedSet` interface also includes these methods:

**comparator()** - returns a comparator that can be used to order elements in the set

**first()** - returns the first element of the set

**last()** - returns the last element of the set

**headSet(element)** - returns all the elements of the set before the specified element

**tailSet(element)** - returns all the elements of the set after the specified element including the specified element

**subSet(element1, element2)** - returns all the elements between the `element1` and `element2` including `element1`

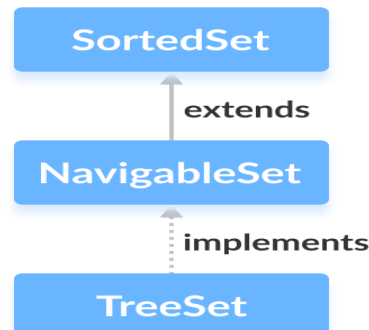
### NavigableSet Interface

The `NavigableSet` interface of the Java Collections framework provides the features to navigate among the set elements.

It is considered as a type of `SortedSet`.

### Class that implements NavigableSet

In order to use the functionalities of the `NavigableSet` interface, we need to use the `TreeSet` class that implements `NavigableSet`.



### How to use NavigableSet?

In Java, we must import the `java.util.NavigableSet` package to use `NavigableSet`. Once we import the package, here's how we can create navigable sets.

```
// SortedSet implementation by TreeSet class
NavigableSet<String> numbers = new TreeSet<>();
```

Here, we have created a navigable set named `numbers` of the `TreeSet` class.

## **Methods of NavigableSet**

The `NavigableSet` is considered as a type of `SortedSet`. It is because `NavigableSet` extends the `SortedSet` interface.

Hence, all `SortedSet` methods are also available in `NavigableSet`. To learn how these methods, visit [Java SortedSet](#).

However, some of the methods of `SortedSet` (`headSet()`, `tailSet()` and `subSet()`) are defined differently in `NavigableSet`.

Let's see how these methods are defined in `NavigableSet`.

### **headSet(element, booleanValue)**

The `headSet()` method returns all the elements of a navigable set before the specified `element` (which is passed as an argument).

The `booleanValue` parameter is optional. Its default value is `false`.

If `true` is passed as a `booleanValue`, the method returns all the elements before the specified element including the specified element.

### **tailSet(element, booleanValue)**

The `tailSet()` method returns all the elements of a navigable set after the specified `element` (which is passed as an argument) including the specified element.

The `booleanValue` parameter is optional. Its default value is `true`.

If `false` is passed as a `booleanValue`, the method returns all the elements after the specified element without including the specified element.

### **subSet(e1, bv1, e2, bv2)**

The `subSet()` method returns all the elements between `e1` and `e2` including `e1`.

The `bv1` and `bv2` are optional parameters. The default value of `bv1` is `true`, and the default value of `bv2` is `false`.

If `false` is passed as `bv1`, the method returns all the elements between `e1` and `e2` without including `e1`.

If `true` is passed as `bv2`, the method returns all the elements between `e1` and `e2`, including `e1`.

### **Methods for Navigation**

The `NavigableSet` provides various methods that can be used to navigate over its elements.

**`descendingSet()`** - reverses the order of elements in a set

**`descendingIterator()`** - returns an iterator that can be used to iterate over a set in reverse order

**`ceiling()`** - returns the lowest element among those elements that are greater than or equal to the specified element

**`floor()`** - returns the greatest element among those elements that are less than or equal to the specified element

**`higher()`** - returns the lowest element among those elements that are greater than the specified element

**`lower()`** - returns the greatest element among those elements that are less than the specified element

**`pollFirst()`** - returns and removes the first element from the set

**`pollLast()`** - returns and removes the last element from the set

### **Implementation of NavigableSet in TreeSet Class**

```
import java.util.NavigableSet;

import java.util.TreeSet;

class Main {

    public static void main(String[] args) {

        // Creating NavigableSet using the TreeSet
        NavigableSet<Integer> numbers = new TreeSet<>();

        // Insert elements to the set
        numbers.add(1);
        numbers.add(2);
```

```
numbers.add(3);
System.out.println("NavigableSet: " + numbers);
// Access the first element
int firstElement = numbers.first();
System.out.println("First Number: " + firstElement);
// Access the last element
int lastElement = numbers.last();
System.out.println("Last Element: " + lastElement);
// Remove the first element
int number1 = numbers.pollFirst();
System.out.println("Removed First Element: " + number1);
// Remove the last element
int number2 = numbers.pollLast();
System.out.println("Removed Last Element: " + number2);
}
}
```

### Output

```
NavigableSet: [1, 2, 3]
First Element: 1
Last Element: 3
Removed First Element: 1
Removed Last Element: 3
```

### **3.TreeSet**

TreeSet class which is implemented in the collections framework and implementation of the SortedSet Interface and SortedSet extends Set Interface.

It behaves like a simple set with the exception that it stores elements in a sorted format.

TreeSet uses a **tree data structure** for storage. Objects are stored in sorted, ascending order.

But we can iterate in descending order using the method **TreeSet.descendingIterator()**.

**Example:**

```
// Java program Demonstrating Creation of Set object

// Using the LinkedHashSet class

import java.util.*;

class TreeSetDemo {

    public static void main(String[] args)

    {

        // Creating object of Set of type String

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

        // Adding elements into the HashSet using add() method

        ts.add("B");

        ts.add("D");

        ts.add("A");

        ts.add("C");

        ts.add("F");

        ts.add("E");

        ts.add("I");

        ts.add("G");

        ts.add("F");

        // Adding the duplicate element

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

```

        // Displaying the HashSet
        System.out.println(ts);

// Removing items from HashSet using remove() method

        ts.remove("A");

        System.out.println("Set after removing "+ "A:" + ts);

        // Iterating over hash set items

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

        // Iterating through iterators

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

        // It holds true till there is a single element

        // remaining in the object

        while (itr.hasNext())

        {

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

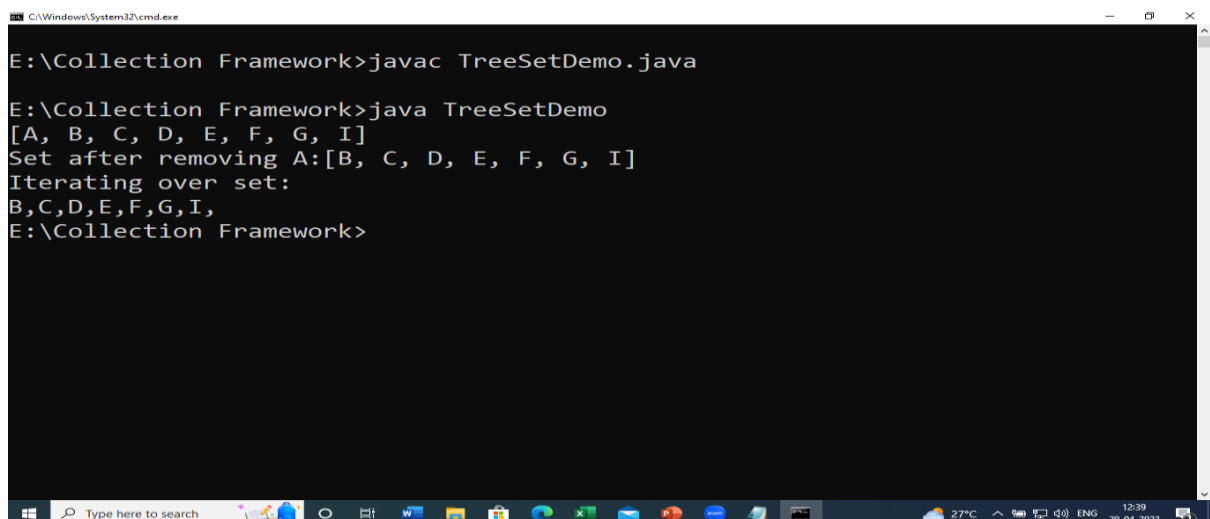
        }

    }

}

```

### **Output:**



```

C:\Windows\System32\cmd.exe
E:\Collection Framework>javac TreeSetDemo.java

E:\Collection Framework>java TreeSetDemo
[A, B, C, D, E, F, G, I]
Set after removing A:[B, C, D, E, F, G, I]
Iterating over set:
B,C,D,E,F,G,I,
E:\Collection Framework>

```

## **Cursors of Collection Framework in Java:**

Cursors are mainly used to access the elements of any collection. We have the following three types of cursors in Collection Framework:

**Iterator**

**ListIterator**

**Enumeration**

## **Iterator Cursors of Collection Framework in Java**

This cursor is used to access the elements in the forward direction only.

This cursor can be applied to any Collection Interfaces. While accessing the methods we can also delete the elements.

An iterator is an interface and we cannot create an object directly. If we want to create an object for Iterator we have to use the iterator() method.

### **Creation of Iterator**

**Syntax : Iterator it = c.iterator();**

Here, the iterator() method internally creates and returns an object of a class that implements the Iterator Interface.

Methods of Iterator:

**boolean hasNext():** returns true if there is a next element in the array list.

**Object next():** returns the next element in the array list.

**void remove():** removes an element from an array list.

Sample Program to demonstrate Iterator Cursors in Java

### **Example:**

```
import java.util.*;
public class IteratorDemo
{
    public static void main (String[]args)
    {
        LinkedList < Integer > ll = new LinkedList < Integer > ();
        ll.add (10);
        ll.add (25);
        ll.add (50);
        ll.add (20);
        ll.add (25);
        ll.add (23);
        ll.add (60);
        ll.add (25);
        ll.add (30);
        ll.add (40);
        ll.add (15);
        ll.add (25);
        System.out.println (ll);
        Iterator it = ll.descendingIterator ();
        while (it.hasNext ())
        {
            System.out.println (it.next ());
        }
        Iterator it1 = ll.descendingIterator ();
```



```

while (it1.hasNext ())
{
Integer e = (Integer) it1.next (); //down casting
if (e == 25)
{
it1.remove ();
}
}
System.out.println (ll);
}
}

```

**Output:**

```

[10, 25, 50, 20, 25, 23, 60, 25, 30, 40, 15, 25]
25
15
40
30
25
60
23
25
20
50
25
10
[10, 50, 20, 23, 60, 30, 40, 15]

```

### **ListIterator Cursors of Collection Framework in Java**

This cursor is used to access the elements of Collection in both forward and backward directions.

This cursor can be applied only for List category Collections. While accessing the methods we can also add, set, delete elements.

ListIterator is an interface and we cannot create Object directly. If we want to create an object for ListIterator we have to use ListIterator() method.

#### **Creation of ListIterator**

**Syntax : ListIterator it = l.listIterator();**

Here, listIterator() method internally creates and returns an Object of a class that implements the ListIterator Interface.

#### **Methods of ListIterator:**

**boolean hasNext():** return true if the given list iterator contains more number of elements during traversing the given list in the forward direction.

**Object next():** return the next element in the given list. This method is used to iterate through the list.

**boolean hasPrevious():** is used to retrieve and remove the head of the deque.

**Object previous():** return the previous element from the list and moves the cursor in a backward position. The above method can be used to iterate the list in a backward direction.

**int nextIndex():** return the index of the element which is returned by the next() method.

**int previousIndex():** return the index of the given element which is returned by a call to previous. The method may return -1 if and only if the iterator is placed at the beginning of the list.

**void remove():** remove the last element from the list which is returned by next() or previous() method.

**void set(Object obj):** is used to replace the last element which is returned by the next() or previous() along with the given element.

**void add(Object obj):** is used to insert the given element into the specified list. The element is inserted automatically before the next element may return by the next() method.

Sample Program to demonstrate ListIterator Cursors of Collection Framework in Java

```
import java.util.LinkedList;
import java.util.ListIterator;

public class ListIteratorDemo
{
    public static void main (String[]args)
    {
        LinkedList < Integer > ll = new LinkedList < Integer > ();
        ll.add (10);
        ll.add (25);
        ll.add (50);
        ll.add (20);
        ll.add (25);
        ll.add (23);
        ll.add (40);
        ll.add (15);
        ll.add (25);
        System.out.println (ll);
        ListIterator < Integer > lit = ll.listIterator ();

        System.out.println ("Elements in Forward Direction");
        while (lit.hasNext ())
        {
            System.out.println (lit.next () + " ");
        }

        System.out.println ("\n elements in Backward direction");
        while (lit.hasPrevious ())
```

```
{
System.out.println (lit.previous () + " ");
}
while (lit.hasNext ())
{
Object o = lit.next ();
Integer e = (Integer) o;
if (e == 23)
{
    lit.add (56);
}
if (e == 15)
{
    lit.set(15000);
}
if (e == 25)
{
    lit.remove();
}
}
System.out.println ("\n New List:" + ll);
}
}
```

**Output:**

```
[10, 25, 50, 20, 25, 23, 40, 15, 25]
Elements in Forward Direction
10
25
50
20
25
23
40
15
25

elements in Backward direction
25
15
40
23
25
20
50
25
10

New List:[10, 50, 20, 23, 56, 40, 15000]
```

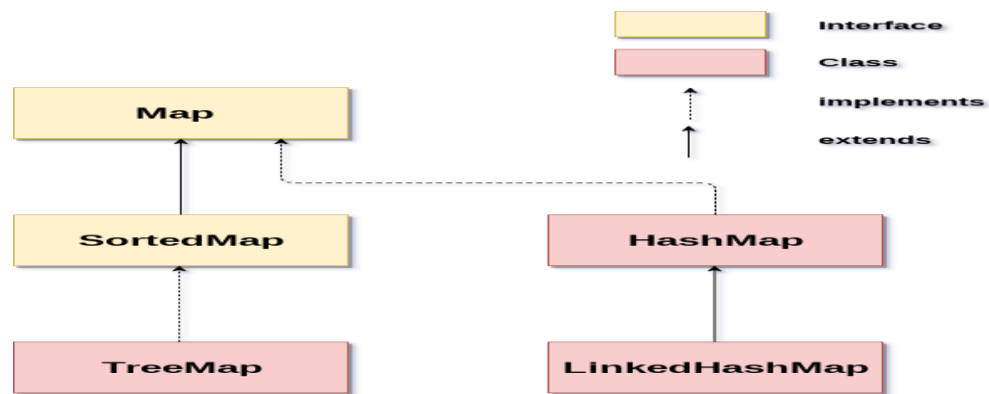
## **Java Map Interface**

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.

## **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.

A Map can't be traversed, so you need to convert it into Set using *keySet()* or *entrySet()* method.

| Class                         | Description  |
|-------------------------------|--|
| <a href="#">HashMap</a>       | HashMap is the implementation of Map, but it doesn't maintain any order.                             |
| <a href="#">LinkedHashMap</a> | LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order. |
| <a href="#">TreeMap</a>       | TreeMap is the implementation of Map and SortedMap. It maintains ascending order.                    |

### Useful methods of Map interface

| Method                          | Description   |
|---------------------------------|---|
| V put(Object key, Object value) | It is used to insert an entry in the map.   |
| void putAll(Map map)            | It is used to insert the specified map in the map.  |
| V putIfAbsent(K key, V value)   | It inserts the specified value with the specified key in the map only if it is not already specified. |

|   |  |
|---|--|
| <code>V remove(Object key)</code>   | It is used to delete an entry for the specified key.   |
| <code>boolean remove(Object key, Object value)</code>   | It removes the specified values with the associated specified keys from the map.   |
| <code>Set keySet()</code>   | It returns the Set view containing all the keys.   |
| <code>Set&lt;Map.Entry&lt;K,V&gt;&gt; entrySet()</code>   | It returns the Set view containing all the keys and values.  |
| <code>void clear()</code>   | It is used to reset the map.   |
| <code>V compute(K key, BiFunction&lt;? super K,? super V,? extends V&gt; remappingFunction)</code>          | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).   |
| <code>V computeIfAbsent(K key, Function&lt;? super K,? extends V&gt; mappingFunction)</code>                | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| <code>V computeIfPresent(K key, BiFunction&lt;? super K,? super V,? extends V&gt; remappingFunction)</code> | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.   |
| <code>boolean containsValue(Object value)</code>  | This method returns true if some value equal to the value exists within the map, else return false.  |
| <code>boolean containsKey(Object key)</code>  | This method returns true if some key equal to the key exists within the map, else return false.  |
| <code>boolean equals(Object o)</code>   | It is used to compare the specified Object with the Map.   |

|   |  |
|---|--|
| <code>void forEach(BiConsumer&lt;? super K,? super V&gt; action)</code>                                   | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.  |
| <code>V get(Object key)</code>  | This method returns the object that contains the value associated with the key.  |
| <code>V getOrDefault(Object key, V defaultValue)</code>   | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.   |
| <code>int hashCode()</code>   | It returns the hash code value for the Map   |
| <code>boolean isEmpty()</code>  | This method returns true if the map is empty; returns false if it contains at least one key.   |
| <code>V merge(K key, V value, BiFunction&lt;? super V,? super V,? extends V&gt; remappingFunction)</code> | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.                                   |
| <code>V replace(K key, V value)</code>  | It replaces the specified value for a specified key.   |
| <code>boolean replace(K key, V oldValue, V newValue)</code>   | It replaces the old value with the new value for a specified key.  |
| <code>void replaceAll(BiFunction&lt;? super K,? super V,? extends V&gt; function)</code>                  | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| <code>Collection values()</code>  | It returns a collection view of the values contained in the map.   |
| <code>int size()</code>   | This method returns the number of entries in the map.  |

## Map.Entry Interface

- Entry is the subinterface of Map. So we will be accessed it by Map.Entry name.
- It returns a collection-view of the map, whose elements are of this class.
- It provides methods to get key and value.

### Methods of Map.Entry interface

| Method   | Description   |
|--|---|
| K getKey()   | It is used to obtain a key.   |
| V getValue()   | It is used to obtain value.   |
| int hashCode()   | It is used to obtain hashCode.  |
| V setValue(V value)  | It is used to replace the value corresponding to this entry with the specified value. |
| boolean equals(Object o)   | It is used to compare the specified object with the other existing objects.           |
| static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>> comparingByKey()   | It returns a comparator that compare the objects in natural order on key.             |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super K> cmp)        | It returns a comparator that compare the objects by key using the given Comparator.   |
| static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>> comparingByValue() | It returns a comparator that compare the objects in natural order on value.           |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super V> cmp)      | It returns a comparator that compare the objects by value using the given Comparator. |



### **Example:**

```
import java.util.*;
class MapExample2{
public static void main(String args[]){
    Map<Integer,String> map=new HashMap<Integer,String>();
    map.put(100,"Amit");
    map.put(101,"Vijay");
    map.put(102,"Rahul");
    //Elements can traverse in any order
    for(Map.Entry m:map.entrySet()){
        System.out.println(m.getKey()+" "+m.getValue());
    }
}
```

## **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.

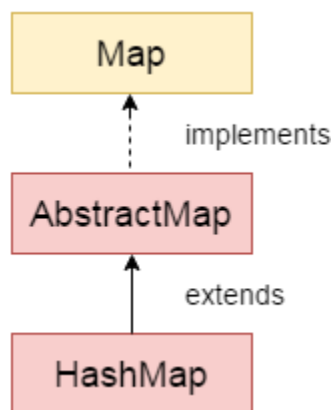
### **Points to remember**

- 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.

## Hierarchy of HashMap class

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.



## HashMap class declaration

Let's see the declaration for java.util.HashMap class.

**public class** HashMap<K,V> **extends** AbstractMap<K,V> **implements** Map<K,V>, Cloneable, Serializable

## HashMap class Parameters

Let's see the Parameters for java.util.HashMap class.

- **K**: It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.

## Constructors of Java HashMap class

| Constructor                             | Description  |
|---|--|
| HashMap()                               | It is used to construct a default HashMap.   |
| HashMap(Map<? extends K,? extends V> m) | It is used to initialize the hash map by using the elements of the given Map object m.       |
| HashMap(int capacity)                   | It is used to initializes the capacity of the hash map to the given integer value, capacity. |

|   |  |
|---|--|
| HashMap(int capacity, float loadFactor) | It is used to initialize both the capacity and load factor of the hash map by using its arguments. |
|---|--|

## **Methods of Java HashMap class**

| Method                                   | Description  |
|--|--|
| void clear()                             | It is used to remove all of the mappings from this map.  |
| boolean isEmpty()                        | It is used to return true if this map contains no key-value mappings.  |
| Object clone()                           | It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| Set entrySet()                           | It is used to return a collection view of the mappings contained in this map.                                |
| Set keySet()                             | It is used to return a set view of the keys contained in this map.   |
| V put(Object key, Object value)          | It is used to insert an entry in the map.  |
| void putAll(Map map)                     | It is used to insert the specified map in the map.   |
| V putIfAbsent(K key, V value)            | It inserts the specified value with the specified key in the map only if it is not already specified.        |
| V remove(Object key)                     | It is used to delete an entry for the specified key.   |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map.                             |

|  |  |
|--|--|
| V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)          | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).   |
| V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)                | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.   |
| boolean containsValue(Object value)  | This method returns true if some value equal to the value exists within the map, else return false.  |
| boolean containsKey(Object key)  | This method returns true if some key equal to the key exists within the map, else return false.  |
| boolean equals(Object o)   | It is used to compare the specified Object with the Map.   |
| void forEach(BiConsumer<? super K,? super V> action)                                     | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.  |
| V get(Object key)  | This method returns the object that contains the value associated with the key.  |
| V getOrDefault(Object key, V defaultValue)   | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.   |
| boolean isEmpty()  | This method returns true if the map is empty; returns false if it contains at least one key.   |

|   |  |
|---|--|
| <code>V merge(K key, V value, BiFunction&lt;? super V,? super V,? extends V&gt; remappingFunction)</code> | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.                                   |
| <code>V replace(K key, V value)</code>  | It replaces the specified value for a specified key.   |
| <code>boolean replace(K key, V oldValue, V newValue)</code>   | It replaces the old value with the new value for a specified key.  |
| <code>void replaceAll(BiFunction&lt;? super K,? super V,? extends V&gt; function)</code>                  | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| <code>Collection&lt;V&gt; values()</code>   | It returns a collection view of the values contained in the map.   |
| <code>int size()</code>   | This method returns the number of entries in the map.  |

## Java HashMap Example

Let's see a simple example of HashMap to store key and value pair.

```
import java.util.*;
public class HashMapExample1{
public static void main(String args[]){
HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
map.put(1,"Mango"); //Put elements in Map
map.put(2,"Apple");
map.put(3,"Banana");
map.put(4,"Grapes");

System.out.println("Iterating Hashmap...");
for(Map.Entry m : map.entrySet()){
System.out.println(m.getKey()+" "+m.getValue());
```

```
}  
}  
}
```

Iterating Hashmap...

```
1 Mango  
2 Apple  
3 Banana  
4 Grapes
```

- In this example, we are storing Integer as the key and String as the value, so we are using `HashMap<Integer,String>` as the type. The `put()` method inserts the elements in the map.
- To get the key and value elements, we should call the `getKey()` and `getValue()` methods. The `Map.Entry` interface contains the `getKey()` and `getValue()` methods. But, we should call the `entrySet()` method of Map interface to get the instance of Map.Entry.

## No Duplicate Key on HashMap

You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the value.

```
import java.util.*;  
  
public class HashMapExample2{  
    public static void main(String args[]){  
        HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating  
        HashMap  
        map.put(1,"Mango"); //Put elements in Map  
        map.put(2,"Apple");  
        map.put(3,"Banana");  
        map.put(1,"Grapes"); //trying duplicate key  
  
        System.out.println("Iterating Hashmap...");  
        for(Map.Entry m : map.entrySet()){  
            System.out.println(m.getKey()+" "+m.getValue());  
        }  
    }  
}
```

**Output:**

Iterating Hashmap...

```
1 Grapes  
2 Apple  
3 Banana
```

## Java HashMap example to remove() elements

Here, we see different ways to remove elements.

```
import java.util.*;  
  
public class HashMap2 {  
    public static void main(String args[]) {  
        HashMap<Integer,String> map=new HashMap<Integer,String>();  
        map.put(100,"Amit");  
        map.put(101,"Vijay");  
        map.put(102,"Rahul");  
        map.put(103, "Gaurav");  
        System.out.println("Initial list of elements: "+map);  
        //key-based removal  
        map.remove(100);  
        System.out.println("Updated list of elements: "+map);  
        //value-based removal  
        map.remove(101);  
        System.out.println("Updated list of elements: "+map);  
        //key-value pair based removal  
        map.remove(102, "Rahul");  
        System.out.println("Updated list of elements: "+map);  
    }  
}
```

### Output:

```
Initial list of elements: {100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav}  
Updated list of elements: {101=Vijay, 102=Rahul, 103=Gaurav}  
Updated list of elements: {102=Rahul, 103=Gaurav}  
Updated list of elements: {103=Gaurav}
```

## Java HashMap example to replace() elements

Here, we see different ways to replace elements.

```
import java.util.*;
```

```

class HashMap3{
public static void main(String args[]){
HashMap<Integer,String> hm=new HashMap<Integer,String>();
    hm.put(100,"Amit");
    hm.put(101,"Vijay");
    hm.put(102,"Rahul");
    System.out.println("Initial list of elements:");
for(Map.Entry m:hm.entrySet())
{
    System.out.println(m.getKey()+" "+m.getValue());
}
System.out.println("Updated list of elements:");
hm.replace(102, "Gaurav");
for(Map.Entry m:hm.entrySet())
{
    System.out.println(m.getKey()+" "+m.getValue());
}
System.out.println("Updated list of elements:");
hm.replace(101, "Vijay", "Ravi");
for(Map.Entry m:hm.entrySet())
{
    System.out.println(m.getKey()+" "+m.getValue());
}
System.out.println("Updated list of elements:");
hm.replaceAll((k,v) -> "Ajay");
for(Map.Entry m:hm.entrySet())
{
    System.out.println(m.getKey()+" "+m.getValue());
}
}
}
}

```

### **Output:**

```

Initial list of elements:
100 Amit
101 Vijay
102 Rahul
Updated list of elements:
100 Amit

```



```
101 Vijay
102 Gaurav
Updated list of elements:
100 Amit
101 Ravi
102 Gaurav
Updated list of elements:
100 Ajay
101 Ajay
102 Ajay
```

### Difference between HashSet and HashMap

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

### Java HashMap Example: Book

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

public class MapExample {
    public static void main(String[] args) {
        //Creating map of Books
        Map<Integer,Book> map=new HashMap<Integer,Book>();
        //Creating Books
        Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
        Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
        Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
        //Adding Books to map
```

```
map.put(1,b1);
map.put(2,b2);
map.put(3,b3);
```

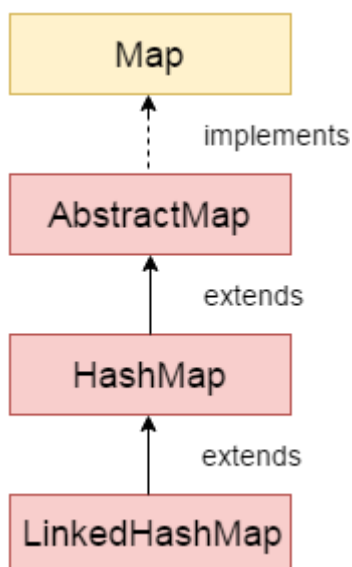
```
//Traversing map
```

```
for(Map.Entry<Integer, Book> entry:map.entrySet()){
    int key=entry.getKey();
    Book b=entry.getValue();
    System.out.println(key+" Details:");
    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.q
    uantity);
}
}
}
```

Output:

```
1 Details:
101 Let us C Yashwant Kanetkar BPB 8
2 Details:
102 Data Communications and Networking Forouzan Mc Graw Hill 4
3 Details:
103 Operating System Galvin Wiley 6
```

Java LinkedHashMap class



Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

### Points to remember

Java LinkedHashMap contains values based on the key.

Java LinkedHashMap contains unique elements.

Java LinkedHashMap may have one null key and multiple null values.

Java LinkedHashMap is non synchronized.

Java LinkedHashMap maintains insertion order.

The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

### LinkedHashMap class declaration

Let's see the declaration for java.util.LinkedHashMap class.

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

### LinkedHashMap class Parameters

| Method                              | Description  |
|-------------------------------------|--|
| V get(Object key)                   | It returns the value to which the specified key is mapped.               |
| void clear()                        | It removes all the key-value pairs from a map.                           |
| boolean containsValue(Object value) | It returns true if the map maps one or more keys to the specified value. |
| Set<Map.Entry<K,V>> entrySet()      | It returns a Set view of the mappings contained in the map.              |
| void forEach(BiConsumer<? super K,? | It performs the given action for each                                    |

|  |  |
|--|--|
| <code>super V&gt; action)</code>   | entry in the map until all entries have been processed or the action throws an exception.  |
| <code>V getOrDefault(Object key, V defaultValue)</code>                                  | It returns the value to which the specified key is mapped or <code>defaultValue</code> if this map contains no mapping for the key.                                    |
| <code>Set&lt;K&gt; keySet()</code>   | It returns a Set view of the keys contained in the map   |
| <code>protected boolean removeEldestEntry(Map.Entry&lt;K,V&gt; eldest)</code>            | It returns true on removing its eldest entry.  |
| <code>void replaceAll(BiFunction&lt;? super K,? super V,? extends V&gt; function)</code> | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| <code>Collection&lt;V&gt; values()</code>  | It returns a Collection view of the values contained in this map.  |

Let's see the Parameters for `java.util.LinkedHashMap` class.

**K:** It is the type of keys maintained by this map.

**V:** It is the type of mapped values.

### Constructors of Java LinkedHashMap class

| Constructor                                    | Description  |
|--|--|
| <code>LinkedHashMap()</code>                   | It is used to construct a default <code>LinkedHashMap</code> .                 |
| <code>LinkedHashMap(int capacity)</code>       | It is used to initialize a <code>LinkedHashMap</code> with the given capacity. |
| <code>LinkedHashMap(int capacity, float</code> | It is used to initialize both the capacity                                     |

|  |  |
|--|--|
| loadFactor)  | and the load factor.   |
| LinkedHashMap(int capacity, float loadFactor, boolean accessOrder) | It is used to initialize both the capacity and the load factor with specified ordering mode. |
| LinkedHashMap(Map<? extends K,? extends V> m)                      | It is used to initialize the LinkedHashMap with the elements from the given Map class m.     |

### Methods of Java LinkedHashMap class

| Method  | Description  |
|---|--|
| V get(Object key)   | It returns the value to which the specified key is mapped.   |
| void clear()  | It removes all the key-value pairs from a map.   |
| boolean containsValue(Object value)                                   | It returns true if the map maps one or more keys to the specified value.   |
| Set<Map.Entry<K,V>> entrySet()  | It returns a Set view of the mappings contained in the map.  |
| void forEach(BiConsumer<? super K,? super V> action)                  | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.          |
| V getOrDefault(Object key, V defaultValue)                            | It returns the value to which the specified key is mapped or defaultValue if this map contains no mapping for the key.                   |
| Set<K> keySet()   | It returns a Set view of the keys contained in the map   |
| protected boolean removeEldestEntry(Map.Entry<K,V> eldest)            | It returns true on removing its eldest entry.  |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the |

|                        |   |
|------------------------|---|
|                        | function throws an exception.                                     |
| Collection<V> values() | It returns a Collection view of the values contained in this map. |

### Java LinkedHashMap Example

```
import java.util.*;
class LinkedHashMap1{
    public static void main(String args[]){

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

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

        for(Map.Entry m:hm.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

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

### Java LinkedHashMap Example: Key-Value pair

```
import java.util.*;
class LinkedHashMap2{
    public static void main(String args[]){
        LinkedHashMap<Integer, String> map = new LinkedHashMap<Integer, Str
ing>();
        map.put(100,"Amit");
        map.put(101,"Vijay");
        map.put(102,"Rahul");
        //Fetching key
        System.out.println("Keys: "+map.keySet());
        //Fetching value
```

```

        System.out.println("Values: "+map.values());
        //Fetching key-value pair
        System.out.println("Key-Value pairs: "+map.entrySet());
    }
}

```

Keys: [100, 101, 102]  
 Values: [Amit, Vijay, Rahul]  
 Key-Value pairs: [100=Amit, 101=Vijay, 102=Rahul]

### Java LinkedHashMap Example:remove()

```

import java.util.*;
public class LinkedHashMap3 {
    public static void main(String args[]) {
        Map<Integer,String> map=new LinkedHashMap<Integer,String>();
        map.put(101,"Amit");
        map.put(102,"Vijay");
        map.put(103,"Rahul");
        System.out.println("Before invoking remove() method: "+map);
        map.remove(102);
        System.out.println("After invoking remove() method: "+map);
    }
}

```

Output:

Before invoking remove() method: {101=Amit, 102=Vijay, 103=Rahul}  
 After invoking remove() method: {101=Amit, 103=Rahul}

### Java LinkedHashMap Example: Book

```

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

```

```

}
}
public class MapExample {
public static void main(String[] args) {
    //Creating map of Books
    Map<Integer,Book> map=new LinkedHashMap<Integer,Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","
    Mc Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to map
    map.put(2,b2);
    map.put(1,b1);
    map.put(3,b3);

    //Traversing map
    for(Map.Entry<Integer, Book> entry:map.entrySet()){
        int key=entry.getKey();
        Book b=entry.getValue();
        System.out.println(key+" Details:");
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b
        .quantity);
    }
}
}

```

### **Output:**

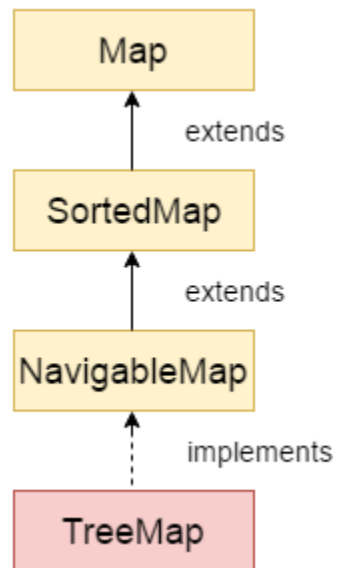
```

2 Details:
102 Data Communications & Networking Forouzan Mc Graw Hill 4
1 Details:
101 Let us C Yashwant Kanetkar BPB 8
3 Details:
103 Operating System Galvin Wiley 6

```

### **Java TreeMap class**





Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

The important points about Java TreeMap class are:

- Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.

| Method  | Description  |
|---|--|
| Map.Entry<K,V> ceilingEntry(K key)                    | It returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key. |
| K ceilingKey(K key)                                   | It returns the least key, greater than the specified key or null if there is no such key.  |
| void clear()  | It removes all the key-value pairs from a map.   |
| Object clone()  | It returns a shallow copy of TreeMap instance.   |
| Comparator<? super K> comparator()                    | It returns the comparator that arranges the key in order, or null if the map uses the natural ordering.                          |
| NavigableSet<K> descendingKeySet()                    | It returns a reverse order NavigableSet view of the keys contained in the map.   |
| NavigableMap<K,V> descendingMap()                     | It returns the specified key-value pairs in descending order.  |
| Map.Entry firstEntry()                                | It returns the key-value pair having the least key.  |
| Map.Entry<K,V> floorEntry(K key)                      | It returns the greatest key, less than or equal to the specified key, or null if there is no such key.                           |
| void forEach(BiConsumer<? super K,? super V> action)  | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.  |
| SortedMap<K,V> headMap(K toKey)                       | It returns the key-value pairs whose keys are strictly less than toKey.  |
| NavigableMap<K,V> headMap(K toKey, boolean inclusive) | It returns the key-value pairs whose keys are less than (or equal to if inclusive is   |

|   |  |
|---|--|
|   | true) toKey.   |
| Map.Entry<K,V> higherEntry(K key)             | It returns the least key strictly greater than the given key, or null if there is no such key.                                     |
| K higherKey(K key)                            | It is used to return true if this map contains a mapping for the specified key.  |
| Set keySet()                                  | It returns the collection of keys exist in the map.  |
| Map.Entry<K,V> lastEntry()                    | It returns the key-value pair having the greatest key, or null if there is no such key.  |
| Map.Entry<K,V> lowerEntry(K key)              | It returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key. |
| K lowerKey(K key)                             | It returns the greatest key strictly less than the given key, or null if there is no such key.                                     |
| NavigableSet<K><br>navigableKeySet()          | It returns a NavigableSet view of the keys contained in this map.  |
| Map.Entry<K,V> pollFirstEntry()               | It removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.                 |
| Map.Entry<K,V> pollLastEntry()                | It removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.              |
| V put(K key, V value)                         | It inserts the specified value with the specified key in the map.  |
| void putAll(Map<? extends K,? extends V> map) | It is used to copy all the key-value pair from one map to another map.   |

|  |  |
|--|--|
| V replace(K key, V value)  | It replaces the specified value for a specified key.   |
| boolean replace(K key, V oldValue, V newValue)   | It replaces the old value with the new value for a specified key.  |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function)                    | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive) | It returns key-value pairs whose keys range from fromKey to toKey.   |
| SortedMap<K,V> subMap(K fromKey, K toKey)  | It returns key-value pairs whose keys range from fromKey, inclusive, to toKey, exclusive.  |
| SortedMap<K,V> tailMap(K fromKey)  | It returns key-value pairs whose keys are greater than or equal to fromKey.  |
| NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)                                  | It returns key-value pairs whose keys are greater than (or equal to, if inclusive is true) fromKey.  |
| boolean containsKey(Object key)  | It returns true if the map contains a mapping for the specified key.   |
| boolean containsValue(Object value)  | It returns true if the map maps one or more keys to the specified value.   |
| K firstKey()   | It is used to return the first (lowest) key currently in this sorted map.  |
| V get(Object key)  | It is used to return the value to which the map maps the specified key.  |
| K lastKey()  | It is used to return the last (highest) key  |

|                                |   |
|--------------------------------|---|
|                                | currently in the sorted map.                                      |
| V remove(Object key)           | It removes the key-value pair of the specified key from the map.  |
| Set<Map.Entry<K,V>> entrySet() | It returns a set view of the mappings contained in the map.       |
| int size()                     | It returns the number of key-value pairs exists in the hashtable. |
| Collection values()            | It returns a collection view of the values contained in the map.  |

- Java TreeMap contains only unique elements.
- Java TreeMap cannot have a null key but can have multiple null values.
- Java TreeMap is non synchronized.
- Java TreeMap maintains ascending order.

### TreeMap class declaration

Let's see the declaration for java.util.TreeMap class.

1. **public class** TreeMap<K,V> **extends** AbstractMap<K,V> **implements** NavigableMap<K,V>, Cloneable, Serializable

### TreeMap class Parameters

Let's see the Parameters for java.util.TreeMap class.

- **K:** It is the type of keys maintained by this map.
- **V:** It is the type of mapped values.

### Constructors of Java TreeMap class

| Constructor                | Description   |
|----------------------------|---|
| TreeMap()                  | It is used to construct an empty tree map that will be sorted using the natural order of its key. |
| TreeMap(Comparator<? super | It is used to construct an empty tree-based map that  |

|  |  |
|--|--|
| K> comparator)                           | will be sorted using the comparator comp.  |
| TreeMap(Map<? extends K, ? extends V> m) | It is used to initialize a treemap with the entries from <b>m</b> , which will be sorted using the natural order of the keys.            |
| TreeMap(SortedMap<K, ? extends V> m)     | It is used to initialize a treemap with the entries from the SortedMap <b>sm</b> , which will be sorted in the same order as <b>sm</b> . |

## Methods of Java TreeMap class

### Java TreeMap Example

```
import java.util.*;
class TreeMap1{
    public static void main(String args[]){
        TreeMap<Integer,String> map=new TreeMap<Integer,String>();
        map.put(100,"Amit");
        map.put(102,"Ravi");
        map.put(101,"Vijay");
        map.put(103,"Rahul");

        for(Map.Entry m:map.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

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

### Java TreeMap Example: remove()

```
import java.util.*;
public class TreeMap2 {
    public static void main(String args[]) {
        TreeMap<Integer,String> map=new TreeMap<Integer,String>();
        map.put(100,"Amit");
        map.put(102,"Ravi");
        map.put(101,"Vijay");
        map.put(103,"Rahul");
```

```

        System.out.println("Before invoking remove() method");
        for(Map.Entry m:map.entrySet())
        {
            System.out.println(m.getKey()+" "+m.getValue());
        }
        map.remove(102);
        System.out.println("After invoking remove() method");
        for(Map.Entry m:map.entrySet())
        {
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}

```

Output:

```

Before invoking remove() method
100 Amit
101 Vijay
102 Ravi
103 Rahul
After invoking remove() method
100 Amit
101 Vijay
103 Rahul

```

### Java TreeMap Example: NavigableMap

```

import java.util.*;
class TreeMap3{
    public static void main(String args[]){
        NavigableMap<Integer,String> map=new TreeMap<Integer,String>();
        map.put(100,"Amit");
        map.put(102,"Ravi");
        map.put(101,"Vijay");
        map.put(103,"Rahul");
        //Maintains descending order
        System.out.println("descendingMap: "+map.descendingMap());
        //Returns key-
value pairs whose keys are less than or equal to the specified key.
        System.out.println("headMap: "+map.headMap(102,true));
    }
}

```

```

//Returns key-
value pairs whose keys are greater than or equal to the specified key.
System.out.println("tailMap: "+map.tailMap(102,true));
//Returns key-value pairs exists in between the specified key.
System.out.println("subMap: "+map.subMap(100, false, 102, true));
}
}
descendingMap: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}
headMap: {100=Amit, 101=Vijay, 102=Ravi}
tailMap: {102=Ravi, 103=Rahul}
subMap: {101=Vijay, 102=Ravi}

```

### Java TreeMap Example: SortedMap

```

import java.util.*;
class TreeMap4{
public static void main(String args[]){
SortedMap<Integer,String> map=new TreeMap<Integer,String>();
map.put(100,"Amit");
map.put(102,"Ravi");
map.put(101,"Vijay");
map.put(103,"Rahul");
//Returns key-value pairs whose keys are less than the specified key.
System.out.println("headMap: "+map.headMap(102));
//Returns key-
value pairs whose keys are greater than or equal to the specified key.
System.out.println("tailMap: "+map.tailMap(102));
//Returns key-value pairs exists in between the specified key.
System.out.println("subMap: "+map.subMap(100, 102));
}
}
headMap: {100=Amit, 101=Vijay}
tailMap: {102=Ravi, 103=Rahul}
subMap: {100=Amit, 101=Vijay}

```

### What is difference between HashMap and TreeMap?

| HashMap                              | TreeMap                              |
|--------------------------------------|--------------------------------------|
| 1) HashMap can contain one null key. | TreeMap cannot contain any null key. |



2) HashMap maintains no order.

TreeMap maintains ascending order.

### Java TreeMap Example: Book

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

public class MapExample {
    public static void main(String[] args) {
        //Creating map of Books
        Map<Integer,Book> map=new TreeMap<Integer,Book>();
        //Creating Books
        Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPP",8);
        Book b2=new Book(102,"Data Communications & Networking","Forouzan",
"Mc Graw Hill",4);
        Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
        //Adding Books to map
        map.put(2,b2);
        map.put(1,b1);
        map.put(3,b3);

        //Traversing map
        for(Map.Entry<Integer, Book> entry:map.entrySet()){
            int key=entry.getKey();
            Book b=entry.getValue();
            System.out.println(key+" Details:");
```

```
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.  
quantity);  
    }  
}  
}
```

Output:

```
1 Details:  
101 Let us C Yashwant Kanetkar BPB 8  
2 Details:  
102 Data Communications & Networking Forouzan Mc Graw Hill 4  
3 Details:  
103 Operating System Galvin Wiley 6
```