

Day 7- Facilitation Guide

Index

- I. Stack
- II. Queue
- III. Set Interface introduction & Implementation classes (HashSet)

For (1.5 hrs) ILT

During the previous session about the List interface in Java, we covered a variety of important concepts:

The List interface is a fundamental part of the Java Collection Framework, offering an ordered collection that allows duplicate elements. It extends the Collection interface and introduces methods for indexing and manipulating elements in a sequence. ArrayList and LinkedList are common implementations of the List interface, providing different trade-offs between fast element access and efficient insertions/deletions. Lists maintain the insertion order of elements, and their methods enable operations like adding, removing, retrieving elements by index, and checking for the presence of certain elements.

In this session, we will explore Stack, Queue and the Set interface along with its diverse implementation classes.

Trainer will ask questions to students.

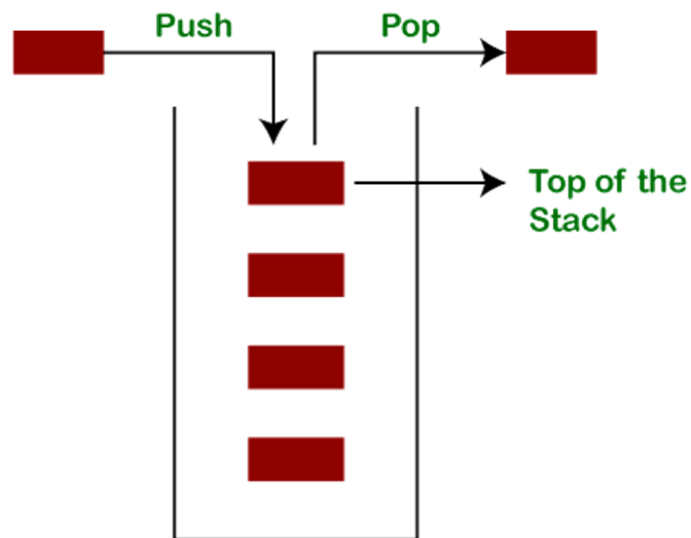
Are you familiar with the concept of stacks in computer science and programming?

If yes, could you briefly explain what a stack is and how it works?

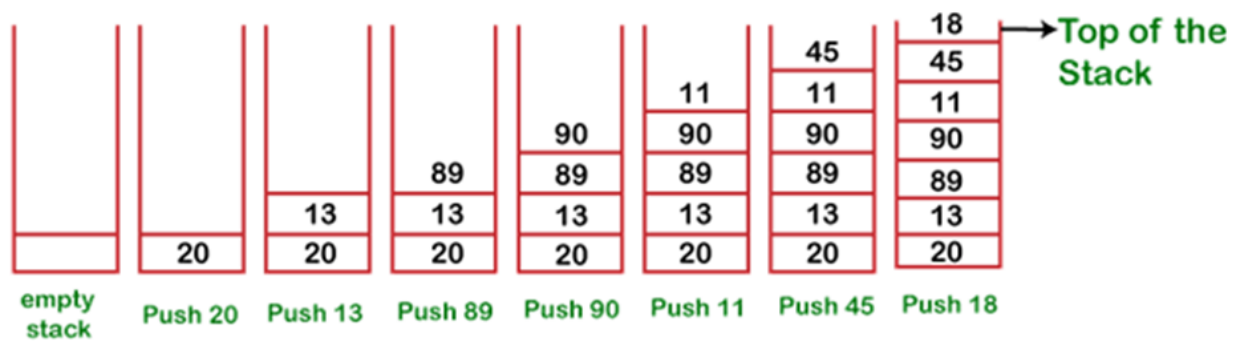
I. Stack

Stack is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out** (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the **Stack class** that provides different operations such as push, pop, search, etc.

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

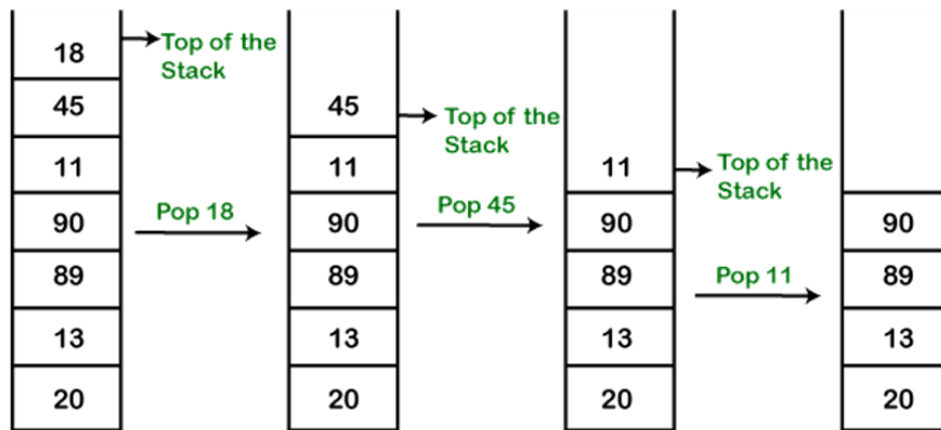


Let's push 20, 13, 89, 90, 11, 45, 18, respectively into the Stack.



Push operation

Let's remove (pop) 18, 45, and 11 from the Stack.



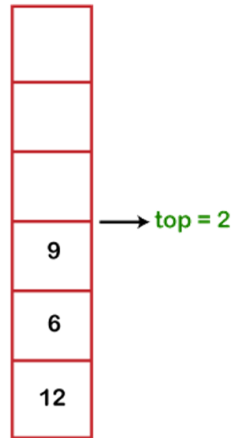
Pop operation

Empty Stack: If the Stack has no element is known as an **empty Stack**. When the Stack is empty the value of the top variable is -1.

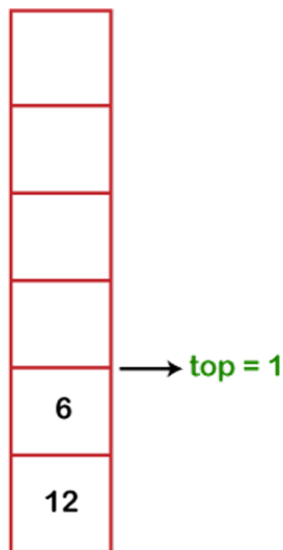


When we push an element into the Stack the top is **increased by 1**. In the following figure,

- Push 12, top=0
- Push 6, top=1
- Push 9, top=2



When we pop an element from the Stack the value of top is **decreased by 1**. In the following figure, we have popped 9.

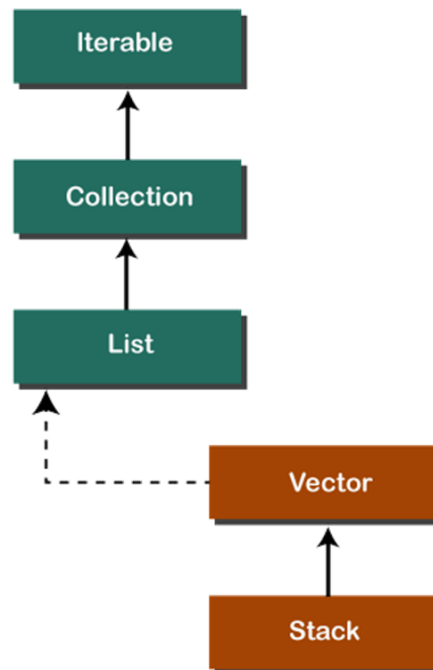


The following table shows the different values of the top.

Top value	Meaning
-1	It shows the stack is empty.
0	The stack has only an element.
N-1	The stack is full.
N	The stack is overflow.

Stack Class

In Java, **Stack** is a class that falls under the Collection Framework that extends the **Vector** class. It also implements interfaces **List**, **Collection**, **Iterable**, **Cloneable**, **Serializable**. It represents the LIFO Stack of objects. Before using the Stack class, we must import the `java.util` package. The Stack class is arranged in the Collections framework hierarchy, as shown below.



Stack Class Constructor

The Stack class contains only the **default constructor** that creates an empty Stack.

1. `public Stack()`

Creating a Stack

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

1. `Stack stack= new Stack();`

Or

1. `Stack<type> stack= new Stack<>();`

Where **type** denotes the type of Stack like Integer, String, etc.

Methods of Stack

public boolean empty(): It is used for returns true provided Stack is empty. It returns false in case Stack is non-empty.

public void push (Object): It is used for inserting the elements into the Stack.

· **public Object pop():** It is used for removing Top Most elements from the Stack.

· **public Object peek():** It is used for retrieving the Top Most element from the Stack.

· **public int search(Object):** It is used for searching an element in the Stack. If the element is found then it returns Stack relative position of that element otherwise it returns -1, -1 indicates search is unsuccessful and element is not found.

Public push(): The method inserts an item onto the top of the stack. It works the same as the method `addElement(item)` method of the Vector class. It passes a parameter item to be pushed into the stack.

Example:

```
import java.util.Stack;

public class StackEmptyMethodExample
{
    public static void main(String[] args)
    {
        //creating an instance of Stack class
        Stack<Integer> stack= new Stack<>();

        // checking stack is empty or not
        boolean result = stack.empty();
```

```

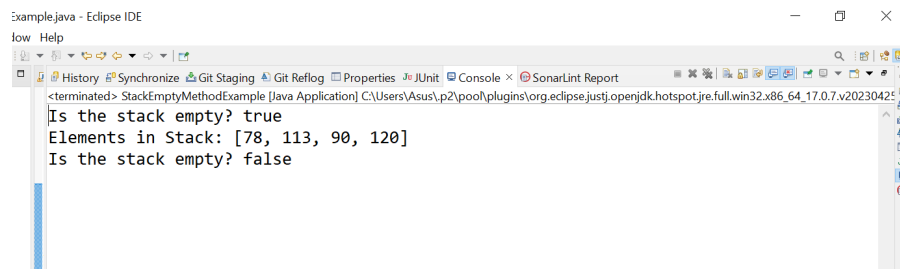
        System.out.println("Is the stack empty? " + result);

        // pushing elements into stack
        stack.push(78);
        stack.push(113);
        stack.push(90);
        stack.push(120);

        //prints elements of the stack
        System.out.println("Elements in Stack: " + stack);
        result = stack.empty();
        System.out.println("Is the stack empty? " + result);
    }
}

```

Output:



The screenshot shows the Eclipse IDE interface with the console window open. The console output displays the following text:

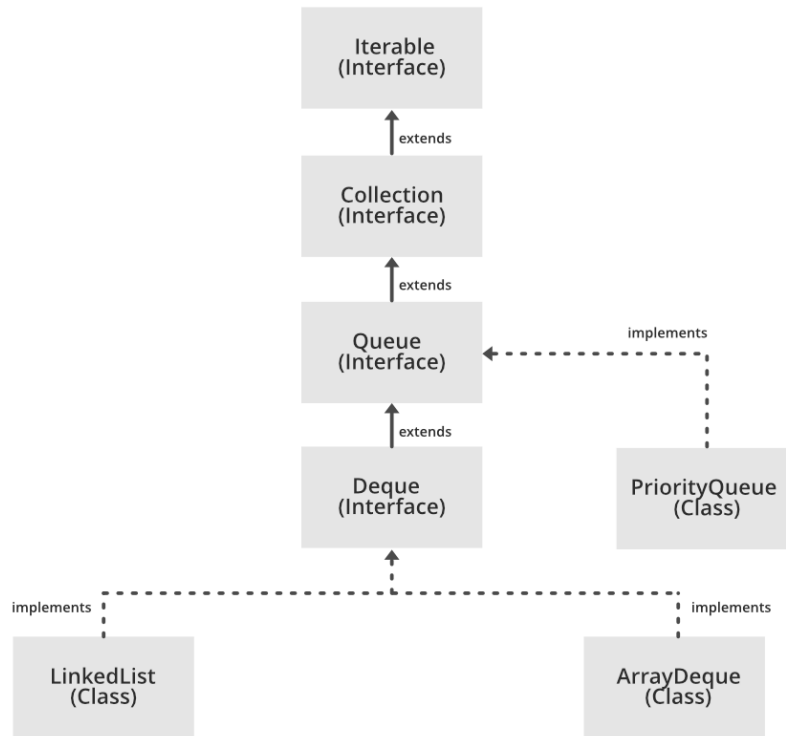
```

<terminated> StackEmptyMethodExample [Java Application] C:\Users\Asus\AppData\Local\Temp\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425\jre\bin\java.exe
Is the stack empty? true
Elements in Stack: [78, 113, 90, 120]
Is the stack empty? false

```

II. Queue

The Queue interface in Java represents a collection of elements in which elements are processed in a First-In-First-Out (FIFO) order. This means that the element that was added first will be the first one to be removed. The Queue interface is part of the Java Collections Framework and extends the Collection interface.



Methods for Basic Operations:

- **add(element):** Adds the specified element to the queue if space is available.
- **offer(element):** Adds the specified element to the queue if possible, returning true if successful.
- **remove():** Removes and returns the element at the front of the queue.
- **poll():** Removes and returns the element at the front of the queue, or returns null if the queue is empty.
- **element():** Returns the element at the front of the queue without removing it.
- **peek():** Returns the element at the front of the queue without removing it, or returns null if the queue is empty.

Implementations of the Queue Interface: There are several classes that implement the Queue interface, each offering different behaviors:

- **LinkedList:** Implements a doubly-linked list and can be used as a queue.
- **PriorityQueue:** Implements a priority heap-based queue, where elements are ordered according to their natural ordering or a custom comparator.

- **ArrayDeque:** Implements a resizable array-based double-ended queue, which can function as both a stack and a queue.
- **Usage Scenarios:**
 - Queues are commonly used in scenarios where elements need to be processed in the order they were added, such as task scheduling and breadth-first search algorithms.
 - The PriorityQueue is used for scenarios where elements have a priority or need to be ordered in a specific way.
- **Performance Considerations:**
 - The performance characteristics of different queue implementations vary. Choose the implementation that best suits your use case.
- **Concurrent Queues:** Java provides several concurrent queue implementations that can be used in multithreaded scenarios. These implementations offer thread-safe operations and blocking behavior to handle concurrency safely.

The Queue interface provides a versatile way to manage elements in a FIFO order. Depending on your requirements and the concurrency context, you can choose the appropriate implementation from the Java Collections Framework or the concurrent collections package.

Example:

```
package queueExamples;
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample1 {
    public static void main(String[] args) {
        /*
        * We cannot create instance of a Queue as it is an
        * interface, we can create instance of LinkedList or
        * PriorityQueue and assign it to Queue
        */
        Queue<String> q = new LinkedList<String>();

        //Adding elements to the Queue
        q.add("Rick");
        q.add("Maggie");
        q.add("Glenn");
        q.add("Negan");
        q.add("Daryl");
    }
}
```

```

System.out.println("Elements in Queue:"+q);
/*
 * We can remove element from Queue using remove() method,
 * this would remove the first element from the Queue
 */
System.out.println("Removed element: "+q.remove());

/*
 * element() method - this returns the head of the
 * Queue. Head is the first element of Queue
 */
System.out.println("Head: "+q.element());

/*
 * poll() method - this removes and returns the
 * head of the Queue. Returns null if the Queue is empty
 */
System.out.println("poll(): "+q.poll());

/*
 * peek() method - it works same as element() method,
 * however it returns null if the Queue is empty
 */
System.out.println("peek(): "+q.peek());

//Again displaying the elements of Queue
System.out.println("Elements in Queue:"+q);
}
}

```

Output:

```

Example1.java - Eclipse IDE
Help
History Synchronize Git Staging Git Reflog Properties JUnit Console SonarLint Report
<terminated> QueueExample1 [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (14-Aug-2023, 5:49:25 pm - 5:49:26 pm) [pid: 11412]
Elements in Queue:[Rick, Maggie, Glenn, Negan, Daryl]
Removed element: Rick
Head: Maggie
poll(): Maggie
peek(): Glenn
Elements in Queue:[Glenn, Negan, Daryl]

```

PriorityQueue Interface

We have seen how a Queue serves the requests based on FIFO(First in First out).

Now the question is: What if we want to serve the request based on the priority rather than FIFO?

In a practical scenario this type of solution would be preferred as it is more dynamic and efficient in nature. This can be done with the help of PriorityQueue, which serves the request based on the priority that we set using Comparator.

Note:*In an upcoming session, we'll delve into the topic of comparators.*

In this example, we are adding a few Strings to the PriorityQueue, while creating PriorityQueue we have passed the Comparator(named as MyComparator) to the PriorityQueue constructor.

In the MyComparator java class, we have sorted the Strings based on their length, which means the priority that we have set in PriorityQueue is String length. That way we ensured that the smallest string would be served first rather than the string that we have added first.

Methods of PriorityQueue Class

boolean add(E e): Adds the element into the PriorityQueue.

void clear(): This method removes all the elements from the PriorityQueue.

boolean contains(Element e): This method returns true, if the specified element is present in the Queue.

boolean Offer(E e): Same as add() method.

E peek(): Returns the head(the first element) of the Queue.

E poll(): Removes the head of the Queue and returns it.

boolean remove(E e): This method removes the specified element from the Queue and returns true if the deletion is successful. If the specified element is not present in the Queue then it returns false.

int size(): Returns the size of the Queue.

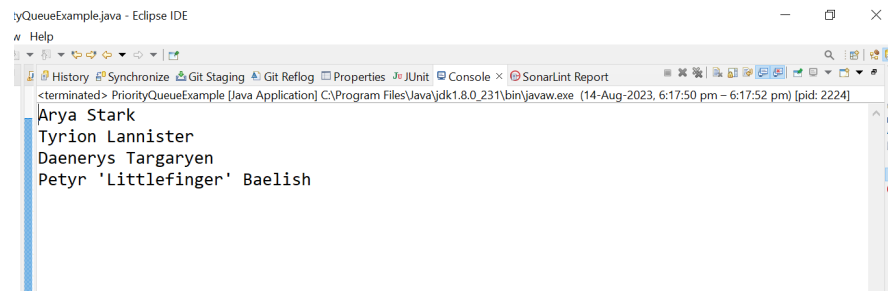
object[] toArray(): Returns array containing all the elements of Queue.

```
package queueExamples;  
import java.util.PriorityQueue;
```

```
public class PriorityQueueExample {  
    public static void main(String[] args) {  
        PriorityQueue<String> queue =  
            new PriorityQueue<String>(15, new MyComparator());  
        queue.add("Tyrion Lannister");  
        queue.add("Daenerys Targaryen");  
        queue.add("Arya Stark");  
        queue.add("Petyr 'Littlefinger' Baelish");  
  
        /*  
        * What I am doing here is removing the highest  
        * priority element from Queue and displaying it.  
        * The priority I have set is based on the string  
        * length. The logic for it is written in Comparator  
        */  
        while (queue.size() != 0)  
        {  
            System.out.println(queue.poll());  
        }  
    }  
}
```

```
package queueExamples;  
import java.util.Comparator;  
public class MyComparator implements Comparator<String> {  
    @Override  
    public int compare(String x, String y)  
    {  
        return x.length() - y.length();  
    }  
}
```

Output:

A screenshot of the Eclipse IDE's console window. The title bar reads "yQueueExample.java - Eclipse IDE". The console shows the output of a Java application. The first line is "<terminated> PriorityQueueExample [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (14-Aug-2023, 6:17:50 pm - 6:17:52 pm) [pid: 2224]". Below this, four names are printed on separate lines: "Arya Stark", "Tyrion Lannister", "Daenerys Targaryen", and "Petyr 'Littlefinger' Baelish". The IDE interface includes a menu bar (File, Edit, etc.), a toolbar, and a package explorer on the left.

Deque Interface

The Deque interface in Java represents a double-ended queue, which is a linear collection of elements that supports adding and removing elements from both ends. It is an acronym for "Double Ended Queue." The Deque interface is part of the Java Collections Framework and extends the Queue interface. Here are some key points and methods associated with the Deque interface:

Methods for Basic Operations:

addFirst(element): Adds the specified element to the front of the deque.

addLast(element): Adds the specified element to the end of the deque.

offerFirst(element): Adds the specified element to the front of the deque if possible.

offerLast(element): Adds the specified element to the end of the deque if possible.

removeFirst(): Removes and returns the element at the front of the deque.

removeLast(): Removes and returns the element at the end of the deque.

pollFirst(): Removes and returns the element at the front of the deque, or returns null if the deque is empty.

pollLast(): Removes and returns the element at the end of the deque, or returns null if the deque is empty.

getFirst(): Returns the element at the front of the deque without removing it.

getLast(): Returns the element at the end of the deque without removing it.

peekFirst(): Returns the element at the front of the deque without removing it, or returns null if the deque is empty.

peekLast(): Returns the element at the end of the deque without removing it, or returns null if the deque is empty.

Example:

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

```

public class ArrayDequeExample {
    public static void main(String[] args) {
        /*
            * We cannot create instance of a Deque as it is an
            * interface, we can create instance of ArrayDeque or
            * LinkedList and assign it to Deque
            */
        Deque<String> dq = new ArrayDeque<String>();

        /*
            * Adding elements to the Deque.
            * addFirst() adds element at the beginning
            * and addLast() method adds at the end.
            */
        dq.add("Glenn");
        dq.add("Negan");
        dq.addLast("Maggie");
        dq.addFirst("Rick");
        dq.add("Daryl");

        System.out.println("Elements in Deque:"+dq);

        /*
            * We can remove element from Deque using remove() method,
            * we can use normal remove() method which removes first
            * element or we can use removeFirst() and removeLast()
            * methods to remove the first and last element respectively.
            */
        System.out.println("Removed element: "+dq.removeLast());

        /*
            * element() method - returns the head of the
            * Deque. Head is the first element of Deque
            */
        System.out.println("Head: "+dq.element());

        /*
            * pollLast() method - this method removes and returns the
            * tail of the Deque(last element). Returns null if the Deque is empty.
            * We can also use poll() or pollFirst() to remove the first element of
            * Deque.
        */
    }
}

```

```

    */
    System.out.println("poll(): "+dq.pollLast());

    /*
    * peek() method - it works same as element() method,
    * however it returns null if the Deque is empty. We can also use
    * peekFirst() and peekLast() to retrieve first and last element
    */
    System.out.println("peek(): "+dq.peek());

    //Again printing the elements of Deque
    System.out.println("Elements in Deque:"+dq);
}
}

```

Output:

```

DequeExample.java - Eclipse IDE
Help
<terminated> ArrayDequeExample [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (14-Aug-2023, 6:25:51 pm - 6:25:53 pm) [pid: 13876]
Elements in Deque:[Rick, Glenn, Negan, Maggie, Daryl]
Removed element: Daryl
Head: Rick
poll(): Maggie
peek(): Rick
Elements in Deque:[Rick, Glenn, Negan]

```

Food for thought..

The trainer can ask the students the following question to engage them in a discussion:

Imagine you have a list of names with potential duplicates. Do you remember which collection you could use to remove duplicates?

I. Set Interface

The Set interface in Java is a part of the Java Collections Framework, which provides a way to work with collections of objects. The Set interface is designed to store a collection of distinct elements, ensuring that each element appears only once in the collection. This makes it useful for scenarios where you need to maintain a collection of unique values.

Key characteristics of the Set interface:

No Duplicates: Set implementations do not allow duplicate elements. If you attempt to add an element that already exists in the set, it won't be added again.

Unordered: Unlike List implementations, Sets do not maintain any specific order of elements. The order of elements in a Set is generally not guaranteed.

Methods: The Set interface provides methods for basic operations like adding elements (add), removing elements (remove), checking for element existence (contains), and getting the size of the set (size).

Implementations: Java offers several implementations of the Set interface, including HashSet, LinkedHashSet, and TreeSet. Each implementation has its own characteristics and use cases.

Hashing and Equality: Hash-based implementations (HashSet and LinkedHashSet) use the object's hash code to determine its position in the Set, while the TreeSet uses a comparator or the natural ordering of elements.

Iterating: You can iterate through the elements of a Set using the enhanced for loop or an iterator. Keep in mind that the order of iteration might not match the insertion order.

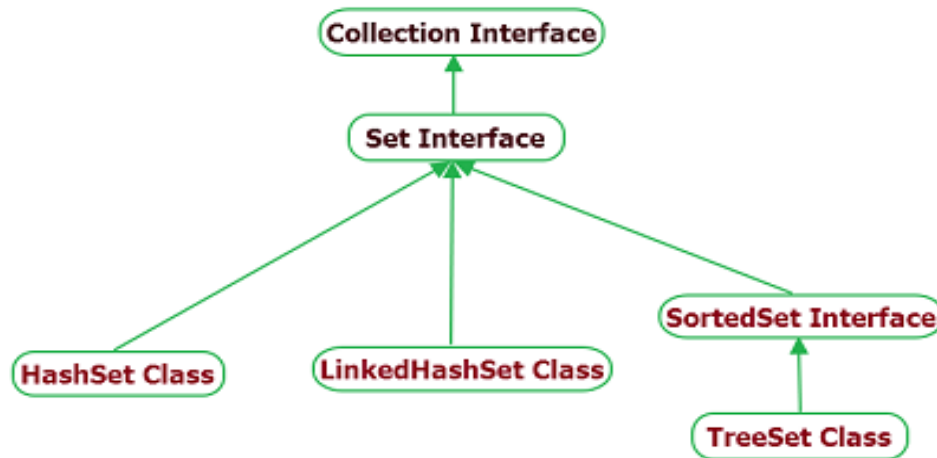
II. Implementation Classes

There are three main implementations of Set interface:

HashSet, which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration.

TreeSet, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet.

LinkedHashSet, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).



HashSet Class

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element. This class is not synchronized. However it can be synchronized explicitly like this: `Set s = Collections.synchronizedSet(new HashSet(...));`

Points to Note about HashSet:

1. HashSet doesn't maintain any order, the elements would be returned in any random order.
2. HashSet doesn't allow duplicates. If you try to add a duplicate element in HashSet, the old value would be overwritten.
3. HashSet allows null values however if you insert more than one null it would still return only one null value.
4. HashSet is non-synchronized.
5. The iterator returned by this class is fail-fast which means the iterator would throw `ConcurrentModificationException` if HashSet has been modified after creation of the iterator, by any means except the iterator's own remove method.

HashSet Methods:

1. **boolean add(Element e):** It adds the element e to the List.
2. **void clear():** It removes all the elements from the List.

3. **Object clone():** This method returns a shallow copy of the HashSet.
4. **boolean contains(Object o):** It checks whether the specified Object o is present in the List or not. If the object has been found it returns true else false.
5. **boolean isEmpty():** Returns true if there is no element present in the Set.
6. **int size():** It gives the number of elements of a Set.
7. **Boolean remove(Object o):** It removes the specified Object o from the Set.

Sr.No.	Constructor & Description
1	HashSet() This constructor constructs a default HashSet.
2	HashSet(Collection c) This constructor initializes the hash set by using the elements of the collection c.
3	HashSet(int capacity) This constructor initializes the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
4	HashSet(int capacity, float fillRatio) This constructor initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments. Here the fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded.

Example:

```
package com.anudip;  
import java.util.HashSet;  
public class HashSetEx {  
    public static void main(String[] args) {  
        //creating a HashSet
```

```
HashSet<String> str= new HashSet<String>();

//displaying the initial size
System.out.println("Size at the beginning "+str.size());

//add elements
str.add("apple");
str.add("kiwi");
str.add("banana");
str.add("orange");

//displaying the HashSet
System.out.println("Added few elements:"+str);

//displaying the size
System.out.println("Size after addition "+str.size());

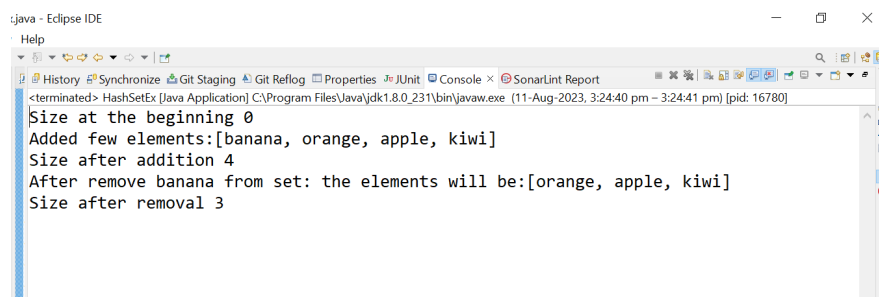
//remove element using value
str.remove("banana");

//display the new HashSet
System.out.println("After remove banana from set: the elements will be:"+str);

//display the new size
System.out.println("Size after removal "+str.size());
}

}
```

Output:

A screenshot of the Eclipse IDE's console window. The title bar shows ':java - Eclipse IDE'. The console output displays the results of the Java program: 'Size at the beginning 0', 'Added few elements:[banana, orange, apple, kiwi]', 'Size after addition 4', 'After remove banana from set: the elements will be:[orange, apple, kiwi]', and 'Size after removal 3'. The console window includes standard IDE toolbars and tabs for 'History', 'Synchronize', 'Git Staging', 'Git Reflog', 'Properties', 'JUnit', 'Console', and 'SonarLint Report'.

```
:java - Eclipse IDE
Help
<terminated> HashSetEx [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (11-Aug-2023, 3:24:40 pm - 3:24:41 pm) [pid: 16780]
Size at the beginning 0
Added few elements:[banana, orange, apple, kiwi]
Size after addition 4
After remove banana from set: the elements will be:[orange, apple, kiwi]
Size after removal 3
```

Let's explore the process of converting an array into a set in Java:

Array is a group of like-typed variables that are referred to by a common name. An array can contain primitive data types as well as objects of a class depending on the definition of the array. In the case of primitive data types, the actual values are stored in contiguous memory locations. In case of objects of a class, the actual objects are stored in the heap segment. Set in Java is a part of java.util package and extends java.util.Collection interface. It does not allow the use of duplicate elements and at max can accommodate only one null element.

Algorithm:

1. Get the Array to be converted.
2. Create an empty Set
3. Iterate through the items in the Array.
4. For each item, add it to the Set
5. Return the formed Set

Method 1: Brute Force or Naive Method: In this method, an empty Set is created and all elements present of the Array are added to it one by one.

// Java Program to convert

// Array to Set

Example:

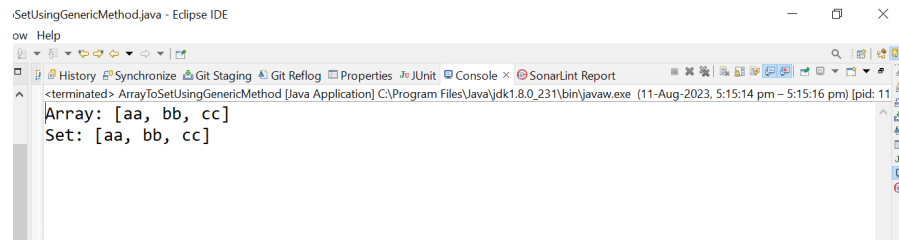
```
package com.anudip;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
public class ArrayToSetUsingGenericMethod {
    // Generic function to convert an Array to Set
    public static <T> Set<T> convertArrayToSet(T array[])
    {
        // Create an empty Set
        Set<T> set = new HashSet<>();
        // Iterate through the array
        for (T t : array) {
            // Add each element into the set
            set.add(t);
        }
        // Return the converted Set
    }
}
```

```

return set;
}
public static void main(String args[])
{
// Create an Array
String array[]
= { "aa", "bb", "cc" };
// Print the Array
System.out.println("Array: "
+ Arrays.toString(array));
// convert the Array to Set
Set<String> set = convertArrayToSet(array);
// Print the Set
System.out.println("Set: " + set);
}
}

```

Output



Way 2: Using Arrays.asList() method: In this method, the Array is passed as the parameter into the Set constructor in the form of an Array with the help of Arrays.asList() method.

Algorithm:

1. Get the Array to be converted.
2. Create the Set by passing the Array as parameter in the constructor of the Set with the help of Arrays.asList() method
3. Return the formed Set

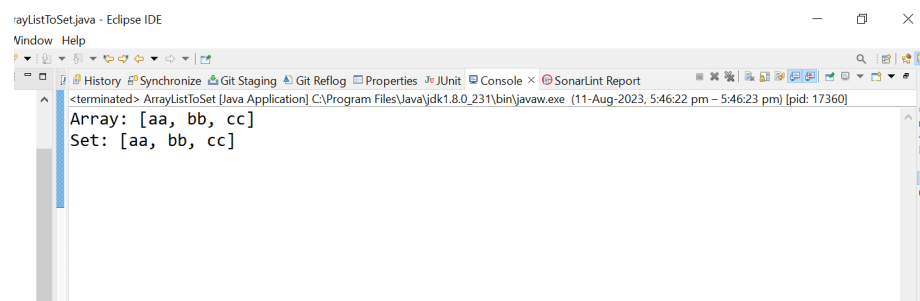
Example:

```
// Java Program to convert
// Array to Set

import java.util.*;
import java.util.stream.*;

public class ArrayListToSet {
    // Generic function to convert an Array to Set
    public static <T> Set<T> convertArrayToSet(T array[])
    {
        // Create the Set by passing the Array
        // as parameter in the constructor
        Set<T> set = new HashSet<>(Arrays.asList(array));
        // Return the converted Set
        return set;
    }
    public static void main(String args[])
    {
        // Create an Array
        String array[]
        = { "aa", "bb", "cc" };
        // Print the Array
        System.out.println("Array: "
        + Arrays.toString(array));
        // convert the Array to Set
        Set<String> set = convertArrayToSet(array);
        // Print the Set
        System.out.println("Set: " + set);
    }
}
```

Output



```
rayListToSet.java - Eclipse IDE
Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console x SonarLint Report
<terminated> ArrayListToSet [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (11-Aug-2023, 5:46:22 pm - 5:46:23 pm) [pid: 17360]
Array: [aa, bb, cc]
Set: [aa, bb, cc]
```

Way 3: Using Collections.addAll(): Since Set is a part of the Collection package in Java. Therefore the Array can be converted into the Set with the help of Collections.addAll() method.

Algorithm:

1. Get the Array to be converted.
2. Create an empty Set.
3. Add the array into the Set by passing it as the parameter to the Collections.addAll() method.
4. Return the formed Set

Example:

```
// Java Program to convert  
// Array to Set
```

```
package com.anudip;  
import java.util.*;  
public class ArrayToSet {  
    public static void main(String args[])  
    {  
        // Creating an array of integers  
        Integer[] array = { 1, 2, 3, 4, 5 };  
  
        // Creating a new HashSet to store the elements  
        Set<Integer> set = new HashSet<>();  
  
        // Converting the array to a set using Collections.addAll()  
        Collections.addAll(set, array);  
  
        // Printing the set  
        System.out.println("Converted Set: " + set);  
    }  
}
```

Output:

