**Collections**

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).

**Collection:**

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

**Framework:**

* 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. It has:

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

**Hierarchy of Collection Framework:**



**Methods of Collection Interface**

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

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 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> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

**Iterator interface**

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

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

**Iterable Interface**

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

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

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

**Collection Interface**

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

**List Interface**

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

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

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

**Set Interface**

A Set Interface is a Collection that prevents duplicate elements. Only the unique elements will be added to the set. The order in which the elements are added is not preserved.

The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Two Set instances are equal if they contain the same elements. As implied by its name, this interface models the mathematical set abstraction.

**Sorted Set and Other Interfaces**

**Generics:**

Generics are given the ability to create generalized classes, interfaces and methods by operating through references of type Object. It provides **compile-time type safety** that allows programmers to catch invalid types at compile time and also expand ability to reuse code.

**public** **class** SampleGenericsDemo {

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

{

SampleGenerics<Integer> integerObject = **new** SampleGenerics<Integer>(45); // LINE A

integerObject.showType();

**int** integerValue = integerObject.getObject();

System.***out***.println("Given Integer Value is : " +integerValue);

SampleGenerics<String> stringObject;

stringObject = **new** SampleGenerics<String>("MeritCampus"); // LINE B

stringObject.showType();

String str = stringObject.getObject();

System.***out***.println("Given String is : " +str);

}

}

**class** SampleGenerics<T> {

T value;

SampleGenerics(T object)

{

**this**.value = object;

}

**public** T getObject() {

**return** value;

}

**void** showType()

{

System.***out***.println("Type of T is : " + value.getClass().getName());

}

}

**//** **Here, T used inside the angle bracket <> indicates the type parameter.**

**Let’s see same example without using generics**

**class** SampleNonGenericDemo

{

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

{

SampleNonGeneric integerObject = **new** SampleNonGeneric(12);

integerObject.displayType();

Integer integerValue = (Integer)integerObject.getObj(); // LINE A

System.***out***.println("The integer value is : "+integerValue);

SampleNonGeneric stringObject;

stringObject = **new** SampleNonGeneric("MeritCampus");

stringObject.displayType();

String string = (String)stringObject.getObj(); // LINE B

System.***out***.println("The given string is :"+string);

integerObject = stringObject; // LINE C

// integerValue = (Integer)integerObject.getObj(); // LINE D

}

}

**class** SampleNonGeneric

{

Object value;

**public** SampleNonGeneric(Object obj)

{

**this**.value = obj;

}

Object getObj()

{

**return** value;

}

**void** displayType()

{

System.***out***.println("The type of object is : "+value.getClass().getName());

}

}

**Generic Class With 2 parameters**

A generic class has the facility to declare more than one type parameter. To specify two or more type parameters, simply use a comma - seperated list.

class GenericsDemo  
{  
    public static void main(String args[])  
    {  
          
        TwoGenerics<Integer, String> obj = new TwoGenerics<Integer, String>(45, "Meritcampus"); // LINE A  
          
        obj.showTypes();  
          
        int integerValue = obj.getOb1();  
          
        System.out.println("The given Integer is : " + integerValue);  
          
        String str = obj.getOb2();  
          
        System.out.println("The given String is : " + str);  
    }  
}  
  
class TwoGenerics<T, V>  
{  
    T ob1;  
  
    V ob2;  
          
    TwoGenerics(T ob1, V ob2)  
    {  
  
        this.ob1 =ob1;  
      
        this.ob2 =ob2;  
      
    }  
          
    T getOb1()  
    {  
        return ob1;  
    }  
          
    V getOb2()  
    {  
        return ob2;  
    }  
    void showTypes()  
    {  
        System.out.println("Type of T is : " + ob1.getClass().getName());  
        System.out.println("Type of V is : " + ob2.getClass().getName());  
    }  
}

**Java Bounded Type – Bounded Type in Java**

The type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter  
Syntax :  
<T extends superclass>

Bounded type parameters can be used with methods as well as classes and interfaces. Genericssupports multiple bounds also, i.e <T extends A & B & C>. In this case A can be an interfaceor class.If A is classthen B and C should be interfaces. We can’t have more than one classin multiple bounds.

Assume that you want to create a generic class that contains a method that returns the average of an array of numbers. Furthermore, you can use the class to obtain the average of an array of any type of number, including integers, floats, and doubles

class AverageDemo  
{  
    public static void main (String args[])  
        {  
            Integer integerNumbers[] = {1, 2, 3, 4, 5};  
      
            Average<Integer> integerObject = new Average<Integer>(integerNumbers);  
      
            double average = integerObject.getAverage();  
      
            System.out.println("The average value of given integers is : " + average);  
      
            String strs[] = {"A", "B", "C", "D"};  
      
          //  Average<String> stringObject = new Average<String>(strs); // LINE A  
        }  
}  
  
class Average<T extends Number> { // LINE B  
      
    T[] numbers;  
              
    public Average(T[] numbers)   
    {  
                      
        this.numbers = numbers;  
    }  
    double getAverage()  
    {  
                      
        double sum = 0.0;  
                          
        for(int i = 0; i< numbers.length; i++)  
              
        {  
            sum += numbers[i].doubleValue(); // LINE C  
              
        }  
              
        return sum / numbers.length;  
    }  
}

**ArrayList**

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

Some of the advantages ArrayList has over arrays are it can grow dynamically. It provides more powerful insertion and search mechanisms than arrays. Let's take a look at using an ArrayList that contains Strings:  
List<String> myList = new ArrayList<String>();

import java.util.\*;  
  
class Array\_List  
{  
    public static void main(String arg[])  
    {  
        ArrayList<String> list = new ArrayList<String>();  
        list.add("Good");  
        list.add("Morning");  
        list.add("Students");  
        System.out.println(list.size());   // LINE A  
        list.remove("Good");   // LINE B  
        System.out.println(list.contains("Morning"));  // LINE C  
        System.out.println(list.get(1));   // LINE D  
        Iterator<String> iterate = list.iterator();  
        while(iterate.hasNext())  
        {  
            System.out.println(iterate.next());  
        }      
    }  
}

When an arraylist is created, its default capacity or size is 10 if not mentioned by the user. The size of the arraylist grows based on load factor and current capacity. Here, the load factor is a measure to decide when to increase its capacity. The load factor of an arraylist is **0.75f.**

ArrayList in Java expands its capacity after each threshold, so we first need to calculate the threshold.

Threshold = (Load Factor) \* (Current Capacity)

For example, if the user creates an arraylist of size 10, then threshold value will be 0.75\*10=7. This means after adding the 7th element to the list, the size will increase as it has reached the threshold value.

Internally, a new ArrayList with a new capacity is created and the elements present in the old arraylist are copied in the new arraylist, as shown in the following image.

Chart

Description automatically generated with medium confidence

In java 8 and later, the new capacity is estimated to be 50% more than the old capacity.

new\_capacity = old\_capacity + (old\_capacity >> 1)

For example, if the array size is 10 and it has reached the threshold value, we have to increase its capacity to add new elements. The new capacity will be 10+ (10>>1) => 10+ 5 => 15. Hence, the size is increased from 10 to 15.

**Takeaway:**

* The backing data structure of ArrayList is an array.
* When the array becomes full and we want to add new elements, a new ArrayList with a new capacity is created.
* The elements present in the old arraylist are copied in the new arraylist, while deleting the old arraylist.

**How to Sort Arraylist in Java?**

We can sort an arraylist using the **sort()** method of the Collection framework in Java.

**Syntax:**

**Collections.sort(ArrayList):** As an argument, it takes the object of an arraylist and sorts an arraylist in the ascending order according to the natural ordering of its elements.

import java.util.\*;

class Test {

public static void main(String[] args)

{

ArrayList<String> names = new ArrayList<String>();

names.add(“Raj”);

names.add(“Priya”);

names.add(“Shashank”);

names.add(“Ansh”);

System.out.println("Before sorting, names : " + names);

//Sorting arraylist in ascending order

Collections.sort(names);

System.out.println("After sorting, names : " + names);

}

}

* **sort()** method doesn't return anything.

**Change an element in ArrayList**

**set(int index, E element):** In this method, we provide the index and the new element as arguments which in return replaces the element present at the specified index with the new element.

import java.util.\*;

class Test {

public static void main(String[] args)

{

ArrayList<Integer> num= new ArrayList<Integer>();

num.add(15);

num.add(9);

num.add(20);

num.add(35);

System.out.println("Arraylist num : " + num);

//Replacing element present at 0th index with 40

num.set(0,40);

System.out.println("Arraylist num after updating : " + num);

}

}

**How to Remove Elements from ArrayList?**

We can remove elements from an arraylist with the help of **remove()** method. This method is also overloaded, allowing it to do several operations based on various parameters. Here's what they are:

**remove(Object o) :** This method is used to directly remove the specified element from the arraylist.

**remove(int index) :** This method is used to remove the element present at the specified index from the arraylist.

import java.util.\*;

class Test {

public static void main(String[] args)

{

ArrayList<String> colors= new ArrayList<String>();

colors.add("red");

colors.add("orange");

colors.add("blue");

colors.add("pink");

colors.add("black");

colors.add("green");

System.out.println("ArrayList colors : "+colors);

// removing element pink from the arraylist

colors.remove("pink");

System.out.println("ArrayList colors : "+colors);

// removing 3rd element from the arraylist

colors.remove(2);

System.out.println("ArrayList colors : "+colors);

}

}

**Iterating a ArrayList**

**Length of a Array List**

Use size()

**Check if ArrayList is Empty**

Use isEmpty

**Java Array vs ArrayList**

The key differences between Array and ArrayList are listed in the table below:

| Array | ArrayList |
| --- | --- |
| Array is a data structure in Java used for storing elements. | ArrayList is a class of Collection framework which implements the List interface. |
| The size of an array is fixed and cannot be changed. | The size of an arraylist is not fixed. We can increase or decrease its size as one adds or removes elements. |
| We need to mention its size while creating an array. | We can create an ArrayList instance without mentioning its size. |
| Arrays can hold both primitive data types and objects of a class. | ArrayList only accepts object entries and not primitive data types. |
| Arrays are faster as they are of fixed length. | ArrayList are relatively slower because of its dynamic nature. |

**Benefits of ArrayList**

* The fact that ArrayList is dynamic in size is one of its main advantages. We can increase and decrease the size of arraylist dynamically.
* ArrayList has various predefined methods which help to manipulate the stored objects.
* In arraylist, we can randomly insert and delete elements.
* We can add different types of objects into the ArrayList.
* In arraylist, list iterator allows us to traverse in both directions.

Conclusion

* The ArrayList class in Java is a part of the Collection framework, and it implements the List interface.
* ArrayLists are based on array data structure and can grow and shrink dynamically.
* ArrayList class in Java has various predefined methods through which we can manipulate the elements.
* **Operations like accessing an element take O(1), and insertions and deletions take O(N).**
* The implementation of an ArrayList is not synchronized, it must be synchronized externally.
* The iterators returned by the ArrayList class's iterator and listIterator methods are fail-fast, the fail-fast behavior of the iterators should be used only to detect bugs.
* Java's ArrayList overcomes the disadvantages of using arrays and makes Java programming much easier.

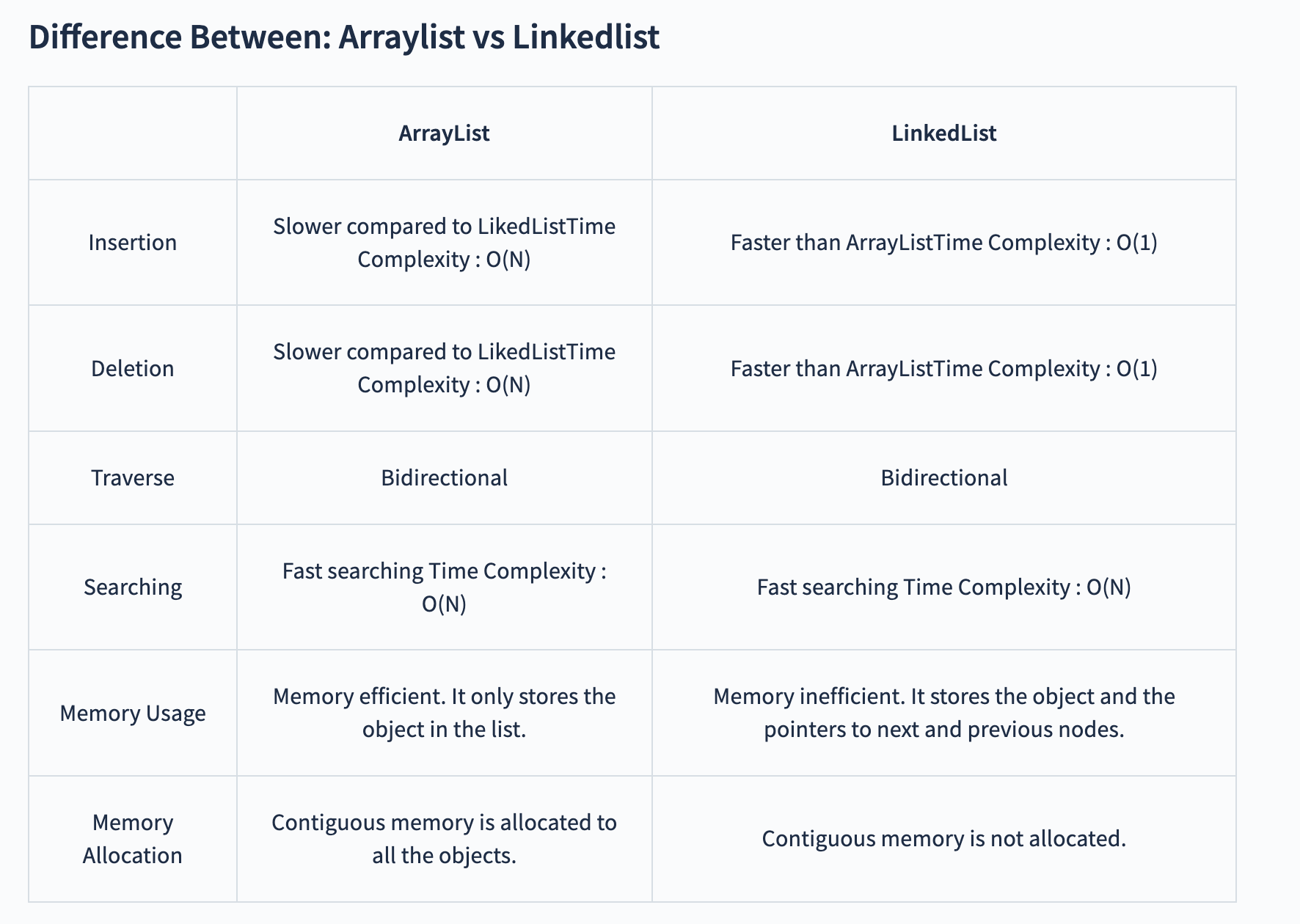
**LinkedList**

LinkedList class extends AbstractSequentialList and implements the List, Java Queue Interface, and Java Deque Interface interfaces.

Declaration of LinkedList:  
class LinkedList< E >  
Here, E specifies the type of objects that the list will hold.

LinkedList has two constructors they are shown below.  
  
Constructors in LinkedList:  
LinkedList()  
Builds an empty linked list.  
  
LinkedList(Collection<? extends E> <code class="code">c</code>)  
Builds a linked list with the elements of the collection c.

import java.util.\*;  
  
class LinkedListDemo  
{  
    public static void main(String arg[])  
    {  
        LinkedList characters = new LinkedList(); // LINE A  
        characters.add('A');  
        characters.add('C');  
        characters.add('D');  
        System.out.println("Original list: " + characters);  
        characters.add(1, 'B'); // LINE B  
        System.out.println("After adding B: " + characters);  
        List numbers = new ArrayList(); // LINE C  
        numbers.add(1);  
        numbers.add(2);  
        numbers.add(3);  
        characters.addAll(numbers); // LINE D  
        System.out.println("After adding numbers: " + characters);  
        characters.addFirst("First"); // LINE E  
        characters.addLast("Last"); // LINE F  
        System.out.println("After adding First and Last: " + characters); // LINE G  
        characters.removeFirst(); // LINE H  
        characters.removeLast(); // LINE I  
        System.out.println("After removing First and Last:" + characters);  
        //converting characters to char[] array  
        Object[] newCharacters = characters.toArray(); // LINE J  
        System.out.println("Conveting LinkedList to Array of objects: ");  
        for (Object newCharacter : newCharacters)  
        {  
            System.out.print(newCharacter + ", ");  
        }  
      
    }  
}



**When to use what?**

We should choose LinkedList when operations like data addition or deletion occur more frequently than reading the data. Conversely, use ArrayList when data reading scenarios are more common than adding or removing data. This is because ArrayList is better equipped for frequent data retrievals.

The data in an ArrayList is stored more compactly than in LinkedList, making ArrayList more cache-friendly and reducing the likelihood of cache misses. In contrast, LinkedList tends to have poorer cache-locality due to its scattered data storage.

Memory overhead is another critical factor. LinkedList consumes more memory than ArrayList due to the additional links needed to connect nodes. Each node in LinkedList holds the address of both the previous and the next nodes, requiring extra space. This additional link structure is not present in ArrayList, making it more memory-efficient.

**Stack class**

A stack is a linear data structure. It provides a container to store data items which can be accessed from one side of the container only.

Here the element that is inserted at last is deleted first.

This is called the Last-In-First-Out (LIFO) principle.

It means that we can only access the latest entered elements.

**A diagram of a number of steps

Description automatically generated with medium confidence**

**Stacks can be implemented using the Stack class provided to us by the Java package.**

**Sample Stack program**

import java.util.Stack;

public class StackDemo {

public static void main(String[] args) {

//declaration of stack

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

//insert elements into stack

s.push(10);

s.push(20);

s.push(30);

s.push(40);

s.push(50);

//printing and deleting top element of stack

while (!s.empty()) {

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

s.pop();

}

}

}

**Methods in Stack class**

1) push()

This method inserts a given value into the stack at the top

2) pop()

This method removes the top element from the stack and returns it until the stack becomes completely empty

3) peek()

This method returns us the top element of the stack. It throws an error when we use it with an empty stack and tells us underflow.

4) search()

This method is used to know whether an element exists in our stack or not.

If the element exists it returns the position of that element from the top of the stack.

If the element is absent it returns -1.

5) empty()

This method returns a boolean value.

If the stack is empty it returns true otherwise it returns false.