**Assignment**

1.Stack Interface

public interface Stack<E> {public interface Stack<E> {  
 boolean isEmpty();  
 int size();  
 E top();  
 void push(E e);  
 E pop();  
}

2. Stack Using Array

public class ArrayStack<E> implements Stack<E> {  
 private static final int *DEFAULT\_CAPACITY* = 10;  
 private E[] array;  
 private int top;  
  
 public ArrayStack() {  
 this(*DEFAULT\_CAPACITY*);  
 }  
  
 public ArrayStack(int capacity) {  
 array = (E[]) new Object[capacity];  
 top = -1;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return top == -1;  
 }  
  
 @Override  
 public int size() {  
 return top + 1;  
 }  
  
 @Override  
 public E top() {  
 if (isEmpty()) {  
 throw new EmptyStackException();  
 }  
 return array[top];  
 }  
  
 @Override  
 public void push(E e) {  
 if (size() == array.length) {  
 resizeArray();  
 }  
 array[++top] = e;  
 }  
  
 @Override  
 public E pop() {  
 if (isEmpty()) {  
 throw new EmptyStackException();  
 }  
 E element = array[top];  
 array[top--] = null;  
 return element;  
 }  
  
 private void resizeArray() {  
 int newCapacity = array.length \* 2;  
 E[] newArray = (E[]) new Object[newCapacity];  
 System.*arraycopy*(array, 0, newArray, 0, array.length);  
 array = newArray;  
 }  
}

3. Stack Using Linked Lists

public class LinkedStack<E> implements Stack<E> {  
 private Node<E> top;  
 private int size;  
  
 public LinkedStack() {  
 top = null;  
 size = 0;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return size == 0;  
 }  
  
 @Override  
 public int size() {  
 return size;  
 }  
  
 @Override  
 public E top() {  
 if (isEmpty()) {  
 throw new EmptyStackException();  
 }  
 return top.data;  
 }  
  
 @Override  
 public void push(E e) {  
 Node<E> newNode = new Node<>(e);  
 newNode.next = top;  
 top = newNode;  
 size++;  
 }  
  
 @Override  
 public E pop() {  
 if (isEmpty()) {  
 throw new EmptyStackException();  
 }  
 E element = top.data;  
 top = top.next;  
 size--;  
 return element;  
 }  
  
 private static class Node<E> {  
 E data;  
 Node<E> next;  
  
 public Node(E data) {  
 this.data = data;  
 next = null;  
 }  
 }  
}

1\

public class StackTransfer {  
 public static <E> void transfer(Stack<E> sourceStack, Stack<E> targetStack) {  
 while (!sourceStack.isEmpty()) {  
 E element = sourceStack.pop();  
 targetStack.push(element);  
 }  
 }  
}

2\

public class StackUtils {  
 public static <E> void removeAll(Stack<E> stack) {  
 if (!stack.isEmpty()) {  
 stack.pop();  
 *removeAll*(stack);  
 }  
 }  
}

3\

import java.util.Stack;  
  
public class PostfixEvaluator {  
 public static int evaluate(String postfixExpression) {  
 Stack<Integer> stack = new Stack<>();  
  
 for (String token : postfixExpression.split("\\s+")) {  
 if (*isOperand*(token)) {  
 stack.push(Integer.*parseInt*(token));  
 } else if (*isOperator*(token)) {  
 int pexp2 = stack.pop();  
 int pexp1 = stack.pop();  
 int result = *applyOperator*(token, pexp1, pexp2);  
 stack.push(result);  
 }  
 }  
  
 return stack.pop();  
 }  
  
 private static boolean isOperand(String token) {  
 return token.matches("\\d+"); // Check if the token is a number  
 }  
  
 private static boolean isOperator(String token) {  
 return token.matches("[+\\-\*/]"); // Check if the token is one of the supported operators  
 }  
  
 private static int applyOperator(String operator, int pexp1, int pexp2) {  
 switch (operator) {  
 case "+":  
 return pexp1 + pexp2;  
 case "-":  
 return pexp1 - pexp2;  
 case "\*":  
 return pexp1 \* pexp2;  
 case "/":  
 return pexp1 / pexp2;  
 default:  
 throw new IllegalArgumentException("Invalid operator: " + operator);  
 }  
 }  
}

4\

import java.util.Arrays;  
  
public class ArrayStack<T> implements Stack<T> {  
 private static final int *DEFAULT\_CAPACITY* = 10;  
 private T[] array;  
 private int size;  
  
 public ArrayStack() {  
 this.array = (T[]) new Object[*DEFAULT\_CAPACITY*];  
 this.size = 0;  
 }  
  
 // Other methods of the ArrayStack class...  
  
 @Override  
 public ArrayStack<T> clone() {  
 ArrayStack<T> clonedStack = new ArrayStack<>();  
 clonedStack.array = Arrays.*copyOf*(this.array, this.size);  
 clonedStack.size = this.size;  
 return clonedStack;  
 }  
}

5\

class PostfixEvaluator:  
 @staticmethod  
 def evaluate(expression):  
 stack = []  
 operators = {  
 '+': lambda x, y: x + y,  
 '-': lambda x, y: x - y,  
 '\*': lambda x, y: x \* y,  
 '/': lambda x, y: x / y  
 }  
  
 for token in expression.split():  
 if token.isdigit():  
 stack.append(int(token))  
 elif token in operators:  
 operand2 = stack.pop()  
 operand1 = stack.pop()  
 result = operators[token](operand1, operand2)  
 stack.append(result)  
  
 return stack.pop()  
  
  
expression = input("Enter the postfix expression: ")  
result = PostfixEvaluator.evaluate(expression)  
print("Result:", result)