Stack-Based Calculator Report

# Introduction

The Stack-Based Calculator is a versatile tool designed to perform various arithmetic and advanced mathematical operations.   
It supports both infix and postfix notations, with proper handling of operator precedence and associativity.   
This document provides an in-depth explanation of the calculator's functionalities, its implementation, and how the code works.

# Features of the Calculator

1. Basic arithmetic operations: addition, subtraction, multiplication, and division.  
2. Advanced operations: exponentiation, square root, and parenthesis handling.  
3. Input handling for both infix and postfix (Reverse Polish Notation) notations.  
4. Operator precedence and associativity management using stacks.  
5. Error handling for invalid inputs and division by zero.

# Code Overview

The Stack-Based Calculator is implemented using Java. It allows users to input expressions in either infix or postfix notation.   
The code is divided into three main functionalities:  
1. Conversion of infix expressions to postfix.  
2. Evaluation of postfix expressions.  
3. Error handling and result display.

# Explanation of Main Components

## 1. Main Method

The main method serves as the entry point of the program. It takes user input for the type of expression (infix or postfix)   
and the mathematical expression itself. Based on the input, it either converts the infix to postfix or directly evaluates the postfix expression.

## 2. Infix to Postfix Conversion

The infixToPostfix method converts an infix expression to its equivalent postfix expression using a stack.   
It considers operator precedence and ensures correct handling of parenthesis. The steps involved are:  
- Reading characters of the input expression.  
- Pushing operators onto the stack based on precedence.  
- Managing parenthesis by pushing '(' and popping till ')'.

## 3. Postfix Evaluation

The evaluatePostfix method evaluates a postfix expression by processing each token and using a stack to keep track of operands.   
Operations such as addition, subtraction, multiplication, division, and exponentiation are performed.   
Error handling is included to catch invalid expressions and division by zero.

## 4. Operator Precedence and Associativity

Operator precedence is crucial for correct expression evaluation. The calculator follows these precedence rules:  
- Exponentiation has the highest precedence.  
- Multiplication and division have higher precedence than addition and subtraction.  
- Parentheses are used to override default precedence.

## 5. Error Handling

The program includes robust error handling to manage:  
- Invalid expressions.  
- Mismatched parentheses.  
- Division by zero errors.

# Java Code Implementation

import java.util.\*;  
  
public class StackBasedCalculator {  
  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.in);  
 System.out.println("Choose input type (1: Infix, 2: Postfix): ");  
 int choice = scanner.nextInt();  
 scanner.nextLine(); // Consume newline  
  
 System.out.println("Enter the expression: ");  
 String expression = scanner.nextLine();  
  
 try {  
 if (choice == 1) {  
 String postfix = infixToPostfix(expression);  
 System.out.println("Postfix Notation: " + postfix);  
 double result = evaluatePostfix(postfix);  
 System.out.println("Result: " + result);  
 } else if (choice == 2) {  
 double result = evaluatePostfix(expression);  
 System.out.println("Result: " + result);  
 } else {  
 System.out.println("Invalid choice!");  
 }  
 } catch (Exception e) {  
 System.out.println("Error: " + e.getMessage());  
 }  
 scanner.close();  
 }  
  
 // Convert infix to postfix  
 private static String infixToPostfix(String infix) throws Exception {  
 StringBuilder postfix = new StringBuilder();  
 Stack<Character> stack = new Stack<>();  
 Map<Character, Integer> precedence = Map.of(  
 '+', 1, '-', 1,  
 '\*', 2, '/', 2,  
 '^', 3  
 );  
  
 for (int i = 0; i < infix.length(); i++) {  
 char ch = infix.charAt(i);  
  
 if (Character.isDigit(ch) || ch == '.') {  
 postfix.append(ch);  
 } else if (ch == '(') {  
 stack.push(ch);  
 } else if (ch == ')') {  
 while (!stack.isEmpty() && stack.peek() != '(') {  
 postfix.append(' ').append(stack.pop());  
 }  
 if (stack.isEmpty() || stack.pop() != '(') {  
 throw new Exception("Mismatched parentheses.");  
 }  
 } else if (precedence.containsKey(ch)) {  
 postfix.append(' ');  
 while (!stack.isEmpty() && precedence.getOrDefault(stack.peek(), 0) >= precedence.get(ch)) {  
 postfix.append(stack.pop()).append(' ');  
 }  
 stack.push(ch);  
 } else {  
 postfix.append(' ');  
 }  
 }  
  
 while (!stack.isEmpty()) {  
 char op = stack.pop();  
 if (op == '(' || op == ')') {  
 throw new Exception("Mismatched parentheses.");  
 }  
 postfix.append(' ').append(op);  
 }  
  
 return postfix.toString().trim();  
 }  
  
 // Evaluate postfix expression  
 private static double evaluatePostfix(String postfix) throws Exception {  
 Stack<Double> stack = new Stack<>();  
 String[] tokens = postfix.split("\s+");  
  
 for (String token : tokens) {  
 if (token.matches("-?\d+(\.\d+)?")) {  
 stack.push(Double.parseDouble(token));  
 } else if (token.equals("^")) {  
 if (stack.size() < 2) throw new Exception("Invalid expression.");  
 double b = stack.pop();  
 double a = stack.pop();  
 stack.push(Math.pow(a, b));  
 } else if (token.equals("sqrt")) {  
 if (stack.isEmpty()) throw new Exception("Invalid expression.");  
 stack.push(Math.sqrt(stack.pop()));  
 } else {  
 if (stack.size() < 2) throw new Exception("Invalid expression.");  
 double b = stack.pop();  
 double a = stack.pop();  
 switch (token) {  
 case "+" -> stack.push(a + b);  
 case "-" -> stack.push(a - b);  
 case "\*" -> stack.push(a \* b);  
 case "/" -> {  
 if (b == 0) throw new Exception("Division by zero.");  
 stack.push(a / b);  
 }  
 default -> throw new Exception("Unknown operator: " + token);  
 }  
 }  
 }  
  
 if (stack.size() != 1) throw new Exception("Invalid expression.");  
 return stack.pop();  
 }  
}