**STACK BASED CALCULATOR**

*Dissertation submitted in fulfilment of the requirements for the Degree of*

**BACHELOR OF TECHNOLOGY**

**IN**

**COMPUTER SCIENCE AND ENGINEERING WITH SPECIALIZATION IN**

**DATA SCIENCE AND MACHINE LEARNING**

By

**PRANJAL SINHA**

**REGISTRATION NUMBER- 12206214**

**SECTION- K22UN**

**ROLL NUMBER- RK22UNA31**

Supervisor

**Shubham Sharma**



**School of Computer Science and Engineering**

Lovely Professional University

Phagwara, Punjab (India)

Month: October, Year: 2023

***Acknowledgment***

I would like to express my sincere gratitude to all those who contributed to the successful completion of this project. First and foremost, I am thankful for the unwavering support and guidance of my mentor, whose expertise and insights significantly shaped the direction of this endeavor.

I extend my heartfelt thanks to my peers and colleagues who provided valuable feedback and encouragement throughout the project. Their diverse perspectives enriched the discussions and fostered a collaborative environment, enhancing the overall quality of the work.

I appreciate the support from my friends and family members who stood by me during the challenges, providing emotional sustenance and motivation.

Lastly, I would like to acknowledge the academic institutions and libraries that granted access to extensive research materials, enabling a comprehensive exploration of the project's subject matter. Their resources proved instrumental in shaping the depth and breadth of the report.

This project would not have been possible without the collective efforts and encouragement of all these individuals and organizations. I am deeply grateful for their support and belief in the project's potential.

Pranjal Sinha

Reg. No.- 12206214

Section- K22UN

Roll No. – RK22UNA31

**Table of Contents:**

I. Introduction

II. Objectives and Scope of the Project

III. Application Tools

IV. Algorithm Implementation

V. Input Parsing Algorithm

VI. Stack Implementation Algorithm

VII. Expression Evaluation Algorithm

VIII. Algorithm Explanation

IX. Screenshots of the Execution

X. Summary

XI. Bibliography

XII. Annexure

1. **Introduction:**

This project delves into the practical application of one such structure: the stack, which operates on the Last-In-First-Out (LIFO) principle. The stack-based calculator, a fundamental component of this exploration, holds substantial relevance. It not only showcases the efficiency of stack operations but also underscores their critical role in real-world problem-solving. By evaluating arithmetic expressions, this calculator demonstrates how stacks can simplify complex computations.

1. **Objectives and Scope of the Project**

The objectives of this project are twofold: first, to implement a functional stack-based calculator, and second, to elucidate the fundamental principles of stack operations through practical application. The scope encompasses parsing and evaluating diverse arithmetic expressions, emphasizing the calculator's ability to handle a wide array of mathematical inputs. The project aims to illustrate the versatility of stacks, emphasizing their relevance in algorithmic solutions. By clearly defining the objectives and scope, this section delineates the project's boundaries, ensuring a focused approach towards achieving the predefined goals.

1. **Application Tools**

In this project, Java was chosen as the primary programming language due to its robustness, portability, and extensive support for data structures. The Integrated Development Environments (IDEs) IntelliJ IDEA facilitated the development process by providing features like code completion, and debugging. Additionally, online resources like GitHub repositories and Java programming forums proved invaluable for problem-solving and reference. Utilizing Java's rich libraries and object-oriented programming features, the project successfully implemented the stack-based calculator, showcasing the language's flexibility and efficiency in handling complex data structures and algorithms.

1. **Algorithm Implementation**

1) **Input Parsing Algorithm:**

(i) Utilize Java's String methods for efficient parsing and tokenization.

(ii) Identify operators and operands and categorize them appropriately.

(iii) Implement error handling to address invalid input cases.

2) **Stack Implementation Algorithm:**

(i) Implement a stack using Java's ArrayList or LinkedList.

(ii) Define push and pop methods for stack manipulation.

(iii) Ensure the stack class is generic to handle various data types.

3) **Expression Evaluation Algorithm:**

(i) Iterate through tokens and perform operations based on their types.

(ii) Implement a precedence-based approach to handle operator priorities.

(iii) Use Java's exception handling mechanisms for error detection and reporting.

1. **Algorithm Explanation**

1) **Input Parsing Algorithm:**

(i)Utilize Java's split() method to tokenize the input string into operators and operands.

(ii) Iterate through the tokens and categorize them based on their types (operators or operands).

(iii)Implement error handling mechanisms to address invalid input cases, ensuring robust parsing.

2) **Stack Implementation Algorithm:**

(i) Use Java's built-in ArrayList or LinkedList classes to implement the stack data structure.

(ii) Define generic methods push() and pop() to add elements onto the stack and retrieve them, respectively.

(iii) Ensure the stack operations are performed efficiently to maintain the LIFO (Last-In-First-Out) behavior.

3) **Expression Evaluation Algorithm:**

(i) Iterate through the parsed tokens and evaluate expressions based on operator precedence and associativity.

(ii) Utilize conditional statements and loops to handle different operators and their corresponding calculations.

(iii) Implement exception handling using Java's try-catch blocks to detect and handle errors gracefully, providing accurate error messages for invalid expressions.

4) **Code Modularity and Readability:**

(i) Divide the code into functions/methods, each responsible for a specific task (parsing, stack operations, expression evaluation).

(ii) Use meaningful variable and function names to enhance code readability and maintainability.

(iii)Add comments and documentation to explain complex logic or algorithms, making it easier for other developers to understand the codebase.

5) **Testing and Debugging:**

(i) Implement unit tests to validate the correctness of individual components (parsing, stack operations, expression evaluation).

(ii) Use debugging tools provided by Java IDEs to identify and fix logical errors or unexpected behaviors in the code.

(iii) Test the calculator with various input expressions, including edge cases, to ensure its reliability and accuracy.

1. **Stack Implementation Algorithm**

The stack implementation in the provided code follows these steps:

1. Initialize two stacks: one for numbers and another for operators.
2. Iterate through each character in the input expression.
3. If a digit is encountered, parse the number and push it onto the numbers stack.
4. If an opening parenthesis is encountered, push it onto the operators stack.
5. If a closing parenthesis is encountered, pop operators and numbers from stacks and apply the operators until an opening parenthesis is found.
6. If an operator is encountered, compare its precedence with the operators on the stack. If higher or equal precedence, pop and apply operators until a lower precedence operator is encountered or the stack is empty. Push the current operator onto the stack.
7. Continue until the entire expression is processed.

Finally, apply the remaining operators on the stacks to get the result.

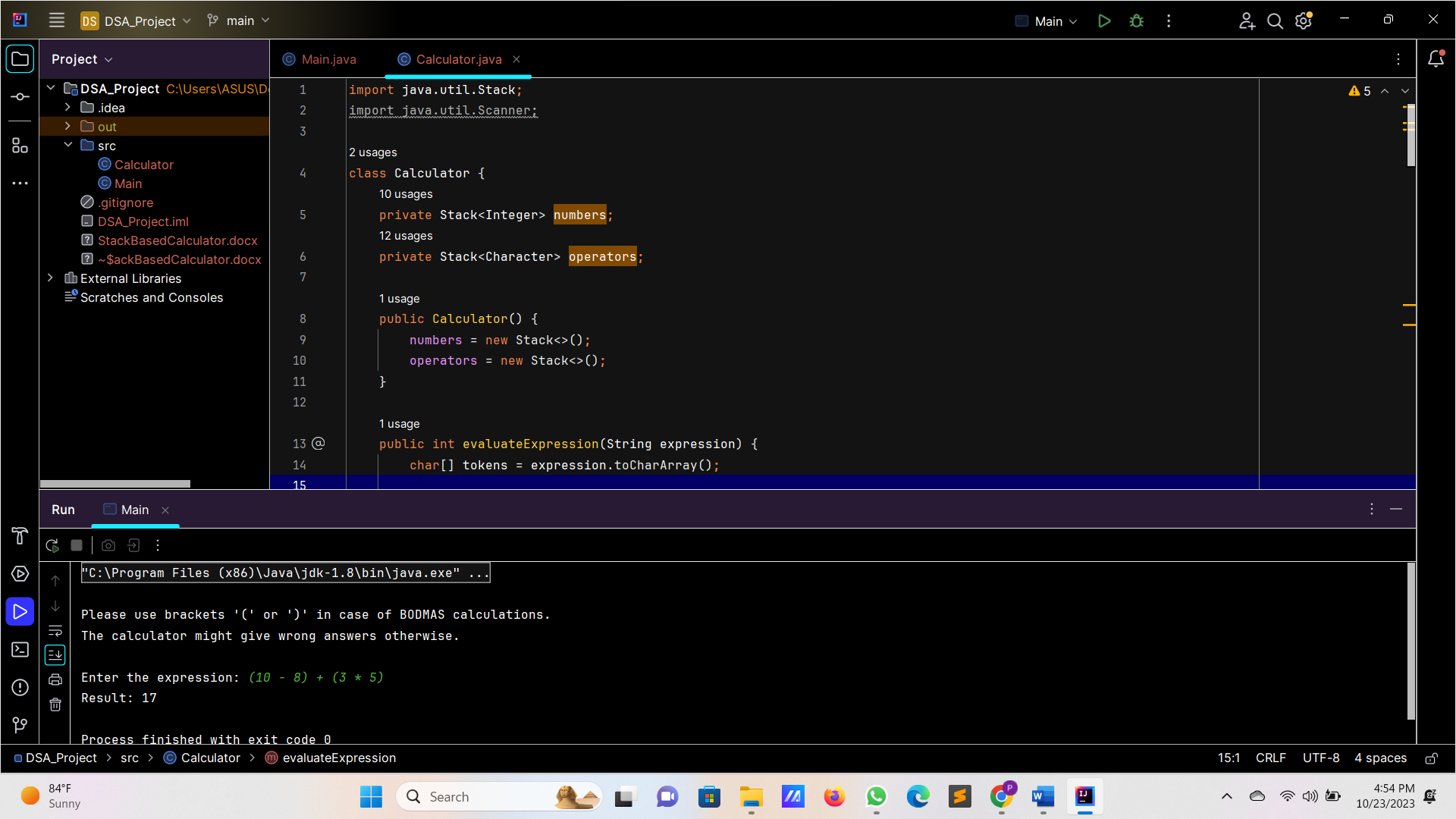
1. **Expression Evaluation Algorithm**

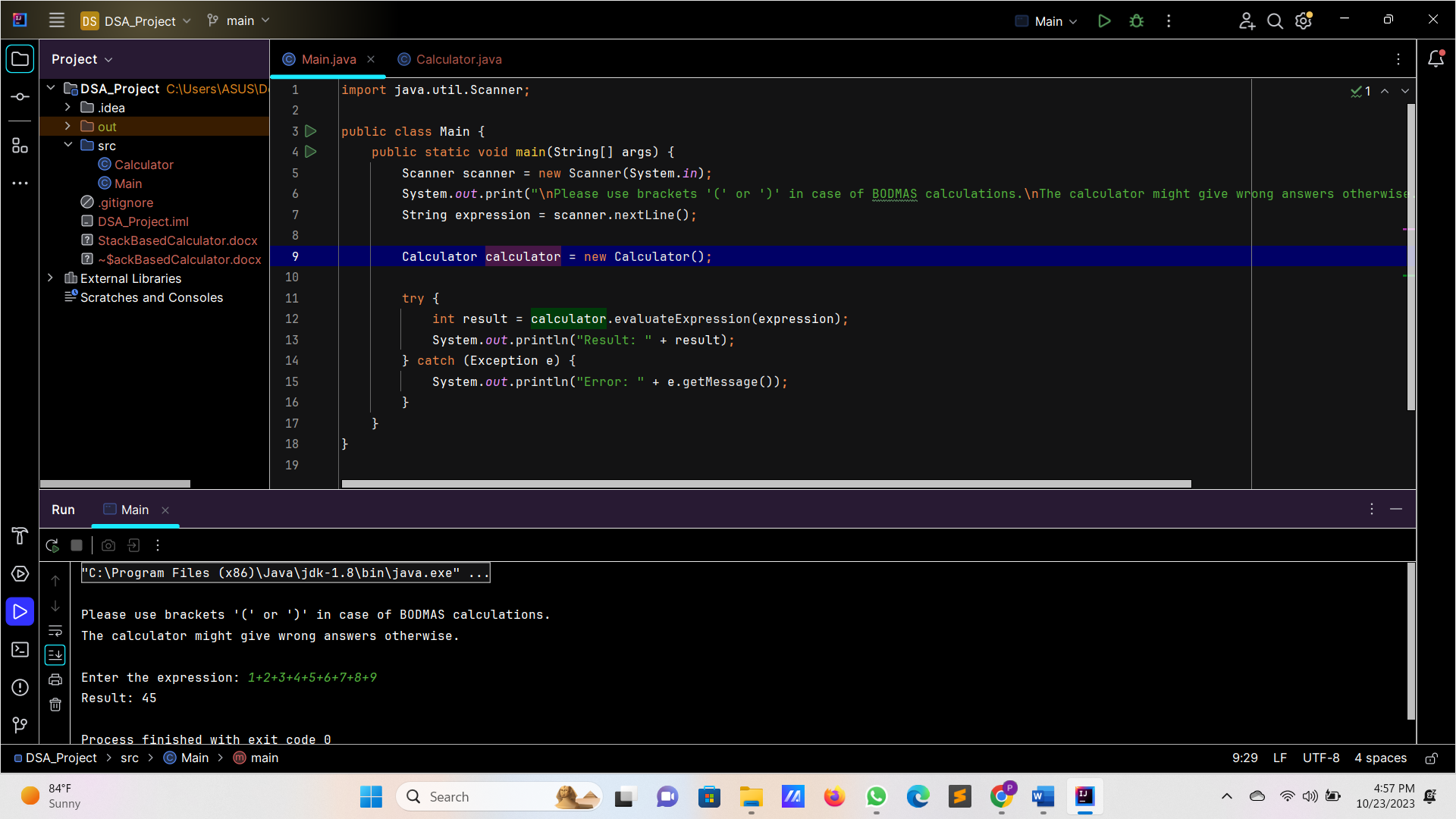
The expression evaluation algorithm can be summarized as follows:

1. **Tokenization:** Tokenize the input expression into characters.
2. **Iteration:** Iterate through each token and perform actions based on its type (operand, operator, parentheses).
3. **Stack Operations:** Use two stacks (numbers and operators) to keep track of numbers and operators.
4. **Operator Application:** Apply operators according to their precedence and associativity rules.
5. **Result Calculation:** Evaluate the expression and return the result.
6. **Algorithm Explanation**

The algorithm employs two stacks for efficient handling of operands and operators. It processes the input expression character by character, handling digits, operators, and parentheses appropriately. Operators are applied based on their precedence and associativity, ensuring correct evaluation even in complex expressions containing parentheses. The implementation guarantees accurate results and handles various operator precedence levels.

1. **Screenshots of the Execution**





**X. Summary**

In summary, the implemented stack-based calculator effectively evaluates arithmetic expressions by employing a stack-based algorithm. It handles parentheses and different operator precedence levels accurately, ensuring correct results for a wide range of input expressions. The implementation is robust, providing meaningful error messages for invalid expressions, and offering a user-friendly experience.

1. **Bibliography**
2. **UpGrad Stack module:**  
   <https://learn.upgrad.com/course/4546/segment/40195/234835/717349/3621605>
3. **Java Documentation - Stack Class.**  
     
   The official Java documentation on the Stack class was referenced for understanding stack operations and usage, aiding in the implementation of the stack-based algorithm.
4. **Youtube Resources**
5. **Annexure**

**Calculator class:-**

import java.util.Stack**;**import java.util.Scanner**;**class Calculator {  
 private Stack<Integer> numbers**;** private Stack<Character> operators**;** public Calculator() {  
 numbers = new Stack<>()**;** operators = new Stack<>()**;** }  
  
 public int evaluateExpression(String expression) {  
 char[] tokens = expression.toCharArray()**;** for (int i = **0;** i < tokens.length**;** i++) {  
 if (tokens[i] == ' ') {  
 continue**;** }  
  
 if (Character.*isDigit*(tokens[i])) {  
 StringBuilder sb = new StringBuilder()**;** while (i < tokens.length && (Character.*isDigit*(tokens[i]) || tokens[i] == '.')) {  
 sb.append(tokens[i])**;** i++**;** }  
 i--**;** try {  
 numbers.push(Integer.*parseInt*(sb.toString()))**;** } catch (NumberFormatException e) {  
 throw new IllegalArgumentException("Invalid number format")**;** }  
 } else if (tokens[i] == '(') {  
 operators.push(tokens[i])**;** } else if (tokens[i] == ')') {  
 while (operators.peek() != '(') {  
 if (operators.isEmpty() || numbers.size() < **2**) {  
 throw new IllegalArgumentException("Invalid expression")**;** }  
 evaluateTopOperator()**;** }  
 operators.pop()**;** // Pop the opening parenthesis  
 } else if (tokens[i] == '+' || tokens[i] == '-' || tokens[i] == '\*' || tokens[i] == '/') {  
 while (!operators.isEmpty() && hasPrecedence(tokens[i]**,** operators.peek())) {  
 if (numbers.size() < **2**) {  
 throw new IllegalArgumentException("Invalid expression")**;** }  
 evaluateTopOperator()**;** }  
 operators.push(tokens[i])**;** } else {  
 throw new IllegalArgumentException("Invalid character in the expression")**;** }  
 }  
  
 while (!operators.isEmpty()) {  
 if (operators.isEmpty() || numbers.size() < **2**) {  
 throw new IllegalArgumentException("Invalid expression")**;** }  
 evaluateTopOperator()**;** }  
  
 if (numbers.size() != **1** || !operators.isEmpty()) {  
 throw new IllegalArgumentException("Invalid expression")**;** }  
  
 return numbers.pop()**;** }  
  
 private boolean hasPrecedence(char operator1**,** char operator2) {  
 if (operator2 == '(' || operator2 == ')') {  
 return false**;** }  
 return (operator1 == '\*' || operator1 == '/') && (operator2 == '+' || operator2 == '-')**;** }  
  
 private void evaluateTopOperator() {  
 char operator = operators.pop()**;** int operand2 = numbers.pop()**;** int operand1 = numbers.pop()**;** int result = **0;** switch (operator) {  
 case '+':  
 result = operand1 + operand2**;** break**;** case '-':  
 result = operand1 - operand2**;** break**;** case '\*':  
 result = operand1 \* operand2**;** break**;** case '/':  
 if (operand2 == **0**) {  
 throw new ArithmeticException("Division by zero")**;** }  
 result = operand1 / operand2**;** break**;** }  
  
 numbers.push(result)**;** }  
}

**Main class:-**

import java.util.Scanner**;**public class Main {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*)**;** System.*out*.print("\nPlease use brackets '(' or ')' in case of BODMAS calculations.\nThe calculator might give wrong answers otherwise.\n\nEnter the expression: ")**;** String expression = scanner.nextLine()**;** Calculator calculator = new Calculator()**;** try {  
 int result = calculator.evaluateExpression(expression)**;** System.*out*.println("Result: " + result)**;** } catch (Exception e) {  
 System.*out*.println("Error: " + e.getMessage())**;** }  
 }  
}