John Kucera

Prof. Ioan Salomie

CMSC 350 Data Structures and Analysis

23 January 2020

Week 2: Project 1 Solution Description

**Infix Expression Evaluation GUI**

1. **Assumptions, Main design decisions, and error handling**

I have assumed that anything can be entered the GUI. This includes white space, characters, and any other possible tokens. The one thing that I have assumed about the numbers themselves is that they will be unsigned integers. Any decimal points in the input will count as illegal tokens.

While I have assumed that valid input only includes the four operators + - \* / and digits 0-9, I have tried to catch each possible error that could be entered through the GUI. This includes:

1. EmptyStackException - for cases where the parentheses and operators might be misplaced
2. NullPointerException – for cases where the input is empty
3. DivideByZero – for cases where the input expression attempts to divide by zero
4. IllegalToken – for cases where the input expression includes tokens that are NOT + - \* / 0-9. White space does not count here

I handle White Space in the InfixEval class where it is bypassed and ignored as the input expression is broken down. In addition, when writing the expression and result to Results.txt, I have IOException in a catch statement to account for failed or interrupted file writing.

As for the main design decisions, I always keep custom exception handlers as their own public constructor classes – for this project, that includes DivideByZero.java and IllegalToken.java. This is to allow simple access for all substantial classes.

The main classes for this project are P1GUI.java and InfixEval.java. P1GUI is meant to display the GUI and grab the data while catching the exceptions for invalid input. P1GUI also initiates the file writing for Results.txt. P1GUI also holds a listener class, EvaluateBtnListener, which tells the program how to respond when the Evaluate button is clicked.

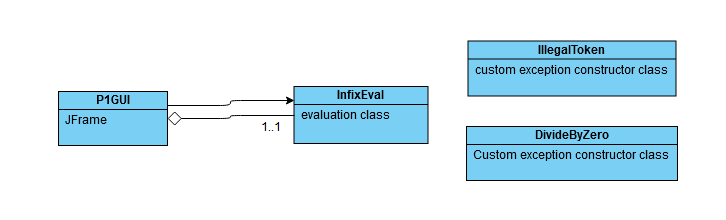
InfixEval is a class that holds four methods:

1. convertToken – converts the input into individual tokens. Eliminates white space and accounts for double digit numbers.
2. precedence – ensures that \* / operate before + -
3. calculate – executes operations between two operands and one operator
4. evaluate – breaks down input expression and puts each token into appropriate stacks. Then, it manages each operation among the stacks and appropriately puts operands/operators through the calculate method. The final result is left in the operand stack.

I decided to design the program in a way that would let each token flow individually so the stacks would be accurately formed. The input expression is retrieved through the GUI; characters are popped and converted into tokens; the operators among these go through a precedence check; the tokens are stacked; finally, operations are executed between two operands and one operator until no operators remain. A final result is left in the operand stack, which is displayed in the “Result” text field.

Finally, I designed Results.txt to NOT be cleared whenever a new GUI is started up. All expressions and results will be recorded during a session and will stay in Results.txt when the current GUI exits. Results.txt will REMAIN THE SAME when the program executes again, and all new expressions/results will be appended.

1. **UML Class diagram**

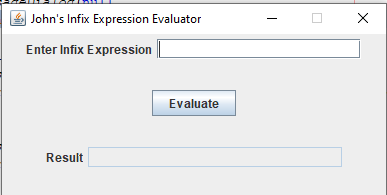


1. **Test Cases**

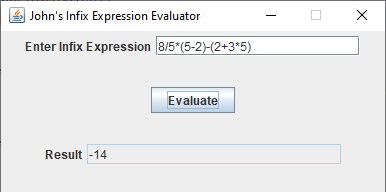
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **What aspect is tested** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| If the user can close the GUI | Click “X” button on frame | GUI is closed and exited | GUI is closed and exited | PASS |
| Addition and multiplication. Precedence of  multiplication | 1+5\*8  Press evaluate | Result: 41 | Result: 41 | PASS |
| Subtraction and division. Precedence of division. White space included | 40- 6 /3  press evaluate | Result: 38 | Result: 38 | PASS |
| Addition and division. Precedence of division. | 16/4+7+7  press evaluate | Result: 18 | Result: 18 | PASS |
| Subtraction and multiplication. Precedence of multiplication. White space included | 9 \* 9 -34  press evaluate | Result: 47 | Result: 47 | PASS |
| Precedence of operations inside parentheses. | (2+3\*5)-8/5\*(5-2)  press evaluate | Result: 14 | Result: 14 | PASS |
| Addition and subtraction, same precedence | 20+30-40  press evaluate | Result: 10 | Result: 10 | PASS |
| Multiplication and division, same precedence. White space included | 5 \* 6 \* 7 / 3  press evaluate | Result: 70 | Result: 70 | PASS |
| Results.txt output file | All above tests ^ | (In Results.txt)  1+5\*8 = 41  40- 6 /3 = 38  16/4+7+7 = 18  9 \* 9 -34 = 47  (2+3\*5)-8/5\*(5-2) = 14  20+30-40 = 10  5 \* 6 \* 7 / 3 = 70 | (In Results.txt)  1+5\*8 = 41  40- 6 /3 = 38  16/4+7+7 = 18  9 \* 9 -34 = 47  (2+3\*5)-8/5\*(5-2) = 14  20+30-40 = 10  5 \* 6 \* 7 / 3 = 70 | PASS |
| Division by zero | 10/0  press evaluate | Message Dialog Box: Division by zero is not allowed. Please try again. | Message Dialog Box: Division by zero is not allowed. Please try again. | PASS |
| Misplaced parentheses | (80+50(  press evaluate | Message Dialog Box: Your Infix Expression is invalid due to misplaced Operators, Operands, or Parentheses. Please try again. | Message Dialog Box: Your Infix Expression is invalid due to misplaced Operators, Operands, or Parentheses. Please try again. | PASS |
| Misplaced parentheses | )70/7)  press evaluate | Message Dialog Box: Your Infix Expression is invalid due to misplaced Operators, Operands, or Parentheses. Please try again. | Message Dialog Box: Your Infix Expression is invalid due to misplaced Operators, Operands, or Parentheses. Please try again. | PASS |
| Misplaced operators | 5--1  press evaluate | Message Dialog Box: Your Infix Expression is invalid due to misplaced Operators, Operands, or Parentheses. Please try again. | Message Dialog Box: Your Infix Expression is invalid due to misplaced Operators, Operands, or Parentheses. Please try again. | PASS |
| Misplaced operators | 60-6+  press evaluate | Message Dialog Box: Your Infix Expression is invalid due to misplaced Operators, Operands, or Parentheses. Please try again. | Message Dialog Box: Your Infix Expression is invalid due to misplaced Operators, Operands, or Parentheses. Please try again. | PASS |
| Illegal tokens: letters | 70y\*90a  press evaluate | Message Dialog Box: Only characters including 0 to 9, +, -, \*, /, (, ) are allowed. Please try again. | Message Dialog Box: Only characters including 0 to 9, +, -, \*, /, (, ) are allowed. Please try again. | PASS |
| Illegal tokens: misc. characters | 60^7%3  press evaluate | Message Dialog Box: Only characters including 0 to 9, +, -, \*, /, (, ) are allowed. Please try again. | Message Dialog Box: Only characters including 0 to 9, +, -, \*, /, (, ) are allowed. Please try again. | PASS |
| No input | No input  Press evaluate | Message Dialog Box: Please enter an Infix Expression. | Message Dialog Box: Please enter an Infix Expression. | PASS |

1. **Screenshots of program execution**

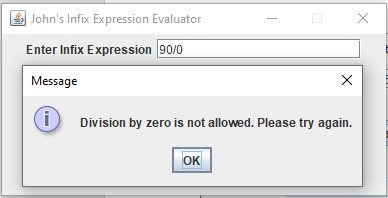
GUI:



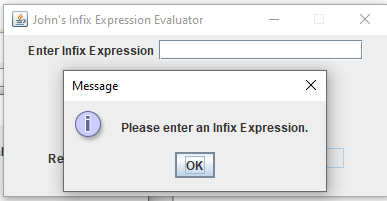
Evaluating an Infix Expression:



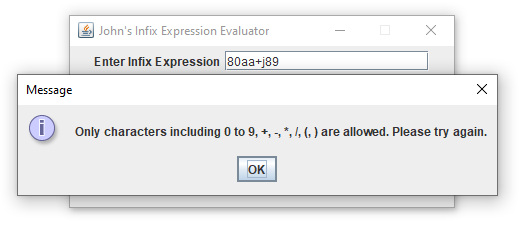
Division by zero error:



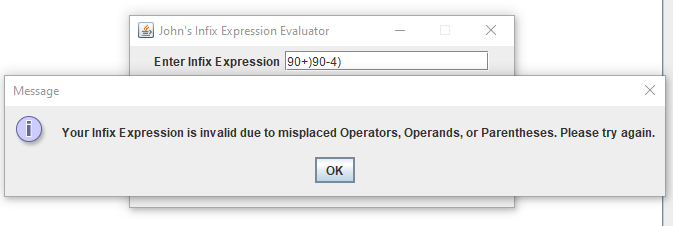
Empty input error:



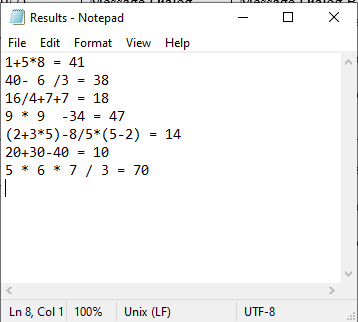
Illegal tokens error:



Misplaced operators/parentheses error:



Results.txt (from the test case):



1. **Lessons learned**

The biggest lesson I learned during this project was that it is important to debug as the code is being written. I have always had a bad habit of trying to write out all the code of a program and testing it very rarely throughout. This is a poor practice because if there are bugs in the program, it is hard to tell where they are coming from and I need to go back and insert print statements. It feels like taking a big step backwards, and just feels overwhelming when I have this fully written program in front of me and I barely have a clue where the bugs might be. However, if I continuously test my program as I write it, I can feel secure about what I have written so far and that it runs. If a bug appears, I usually only need to debug through the most recently written code. Simply put, debugging is a lot more efficient this way.

A minor lesson I learned during this project was how convenient it is to write custom exceptions as separate constructor classes (I did this with DivideByZero and IllegalToken). While I have done this before previous Java programs, those did not have multiple main/important classes so I never saw much importance in the separate constructor classes. This program, however, uses both P1GUI and InfixEval, and both of those classes require the use of the custom exceptions. It is more simple than I thought: I create the custom exception constructor classes, throw the exceptions in InfixEval, and catch them in P1GUI. I will definitely use the same format for future projects.

Finally, this was my first experience with stacks, and I am glad that I can understand them well already. This week’s readings and examples helped immensely for grasping the concept of stacks, which I could then successfully apply to this project. I also researched outside videos that introduce how stacks work in Java. I look forward to using stacks again in future projects.