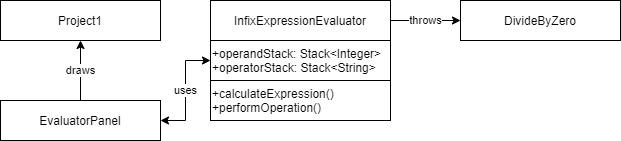
# CMSC350 Project 1 Documentation

By Brian Yu

## Goal

The goal of this project was to develop a in-fix expression calculator that would read in an in-fix expression and display the result. The calculator would observe order of operations and prevent the user from dividing by zero. It would also be able to parse the expression with our without spaces.

## UML Diagram

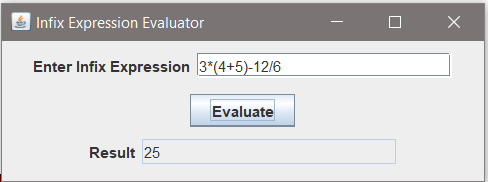


## Test Cases

I performed basic functional testing to make sure my program could handle the expected use cases of in-fix expressions with only integers, arithmetic operators, and parenthesis. I included an edge case to test for a mix of spaces and no spaces and an error case for dividing by zero.

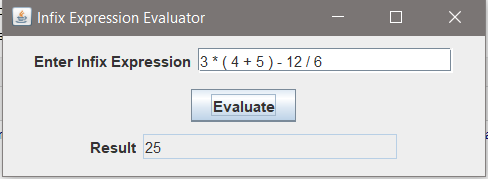
### Test Case 1

This test case inputted an in-fix expression containing all the arithmetic operators and a set of parenthesis (without spaces) to test to see if the algorithm respects order of operations. I inputted “3\*(4+5)-12/6” with the expected result of 25. This test case passed.



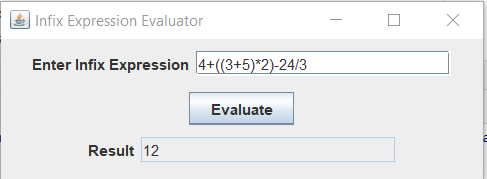
### Test Case 2

This test case inputted an in-fix expression containing all the arithmetic operators and a set of parenthesis (with spaces) to test to verify spaces did not affect the result. I inputted the same expression as test case 1 to verify “3 \* ( 4 + 5 ) – 12 / 6”. This test case passed.



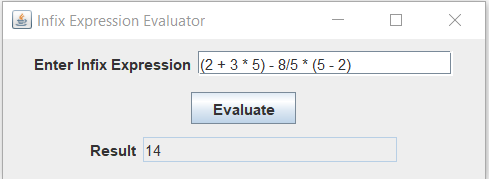
### Test Case 3

This test case is an edge case that tested an in-fix expression with nested parenthesis to verify that the calculator could handle them. I inputted the expression “4+((3+5)\*2)-24/3” with the expected result of 12. This test case passed.



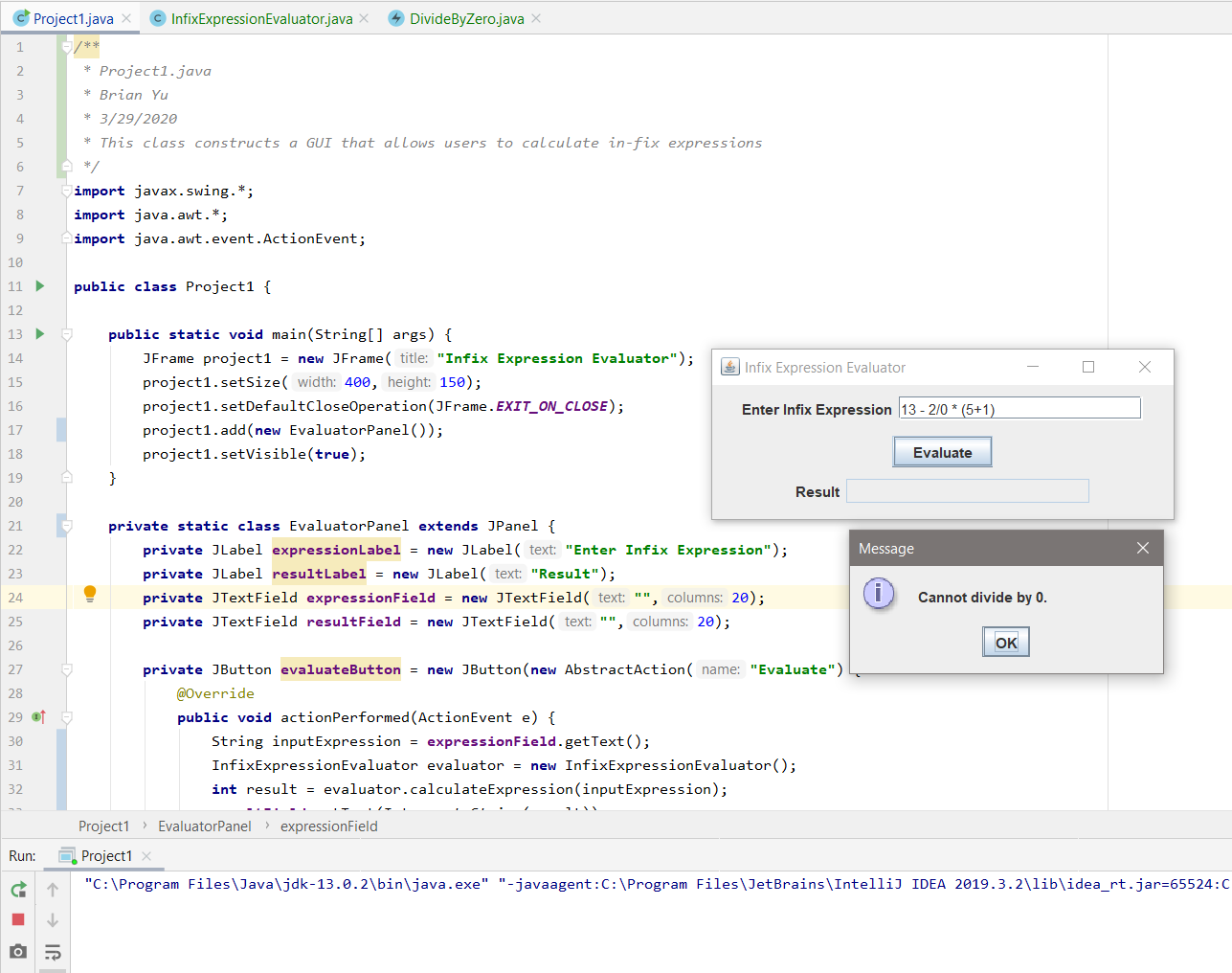
### Test Case 4

This test case is an edge case that tested a mix of spaces and no spaces between the operands and operators to see if the calculator could handle this mix. I inputted the expression shown in the project documentation “(2 + 3 \* 5) – 8/5 \* (5 - 2)” with expected result of 14. This test case passed.



### Test Case 5

This test case is an error case that tested an in-fix expression that divided by 0 to verify an dialog would appear and the calculator would handle it gracefully. I inputted the expression “13 – 2/0 \* (5+1)” with the expected result that no exception was thrown in the console and a dialog pop-up appears. This test case passed.



### Lessons Learned

Since the algorithm was already written out for me and I used the built-in stack collections in Java, my main challenge in this project was how to parse the inputted in-fix expression string with and without spaces. Since I was required to handle the expression with or without spaces, I couldn’t use the spaces as a delimiter to split the string. I also couldn’t use the operators since I needed to include those to add to the operator stack. Initially I tried using a character array, but I could not easily handle integers with multiple digits. I then tried to use a regex expression to look ahead and look behind when I set the arithmetic operators as delimiters, but this did not work for the parenthesis. I finally settled on using any non-digit character as a delimiter which works for the requirements for the project but does not protect against unsupported characters.

My method for parsing also created other problems as the tokens were stored in a String array. This made it difficult for detecting operands as there is no easy way to determine whether a String is an integer or not. I ended up using a clunky workaround where I tried to see if the first character of each string (operator or operand) was a digit. While this works in practice since even multiple digit integers will have the first digit as an integer, it isn’t the most elegant solution. Another solution I considered was to write a method to catch an ArithmeticException that would be thrown when an operator string was encountered, but I thought that would be too inefficient to do so for each string in the array.

I also deviated slightly from the provided algorithm by adding an additional step in the else-if statement for handling right-sided parentheses. Outside of the while loop, I added a statement to pop out the top of the operator stack. Since this statement was outside of the while loop which checked for all operators except the left-sided parenthesis, this statement served to pop-up out the remaining left-sided parenthesis after the expression inside the parenthesis had been executed. I had to add this statement as I was running into EmptyStackExceptions when the algorithm attempted to perform an operation with the left-sided parenthesis as an operator.

Finally, I recognize that there might be some controversy around the way I implemented how to check for operator precedence. Following the algorithm, I realized that if I checked to see if the operator on top of the operator stack was either a multiplication or division operator, it would always fulfill the requirement of being a higher precedence or same precedence as the operator going through the if-else statements. While technically if an addition or subtraction operator going through the if-else statements would have the same precedence as an addition or subtraction operator on top of the operator stack, this case does not affect the end result due to the stack’s LIFO structure. Parenthesis are also handled by earlier if-else statements so I decided that my method was sufficient for the requirements.