Project 2 Write Up: Dolev Peleg

Approach, Design, and Algorithm

My approach to this project was to first design the data structures classes MyQueue and MyStack. Because the Notation utility class is built upon using these data structures.

I started with the MyQueue class, because it was pretty similar to our last lab, so I felt more comfortable designing it. As I expected, designing a circular array based queue for the second time was not as daunting as it was the first time. I did not encounter any issues or obstacles when creating this class.

After testing each of the MyQueue's methods in a separate driver class, I moved on to creating the MyStack class. I decided to implement it using a linked list, because I never worked with linked lists before, and I wanted to get familiar with this technique. Using both my notes, the book, and the lectures, I managed to successfully create and test MyStack without encountering any issues.

It is worth mentioning, that as I created the data structures, I also created the exception classes that these classes' methods throw: InvalidNotationFormatException, StackUnderflowException, StackOverflowException, QueueUnderflowException, and QueueOverflowException.

After creating and ensuring the data structures, and the exception classes work as expected, I moved on to creating the Notation class. Creating this class was the most challenging task for me during this project, and I learned a lot by designing it. The first challenge I encountered was that the algorithms of the methods were hard to understand. After watching the "video notes" provided by my instructor, I understood these algorithms better, and I created comments throughout my class to ensure I know what I need to do.

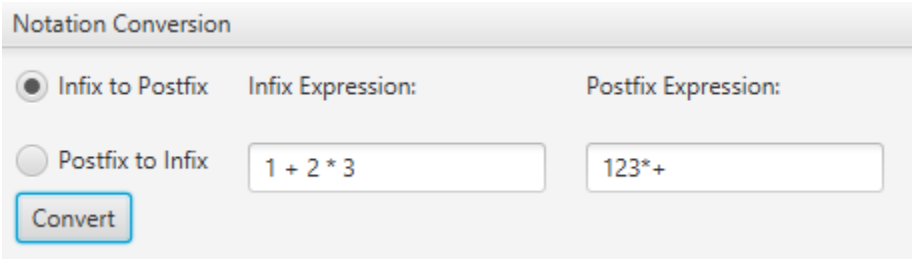
The next challenge I faced was figuring out what type of objects should each one of my stacks hold. After some trial and error, I decided that in the convertInfixToPostfix method, the stack will hold Character objects, while in convertPostfixToInfix I will use String objects instead. That is because in the provided algorithm there is a requirement for "If there is only one value in the stack – it is the infix string, if more than one value, throw an error" at the end of this method. When trying to work with the Character class in this stack, my method kept throwing the InvalidNotationFormatException, because the stack almost always holds more than one character. After learning this, I decided to hold Double objects in my evaluatePostfixExpression. That is because it is much easier returning a double from this method if Double objects are used in this stack.

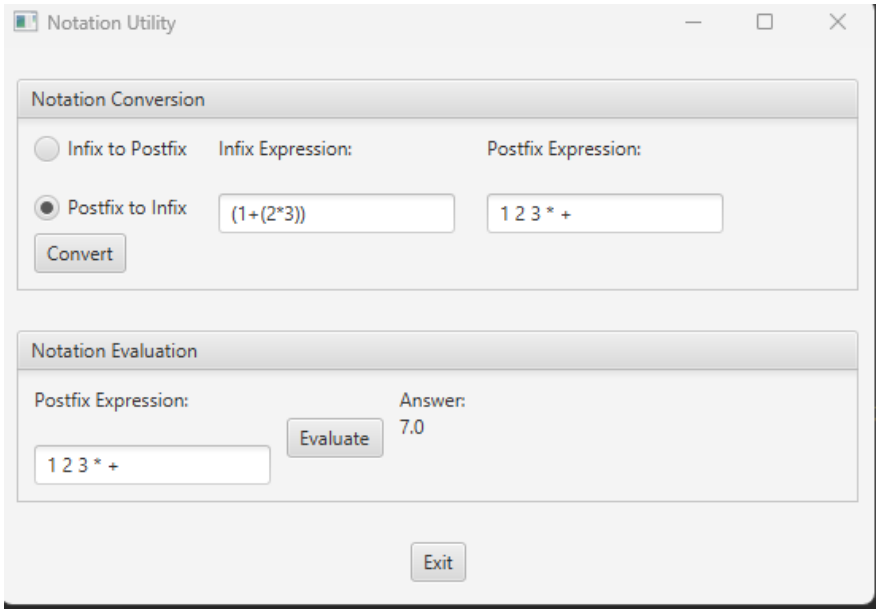
One important thing is, it was instructed by the write-up, and by the provided interface, that some methods will throw an exception (and therefore will have 'throws' in their header) . In order to use the provided test cases, I had to add a few try-catch blocks. I commented where I added these blocks.

Test Runs and Cases

Case 1:

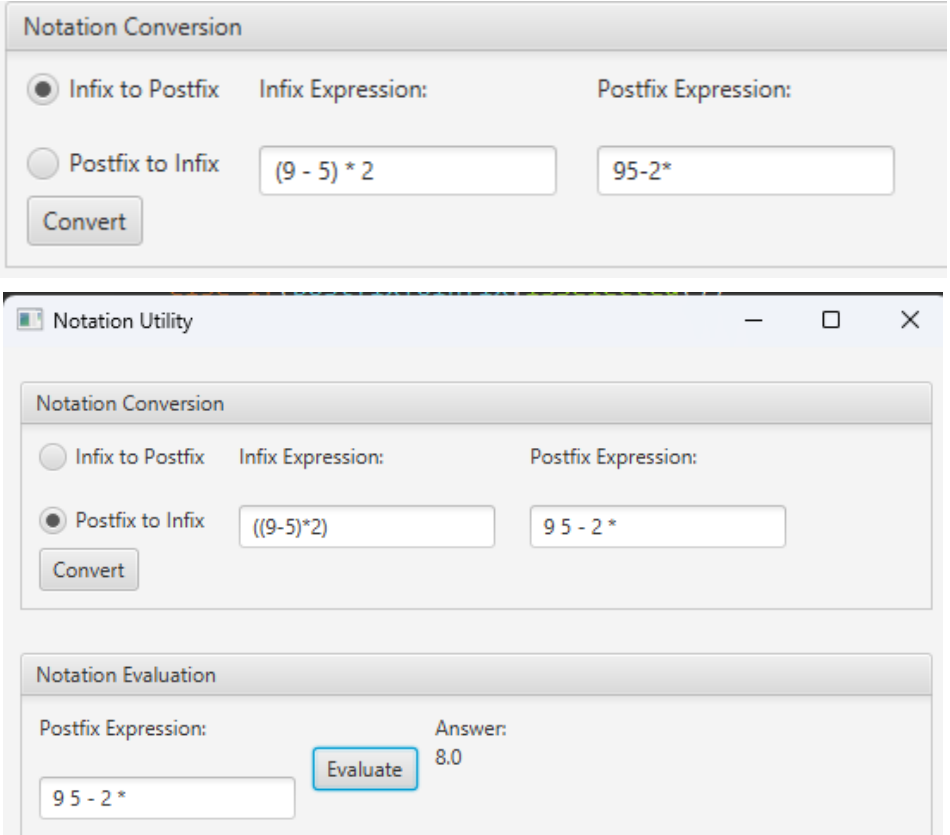
Input	Expected Output	Actual Output
Infix to postfix: 1 + 2 * 3 Postfix to infix:1 2 3 * + Postfix Evaluation: 1 2 3 * +	Infix to postfix:123*+ Postfix to infix:(1+(2*3)) Postfix Evaluation: 7.0	Infix to postfix:123*+ Postfix to infix:(1+(2*3)) Postfix Evaluation: 7.0





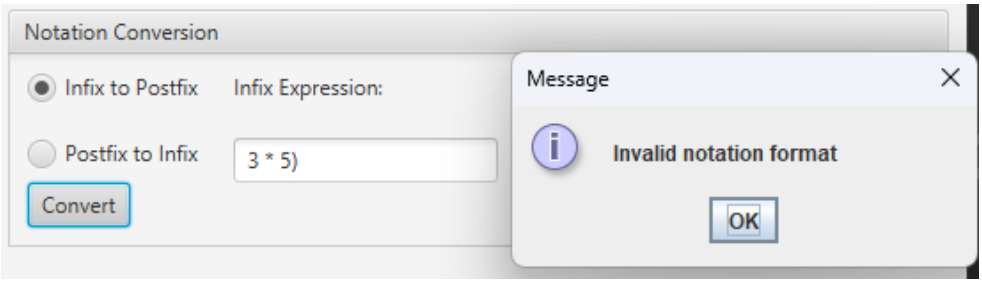
Case 2:

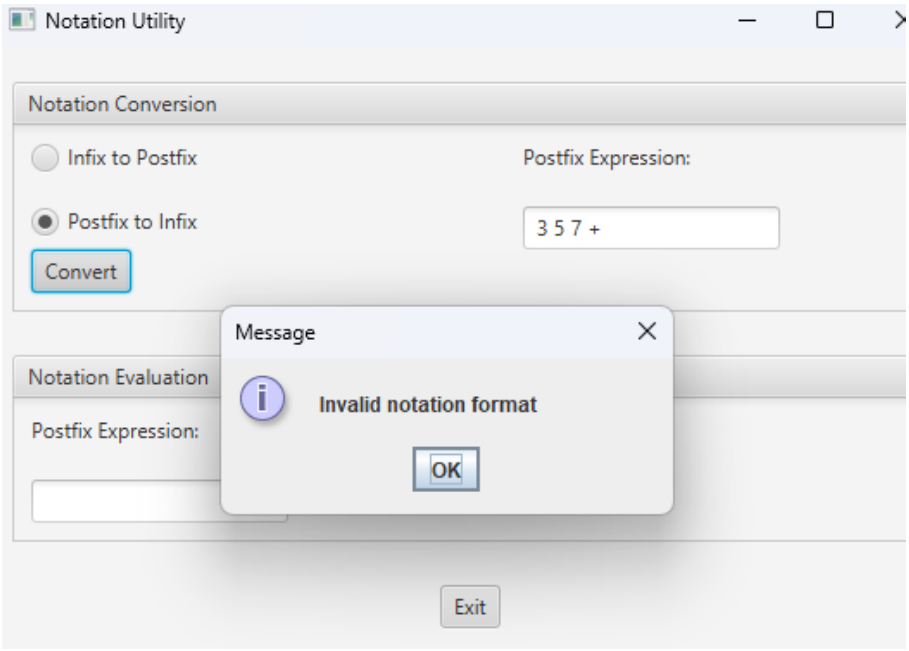
Input	Expected Output	Actual Output
Infix to postfix: (9 - 5) * 2 Postfix to infix: 9 5 - 2 * Postfix Evaluation: 9 5 - 2 *	Infix to postfix: 95-2* Postfix to infix: ((9-5)*2) Postfix Evaluation: 8.0	Infix to postfix: 95-2* Postfix to infix: ((9-5)*2) Postfix Evaluation: 8.0

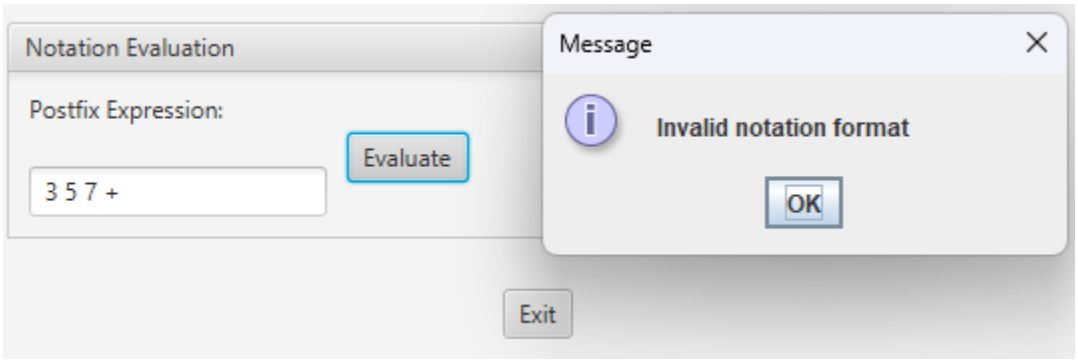


Case 3:

Input	Expected Output	Actual Output
Infix to postfix: 3 * 5) Postfix to infix: 3 5 7 + Postfix Evaluation: 3 5 7 +	Infix to postfix: Exception Postfix to infix: Exception Postfix Evaluation: Exception	Infix to postfix: Exception Postfix to infix: Exception Postfix Evaluation: Exception

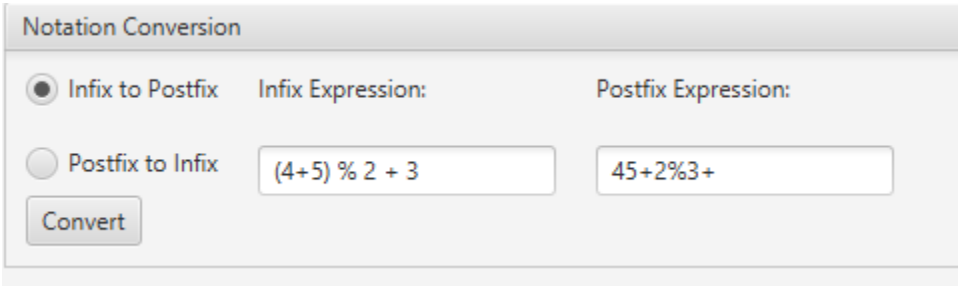


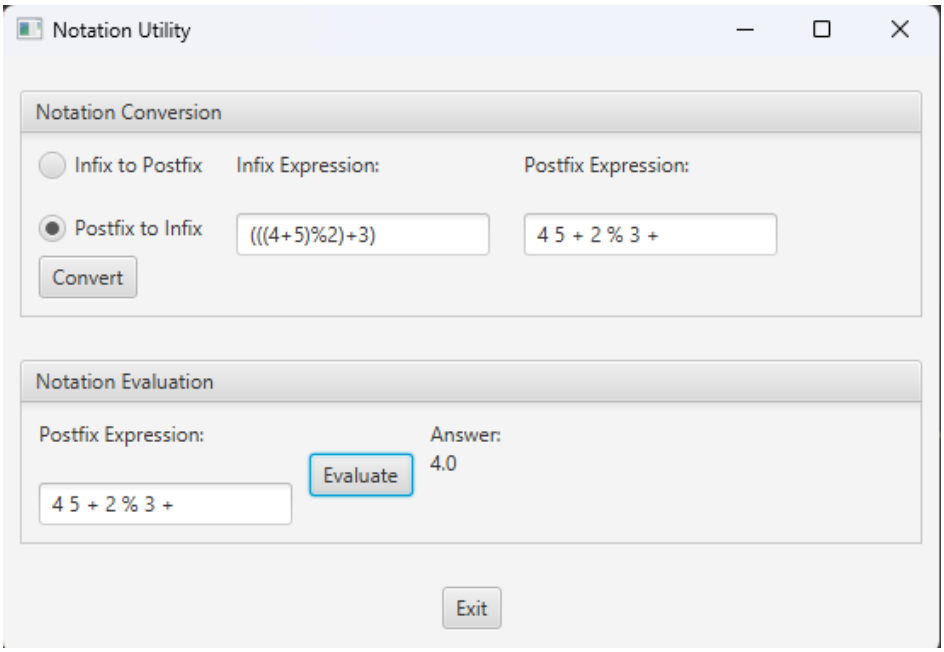




Case 4:

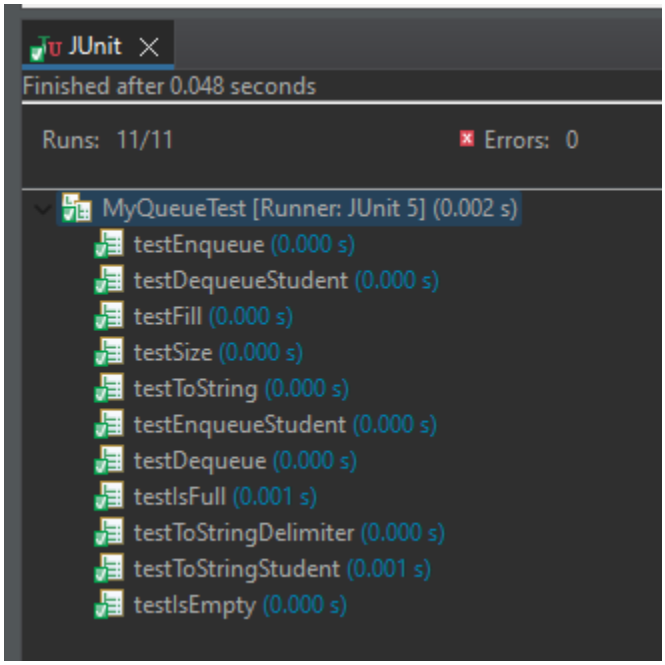
Input	Expected Output	Actual Output
Infix to postfix: (4+5) % 2 + 3 Postfix to infix: 4 5 + 2 % 3 + Postfix Evaluation: 9 5 - 2 *	Infix to postfix: 45+2%3+ Postfix to infix: (((4+5)%2)+3) Postfix Evaluation: 4.0	Infix to postfix: 45+2%3+ Postfix to infix: (((4+5)%2)+3) Postfix Evaluation: 4.0



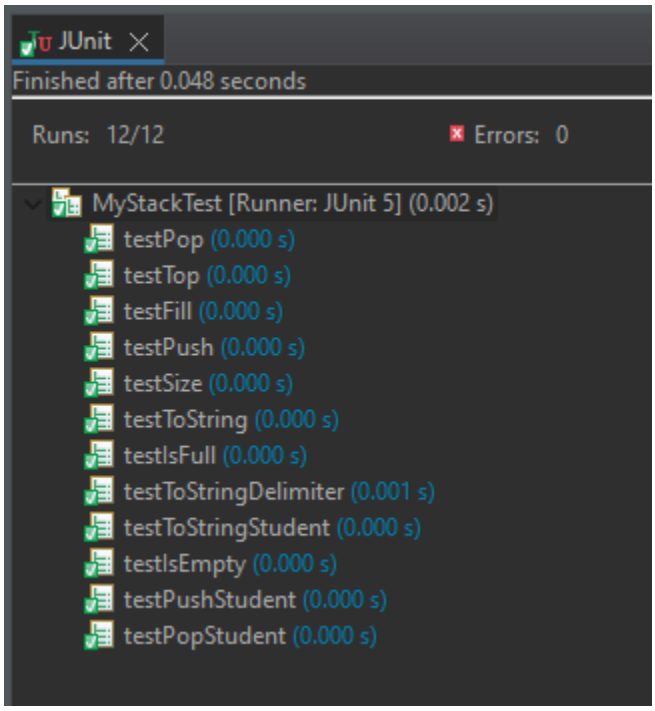


JUnit Testing

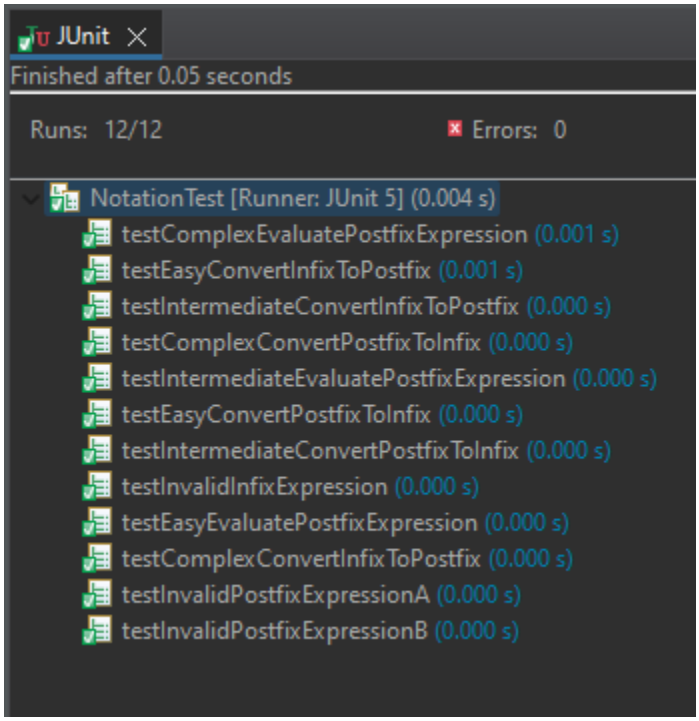
MyQueueTest



MyStackTest



NotationTest



<u>Learning Experiences</u>

One thing I learned during this project was how to work with this many try-catch blocks. Almost every step in my algorithm could throw an exception, and tracking which block throws which exception can be challenging. To solve this, I decided on catching all the exceptions with one catch block, by using *catch(Exception e)*, and the printing each exception message with *System.out.println(e.getMessage()),* thus, I could keep coding without worrying about which exception is thrown by the block. I am sure that next time I have to deal with this many try-catch blocks I will do it with more ease and my code will look more organized.

Working with recursion, even if just for one method, was new to me. It was confusing to understand how to use the toString methods of the MyStack. I did not understand how to reverse the order of the string (duo to the method using recursion). In order to pass the provided tests, I had to fix it by using a nested for loop in order to reverse each string in addition to the entire stack's string.

Besides my notes, the book, and the lectures, I did not use any other material.

Assumptions

1. The user will be using JUnit 5, Java, and JavaFX
2. JUnit test cases will use try-catch blocks when using methods that were instructed to have a ‘throws’ keyword in their header.
3. Operands will be single digit numbers

Enhancements

- No enhancements were made.