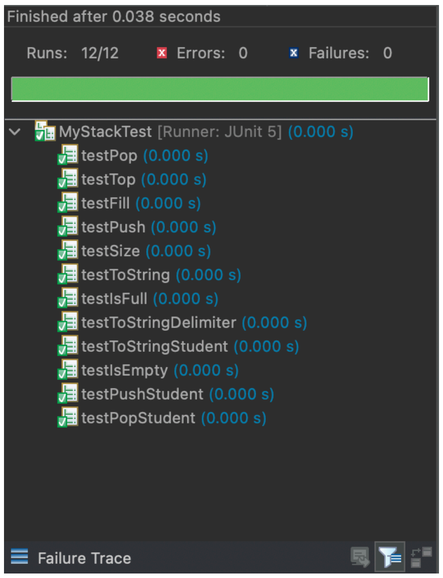
Design:

For this project, I decided to work on the exceptions first and got those out of the way. I figured it made more sense to knock out the small, simple enough to allow myself more time on the complex parts. I then worked on the stack data structure since I felt more confident than starting with the queue. Following your interface, I first created two constructors, one that took in an int for the size of the stack and then another as the default. Both isEmpty() and isFull() were straight forward, just comparing the number of elements to 0 or to the stack capacity. I worked on the pop method next. I first created a T variable which would be the variable we would be returning. I checked is the stack was empty and to throw an error message. If not, I then assigned the element at the top of the stack to the variable. I deleted that element from the stack and decreased the number of elements count in the stack. For the top() method, I used the same methodology as in pop(), but didn't remove the element from the stack. I then worked on the push method. I checked if the stack was full. If not, I then added the given element to the stack and increased the number of elements. For the toString method, I used a for loop to run through the stack and grab each element from the ArrayList stack and add to the string. For the toString(delimiter) method, in the for loop, I first checked to see if the index was not at the end of the stack. If not, I added the element plus the delimiter to the string. Once the stack got to the last element, I used the else statement to add the remaining element to the stack. For the fill method, I first created a copy ArrayList of the given ArrayList to prevent any security breeches. I then added all of the elements in the given ArrayList to the stack. I made sure to update the number of elements in the stack to the newly copied stack.

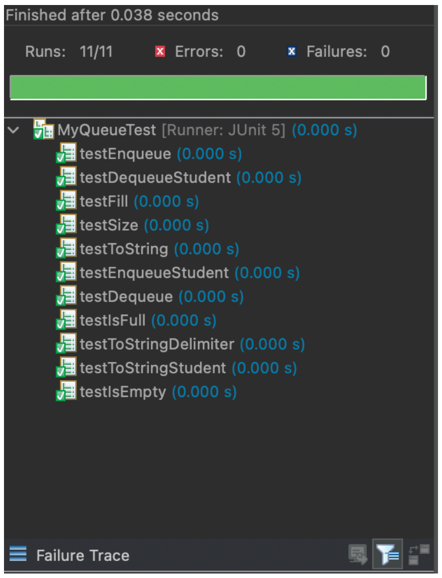
Next, I worked on the queue data structure. I started off with the two constructors. The first one takes in an int for the stack size. It updates the queue to have the given capacity as well as establishing values for the front and last element count. I then made a default constructor to establish all the variable values. Both isEmpty() and isFull() were straight forward, just comparing the number of elements to 0 or to the queue capacity. For dequeue method, I initialized a T variable. I checked if the queue was empty, if so, to throw an error. If not, then it returns the element at the front of the queue. It also increases the front element count as well as deleting the element from the queue. For enqueue method, I checked if the queue was full or not. If not, I then added the given element to the end of the queue based off of the last element count. I then increased increased the last element count as well as the number of elements count. For both toString methods and the fill method, I used the exact same template as MyStack.

Lastly, I moved onto the Notation class. I first started with the convertPostfixtoInfix method. Using a try, I used a for loop to run through the length of the given postfix expression. I then checked if the current character was a space. If so, continue. I checked if the current character was a digit. If so, it pushes the currentCharacter to the stack. I checked if the element was an operator and if the stack size was greater than 2. If so, it pops the first two values. The second popped element goes on the left with the operator in the middle along with the first popped element on the right, all enclosed in parenthesis. It then pushes the string to the stack. For the convertInfixtoPostfix method, using a try, I used a for loop to run through each character of the given infix. I checked if the current character was a space, if so, to continue. I checked if the current character is a digit. If so, to enqueue (copy) the current character into the queue. I then checked if the current character was a left parenthesis. If so, it pushes the current character to the stack. I then divided up the two levels of precedences with the higher and lower. I used a if statement to check for */%. If so, it checks if the stack is not empty. If not empty, while the top of the stack as equal or higher precedence to the higher operator, then insert the popped operator to the queue. I used a if statement to check for +-. If so, it checks if the stack is not empty. If not empty, while the top of the stack as equal or higher precedence to the higher operator, then insert the popped operator to the queue. I then checked if the current character was a right parenthesis. I used a while statement (checks if stack is not empty and not the top is not equal to a left parenthesis. It then copies the popped operators to the queue. I then used a if statement to check is the stack was empty or the top of the stack didn't have a left parenthesis, it throws an errors. Then using another if statement, I checked if the stack was not empty and the top of the stack is a left parenthesis. If so, then it pops the left parenthesis from the stack. Once everything has been read, I used a while statement that checks if the stack is not empty and the top of the stack is not a left parenthesis. If so, it pops the remaining operators and inserts them to the queue. I then worked on the evaluatePostfixExpression. Using a try, I used a for loop to run through the postfix expression. If the current character is a space, then continue. If the current character is a digit or left parenthesis, then it pushes the current character to the stack. If the current character is a operator, it then checks if the stack size is less than 1. If not, then it pops the top two values from the stack. Both popped elements are then put into my arithmeticCalculation method. This method takes in a two strings (first popped number and second popped number), and the operator. I parsed both of the strings to doubles. I then used if else if statements to check for each operator and to solve based on the operator and parsed values. If there is no operator, it throws an error. It then returns the calculated value in double form. Going back to the evaluatePostfixExpression method, I then pushed the newly calculated double to the stack. Finally, after reading everything, if the stack is greater than one, throws an error. If not, then returns the calculated expression.

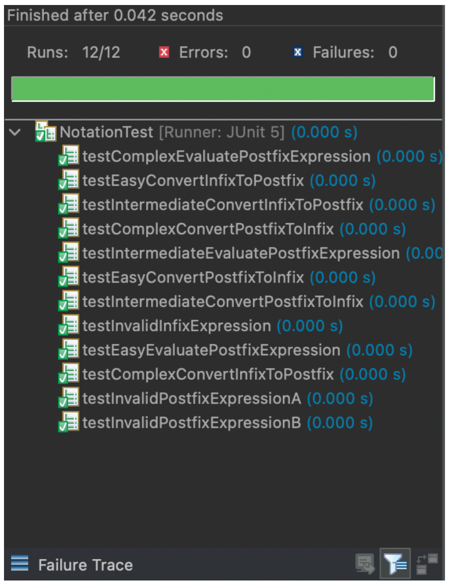
J Unit tests:



```
Finished after 0.038 seconds

Runs:  12/12     Errors:  0     Failures:  0

MyStackTest [Runner: JUnit 5] (0.000 s)
    testPop (0.000 s)
    testTop (0.000 s)
    testFill (0.000 s)
    testPush (0.000 s)
    testSize (0.000 s)
    testToString (0.000 s)
    testIsFull (0.000 s)
    testToStringDelimiter (0.000 s)
    testToStringStudent (0.000 s)
    testIsEmpty (0.000 s)
    testPushStudent (0.000 s)
    testPopStudent (0.000 s)

Failure Trace
```



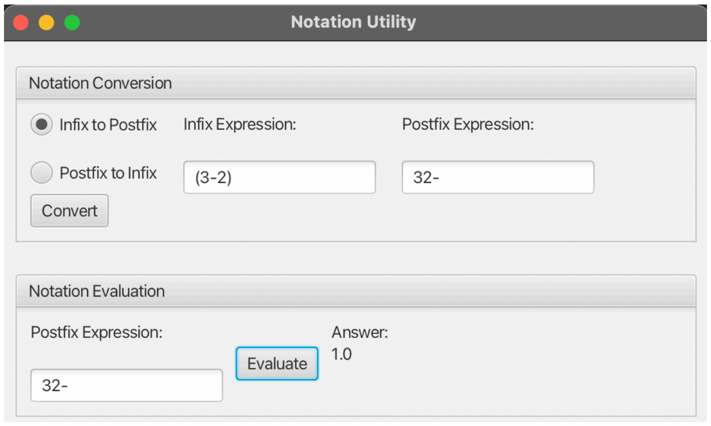
```
Finished after 0.038 seconds

Runs:  11/11     Errors:  0     Failures:  0

MyQueueTest [Runner: JUnit 5] (0.000 s)
    testEnqueue (0.000 s)
    testDequeueStudent (0.000 s)
    testFill (0.000 s)
    testSize (0.000 s)
    testToString (0.000 s)
    testEnqueueStudent (0.000 s)
    testDequeue (0.000 s)
    testIsFull (0.000 s)
    testToStringDelimiter (0.000 s)
    testToStringStudent (0.000 s)
    testIsEmpty (0.000 s)

Failure Trace
```



```
Finished after 0.042 seconds

Runs:  12/12     Errors:  0     Failures:  0

NotationTest [Runner: JUnit 5] (0.000 s)
    testComplexEvaluatePostfixExpression (0.000 s)
    testEasyConvertInfixToPostfix (0.000 s)
    testIntermediateConvertInfixToPostfix (0.000 s)
    testComplexConvertPostfixToInfix (0.000 s)
    testIntermediateEvaluatePostfixExpression (0.00
    testEasyConvertPostfixToInfix (0.000 s)
    testIntermediateConvertPostfixToInfix (0.000 s)
    testInvalidInfixExpression (0.000 s)
    testEasyEvaluatePostfixExpression (0.000 s)
    testComplexConvertInfixToPostfix (0.000 s)
    testInvalidPostfixExpressionA (0.000 s)
    testInvalidPostfixExpressionB (0.000 s)

Failure Trace
```

Test Cases:

Simple infix- (3-2) Expected: 32- , 1.0
Actual: 32-, 1.0



Notation Conversion

[✓ Infix to Postfix] Postfix to Infix

Infix Expression: (3-2)

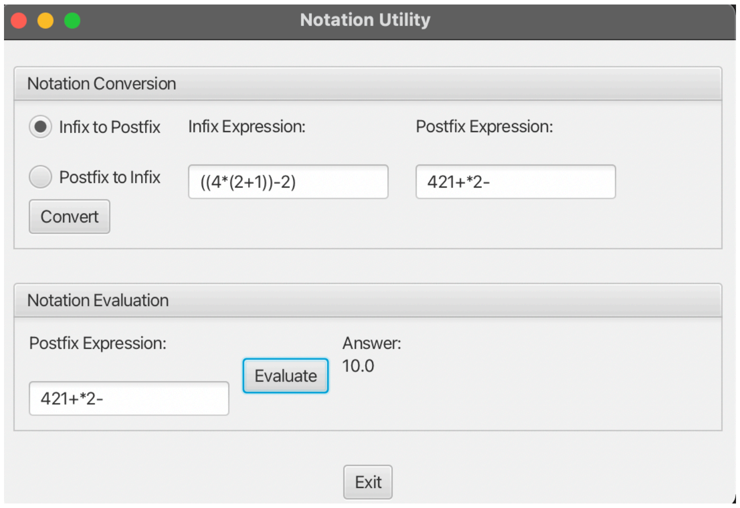
Postfix Expression: 32-

Notation Evaluation

Postfix Expression: 32-

Answer: 1.0

Intermediate infix- ((4*(2+1))-2) Expected: 421+*2- , 10.0
Actual: 421+*2- , 10.0



Notation Conversion

[✓ Infix to Postfix] Postfix to Infix

Infix Expression: ((4*(2+1))-2)

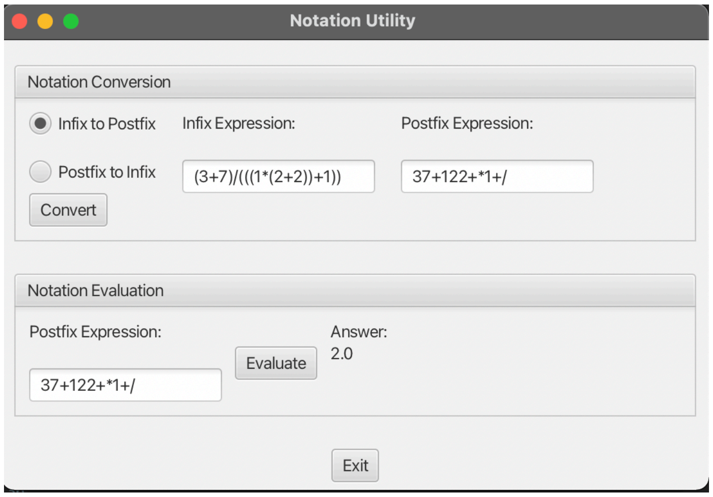
Postfix Expression: 421+*2-

Notation Evaluation

Postfix Expression: 421+*2-

Answer: 10.0

Complex infix- (3+7)/(((1*(2+2))+1)) Expected: 37+122+*1+/ , 2.0
Actual: 37+122+*1+/ , 2.0



Notation Conversion

[✓ Infix to Postfix] Postfix to Infix

Infix Expression: (3+7)/(((1*(2+2))+1))

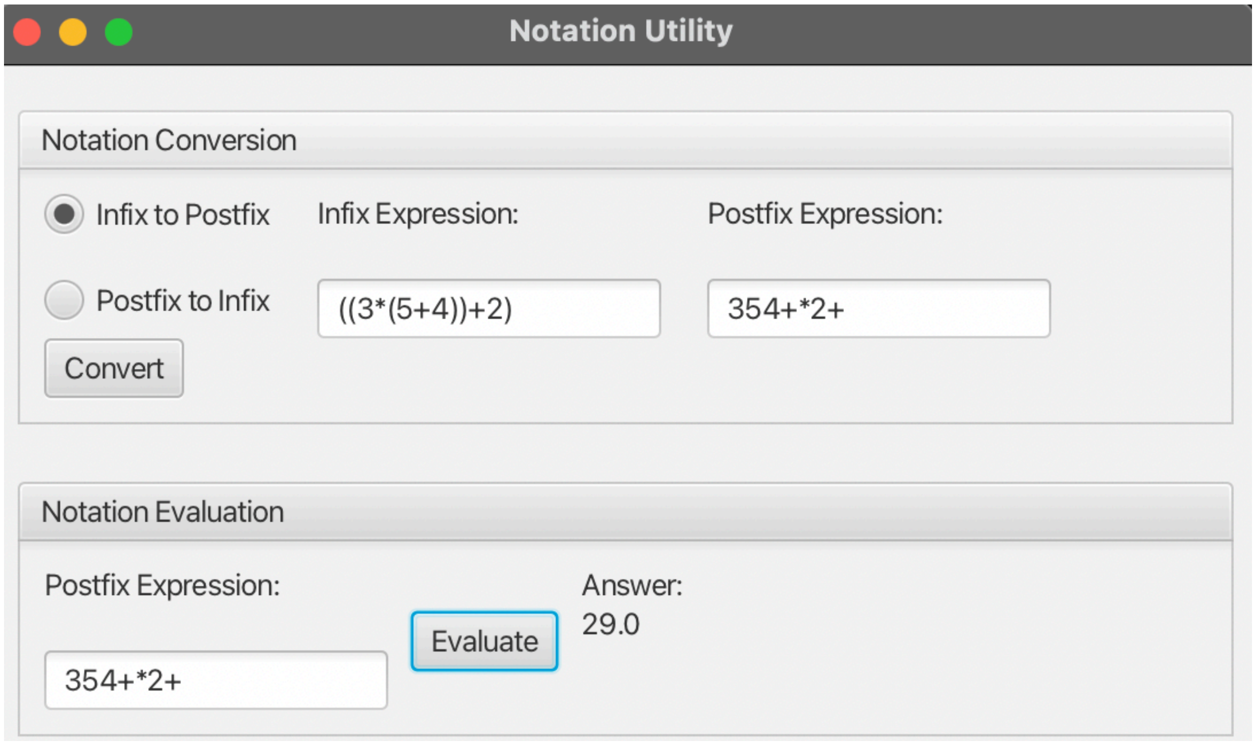
Postfix Expression: 37+122+*1+/

Notation Evaluation

Postfix Expression: 37+122+*1+/

Answer: 2.0

Application Running:



Infix Expression: ((3*(5+4))+2)

Postfix Expression: 354+*2+

Postfix Expression: 354+*2+

Answer: 29.0

Learning Experience:

I have learned a lot through this project. I have gotten comfortable creating my own stack and queue class. It's pretty straight forward, given the interface information we were to follow. I can now see the similarities and differences between the two and there isn't much confusion anymore about them. I got comfortable using ArrayLists like the last project we submitted. For the exceptions, I liked that we had multiple exceptions like the last project. It gives me good practice remembering how to write exceptions and keeps it drilling into my head. For the notation class, I honestly learned a lot about following instructions and not moving ahead too quickly. I find myself getting excited to work on the projects and kind of jump right into it. I need to slow down and really think about my design process. I need to go line by line, and make sure I am following the pseudocode instruction you have given us. Once I realized that I need to just focus on line by line in the given write up, the more smoothly the project actually went.

Assumption:

For the notation test that you provided us, I had to add the throws InvalidFormatException() to the provided tests since these were thrown on the methods in Notation class.

Question:

Please correct me if I'm wrong but based on my code, I would have to completely scrap my arithmeticCalculation method, right? In my theory on how to read double digit, I feel like I would have to take both strings and convert them into separate char arrays. Once that's been done, there must be some way to then convert it to a char arrays into that can then be computed in the if else if statements. Although I did not get to the extra credit, I would still like to know how this can be actually done since I’m having a brain fart on this. Any guidance would be greatly appreciated!