Postfix Calculator

Introduction

The purpose of this lab is to design a program to implement a calculator, but there is a difference this time around: the calculator will first perform an infix to postfix conversion and then evaluate the resulting postfix expression.

You should implement the calculator in two parts:

A **Converter** class that will convert the input string to postfix expression.
A **Calculator** class that will evaluate the postfix expression.

Both the Converter and Calculator classes should use either the ArrayStack or the LinkedStack adapted from the book. You are not allowed to use any pre-built classes in the Java library (such as ArrayList or something else).

Implementation and Design

The PostfixCalculator class

You will need to do the following in your PostfixCalculator class:

- The Calculator class will instantiate an object of the Converter class in order to have the infix expression converted to a postfix expression.

- An additional operator, "^", will be added. Java has a built in exponent function: Math.pow(x, y). Use this to evaluate x^y.

- Parentheses are legal in the infix expression (Note: the parentheses are needed in the infix expression. After your Converter class' algorithm converts the expression to postfix, it will no longer have parentheses because they are not necessary in postfix expressions.)

In postfix notation, the operator is written after the two operands. An expression with multiple operators is evaluated in the same way, from left to right with each operator applied to the two previous operands.

If you are having trouble with figuring out the algorithm for how to evaluate postfix expressions, consult the Postfix notation definition in the textbook. You can also read more here: https://en.wikipedia.org/wiki/Reverse_Polish_notation

The PostFixConverter class

The purpose of the PostFixConverter class is to take an infix expression that is generated by the user, and convert it to a postfix expression that can be evaluated using code. The Converter class uses a stack to accomplish this. Here is how it works:

- When the Converter class is instantiated, a String is passed representing the infix expression entered by the user. This should be saved as an instance variable.
- The method that does all the work in the Converter class should be called toPostFix(). It will convert the infix expression to a postfix expression. The postfix expression should be a string that can be evaluated by the calculator. Each operator and operand should be separated by spaces. This is not necessarily true of the input (i.e., the original infix expression might look like “2+2” instead of “2 + 2”).
- The first thing the converter class does is to tokenize the input string into a list of tokens representing operators and operands. Parenthesis are also allowed in infix expressions, so "(" and ")" are valid tokens. A parser method has been provided to you for your convenience – see the ParserHelper.java file.
- Once the input is tokenized, a stack is used to convert to a postfix expression. We will go over the algorithm in recitation. Here is the basic idea: create a stack and an output string. Read all of the tokens from left to right.

  The algorithm:

  1. Whenever you come upon an operand, append it onto the end of the output string.

  2. Whenever you come upon an operator, look at the top of the stack to make sure the operator on the stack has a lower precedence.

     a. If the token has higher precedence, then push it onto the stack.

     b. If the token has lower precedence, then pop operators out of the stack and append them to the output string. You may stop popping stack elements once the top of the stack has lower precedence than the current token. You will of course have to stop popping elements if the stack becomes empty.

  3. Whenever you come upon an open parenthesis, always put it on the stack.

  4. Whenever you come upon a closed parenthesis, pop out all the operators on the stack and append them to output until you find the matching parenthesis. Pop out the matching parenthesis and don't add either paren to the output (remember: postfix doesn't have parenthesis!)

- The details and helper methods used in this class are left up to you.

The input will always be some mathematical expression that contains a combination of the operations +,-,*,/, and ^. All of the numbers will be integers. You may assume that the expressions will never miss an operand (e.g., you won't encounter something like "3 + "). Here are some example inputs to try out:

- 3+4*5/6
- (300+23)*(43-21)/(84+7)
- (4+8)*(6-5)/((3-2)*(2+2))

Example output
Given (4+8)*(6-5)/((3-2)*(2+2)), your output should show the converted postfix string and the result of the calculation.

```
type your infix expression:
(4+8)*(6-5)/((3-2)*(2+2))
converted to postfix: 4 8 + 6 5 - * 3 2 - 2 2 + * /
answer is 3.0
```

Formatting and Style

The output for this assignment is pretty simple, so just make sure that it is legible and clear. Make sure not to leave out any operands and/or operators.

Coding style will be important, especially since this assignment brings together a lot of the concepts of OOP that we have recently covered. Put to use the best practices we've talked about so far in the course.

Grading Rubric

Points	Description
1	Proper implementation of Stack class
4	Converter class performs infix to postfix expression successfully, uses Stack class
4	Calculator class evaluates postfix expressions correctly, uses Stack class
1	Good style demonstrated in the code; sensible formatting