PROJECT: **3**
DUE DATE: **Thursday March 12, 2015**

Description:

We are used to write arithmetic expressions with the operator between the two operands: a + b or c / d. If we write a + b * c, however, we have to apply precedence orders to avoid ambiguous evaluation. This type of expression is call **infix** expression. There are other two types of different but equivalent ways of writing expressions.

Infix: X + Y: Operators are written in-between their operands. Infix expression needs extra information to make the order of evaluation of the operators clear: precedence and associativity, brackets ( ). For example, A * (B + C) / D.

Postfix: X Y +: Operators are written after their operands. The above infix expression can be written using postfix notation as A B C + * D /

Prefix: + X Y: Operators are written before their operands. The above infix expression can be written using prefix notation as / * A + B C D

More examples of the above three expressions are given below:

Infix	Postfix	Prefix
a + b + c	a b + c +	+ + a b c
a + b * c	a b c * +	+ a * b c
(a + b) * (c - d)	a b + cd - *	* + a b – c d

You are to implement two class methods that can convert an infix expression to its equivalent postfix and prefix expressions.

A term can be either a number or variable name. The parameter to your methods will be an array of Strings and so is the return value. Each element of the array can either be a variable name, a number, or a binary operator + - * /.

Class name: `InfixExpression`

Conversion of Infix to Postfix:
Method name: `String[] convertToPostfix (String[] infixExpression)`

In infix expressions, we know that operators have different precedence orders to avoid ambiguous meaning. The operators "+" and "-" have the same precedence. The operators "*", "/" also have the same precedence, but have higher precedence than "+" and "-". Operators have a higher precedence than the left parenthesis.

An infix expression can be converted to a postfix expression using a stack. Starting from the left most symbol in the infix expression, we follow the following steps and advance to the next symbol in the infix expression until we reach the end of the expression.

- Variables (such as a, b, and c) are directly copied to the output.
- Left parentheses are always pushed onto a stack.
- When a right parenthesis is encountered, the symbol on the top of the stack is popped off the stack and copied to the output. This process repeats until the top of the stack is a left parenthesis. When that occurs, both parentheses are discarded.
- If the symbol been scanned has a higher precedence than the symbol on the top of the stack, the symbol being scanned is pushed onto the stack.
- If the symbol been scanned has a lower or the same precedence than the symbol at the top of the stack, the symbol at the top of the stack is popped off to the output. The symbol been scanned will be compared with the new top element on the stack, and the process continues.
- When the input is empty, the stack is popped to the output until the stack is empty. Then the algorithm terminates.
- If the input is empty but the stack still has a left parenthesis, or a right parenthesis is scanned when the stack is empty, the parenthesis of the original expression were unbalanced.

Conversion of Infix to Prefix
Method name: `String[] convertToPrefix (String[] infixExpression)`

An infix expression can also be converted to a prefix expression using two stacks: one for operators and the other for operands. Starting from the left most symbol in the expression, we follow the following steps and advance to the next symbol in the infix expression until we reach the end of the expression.

- Variables (operands) are pushed onto the operand stack.
- Left parentheses are always pushed onto the operators stack.
- When a right parenthesis is encountered, the operator in the operator stack is popped off and saved to *op*, the expression on the top of the operand stack is popped off and saved to *RightOperand*, another expression on the top of the operand stack is popped off and saved to *LeftOperand*. A new expression is formed by "*op LeftOperand RightOperand*" and is pushed back to the operand stack. This process repeats until the top of the stack is a left parenthesis. When that occurs, both parentheses are discarded.
- If the symbol been scanned has a higher precedence than the symbol on the top of the operator stack, the symbol being scanned is pushed onto the operator stack.
- If the symbol been scanned has a lower or the same precedence than the symbol on the top of the operator stack, the symbol on the top of the stack is popped off and saved to *op*. Popped two expressions from the operands stack and saved them to *RightOperand* and *LeftOperator* respectively. They form a new expression "op LeftOperand RightOperand" and is pushed to the operand stack. The symbol been scanned will be compared with the new top element on the operator stack, and the process continue.
- If the operator stack is not empty, continue to pop operator and operand stacks building prefix expression until the operator stack is empty.
- Make sure you also check for unmatched parentheses.

What to test?

Think of your own cases: correct input, without parentheses or with matched parentheses; and incorrect input (with unmatched parentheses). Your program should catch the error with unmatched parentheses and when one of the operands is missing. Make sure that you deal with all error conditions, for example, pop/top on an empty stack.

Multiple spaces should be allowed between operands and operators if your main is query the user for the input.

Alternatives:

You can choose to implement your conversion algorithms based on the standard Stack interface provided by Java (java.util.stack) or your own stack implementation.

Project report:

- Page 1: Cover page with your name, class, project, and due date
- Page 2 to 3:
    - Section 1 (**Project specification**): Your ADT description. Description of data structures used and a description of how you implement the ADT
    - Section 2 (**Testing methodology**): Description of how you test your ADT, refer to your testing output. Explain why your test cases are rigorous and complete. Demonstrate that you test each method.
    - Section 3 (**Lessons learned**): Any other information you wish to include.

Turn in:

1. Print out of project report and output of your test cases.
2. Email the source code with the file name *flast*-**InfixExpression.java** (ex. tnguyen-InfixExpression.java) to tvnguyen7@cpp.edu with the subject: **cs240w15 p3**. Rename the file before submitting.

Grading Guide:

- 10%: Project report and project output.
- 80%: Program correctness
- 10%: Coding – efficiency, style, comments, formats

Notes:

1. The following information is required in the beginning of every source file.

```
//
//    Name:        Last, First
//    Project:     #
//    Due:         date
//    Course:      cs-240-02-w15
//
//    Description:
//                 A brief description of the project.
//
```

2. The submission **must** be legibly printed.