CS 445: Data Structures
Fall 2018

Assignment 2

Assigned: Friday, September 28 **Due:** Thursday, October 11 11:59 PM

---

1 Motivation

In lecture, we discussed an stack algorithm for converting an infix expression to a postfix expression ("shunting-yard"), and another for evaluating postfix expressions. We briefly described how one might pipeline these algorithms to evaluate an infix expression in a single pass using two stacks, without generating the full postfix expression in between. In this assignment, you will implement this combined algorithm. Your completed program will evaluate an infix arithmetic expression using two stacks.

You are provided with the skeleton of `InfixExpressionEvaluator`. This class accepts input from an `InputStream` and parses it into tokens. When it detects an invalid token, it throws an `InvalidExpressionException` to report the error. To facilitate ease of use, this class also contains a `main` method. This method instantiates an object of type `InfixExpressionEvaluator` to read from `System.in`, then evaluates whatever expression is typed. It also catches any `InvalidExpressionException` that is thrown, and prints the reason for the error.

`InfixExpressionEvaluator` uses composition to store the operator and operand stacks, and calls several private helper methods to manipulate these stacks when handling various tokens. You will need to complete these helper methods and add error checking to ensure the expression is valid.

2 Tasks

First, carefully read the provided code. You can find this code on Pitt Box in a folder named `cs445-a2-abc123`, where `abc123` is your Pitt username. You are allowed to modify the provided code in any way, as long as you do not change how it is used (e.g., do not accept multiple expressions in one execution, or read expressions from a file rather than `System.in`).

2.1 Implement helper methods, 70 points

As tokens are parsed, helper methods are called to handle them. In the included code, these helper methods do not do anything. You must implement the following methods to handle the various types of tokens.

- `handleOperand(double)`: Each time the evaluator encounters an operand, it passes it (as a double) to this method, which should handle it by manipulating the operand and/or operator stack according to the infix-to-postfix and postfix-evaluation algorithms.

- `handleOperator(char)`: Each time the evaluator encounters an operator, it passes it (as a char) to this method, which should handle it by manipulating the operand and/or operator stack according to the infix-to-postfix and postfix-evaluation algorithms.

  Each of the following operators must be supported. Follow standard operator precedence. You can assume that '-' is always the binary subtraction operator (e.g., no negative operands).

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`^`	Exponentiation

- `handleOpenBracket(char)`: Each time the evaluator encounters an open bracket, it passes it (as a char) to this method, which should handle it by manipulating the operand and/or operator stack according to the infix-to-postfix and postfix-evaluation algorithms.

  You *must* support both round brackets `()` and curly brackets `{}`. These brackets can be used interchangeably, but must be nested properly—a `(` cannot be closed with a `}`, and vice-versa.

- `handleCloseBracket(char)`: Each time the evaluator encounters a close bracket, it passes it (as a char) to this method, which should handle it by manipulating the operand and/or operator stack according to the infix-to-postfix and postfix-evaluation algorithms.

- `handleRemainingOperators()`: When the evaluator encounters the end of the expression, it calls this method to handle the remaining operators on the operator stack.

2.2 Error checking, 30 points

This task requires that you modify your program to account for errors in the input expression. The provided code throws `InvalidExpressionException` when encountering an unknown token (for instance, `&`). You must modify your program so that, whenever it encounters any other type of error in an expression, it throws `InvalidExpressionException` with a **specific error message**. The message should explain what is wrong with the expression (i.e., what the user did wrong), and *not* what internal implementation detail allowed you to detect it. For instance, your error message should never mention the stack, as the user should not be required to know how the program works.

This requires careful consideration of all the possible syntax errors. What if an operand follows another operand? An operator following an open bracket? What about brackets that do not nest properly? All such syntax errors must be handled using `InvalidExpressionException`.

Hints:

1. Which pairs of adjacent token types are valid vs. invalid? Consider creating a data member that tracks the most recent token type. When handling a new token, first ensure that it is legal for the current token type to follow the previous token type.

2. Trace through the algorithm by hand using expressions with various bracket errors. When during the process can each error be detected?

3 Grading

The following points breakdown will be used to assign grades:

- **70 points** are allocated for correctly evaluating **valid** expressions. These expressions will range from very simple to very complex, but none will contain syntax errors. You should test your program extensively to ensure that it works properly even for complex expressions with multiple sets of nested brackets and combinations of all operators.
- **30 points** are allocated for correctly identifying the syntax errors in **invalid** expressions. These expressions will contain a wide variety of syntax errors. You should test your program with many invalid expressions to ensure that it correctly detects all possible syntax errors.

Note: Code that cannot be compiled and executed will be given a grade of 0.

3.1 Extra credit

In addition to the above tasks, **for up to 8 bonus points**, you may add additional features to your program. If you do so, include a file named `README.txt` describing these extra features (please place this file in your `cs445-a2-abc123` folder, and not within the `cs445/a2/` package folder).

For instance, you may consider errors beside those of syntax. Can you detect and report divide-by-zero? What about other errors in value? Can you support both the binary subtraction operator and the unary negation operator (using the same symbol or a different symbol)? Can you add support for additional operators? Hexadecimal operands of the form `0x6a` or octal operands like `0o73`?

This task encourages you to brainstorm outside of the assignment brief. Try to add unique and interesting features to your evaluator.

4 Submission

Upload your java files in the provided Box directory as edits or additions to the provided code.

All programs will be tested on the command line, so if you use an IDE to develop your program, you must export the java files from the IDE and ensure that they compile and run on the command line. Do not submit the IDE's project files. Your TA should be able to download your `cs445-a2-abc123` directory from Box, and compile and run your code. Specifically, `javac cs445/a2/InfixExpressionEvaluator.java` and `java cs445.a2.InfixExpressionEvaluator` must compile and run your program, when executed from the root of your `cs445-a2-abc123` directory.

In addition to your code, you may wish to include a `README.txt` file that describes features of your program that are not working as expected, to assist the TA in grading the portions that do work as expected. This file should also specify any **extra credit** tasks that you completed. If you include one, place it in the root of your `cs445-a2-abc123` folder.

Your project is due at 11:59 PM on Thursday, October 11. You should upload your progress frequently, even far in advance of this deadline: **No late submissions will be accepted.**