Joanna Klukowska
joannakl@cs.nyu.edu

Programming Project 3: Calculator

Due date: November 16, 11:55PM EST.

You may discuss any of the assignments with your classmates and tutors (or anyone else) but all work for all assignments must be entirely your own. Any sharing or copying of assignments will be considered cheating. If you get significant help from anyone, you should acknowledge it in your submission.

In this project you will work on the program that is a backend for a very simple calcular (for extra credit you can implement the frontend as well). Your calculator takes as input simple mathematical expressions like

```
23 - 2 * 5
4 - 9 * 11 + 155 * ( 21 - 17 ) / 3
( ( 15 / ( 7 - ( 1 + 1 ) ) ) - ( 2 + ( 1 + 1 ) )
```

and determines their value.

The Program Input and Output

Input File Your program is given the name of the input text file on its command line (the first argument). The text file contains mathematical expressions (like the ones above). There is always only one expression per line. Each valid expression consists of tokens separated by one or more spaces. A token is either an integer, an operator (`+`,`-`,`*`,`/`), or an open or a closed parenthesis.

If the filename is omitted from the command line, it is an error. If the filename is given but it does not exist or cannot be opened for reading by the program, for any reason, it is an error. The program should display an error message if any of these conditions occur, and it should be as specific as possible (it is not enough for it to say ”could not open file”). Your program is NOT ALLOWED to hardcode the input filename in its own code.

Output File Your program is given the name of the output text file on its command line (the second argument). As your program evaluates each of the expressions in the input file, the results of computations should be printed to the output time. If a corresponding expression in the input file is not valid, your program should print INVALID instead of the result and continue with the next expression. The results should be printed one per line.

If the filename is omitted from the command line, it is an error. If the filename is given but it does not exist or cannot be opened for reading by the program, for any reason, it is an error. The program should display an error message if any of these conditions occur, and it should be as specific as possible (it is not enough for it to say ”could not open file”). Your program is NOT ALLOWED to hardcode the input filename in its own code.

User Input This program is not interactive.

Program Design

Your program must contain multiple classes and must be developed in an object oriented way. You will need to develop the following classes:

- ConsoleCalculator - the class that provides the main() method. This class is responsible for all input and output operations (including validation of the input).
- ExpressionTools - the class that provides the methods for infix to postfix conversion and for postfix evaluation.

Joanna Klukowska
joannakl@cs.nyu.edu

- Stack - the class that provides a linked list based implementation of the interface provided below.
- PostFixException - the class that represent the exception that should be thrown when errors in the expression occur.

The program should read each of the expressions from the input file, convert it to the postfix expression, evaluate the postfix expression and print the result to the output file .

Converting From Infix to Postfix

The algorithm for converting the infix expressions to postfix expressions is as follows:

Algorithm for converting infix to postfix expressions

```
for each token in the input infix string expression
    if the token is an operand
        append to postfix string expression
    else if the token is a left brace
        push it onto the operator stack
    else if the token is an operator
        if the stack is not empty
            while top element on the stack has higher precedence
                pop the stack and append to postfix string expression
        push it (the current operator) onto the operator stack
    else if the token is a right brace
        while the operator stack is not empty
            if the top of the operator stack is not a matching left brace

                pop the operator stack and append to postfix string expression
            else
                pop the left brace and discard
                break
while the operator stack is not empty
    pop the operator stack and append to postfix string expression
```

If any errors are encountered, your method should throw an exception of type `PostFixException` with an appropriate message. You should use the following definition of the exception class:

```java

public class PostFixException extends Exception {

    public PostFixException() {
        super();
    }

    public PostFixException(String message) {
        super(message);
    }
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

Evaluating the Expressions

The algorithm for evaluating the postfix expressions is as follows:

```
Algorithm for evaluating postfix expressions

Scan the given postfix expression from left to right
for each token in the input postfix expression

    if the token is an operand
        push it (its value) onto a stack
    else if the token is an operator
        operand2 = pop stack
        operand1 = pop stack
        compute operand1 operator operand2
        push result onto stack

return top of the stack as result
```

If any errors are encountered, your method should throw an exception of type `PostFixException` with an appropriate message.

Stack

Both of the above algorithms need to use a stack. You should implement a linked list based stack that implements the following interface:

```java
public interface Stack <E> {

    /**
     * Add an object to the top of the stack
     * @param item character to be added to the stack
     */
    public void push ( E item ) ;

    /**
     * Remove and return an object from the top of the stack
     * @return an object from the top of the stack is returned and removed
     *   from the stack. If stack is empty, null is returned.
     */
    public E pop () ;

    /**
     * Return an object from the top of the stack.
     * @return an object from the top of the stack is returned.
     *   If stack is empty, null is returned.
     */
    public E peek () ;


    /**
     * Produces string representation of the stack.
     * @return returns a String object that contains all elements
     *   stored on the stack. The elements are separated
     *   by spaces. The top of the stack is the rightmost character
     *   in the returned string.
     */

    public String toString () ;
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

Extra Credit

For additional 20 points, you can implement one of the following options. You will get credit only for one of them.

WARNING: Do not attempt the extra credit unless the base assignment is thoroughly tested and working.

Option 1: Calculator FrontEnd

Implement the front end of the calculator (i.e., the graphical user interface) using Processing. The calculator should contain clickable buttons for all ten digits, clickable buttons for the four operators and a clickable button for the equal sign. The user enters the expression by clicking the buttons (your program should make sure that the numbers and operators are separated by spaces. As the user clicks buttons, the expression should appear in the application window. When the user clicks the button with the equal sign, the expression should be evaluated and the result displayed in place of the expression.

The Processing should be used only for providing the user interface. All the computation should be done by the classes discussed above.

You may find many useful resources at `https://www.processing.org/reference/`.

Option 2: Prefix Calculator

You need to research and implement the algorithm for infix to prefix conversion. You also need to implement the algorithm for evaluating prefix expressions (see the algorithm in the lecture notes). These methods should be added to the `ExpressionTools` class.

Your program should allow optional third command line argument: a name of another output file. For each expression in the input file, your program should convert it to prefix expression, print that prefix expression to the output file, evaluate the prefix expression and then print the result to the output file. This means that for each input expression, you should have two lines in the output file.

If the program is run with only two command line arguments it should run correctly and according to non-extra credit description.

You should define your own `PreFixException` class to be used for this part.

Grading

If your program does not compile or crashes (almost) every time it is ran, you will get a zero on the assignment.

If your program does not use command line arguments, you will lose 20 points on this assignment.

15 points	validating the command line arguments, reading the input file, writing to the output file
35 points	design and implementation of the Stack class
30 points	design and implementation of the ExpressionTools class
20 points	proper documentation
20 points	extra credit part

How and What to Submit

Your should submit all your source code files (the ones with .java extensions only) in a single **zip** file to NYU Classes. You should have at least three different files.