CSC 115: Fundamentals of Programming II
Assignment #4: Stacks

Due date

Wednesday, March 18, 2015 at 9:00 pm via submission to conneX.

How to hand in your work

Submit the requested files through the Assignment #4 link on the CSC 115 conneX site. Please ensure you follow all the required steps for submission (including confirming your submission).

Learning outcomes

When you have completed this assignment, you should have become acquainted with:

- How to create a *reference-based implementation of a stack* for strings.
- Using stacks and lists to *evaluate an infix arithmetic expression*.
- Using exceptions as part of your work writing code handling typical errors that occur when writing arithmetic expressions.
- Using such exceptions with *try blocks that have more than one catch block* in order to properly handle error conditions.

Expression evaluation

Problem description

In lectures we have explored a few forms taken by arithmetic expressions. For example, an *infix expression* has operators (in-)between operands:

```
(314 - 5) * (10 + 20 / 3)
```

Operators in a *postfix expression* follow the operands (i.e., "post" means "after"):

```
314 5 - 10 20 3 / + *
```

You have seen in lectures how a postfix expression may be evaluated. This typically involves using a stack to store both operands and computed values. (Refer to pages 373 and 374 for a description of such an evaluation technique.)

Once you have completed this assignment, you will have code that may be executed as follows:

```
$ java EvalInfix "(314 - 5) * (10 + 20 / 3)"
4944
```

Note the use of the quotation marks around the infix expressions. They are needed to prevent the asterisk ("*") symbol from being "globbed" (i.e., command shell otherwise converts * into a list of all files in the current directory).

The only operators for this assignment are +, -, * and /. The only form of parentheses allowed are "(" and ")" (i.e., curly braces such as "{" and "}" or square braces such as "[" and "]" are not allowed.) Assume all our operators left-associate (i.e., operators of the same precedence are evaluated from left to right). Only integers may be operands (i.e., no floating-point numbers).

Your program must handle the following: (a) mismatched braces; (b) use of a non-integer operand; (c) too many operands; and (d) too many operators.

```
$ java EvalInfix "((10 * 2) + 5"
<unbalanced ()>

$ java EvalInfix "3.14159 * 4 * a"
<noninteger>

$ java EvalInfix "20 + 10 30"
<syntax error>

$ java EvalInfix "20+10-30+"
<syntax error>
```

A pseudocode solution to *evaluateExpression()* appears as follows:

```
If expression has unbalanced parentheses then
    return the string "<unbalanced ()>" (A)

Split string into an array of strings, one string per token (B)

Pass the string array to a method returning a StringList that is
the postfix representation of the infix expression (C)

Use the StringList as an argument to a method that evaluates the
postfix expression and returns a string as a result (D)

Return the String produced in the last step
```

Elements of this pseudocode solution have been described in lectures:

- *Finding balanced parentheses (A):* Code to check for balanced braces was shown during week 8 and is available on conneX.
- *Splitting up string-expressions into separate strings (B)*: This is a strangely tricky operation. I have provided the code for you to do this already in *EvalInfix.java* within the method named *tokenize()*, i.e., it accepts a String as an argument and returns an a array of Strings as a result.
- *Converting infix to postfix (C)*: Pages 375 to 379 in the text describe one approach to this, and an explanation was given in class during week 9. Some code using a List and a Stack to perform his conversion is available from the week's code folder on conneX.

I have already referred to one approach to evaluating the postfix expression (i.e., the step labeled D in the pseudocode above). Note that to accomplish this you will need:

- To convert a string into an int via *Integer.parseInt(),* and to properly catch the *NumberFormatException* if the argument to this method cannot be converted into an int.
- To catch an *ArithmeticException* when a division-by-zero occurs.
- To catch some kind of StackException when there are too many operators for the operands provided.

Feel free to use online resources to learn how to use Integer.parseInt() method or the exceptions listed above. ***However, you are not to use an expression evaluator such as the built-in JavaScript engine or Java Server Faces functionality, etc.***

What you are to write

1. You are provided with these files:
    a. *EvalInfix.java*
    b. *StringStack.java*
    c. *StringStackException.java*
    d. *StringNode.java*
    e. *StringList.java*
    f. *StringListRefBased.java*
    g. *testcases.txt* and *results.txt*

2. Please keep all of your code for this assignment within a single directory. We will take advantage of what is called the *default package* in Java, i.e., all Java files in the same directory are considered to be in the same package. This will permit you to use package-private access for the assignment without a complex directory structure.

3. Write your solution to *StringStackRefBased.java*. You do not need a separate tester class, but you must write testing code for this class with a *main()*

method for the class. Do not use *StringListRefBased* to implement *StringStackRefBased*.

4. Write your implementation of *boolean isBalanced(String expr)* within *EvalInfix.java*. This method will be a static.

5. Write your implementation of *StringList toPostfix(String[] tokens)* within *EvalInfix.java*. This method will be static. Amongst other things you will need to write code (perhaps additional methods?) to determine the precedence of an operator, or even whether or not a string actually is an operator.

6. Write your implementation of *String evaluatePostfix(StringList postfix)* within *EvalInfix.java*. This method will be static.

7. Write your complete implementation of *String evaluateExpression(String expr)*. This will call the functionality you have developed steps 3, 4 and 5.

8. When you are finished your own tests of your work, please use the tests described in *testcases.txt*. There is one test per line. Remember to surround the expression with double-quotes when testing. Each line in *testcases.txt* has a corresponding line in *results.txt* indicating what is expected (i.e., the fifth line in *results.txt* is the result expected for the expression on the fifth line of *testcases.txt*).

9. In all of your coding work, please follow the coding conventions posted at conneX (i.e., a document in the "Lectures & stuff" section).

Files to submit (three in total):

1. *EvalInfix.java*
2. *StringStackRefBased.java*
3. *A file named test_output.txt (which also includes a description of what each test is checking followed by the test input and output)*

Grading scheme

- "A" grade: An exceptional submission demonstrating creativity and initiative going above and beyond the assignment requirements. The program runs without any problems and implements the required classes and methods in those classes. Public methods are tested with output showing tests and their results. Any extra work appears in class files having the name *<file>Extra.java*, and identified within the file (i.e., Class comment) are details of your extension to the assignment demonstrating creativity and initiative.

- "B" grade: A submission completing the requirements of the assignment. The program runs without any problems and implements the required classes and methods in those classes. Public methods are tested with output showing tests and their results.

- "C" grade: A submission completing most of the requirements of the assignment. The program runs with some problems but does implement the required classes and methods, and yet might not have the expected output. Methods are tested with output showing tests and their results.

- "D" grade: A serious attempt at completing requirements for the assignment. The program runs with major problems, or does not contain implementations of all required classes and methods.

- "F" grade: Either no submission given, the submission does not compile, or submission represents very little work.

Appendix

Suppose we have the following invocation of the program:

```
$ java EvalInfix "(50+3) * -12”
```

What appears below is a visualization of what each step of the pseudocode would produce.

The expression in the command-line argument will be passed to *evaluateExpression()* such that the following is stored in the string parameter named *expr*:

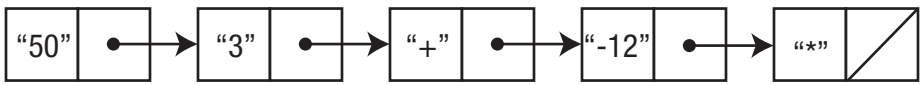
```
“(50+3) * -12”
```

When checking for balanced parentheses, our algorithm would discover the string referenced by *expr* is indeed balanced. Then the string can be tokenized, and the array returned from the tokenizing method will be:

```
{“(“, “50”, “+”, “3”, “)”, “*”, “-12”}
```

That is, for this example the string array would contain seven (7) strings, with each string corresponding to some element in the original expression and appearing in an order matching the original string passed into the program.

This string array will be passed as the single parameter of *toPostfix()* which will return a list corresponding to that below:



This list of strings is now suitable for evaluation as a postfix expression. Given that there are no errors in this expression (or, rather, its evaluation will not lead to any errors), the value returned will be a string:

```
“-636”
```

This will then be the final value returned by *evaluateExpression()* and output by the call to *println()* in the *main()* method of *EvalInfix*. (Note that your implementation of postfix evaluation within evaluatePostfix() will need to convert to and from strings and ints.)