CSC 115: Fundamentals of Programming II
Assignment #2: Building a Calculator

Due date

Monday, February 19, 2018 at 9:00 am via submission to conneX.

How to hand in your work

Submit the requested files through the Assignment #2 link on the CSC 115 conneX site. Please make sure you follow all the required steps for submission (including confirming your submission).

Learning outcomes

When you have completed this assignment, you are expected to be able to:

- Create a simple reference-based Stack.
- Use a stack to:
  - create a postfix expression from an infix expression and
  - calculate the value from a postfix expression.
- Both throw and catch exceptions in Java.
- Gain some experience with the Object-Oriented design concepts: data abstraction, information hiding, and encapsulation.

Building a Basic Calculator

It seems intuitive that a computer can solve arithmetic expressions with speed and accuracy. Something like 3+2 is pretty easy. But what about 3*(4+2)-15/3? The computer would need to parse out the parts that need to be calculated before other parts. It would need to know about the parentheses and the fact that division '/' takes precendence over substraction '-'.

Problem description

It would be much easier to develop an algorithm that allows the computer to *parse* (move strictly left-to-right) each of the tokens (operands and operators) and when finished, deliver the correct answer.

If the expression was ordered in *postfix* order, then the computer can parse the answer using an algorithm that evaluates the postfix expression. Of course, we need to rearrange the order of the *tokens,* (the operands and operators) and remove the parentheses.

We can implement this re-sorting of the original *infix* expression by using a Stack abstract data type. The algorithm parses the infix tokens, and temporarily places some tokens onto the stack, to wait their turn in the new ordering.

This may sound rather complicated, but all of these definitions and algorithm descriptions are well-defined in the textbook.

Programming requirements

You will write the source code for the following classes:

- *StringStack*, using a reference-based list as its main data structure.
- *PostfixTokenizer*, using whatever type of list you wish to construct an array or linked list as its main data structure.
- *Evaluator*, which has a single static method that computes the solution of a postfix expression.

The following fully-developed Java files are provided to support the code you write:

- *Node.java* which provides the container for items in a singly-linked list.
- *StackException.java* and *IllegalExpressionException.java* which are the exceptions that indicate problems with either the stack or the expression.
- *Operator.java* which contains some helpful constants and some methods that deal with operators.
- *Tokenizer.java* which is an interface for two implementing classes:
  - *OperatorTokenizer.java* has been fully implemented.
  - The other is the one you write.

Two other optional files are provided for your fun and pleasure once you have completed the assignment:

- *Memory.java* which manages input and outputs of a graphical representation of a calculator.
- *Calculator.java* which, when run, produces a visual representation of a calculator that allows the input of the GUI (graphical user interface) to commmunicate with the completed *Evaluator* class and send the answer to the calculator screen.

Implementation details

All the specifications can be accessed through the following link. (Make sure you are logged onto conneX when accessing the link directly.)

Additional requirements are supplied in the detailed descriptions, following the tips to ***Get Started:***

1. Download the files associated with this assignment into a fresh working directory (csc115/assn2 for example).
2. Create the minimal class for StringStack, PostfixTokenizer and Evaluator by copying the required public methods and any private data fields into these files.

3. Compile each of the files.

- Note that PostfixTokenizer will require that you add methods from the Iterator interface as well. Click on the Iterator link to find these or look at the source code for OperatorTokenizer.

4. Approach the programming exercise in the order described in the details below:

StringStack:

The instructions for a reference-based Stack are in the textbook. The Node.java is provided for you.

PostfixTokenizer:

This class implements Tokenizer. Note that since Tokenizer extends the Iterator interface, it must also contain the required methods that are found in the Iterator. If you follow the pattern of the provided OperatorTokenizer, it will make sense.

For this class, you will need to create some sort of list, to contain the postfix tokens. You are welcome to use whatever you wish: either an array or a linked list. You **may not use any of the Java.util.List** objects. Note that the PostfixTokenizer does not have any public access to the list that is created. The only way a user can access the tokens is by invoking the methods that are part of the Iterator. This is an example of *Information Hiding.*

Note also that in the constructor for this class, you will need to use the Iterator methods to parse the tokens that were created by the OperatorTokenizer. The algorithm to create a list of postfix tokens by parsing the infix tokens is detailed in the textbook. The completed and thoroughly-tested StringStack will be very useful here.

Evaluate:

The algorithm for parsing postfix tokens to evaluate an expression is also in the textbook. Here you get another opportunity to put the StringStack to work. Review the Operator class: there are some very helpful methods in this class.

Dealing with bad expressions

Note that the input to the OperatorTokenizer constructor is a String, which could be anything. We could do a thorough check to make sure it is a valid arithmetic expression before we start all the processing steps to get to the final result. However, that process will like take as many lines of code as it takes just to start processing and then deal with the errors when they happen to show up.

So for instance, the OperatorTokenizer makes no claims to convert a string into a proper infix expression; it only promises to extract the operators and parens. If you look at the main method (where some internal testing was done), there is no indication that an input string is valid or not.

When you go through the algorithm in the textbook to convert the tokens to a PostfixTokenizer, you will be able to determine a condition that will occur if there are unmatched parentheses in the original expression. If this happens, just throw the IllegalExpressionException with the approprate message.

During the Evaluator.evaluate algorithm, you will have several cases that can will indicate an error:

- If you try to convert a String operand to a double, and a NumberFormatException is thrown because of that call, you will know that a token that is supposed to be an operand is not a number.
- If there are still values on the stack after evaluating a postfix expression, then you know that there are too many operands.
- If you encounter an empty stack while popping operands to match an operator, then you know there are too few operands.

Internal testing

For each of the classes you create, make sure you do the necessary internal testing of the methods in the main method. Note that Operator.java and OperatorTokenizer contain internal testing (implemented by the author to test the code).

PostfixTokenizer example: Suppose the main method of PostfixTokenizer had the following sub-set of statements:

```
String asString = "3*(4+-2)";
OperatorTokenizer asInfix = new OperatorTokenizer(asString);
PostfixTokenizer postfix = new PostfixTokenizer(asInfix);
System.out.print("as infix tokens: ");
System.out.println(asInfix);
System.out.println("By individual postfix tokens:");
while(postfix.hasNext()) {
        System.out.println("next token: "+postfix.next());
}
```

After compiling and running PostfixTokenizer, one would expect the following output:

```
as infix tokens: 3 * ( 4 + -2 )
By individual postfix token:
next token: 3
next token: 4
next token: -2
next token +
next token *
```

Evaluator example: Suppose the main method of Evaluator had the following subset of statements:

```
String good = "3*(4+2)-4*(-6--3)";
String badParens = "(4))";
String toomanyOperands = "(4+6)12";
String notEnough = "3--6+2*";
System.out.println(good);
System.out.println(evaluate(good));
System.out.println(badparens);
try
        evaluate(badparens);
} catch (IllegalExpressionException iee) {
        System.out.println(iee.getMessage());
}
System.out.println(toomanyOperands);
try {
        evaluate(toomanyOperands);
} catch (IllegalExpressionException iee) {
        System.out.println(iee.getMessage());
}
System.out.println(notEnough);
try {
        evaluate(notEnough);
} catch (IllegalExpressionException iee) {
        System.out.println(iee.getMessage());
}
```

After compiling and running Evaluator, one would expect the following output:

```
3*(4+2)-4*(-6--3)
30.0
(4))
Mismatched parens
(4+6)12
Too many operands
3--6+2*
Too few operands
```

Putting it all together (the finished product)

Once all the parts are working, you can run some thorough testing by using the Calculator class provided. Simply compile and run Calculator and input all kinds of values as you would for any simple calculator. Note that the input window displays "INPUT ERROR" for any expression that will throw an IllegalExpressionException.

Files to submit:

1. *StringStack.java*
2. *PostfixTokenizer.java*
3. *Evaluator.java*

Grading

- Submissions are expected to follow all of the requirements for the submitted files in the specification documents.
- Evidence of internal testing must be present: you may comment out any testing the main methods if you wish.
- The coding conventions, specifed in the codingConventions.pdf document on conneX, must be followed.