CSC220 Lab 7
Stacks

The goal of this week's assignment is:

1. Practice using stacks
2. Learn about the importance of debugging

Things you must do:

1. There are many details in this assignment. Make sure you read the whole thing carefully before writing any code and closely follow this instruction.
2. You must complete your assignment **individually**.
3. Always remember Java is case sensitive.
4. Your file names, class names, package name, and method signatures must match exactly as they are specified here.

Things you must not do:

1. You must not change the file names, class names, package names.
2. You must not change the signature of any of these methods (name, parameters, …).
3. You must not create any different class (other than the ones instructed).

Important Note: **You are going to write only two functions this week but there are many details involved. Start early! We are providing test cases here for you but you are encouraged to write your own tests as well.** ***You can make the assumption that the given expression is valid and you don't need to worry about receiving an invalid expression as the input.***

Part 0

- You are going to create a new project for this (and each of the remaining labs). You learned how to create a project in Eclipse before. Create a new Java project and call it **Lab07** (with no space and no other name – notice the capital 'L')
- Create a package inside your project and call it **lab07** (no space and no other name – all lowercase).
- Create a class (or Java file) in this package and call it: **StringSplitter.java** (nothing else). This class breaks up a string into a sequence of tokens using both whitespace and a list of special characters. The special characters in this case are used to tokenize an arithmetic expression. For example, the expression: 2*3.8/(4.95-7.8) would be tokenized as 2 * 3.8 / (4.95 - 7.8) even though it has no whitespace to separate these tokens. This task is accomplished using a Queue to hold the tokenized representation of an input string. The implementation of this class is provided for you in Blackboard. Copy the context of **StringSplitter.java** on Blackboard into your version of the file.
- Create another class (or Java file) in this package and call it: **Postfixer.java** (nothing else). You will be working on developing methods for this class

Part 1 – The problem description

DO NOT MOVE TO NEXT STEPS BEFORE READING THIS PART CAREFULLY

For this lab, you are asked to write a method (as part of Postfixer class) that accepts an infix expression, use StringSplitter class to tokenize that expression and then evaluate it.
To accomplish this task, you are asked to use a variation of a famous algorithm known as shunting-yard algorithm (invented by Edsger Dijkstra). This algorithm uses two stacks to save the intermediate results. One stack stores the operands or numbers and the other stack stores the operators.

The basic idea is that we use the stacks to store values (operands or operators) inside them till we are ready to ***evaluate*** the expression. How we are going to do this? We will see that in the next section.

Part 2 – InfixEvaluator method

First let us define the signature of the method that will accomplish this task:

```
public static double InfixEvaluator(String line)
```

This function first needs to tokenize the input expression (in the form of a String and stored in input variable "line"). You can tokenize the input expression by creating an object of StringSplitter and passing it the String as follows:

```
StringSplitter data = new StringSplitter(line);
```

After this line the input expression is tokenized and positioned inside a Queue (look at the implementation of StringSplitter and look for its fields).

Next create your two stacks. Remember one will contain the operators and the other one will include the operands. Define them as follows:

```
Stack<String> operators = new Stack<String>();
Stack<Double> operands = new Stack<Double>();
```

Now everything is ready to process the input expression using the two stacks you created. We only need to know how do it. Next section, explains shunting-yard algorithm. Follow the algorithm ***exactly!***

Part 3 – Shunting-yard algorithm

DO NOT STAR CODING BEFORE READING THIS SECTION TO THE END

Here is the pseudo-code of the algorithm you have been asked to implement:

```
1. Scan the input string (infix notation) from left to right. One pass is sufficient, take one token at a time.
2. if the token is:
   2.1. number: push it onto the operand stack.
   2.2. a left parenthesis: push it onto the operator stack.
   2.3. a right parenthesis:
        2.3.1. while the thing on top of the operator stack is not a left parenthesis
               2.3.1.1. pop an operator from the operator stack, pop two operands from the operand stack, apply the operator to the operands, in the correct order, push the result onto the operand stack
        2.3.2. pop the left parenthesis from the operator stack and discard it
   2.4. an operator (call it current operator):
        2.4.1. while the operator stack is not empty, and the top thing on the operator stack has the same or greater precedence as the current operator
               2.4.1.1. pop an operator from the operator stack, pop two operands from the operand stack, apply the operator to the operands, in the correct order, push the result onto the operand stack
        2.4.2. push the current operator onto the operator stack
3. While the operator stack is not empty
   3.1. pop an operator from the operator stack, pop two operands from the operand stack, apply the operator to the operands, in the correct order, push the result onto the operand stack
4. At this point the operator stack MUST be empty, and the operand stack MUST have a single value, which is the final result.
5. pop that value and return it.
```

Before start writing your code, grab a piece of paper and try to evaluate the following expression following this algorithm: `100 * ( 2 + 12 ) / 14`. The result should be `100`.

DO NOT START CODING BEFORE YOU ARE ABLE TO FOLLOW THE ALGORITHM WITH PEN AND PAPER.

Now, start thinking about the following points:

1. How can use access each token from the expression? Remember we have tokenized it using StringSplitter. Look at different methods in this class and see how you can get a token from StringSplitter object you defined in the previous section (i.e., `data`).
2. Next, you need to figure out how you can check whether there is any other token left.
3. You need to inspect each token to see whether it is a number, an operator or a parenthesis. How can you do that? Think for a minute before you go to the next line.
    a. There are many valid ways to do this. However, an easy way is to first check whether the token is a parenthesis, then check whether it is an operator, if none of these two were true, then it is an operand. `ParseDouble()` from `Double` class can come handy to convert an operand token to a double value.
    b. You should consider writing a helper function to figure out whether a token is an operand, an operator, or parenthesis.
    c. The operators we want you to be able to handle are: `+, -, *, /, ^`.
4. Whenever you need to apply an operator, make sure you are careful about the order of operands. This requires a little bit of thought to get it right.
5. Note that you need to be able to check the precedence of operators (only the ones mentioned above). You should already know the precedence of mathematical operators from CSC120. If in doubt, consult the first week's lecture notes. Again, there are many different ways to accomplish this. One way to do this is to write a helper function that accepts two operators and returns a Boolean based on the precedence.

`You can finally start coding, good luck!`

Part 3 – Test your code

After you are done implementing your `InfixEvaluator`, you need to test it, as always!

We have provided different test scenarios for you in ***test1.txt***. Grab a copy of this file from Blackboard and copy the content into your main function.

Run your code, if you see any red text that says "test failed", you need to debug your code.

How to debug your code?

1. Use the Eclipse debugger you learned about during the first lab
2. Think about writing helper method that help you inspect the status of your stacks, etc.
3. Go back to your pen/paper example and follow your code to see if indeed it does what it is supposed to.
4. There are other ways to debug your code as well…

Make sure to grab your code before your leave the lab, you will develop more functionality for the classes for your assignment. You are required to submit the code you wrote during the lab as part of your assignment.

For all your assignments, please start early and seek help early (either from the instructor or the TAs).