CSC220 Lab07
Stacks

The goal of this week's lab is:

1. Practice using stacks
2. Practice using Javadoc
3. 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, …).

Important Note: There are many details involved. 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

- Create a new Java project and call it **Lab07.**
- Create a package inside your project and call it **lab07**.
- Download the Lab07-Assignment07 ZIP folder from Blackboard. Copy and paste the following files into your new lab07 package:
    - **Postfixer.java**. You will be working on developing methods for this class.
    - **StringSplitter.java**. 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.

Part 1 – Problem Description

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

You are required to implement the following method:

```
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 uses a queue to accomplish the tokenization (see the class for details).

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

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>();
```

Before we continue further in this method, it will be helpful to consider helper methods we might need (which will make our job easier :-).

Part 3 – Helper methods

In this section, we consider ways we can break down a larger (harder) problem into smaller pieces. The Postfixer class contains method stubs with Javadoc comments. Review and implement. Be sure to test each individual method to confirm it works as intended.

We are now ready to turn to the shunting-yard algorithm. You must follow the algorithm given in Part 4 ***exactly!***

Part 4 – Shunting-yard algorithm

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.

Try to follow the algorithm on paper using the expression: 100 * (2 + 12) / 14.
The result should be 100.

The operators we want you to be able to handle are: `+, -, *, /, ^.`

Keep in mind the following points:

1. How can you access each token from the expression? Remember we have tokenized it using the queue 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. How can you check whether there are any tokens left in the queue?
3. You need to inspect each token to see whether it is a number, an operator or a parenthesis. How can you do that?
    a. There are many 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. An alternative is to write a helper function to figure out whether a token is an operand, an operator, or parenthesis.
4. Whenever you need to apply an operator, make sure you are careful about the order of operands (see Javadoc in the evaluate helper method). This requires a little bit of thought to get it right.

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 the main method.

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 additional helper methods 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. Many other ways…

Make sure to upload your code to Box when you have completed the lab.
Don’t forget: lab is due tomorrow (Thursday) night @ 11:59pm!