

# 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 (google drive link).

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 (3+5 as opposed to 3 5 +)**, use the **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 the **shunting-yard algorithm** (invented by Edsger Dijkstra). This algorithm uses two stacks to save the intermediate results. One stack stores the operands (values being operated on) 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 are we going to do this? We will see that in the next section.

## Part 2 – 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 3 – 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 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. Each type of token is treated differently:
  - 1.1. number: push it onto the operand stack.
  - 1.2. a left parenthesis: push it onto the operator stack.
  - 1.3. a right parenthesis:
    - 1.3.1. while the thing on top of the operator stack is not a left parenthesis
      - 1.3.1.1. 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.
    - 1.3.2. pop the left parenthesis from the operator stack and discard it
  - 1.4. an operator (call it *current operator*):
    - 1.4.1. 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* [*“(“ should have the smallest precedence as is suggested in the code comments*]
      - 1.4.1.1. 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
    - 1.4.2. push the *current operator* onto the operator stack
2. (After you have traversed the entire StringSplitter queue) While the operator stack is not empty:
  - 2.1. 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
3. At this point the operator stack **MUST** be empty, and the operand stack **MUST** have a single value, which is the final result.
4. 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**.

----> Top of stack ---->  
Operators = [ ]  
Operands = [ 100 ]

## PSEUDOCODE:

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 the `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 are 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.
3. 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 5 – 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  
(Refresher from YouTube: <https://www.youtube.com/watch?v=9gAjlQc4bPU>)
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...

## Part 1 – The problem description

For this assignment, you are asked to write a method that accepts a fully parenthesized infix expression and returns a string representing an equivalent postfix expression. As we saw during the lecture, postfix expressions are ones where each operator follows its two. For example, “1 2 -” is the postfix representation of “1 - 2”

**Do not continue unless you have finished the lab.**

## Part 2 – toPostfix method

Your method should accept a string (a fully parenthesized infix expression) and have the following signature:

```
public static String toPostfix(String line)
```

toPostfix uses the StringSplitter you worked with during the lab to tokenize the input String that contains the infix expression, and then follow the algorithm below to convert it into a postfix representation. In order to accomplish this task, we are still going to use stack. In this case, we only need one stack to store the operators (you no longer need a stack to store the operands).

You already should have learned how to use StringSplitter and how to declare your operator stack in the lab instruction. The helper methods you implemented during lab will prove very helpful!

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

1. Scan the input string (infix notation) from left to right (one pass)
2. If the current token is an operand
  - 2.1. append it to the postfix string.
3. If the current token is an operator (call it *current operator*).
  - 3.1. Pop from the operator stack and append to the postfix string every operator on the stack that
    - 3.1.1. is above the most recently scanned left parenthesis, and
    - 3.1.2. has precedence higher than to that of the current operator
  - 3.2. Push the current operator onto the stack.
4. If the current token is a left parenthesis
  - 4.1. push it onto the stack.
5. If the current token is a right parenthesis
  - 5.1. Pop all operators down to the most recently scanned left parenthesis and append them to the postfix string.
  - 5.2. Pop the corresponding left parenthesis and discard this pair of parentheses

Before start writing your code, grab a piece of paper and apply the algorithm on the following expression: " $((4+5)*6)$ ". The output should be: " $45+6*$ ". Follow the algorithm ***exactly!***

## Part 3 – Test your code

After you are done implementing your `toPostfix`, you need to test it. We have provided different test scenarios for you in the main method of the class. You should uncomment these tests.

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.