



**Joe Del Rocco**  
jdelrocco [at] stetson [dot] edu  
Assistant Professor of Practice, Computer Science  
Stetson University  
421 N Woodland Blvd, DeLand, FL, 32723  
www.stetson.edu

**CSCI 142**  
(all sections)  
Spring 2022  
Assignment

## Assignment 2

---

Due: Monday 2/21/2022 11:59pm

### Contents

<b>Program</b>	<b>2</b>
Warning! . . . . .	2
Stack(s) . . . . .	3
infix2postfix() . . . . .	3
solve() . . . . .	4
Testing . . . . .	4
<b>Submission</b>	<b>4</b>
<b>Rubric</b>	<b>5</b>
<b>Example Output</b>	<b>5</b>

## Program

This program will give you practice with writing and using a Stack ADT. You will develop a program that computes mathematical expressions by converting the expression from infix notation to postfix notation, [respecting the order of operations \(PEMDAS\)](#), and then solving it. You will use two Stacks to do this. One Stack will be used to convert the infix expression to a postfix expression, and one Stack will be used to solve the postfix expression. You must write your own Stack class(es). Examples of this were shown in lecture. See the [Example Output](#) for an example of how this program should perform.

Here is the link to the GitHub Classroom assignment:

<https://classroom.github.com/a/mf7sEewx>

## Warning!

Do not make any changes to the provided `Main.java` file. Also, do not change the name of the class or method signatures of the provided class `PostfixCalculator`. You can add more methods and functionality to this class, but you cannot change the name of anything that is already provided (or `Main` won't compile).

### Tip

The `PostfixCalculator` class already comes with the following two methods...

```
public static String infix2postfix(String infix)
```

This takes an infix expression `String` and returns the equivalent postfix expression `String`.

```
public static double solve(String postfix)
```

This takes a postfix expression `String` and should return the answer as a `double`.

### Tip

Keep in mind that all infix and postfix expressions used in this assignment have spaces between each symbol (number or operator or parentheses). And numbers can be either positive or negative.

### Tip

Recall the order of operations (PEMDAS) stands for:

P = Parentheses

E = Exponentiation

MD = Multiplication and Division

AS = Addition and Subtraction

where P has the highest precedence and AS has the lowest precedence.

However, please note that “PEMDAS” taken literally has a major shortcoming:

1. division is not associative
2. and the right-to-left or left-to-right reading is not specified for operators with the same precedence (Mult and Div; Add and Sub)

This means that unless the expression uses parenthesis to dictate the order or similar precedence operators, some expressions are inherently ambiguous, and different programs/algorithms will

result in different answers. [Please read this article for a great explanation.](#)

## Stack(s)

You will need to write at least 1 Stack class, if not 2. It is fine if you write 2 separate Stack classes. You could also write a single Stack class that uses Generics and supports multiple types. You may create your Stack class(es) as array-based or linked-based, the choice is yours. Once you have your Stack class(es), you will use them in the methods below. One method will require a Stack of **Strings** and the other method will require a Stack of **doubles**. Examples of writing a Stack class were shown in lecture (both array-based and linked-based).

You cannot use any provided classes from the [Java Collections Framework](#), including **Stack**, **Queue**, **List**, **Vector**, **Deque**, etc. The reason for this is so you get practice writing your own Stack class. In fact, you cannot import and use any classes provided in `java.util`, except for **StringTokenizer**. You should use a **StringTokenizer** for both methods below to make it easier to parse out negative numbers.

### Tip

We can easily test your submission to verify what you have imported from `java.util` with this JDK tool from the command shell:

```
jdeps -verbose:class .
```

## infix2postfix()

This method will use a Stack of Strings to convert an infix expression into a postfix expression. Please [search the internet](#) for many examples ([like this lecture](#)) which explain what postfix expressions are and how to convert an infix expression to a postfix expression. Your book also talks about postfix expressions. We have provided an algorithm that you can follow in the comments of the starter code of your assignment repo. Basically, you separate all the tokens between the spaces of an infix expression (usually with a **StringTokenizer**) and then you visit each token and decide if it should be appended directly to a postfix expression or put into a stack to retrieved later depending on the type of token it is - operator, operand, open parenthesis, or close parenthesis. The algorithm needs to respect the order of operations (PEMDAS), so when you come across an operator, you need to check its precedence against the existing operators in the Stack before pushing the new one.

### Tip

Here is an example of using the **StringTokenizer** class:

```
String message = "Hello there my friend";
StringTokenizer tokenizer = new StringTokenizer(message, " \\t\\n");
while (tokenizer.hasMoreTokens()) System.out.println(tokenizer.nextToken());
```

Which produces the following output:

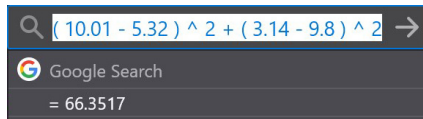
```
Hello
there
my
friend
```

## solve()

This method will use a Stack of doubles to solve a postfix expression. We have provided an algorithm that you can follow in the comments of the starter code of your assignment repo. Basically, you separate all the tokens between the spaces of a postfix expression (usually with a `StringTokenizer`) and then you visit each token, if it is an operator then you pop off the top two numbers from the Stack and compute them appropriately depending on the operator, whereas if the token is a number you push it onto your Stack. Your book also talks about a postfix expression evaluator in detail ([Java Foundations, Chapter 12.4 of the 5th edition](#)).

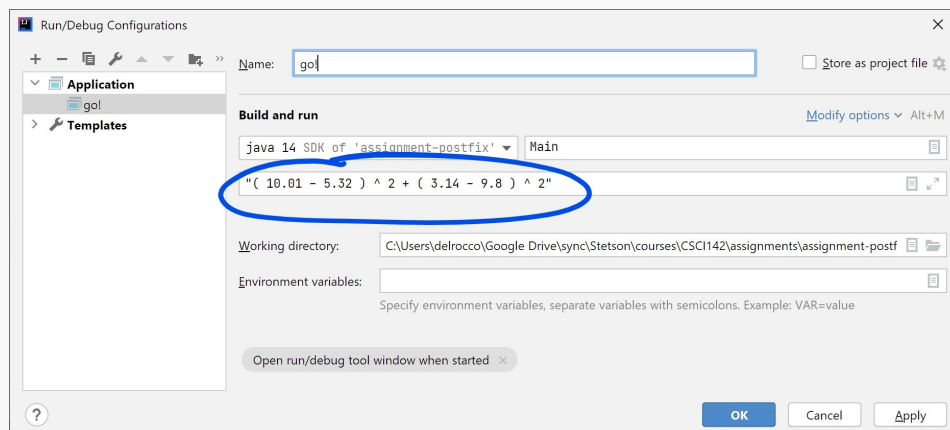
## Testing

If you compile and run your program with the provided starter code without any changes, it will print out 0.0 as the answer. Once you get the program working, the answer to the default postfix expression input ("2 2 5 4 \* \* +") will be printed out: 42.0. However, you can also test your program by passing in an infix expression as an argument. You can do this in the command shell as shown in the [Example Output](#). You can also do this from your IDE in your Debug/Run configuration options. You can also quickly covert infix expressions to postfix expressions using an [online calculator](#). And finally, you can type infix expressions into Google and it will solve them for you.



### Tip

Here is an example of passing an argument to a program in JetBrains IntelliJ IDEA:



## Submission

You will commit and push your changes to your specific GitHub Classroom repository for this assignment. You are encouraged to use an IDE for development, but we will compile and run your program using the shell/terminal during grading, so it isn't a bad idea to test it in that environment to make sure it works. Please follow the directions in this assignment, make the requested code changes, and commit and push your changes any time before the due date. Please see the advice below; it is important for grading purposes. **Failure to follow these directions will result in a loss of points.**

Always make sure to:

- Keep all source files in the folder called **src**, which is one directory in from the root of your repo
- Do not commit multiple copies of the same named source file; modify the ones provided to you. In other words, do not make an old and new version of the same file
- The main starting source file should always be called **Main**
- When loading resources, do not use absolute paths to files on your drive; [use relative paths](#)
- Do not have the keyword **package** at the top of any files. Some IDEs add your files to a custom package by default. Please remove this, as it complicates grading.

## Rubric

Task	Percentage
Assignment files not pushed to GitHub	Grade is 0%
Use of any class from <code>java.util</code> except <code>StringTokenizer</code>	Grade is 0%
General attempt at solving the assignment	40%
Separate Stack class(es) provided and working properly	20%
<code>infix2postfix()</code>	20%
<code>solve()</code>	20%
Total	100%

## Example Output

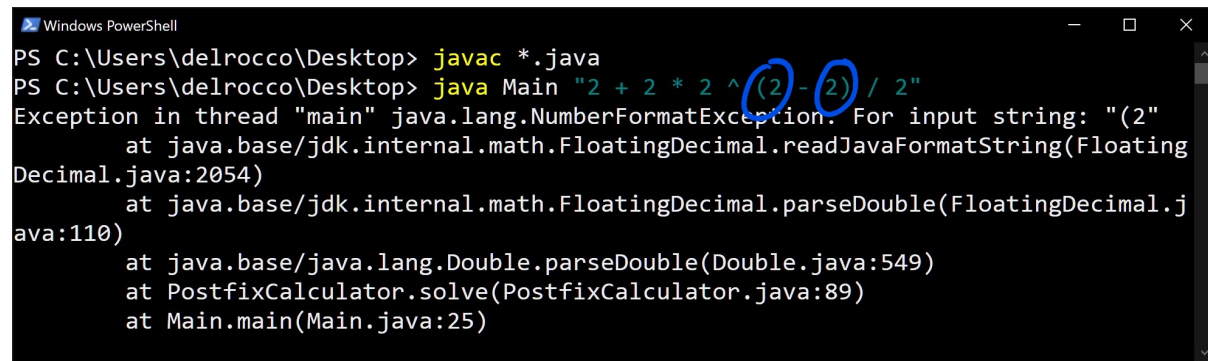


```

Windows PowerShell
PS C:\Users\delrocco\Desktop> javac *.java
PS C:\Users\delrocco\Desktop> java Main
Answer: 42.0
PS C:\Users\delrocco\Desktop> java Main "3 + -2 * 2"
Answer: -1.0
PS C:\Users\delrocco\Desktop> java Main "( 3 + -2 ) * 2"
Answer: 2.0
PS C:\Users\delrocco\Desktop> java Main "1 + 2 * 2 - 1"
Answer: 4.0
PS C:\Users\delrocco\Desktop> java Main "-1 + -1 - -1 * -1 / -1"
Answer: -1.0
PS C:\Users\delrocco\Desktop> java Main "2 ^ 2 ^ 2 ^ 2"
Answer: 65536.0
PS C:\Users\delrocco\Desktop> java Main "2 + 2 * 2 ^ ( 2 - 2 ) / 2"
Answer: 3.0
PS C:\Users\delrocco\Desktop> java Main "45 + ( ( 88 * 90 ) - ( 67 - -32 ) )"
Answer: 7866.0
PS C:\Users\delrocco\Desktop> java Main "( 10.01 - 5.32 ) ^ 2 + ( 3.14 - 9.8 ) ^ 2"
Answer: 66.3517

```

Remember to put spaces between each token of both infix and postfix expressions:



```
Windows PowerShell
PS C:\Users\delrocco\Desktop> javac *.java
PS C:\Users\delrocco\Desktop> java Main "2 + 2 * 2 ^ (2 - 2) / 2"
Exception in thread "main" java.lang.NumberFormatException: For input string: "(2"
    at java.base/jdk.internal.math.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2054)
    at java.base/jdk.internal.math.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
    at java.base/java.lang.Double.parseDouble(Double.java:549)
    at PostfixCalculator.solve(PostfixCalculator.java:89)
    at Main.main(Main.java:25)
```