Lab 4

1. **Due** Start of your next lab period

2. **Purpose**

    a. Improve Java programming proficiency with NetBeans IDE
    b. Employ the Java Swing GUI library for a desktop application
    c. Understand the use of Java event handlers

3. **What you are expected to know prior to completing this lab**

    a. Review Java programming coding structure, classes, interfaces
    b. Basic use of the NetBeans version 8 IDE, debug process
    c. Understand the purpose and operations of a stack structure
    d. Although this lab uses the Java Swing API to define the GUI, the graphical elements used are quite simple (buttons and a text area display). We will be covering the Swing library in class.

4. **Resources**
    a. The Java class Float - http://docs.oracle.com/javase/8/docs/api/java/lang/Float.html
    b. The Java class Stack – http://docs.oracle.com/javase/8/docs/api/java/util/Stack.html
    c. The Java class String – http://docs.oracle.com/javase/8/docs/api/java/lang/String.html
    d. What is Reverse Polish notation http://www.calculator.org/rpn.aspx

5. **Overview**

    a. Design and implement a working postfix calculator as a standalone Java application using Java Swing GUI components.

    b. In grade school we learned math using what is known as the algebraic or *infix notation* for resolving mathematical expressions, which may look like this:

```
4 + (5 - 1) × (7 - 3) / 2
```

or

$$x = (-b \pm \sqrt{(b\text{^}2 - 4ac)})/2a$$

In grade school we learned that *operands* are what appear to the left and right of the *operator*. For example, in the expression `a + b`, the operands are `a` and `b`, the operator is the plus sign. Operands can be expressions,

too, as in `(a+b) × (c-d)` where the operands are `(a+b)` and `(c-d)` for the multiplication operator `×`.

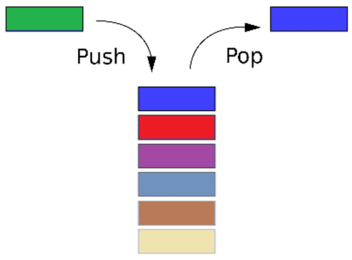
The infix notation for math expressions is easy for humans to read, understand, and figure out. The parentheses in the math expressions serve to show which calculations in the expressions have priority and the order of operations is respected (multiplication and division ahead of addition and subtraction). You can probably work out the first infix notation example above in your head without too much difficulty. (answer is the number twelve)

Humans can read and process infix notation expressions without too much difficulty. On the other hand, for computers to understand and perform calculations on infix notation expressions there are some complications. Parentheses supplied with the infix notation must be processed in some way. The parentheses in the expressions must always balance or there is an error. Also, there could be many levels of parentheses in the expression so that makes it a bit more of a challenge for the computer to determine the calculation. For example for the computer to evaluate this infix notation expression requires work to confirm the parentheses balance correctly and to process the operations in parenthesis priority order

```
( 2 + ( (4 + 2) × 3 - ( 5 - 2) ) × 2)  / 4
```

Fortunately there is a simple way to form any math expression so such a way that the computer can quickly evaluate it and parentheses are never required.

The *postfix* notation (also called reverse Polish) for math expressions places the operator *after* the two operands. This notation was first developed in the late 1950's based on earlier work (the "prefix notation" from a noted Polish mathematician Jan Lukasiewicz in 1923) to make the scientific calculators of the time easier to program and preserve the priority of operations without the bother of entering parentheses symbols.



How postfix works is simple and its implementation requires the understanding how a *stack* mechanism implements LIFO ("last in, first out"). Reading left to right, we look for operands and operators and deal with each one at a time. A number value (an operand) found in the expression is pushed on a stack. Any operator found in the expression causes the stack to immediately *pop* two operands, perform that operator's operation, then *push* the new value onto the top of the stack. This process continues until all the operators and operands are examined. The remaining value left on the stack is the final calculation of the postfix expression.

As a demonstration assume you have an input stream consisting of separate tokens – a token is either a number (an operand) or a math operator.

In **post**fix notation the input stream of three tokens: `2 3 +` is processed as if it were the **in**fix notation `2 + 3`. First the operand `2` is pushed on the stack, then the operand `3` is pushed on the stack. The plus sign operator causes both `2` and `3` to be popped, added together, then the sum is pushed on the stack. Note the order of the operands; it is processed as `2 + 3` not `3 + 2`.

A postfix input stream of these five tokens: `2 3 4 × +` is processed as `(3 × 4) + 2`.

An input stream of these five tokens: `2 3 + 4 ×` is processed as `(2 + 3) × 4`.

An input stream of these seven tokens: `1 2 + 3 × 4 −` is processed as `(1 + 2) × 3 − 4`.

Notice the total lack of parenthesis in the postfix notation expressions—they are not required. This makes postfix notation much simpler to express and process more efficiently within a computer program.

A stack data structure makes evaluating postfix expressions simple. Think of a stack of dinner plates. When you add a new plate, it is placed on the top of the stack. That is a *push* operation for the stack. When you remove a plate, it must be the one on the top. That is a stack's *pop* operation. A *peek* operation just examines what value is at the top of the stack without removing anything. Each operand (value) is *pushed* onto the stack and when an operator (like a plus sign) is seen, the top two operands are *popped* off the stack (that represents two pop actions) and the operator is applied. The sum is then *pushed* onto the top of the stack. If there are more operands following in the expression, those are pushed on top of the stack.

Here's another example:

Postfix expression: `3 4 × 5 +`
The stack starts empty.
`3` is an operand, so push `3` onto stack. Stack contains: `3`
`4` is an operand, so push `4` onto top of stack. Stack contains: `4 3`
`×` is an operator, so pop `4` and pop `3` then apply `×`, then push 12. Stack: `12`
`5` is an operator, so push `5` onto stack. Stack: `5 12`
`+` is an operator, so pop `5` and pop `12`, apply `+`, then push 17. Stack: `17`

6. Requirements of this Lab

a. Using the NetBeans IDE (or appropriate equivalent IDE) create a new Java application which implements a postfix calculator. Download the provided zip file (`RPNCalculator.zip`) of a starter calculator NetBeans project you can import as a project. To do this, start the NetBeans application, then select from the main menu File | Import Project. Use the provided Java Swing GUI frame code named `CalculatorGUI.java` as the front end. Do not modify this file. In fact you won't be able to alter a large portion of it anyway because NetBeans won't let you do that outside of the design tab.

   The required Java class you will need to add to the project is called `CalcBrain.java` and it will need to implement the provided Java interface named `Calculations.java`. The javadoc for the class is also provided for you to get hints as to the implementation details. A sample working calculator application jar file is also provided. A recommended approach would be to start with just handling the digits (display them in the GUI textbox area, then "push" the complete operand onto the stack).

b. There are JUnit test cases provided as well to confirm that your application is working correctly. When you are ready to use the JUnit test, either press alt-F6 or select Run | Test Project to run the test cases.

c. Required use cases for the calculator:
   i. Any number of digits can be pressed to define the operand value. Think about how you could handle this requirement in your code. Hint: it is not a complicated detail.
   ii. A decimal point may be pressed once at any time during entry to provide a fractional value to the operand. Subsequent decimal point entries for that operand are ignored.
   iii. A clear entry button causes the current operand value to be removed.
   iv. A clear button causes all operand values in memory to be removed.
   v. The five operator buttons (`+ - × / ^`) are not enabled until the second operand is entered.
   vi. First operand is confirmed when the enter button in the GUI panel is pressed.
   vii. The second and subsequent operands are entered in the same manner as the first operand.
   viii. Pressing the enter button twice in succession does not affect the calculation. Pressing the operator buttons functions only where there are values to operate on (in other words pressing the + button twice in succession when there are only two operands in memory will only do a single addition calculation on the two operands).

d. Assumptions
    i. All operands are entered as positive numbers.
    ii. All operands can be of any arbitrary size; however, the precision of the operand is limited to Java's float precision as you will see when you experiment with the provided sample Calculator application.

e. Create an additional appropriate JUnit test case using the attached test file for your application and show the results of the tests.

7. Hand In / Demonstration:

A. Demonstrate your working Calculator. Use the D2L dropbox to submit your `CalcBrain.java` and test results.

B. Marks
    a. Completed JUnit test results: 10
    b. Java Source code CalcBrain.java: 10