

## Problem Statement

We are required to write a solution to an assignment that requires us to write a calculator that is stack based. This means that we are required to first begin by converting the expression to a postfix expression from infix. This is probably the most challenging part of the assignment because the instructions in the assignment are vague for implementing this algorithm so we are required to figure it out on our own. So there is the conversion to infix, and then there is the calculator which will evaluate the operands in the stack. The calculator will need to be designed in such a way that it accounts for the way that operands are usually evaluated - their order, and the fact that the stack has a last in first out order of operations.

## Analysis and Design Notes

Fortunately, we don't need to consider implementing the stack itself, or even editing it. All we need to focus on is conversion from infix to postfix and then finally evaluating the expression and printing out each operation that the stack calculator does. First things first, I need to create a constructor that will instantiate an object from the stack. Then I need to create an object that will scan in the infix expression from the user. I will need to validate it to ensure that the expression is between the length of 3 and 20. If that is so, I will enter and check if the expression has the correct operations and operands. If that is not so, it will print a message to the user, clear the input string, then it will break and then ask the user for another input. This will continue until the expression is valid, at which point, I will then set the while loop to a variable that will end the while loop. When the while loop is which will begin the calculation of this expression. The assignment suggests that I use a character array but since it is only a suggestion, I have instead decided to use a string. The reason being is that I can access the methods of strings that make this process of character conversion easier because there are inbuilt methods that make it so that I don't have to worry about indexes and the shuffling of characters in a character array. For this reason, I have decided to just transfer directly the successful input string straight to the conversion method. In the conversion method, I am creating three variables that will handle the conversion of infix to postfix. It uses the implementation that Frank describes. The temp string will handle operators and their precedence, which is determined by a separate method that will return values based on a given character. Notably, since it is strings that I am dealing with, there is a lot of repetitive code that I have compartmentalised into functions that will handle the 'popping', 'peaking' and also updating the string so that each time an element is removed, it is updated not just locally. Each time I use the local variables, first and second, I make sure I clear them to prevent an infinite loop of adding more and more variables. Eventually, after converting from infix to postfix, my function will return and output, which is a field in my class, will be updated and be visible across all of the object. When I have converted the expression, I will feed this expression into my calculator method, which works the exact same as a regular calculator with some differences. I will continuously feed values into my stack that return a -1 when my precedence method is called. This means that I am just feeding digits into my stack and not operands. If when I am going through each of the values of my output string, the postfix string, it comes across an operator, it will take two values from the stack. Since it is a stack, it will assign the first value popped off the stack to be the second value that is operated on. The local variables that will handle the calculation of these values and the are going to be printed to the console. These values are going to be doubles

because then `math.pow` will not work because simple floats will result in lossy conversion. Finally, the total is printed to the console. The calculation is complete.

## Code

```
import javax.swing.JOptionPane;

public class Postfix_To_Infix
{
    // This string will be used for the value taken in from the user
    private String input;
    // This will be false and it will be used as a condition for the
    while loop that is initiated in the constructor
    private boolean validExpression = false;
    // This will store the postfix value
    private String output = new String();
    // This reference will store a stack object
    private ArrayStack S;
    // These will store the local variables and are accessed in the
    calculation
    private double var1 = 0F;
    private double var2 = 0F;

    //For now I will just use a scanner
    //Scanner scan = new Scanner(System.in);

    public Postfix_To_Infix()
    {
        // Here I am instantiating an array stack
        S = new ArrayStack();
        // // // //

        // This is the loop condition that will continue until the
        condition is met. This is to prevent an invalid expression from being
        evaluated by the other methods
        while (!validExpression) {
            input = JOptionPane.showInputDialog(null, "Please enter
            input:");
            // Here I am checking if the length is in the right range 3
            to 20
            if (input.length() > 3 && input.length() < 20) {
```

```

        // This will provide an additional check that will be
        updated constantly in the loop. If the flag is set to false, then the
        condition for

        // exiting the while loop will not be reached
        boolean good = false;
        // If that is the case, then I will go through each
        value and check certain conditions
        for (int i = 0; i < input.length(); i ++ ) {
            // First I am checking that all of the characters
            are correct - either an operator or an operand
            if (!(input.charAt(i) == '+' || input.charAt(i) ==
            '-' || input.charAt(i) == '*' || input.charAt(i) == '/' ||
            input.charAt(i) == '^' || input.charAt(i) == ')' || input.charAt(i) ==
            '(' || Character.isDigit(input.charAt(i)))) {
                // If not, I clear the input string, print a
                message, set good to false and break the for loop. There is no need to
                continue evaluating
                JOptionPane.showMessageDialog(null, "Make sure
                that your input contains * / ^ + - 0-9");
                input = "";
                good = false;
                break;
            }
            // I am accounting for a case where there are two
            operators that are placed beside each other in an expression. This of
            course is not valid and must be evaluated
            /*else if ((i > 1) &&
            (!Character.isDigit(input.charAt(i - 1)) || (input.charAt(i - 1) ==
            ')') || (input.charAt(i - 1) == '(')) &&
            (!Character.isDigit(input.charAt(i)) || (input.charAt(i) == ')') ||
            (input.charAt(i) == '('))) {
                System.out.println("Invalid expression");
                input = "";
                good = false;
                break;
            } */
            // Otherwise, I set the flag to true
            else {
                good = true;
            }
        }
        // If the flag is true, then the while loop will end
        if (good) {

```

```

        System.out.println("Valid expression");
        validExpression = true;
    }
}
// Check that this expression breaks and evaluates to true
when whanted
    else {
        JOptionPane.showMessageDialog(null, "Expression must be
between 3-20 characters");
        input = "";
    }
}
infixToPostfix();
calculator();
}
// This is used to avoid the repetition of the calling of the same
methods that are used in strings as a part of a process of popping the
element
public char popString(String str) { // To avoid repetition
    char element = str.charAt(str.length() - 1);
    if (str != null && str.length() > 0) {
        str = str.substring(0, str.length() - 1);
    }
    else {
        System.out.println("ERROR popping string");
    }
    return element;
}
// This is the equivalent of the 'top' method that frank has, but
for strings
public char peekString(String str) { // to avoid repetition
    return str.charAt(str.length() - 1);
}
// This will then complete the popping method and update the string
by removing the last element
public String popStringUpdate(String str) {
    if (str != null && str.length() > 0) {
        return str.substring(0, str.length() - 1);
    } else {
        System.out.println("ERROR popping string");
        return str;
    }
}
}

```

```

// This method is used for converting infix to postfix notation
public String infixToPostfix() { //(3+4)*8+9*(5-6)
// These are temporary values that are used to check values
String temp = new String();
String first = new String();
String second = new String();
// Then I will loop for the length of the input
for(int i=0;i < input.length(); i++) {
    // I am letting ch be the initial value. I am storing it in ch
for the sake of modularising my code
    char ch=input.charAt(i);
    // Temp will be used to deal with operators. Temp will store
the bracket rather than adding it to the real output
    if(ch=='(') temp += ch ;
    // I am then checking if ch is a digit. If it is, due to the
nature of stack calculations, I will simply add it to my stack
    else if(Character.isDigit(ch)) {
        output += (ch+"");
    }
    // If it is a closing bracker, I will need to deal with all of
the values in the brackets so that they are given priority in the stack
and that the
    // values in the brackets are kept altogether
    else if (ch == ')') {
        // If the character is ')', handle the closing brackets
case

        // Continue until an opening bracket '('
        while (peekString(temp) != '(') {
            // Pop the operator and update temporary stack
            char op = popString(temp);
            temp = popStringUpdate(temp);

            // Pop operands from output stack and update the string
            first += popString(output);
            output = popStringUpdate(output);
            second += popString(output);
            output = popStringUpdate(output);

            // Create a postfix expression and update output string
            String new_postFix = second + first + op;

```

```

        output += new_postFix;

        // Reset operand variables
        first = "";
        second = "";
    }

    // Pop the opening parenthesis '(' from the temporary stack
    popString(temp);
    temp = popStringUpdate(temp);
}

else if(ch=='+' || ch=='-' || ch=='*' || ch=='/' || ch ==
'^') {
    // If the character is an operator handle the operator

    // Continue while there are operators in the temp string,
    // and the end of the string is not an opening bracket '(',
    // and the precedence of the operator is less than or equal
to the precedence of the operator at the end of the string
    while(temp.length()>0 && peekString(temp) !='(' &&
precedence(ch) <= precedence(peekString(temp))) {
        // Pop the operator and update
        char op = popString(temp);
        temp = popStringUpdate(temp);
        first += popString(output);
        // Pop operands from the output
        output = popStringUpdate(output);
        second += popString(output);
        output = popStringUpdate(output);
        String new_postFix = second+first+op;
        output += new_postFix;
        // Empty string for the next iteration
        first = "";
        second = "";
    }

    // Append current character to temp
    temp += ch;
}

}

// Then empty the rest of the string into the output until temp is
empty
while(temp.length()>0) {

```

```

        char op = popString(temp);
        temp = popStringUpdate(temp);
        first += popString(output);
        output = popStringUpdate(output);
        second += popString(output);
        output = popStringUpdate(output);
        String new_postFix = second+first+op;
        output += new_postFix;
        first = "";
        second = "";
    }
    System.out.println(output);
    return output;
}

public void calculator () {
    for (int i = 0; i < output.length(); i++){
        if (precedence(output.charAt(i)) == -1) {
            S.push(output.charAt(i));
        }

        if (precedence(output.charAt(i)) == 1) {
            if (output.charAt(i) == '+') {
                var2 =
Double.parseDouble(String.valueOf(S.pop())); // 4+28-7/4* - number
format exception
                var1 =
Double.parseDouble(String.valueOf(S.pop()));
                S.push(Double.toString(var1 + var2));
                System.out.println(var1 + " + " + var2 + " = "
+ S.top());
            }
            else {
                var2 =
Double.parseDouble(String.valueOf(S.pop()));
                var1 =
Double.parseDouble(String.valueOf(S.pop()));
                S.push(Double.toString(var1 - var2));
                System.out.println(var1 + " - " + var2 + " = "
+ S.top());
            }
        }
        if (precedence(output.charAt(i)) == 2) {

```

```

        if (output.charAt(i) == '*') {
            var2 =
Double.parseDouble(String.valueOf(S.pop()));
            var1 =
Double.parseDouble(String.valueOf(S.pop()));
            S.push(Double.toString(var1 * var2));
            System.out.println(var1 + " * " + var2 + " = "
+ S.top());
        }
        else {
            var2 =
Double.parseDouble(String.valueOf(S.pop()));
            var1 =
Double.parseDouble(String.valueOf(S.pop()));
            S.push(Double.toString((var1 / var2)));
            System.out.println(var1 + " / " + var2 + " = "
+ S.top());
        }
    }
    if (precedence(output.charAt(i)) == 3) {
        var2 = Double.parseDouble(String.valueOf(S.pop()));
        var1 = Double.parseDouble(String.valueOf(S.pop()));
        S.push(Double.toString(Math.pow(var1, var2)));
        System.out.println(var1 + " ^ " + var2 + " = " +
S.top());
    }
}
}

// This will return a number depending on the symbol that is taken
as an argument
private int precedence(Character operator) {
    switch (operator) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        default:
            return -1;
    }
}

```



```

    }

    // The entry of the main application
    public static void main(String args[]) {
        Postfix_To_Infix p = new Postfix_To_Infix();
    }
}

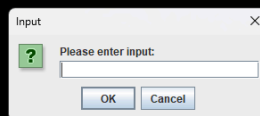
```

## Testing

```

Valid expression
67+4*4-
6.0 + 7.0 = 13.0
13.0 * 4.0 = 52.0
52.0 - 4.0 = 48.0
PS C:\Users\sadeg\Downloads\Assignment5_SADEGRAY_22337743> java Postfix_To_Infix.java

```



2

PROBLEMS 3 OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

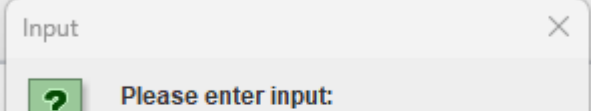
```

Valid expression
67+4*4-
6.0 + 7.0 = 13.0
13.0 * 4.0 = 52.0
52.0 - 4.0 = 48.0
● PS C:\Users\sadeg\Downloads\Assignment5_SADEGRAY_22337743> java Postfix_To_Infix.java
Valid expression
80-4*6-
8.0 - 0.0 = 8.0
8.0 * 4.0 = 32.0
32.0 - 6.0 = 26.0
○ PS C:\Users\sadeg\Downloads\Assignment5_SADEGRAY_22337743>

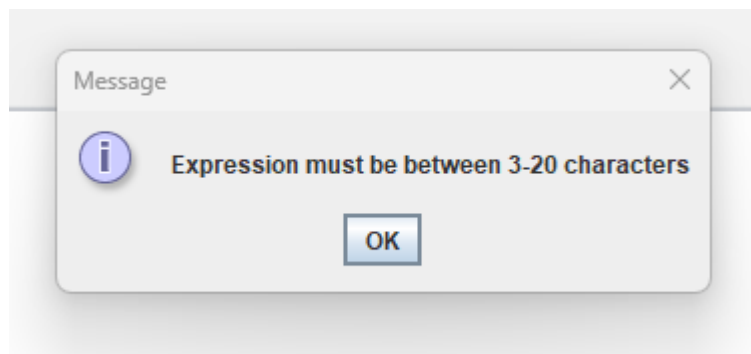
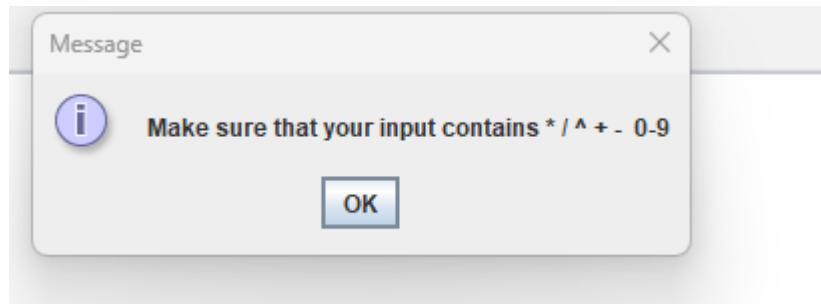
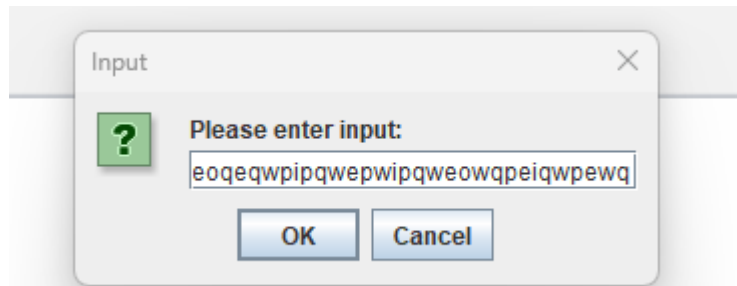
```

[illegible]

```
3.0 + 6.0 = 9.0
● PS C:\Users\sadeg\Downloads\Assignment5_SADEGRAY_22337743> java Postfix_To_Infix.java
Valid expression
58^625+*-
5.0 ^ 8.0 = 390625.0
2.0 + 5.0 = 7.0
6.0 * 7.0 = 42.0
390625.0 - 42.0 = 390583.0
○ PS C:\Users\sadeg\Downloads\Assignment5_SADEGRAY_22337743> 
```



```
PS C:\Users\sadeg\Downloads\Assignment5_SADEGRAY_22337743> java Postfix_To_Infix.java
Valid expression
36^8+5-
3.0 ^ 6.0 = 729.0
729.0 + 8.0 = 737.0
737.0 - 5.0 = 732.0
```



From the initial screenshot, when I begin the program, an input dialog box is created from the class `JoptionPane` using the method `showInputDialog()`. From the second screenshot, when I input an infix expression, it outputs the postfix notation and then each calculation that is performed from taking the last two values. This demonstrates that my code works well with calculations that are not anomalies - that they contain the correct operators and brackets and operands. When I input characters that are not operators or operands, as demonstrated in the screenshots above, it will inspect the characters and inform the user that this is not the correct input. Then it will also check the length. If this is also not correct, then it will inform the user. In my case, I input a string of characters that were much longer than 20 characters so my program informed me that this is too long. If I decide to mix characters and the correct operators and operands, then my program will resolve this by sending the message a message that this is not correct and that the user has input the incorrect characters. Evidently my program deals with these issues gracefully, even if the user decides not to input anything, the program will print out an error message and ask the user to input something else.