Calculator Project Year 2



**Author:** Derry Brennan C00231080

**Tutors:** Jason Barron and Aine Byrne

Contents

[Introduction 3](#_Toc536360740)

[Requirements 3](#_Toc536360741)

[Graphical User interface 4](#_Toc536360742)

[Calculation Logic 6](#_Toc536360743)

[Calculator Testing 7](#_Toc536360744)

[Algorithms 12](#_Toc536360745)

[Split a String 12](#_Toc536360746)

[Infix to postfix 13](#_Toc536360747)

[Post fix resolution 16](#_Toc536360748)

[Data Structures 18](#_Toc536360749)

[Errors and changes for the future 19](#_Toc536360750)

[Appendicies 20](#_Toc536360751)

[Infix to Postfix code 20](#_Toc536360752)

[Post fix resolution code 22](#_Toc536360753)

# Introduction

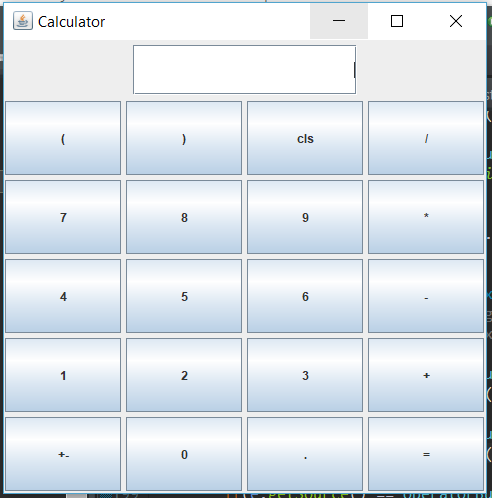
We were tasked with providing a functional calculator written in java. We were to supplement this with the relevant algorithms and data structures which we were exposed to in on discrete structures course.

# Requirements

We were required to develop a calculator application in Java using the Graphical User Interface (GUI) components. We are to develop an application that can carry out at least the following functions: Addition, Subtraction, Multiplication and Division. Additionally we can provide a way to clear the screen of numbers and provide memory functions to remember numbers. Also we could implement a programmer defined Exception classes for Error Handling (i.e. use your own Exception Handling Classes).

# Graphical User interface

Version 1 of the calculator initially used a grid layout.



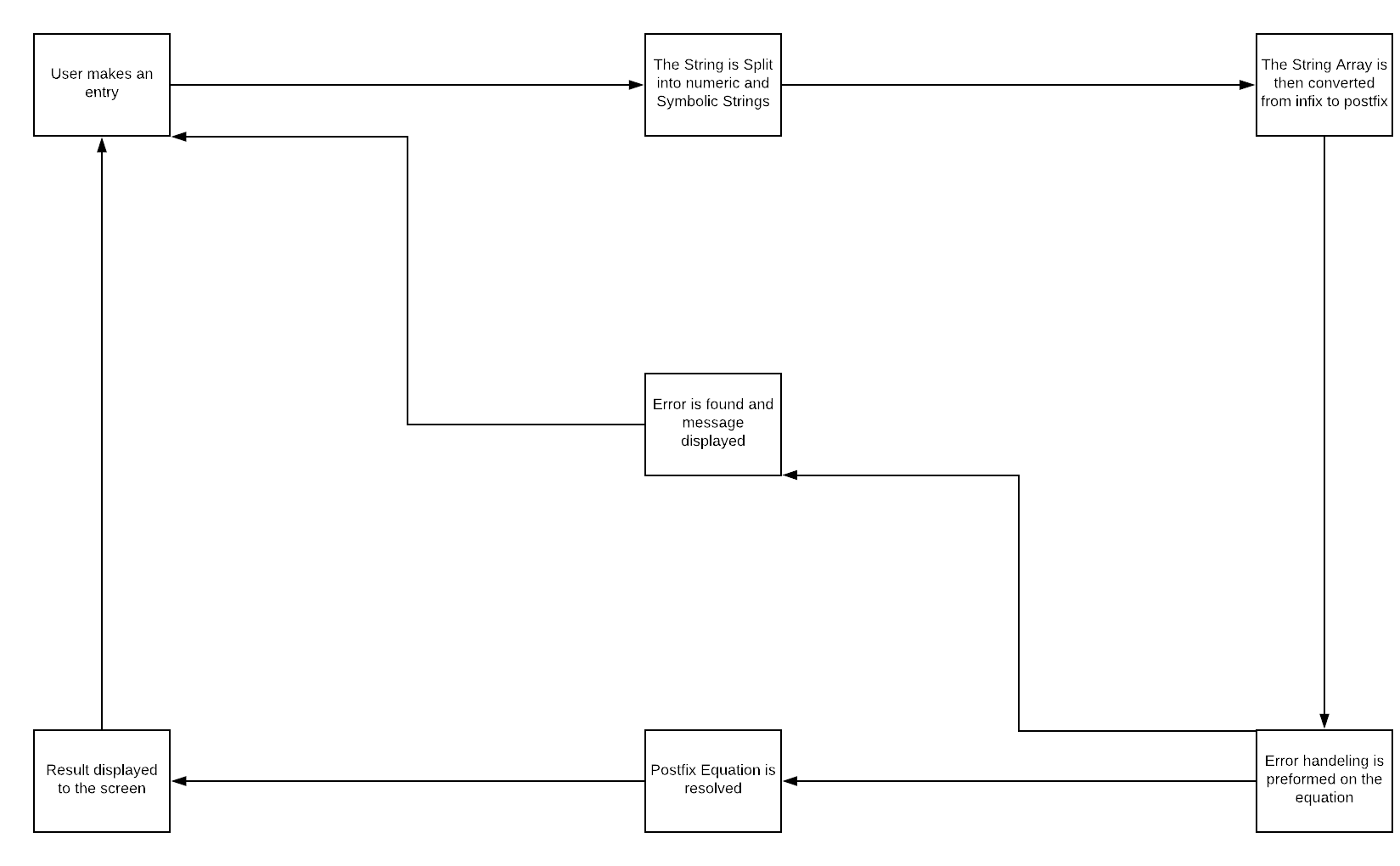
It did not change much from version to version 2 but I remembered that we were told that the Grid Bag Layout was the most versatile Layout that we could work with but also a bit more complex. So I decided to give it a go.



And this was the version that I settled on. Here I was able to have some elongated buttons both horizontally and vertically. I also added the memory buttons and the clear screen and backspace button.

The addition of some colour helps break up the buttons making them easier to see.

# Calculation Logic



As shown above the calculator is very cyclical. It takes an input, then splits that input into numeric and symbolic Strings and places them in order into an array.

That String array is then sent to my infix to post fix class where using the reverse polish notation algorithm (which I will go over in detail later) I reorganise the array into a postfix expression.

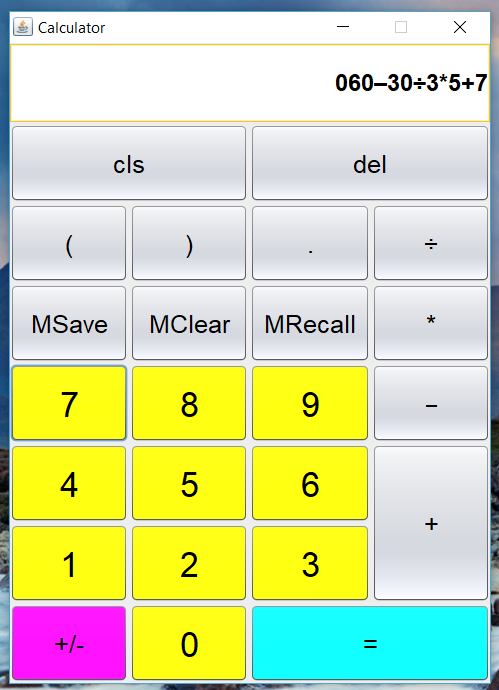
We preform some error handling on the expression at this stage to make sure it is a valid expression and the pass it on to the Postfix class where the expression is resolved.

Once that is done the result is displayed on screen and the user can then enter more, save the answer to memory or just close down the application.

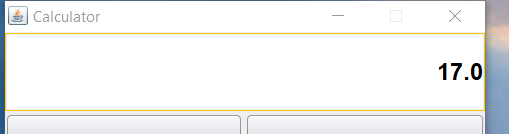
# Calculator Testing

All equation results generated on the Samsung calculator\*

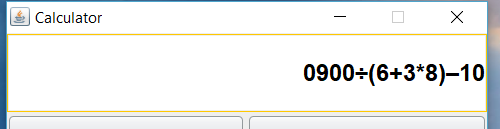
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test Scenario | Test Case | Expected Result | Actual Result | Success? |
| 1 | User types a number or operator | Any number/operator button pressed | The character should appear on screen | The character appears | Success |
| 1.1 | The minimize or X button Pressed | Pushed the minimize button and the later the X button | The calculator should be minimized to the task bar and closed when X is pressed | It was both minimized and closed when the respective buttons were pressed | Success |
| 2 | MSave pressed while a numeric character is on screen | 75 entered and MSave pressed | The number should be saved to memory | The number is saved to memory | Success |
| 3 | MRecall pressed while a number is saved to memory | 75 has been saved to memory and we press the recall button | The memory number should appear in the screen | The number appears in the screen | Success |
| 4 | MClear button pressed while a number is saved in memory | 75 has been saved in memory and we hit the Mclear button | Upon hitting the MRecall button now nothing should happen | The memory is reset to an empty string | Success |
| 5 | Cls(clear screen) pressed while the screen is displaying a string | 888\*999-666+333  Typed to screen and cls button pressed | The screen should be reset back to default “0” displayed | The screen resets back to “0” | Success |
| 6 | Del(delete) button pressed while a string is displayed on screen | 888\*999-666+333  Is on screen and del button pressed | A Last in First out approach is taken to the characters entered and each time del is depressed the next last character entered is deleted | 888\*999-666+33  Is displayed after 1 press.  888\*999-66 after 5 presses etc. | Success |
| 7 | MSave button pressed while anything other than a numeric value is displayed on screen | 8+8-6 is displayed on screen | 10 should be saved into memory | Nothing is saved into memory as the only thing saved is a numeric value at the moment | Failure |
| 8 | Simple addition | 4+5 and the equals button pressed | 9 should be displayed | 9 is displayed | Success |
| 9 | Multistep Addition | 1+2+3+4+5+6+7 | 28 should be displayed | 28 is displayed | Success |
| 10 | Subtraction | 42-37 is displayed and equals pressed | 5 should be displayed | 5 is displayed | Success |
| 11 | MultiStep Subtraction | 55-20-14-37 displayed and equals pressed | -16 should be displayed | -16 is displayed | Success |
| 12 | Multiplication | 42\*57 is displayed and equals is pressed | 2394 should be displayed | 2394 is displayed | Success |
| 13 | MultiStep Multiplication | 3\*4\*5\*6\*7  Equals pressed | 2520 should be displayed | 2520 is displayed | Success |
| 14 | Division | 10÷3 and equals pressed | 3.3333333333 should be displayed | 3.3333333333333335  Is displayed. Limitation of double values | Success? |
| 15 | Multi step division | 5000÷30÷17÷6 | 1.6339869281  Should be displayed | 1.633986928104575  Is displayed, even more accurate than Samsung | Success! |
| 16 | Order of operations | 60-30÷3\*5+7 | 17 should be displayed | 17 is displayed | Success |
| 17 | Order of operations with brackets | 900÷(6+3\*8)-10 | 20 should be displayed | 20 is displayed | Success |
| 18 | Brackets multiplication | 6÷(3)(2) | 4 should be displayed | Returns 1.5  Gives back 3 divided by 2 | Failure |
| 19 | Negative numbers | -14 - -16 | 2 should be displayed | 2 is displayed | Success |
| 20 | Order of operation, multiple brackets, negative numbers and decimal points | (3.9\*0.5)+(-4.87÷3.12) | 0.3891025641  Should be displayed | 0.38910256410256405  Was displayed, again slightly more accuracy than samsung |  |

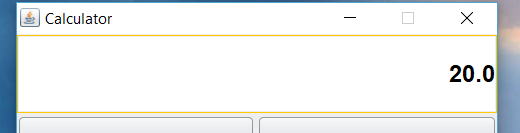


What follows is some screen shots of the more complicated calculations and their results.

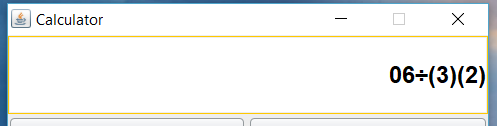


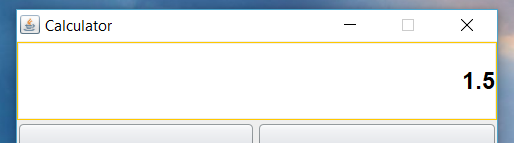
The answer to the previous equation



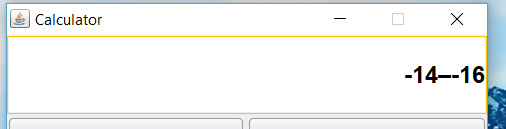


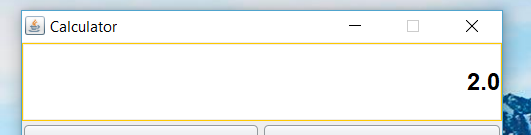
Happily it worked out to be a full number.



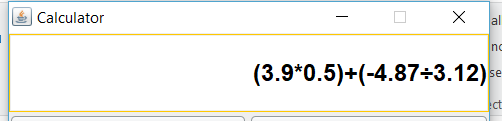


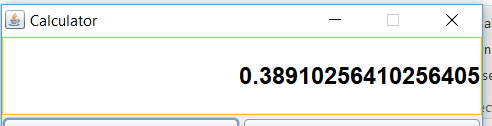
My calculator falls down when presented by bracket multiplication.





Here is an example of how I used different symbols for negative and subtraction to allow my regex to be able to tell the two apart.





Finally a stern test for the calculator.

# Algorithms

## Split a String

Within the calculator there were 3 main algorithms used.

A regex was used to split the String input into numeric and symbolic strings.

public static String[] StringArray(String str) {

int count = 0;

// regex for finding the division symbol, my custom subtraction symbol and the other operators plus and negative or positive digits decimal points and all

String regex = "(\u00F7)|(\u2013)|(-\\d+\\.\\d+)|(\\d+\\.\\d+)|(\\d+)|(-\\d+)|([+-/\*///^])|([/(/)])";

//String regex = "[0-9]\*\\.?[0-9]+([eE][-+]?[0-9]+)?|[-^+\*/\\u00F7()]|\\w+"; this was another possibility but i did not like it as much

Matcher m = Pattern.*compile*(regex).matcher(str);

// make a new array called list

String[] list = new String[50];

//runs the String through a regex matcher and once a match is found store it in the array, add to count and repeat

while (m.find()) {

list[count] = m.group();

count++;

}

// this was just for me to print out the results in the console to see how it was doing

// added the if(i != null) to ignore the empty sections of the array

for(String i : list) {

if (i != null) {

System.***out***.println(i);

}

}

return list;

One of the main challenges I had in my calculator was to differentiate between a negative symbol and a subtraction symbol. It would either see the subtraction symbol and then see any negative numbers as another subtraction or see negative numbers and mistakenly see subtraction symbols as another negative number. To get around this I used the “–“ for negative numbers and used the Unicode symbol \u2013 for the subtraction. That way the string was split into its correct components and we could proceed.

"(\u00F7)|(\u2013)|(-\\d+\\.\\d+)|(\\d+\\.\\d+)|(\\d+)|(-\\d+)|([+-/\*///^])|([/(/)])"

This is the Regex used. The first 2 it looks for is the division symbol and the custom subtraction symbol. The next is and negative digit followed by a decimal point followed by more digits. The next is the same but with positive a number. The 5th and 6th are looking for positive and negative integers and the 7th and 8th is looking for the remaining operators. It runs through a String from left to right and once it finds a match it puts that into an array and adds to count and continues until the string is split.

## Infix to postfix

The next algorithm used was for converting an infix expression to a post fix expression.

For the sake of neatness I will like to the code in to appendices and just run through pseudo code here.

Each operator is assigned a precedence value.

+ - = 1

/ \* = 2

( = -2

) = -3

And numbers were = -1

For (i=0; i<infixarray.length; i++)

String s = infixarray[i]

If (s = number)

Output s

//prec = precedence

Else if (Prec(s) == -2) // ‘(‘

Stack.Push(s)

Else if (Prec(s) == -3) // ‘)‘

While(Prec(stack.peek()) != -2)

Output s = stack.pop()

If (Prec(stack.peek()) == -2)

Stack.pop() // don’t save it anywhere

Else If(stack.isEmpty || Prec(s) > Prec(stack.peek()))

Stack.push(s)

Else

While(!stack.isEmpty() && Prec(stack.peek() >= Prec(s))

Output s = stack.pop()

Stack.push(s)

While(!stack.isEmpty())

Output s = stack.pop()

To put the above into a few sentences. Scanning through the array, if it is a number save it to a new array if it is a ‘(‘ or an operator push it to the stack. Continue this until an operator of lesser value to that of the one on the stack is found. Pop all the operators that are of greater precedence from the stack to the new array and then push the scanned one on. If you encountered an ‘(‘ along the way stop.

Always push an ‘(‘ and if the scanned string is a ‘)’ pop the stack until its matching ‘(‘ is found. And discard both brackets.

Repeat all of this until the entire expression is scanned through.

And finally pop the remainder of the stack until it is not empty.

## Post fix resolution

The next algorithm that was implemented is one to resolve the postfix expression we just created. We pass the postfix array to this class and it then uses a stack again to go through the expression and resolve it down to the single answer. Again I have supplied the actual code in the appendices and will go through the pseudo code here.

String[] exp // is the passed in postfix array

Stack<Double> stack = new stack<>

For (i= 0; i<exp.lenght; i++)

If(exp[i] != null)

String s = exp[i]

Double d = 0;

// a method I wrote that checks if the String is a number

If(isNumeric(s))

D = Double.parseDouble(s)

Stack.push(d)

Else

Double val1 = stack.pop()

Double val2 = stack.pop()

Switch(s)

Case “+”:

Stack.push(val2+val1)

Case “-”:

Stack.push(val2-val1)

Case “÷”:

Stack.push(val2÷val1)

Case”\*”:

Stack.push(val2\*val1)

If (!stack.isEmpty())

Return stack.pop()

Else

Return 0

This time around we pushed or operands onto the stack and popped 2 off when we encountered an operator. Depending on the operator we then applied it to the numbers whith the second pop coming 1st and the 1st pop coming second. We then pushed back the result to the stack and repeat until the whole expression has been looked at.

We return the (hopefully only remaining value in the stack) value left in the stack or return 0 if the stack is empty. This would happed in the event of an invalid expression trying to resolve itself.

# Data Structures

The two main data structures I used during this project were the Stack and the Array.

The Array is great for holding inputs and then being able to search through it linearly. So we split our String expression up into its component parts and then saved them to an array. This made manipulating it and scanning through it a lot easier in our infix to post fix method.

Again once we had scanned through our infix expression a new array allowed me to save the postfix expression in the correct order.

The Stack is basically an array where you can only interact with the top object. So you can put as many thing on here as you like but you will only be able to see or interact with the last value pushed on to here. It Was essential in being able to break up our infix expression to a post fix expression using order of operation values. We could thus order our operators to have the more important one appear first in our expression

In the postfix resolving method it was the operands we pushed onto the stack, since we would only ever need the last two operands we saw once we encountered an operator the stack is perfect for this again as a double pop will give us the correct numbers every time and we then push back their resulting answer back onto the stack.

Since I wanted to get a better understanding of the stack as it was my 1st real attempt at using it in an application I encountered a lot of empty stack exceptions but through those errors in my code I feel it helped me learn and understand the ins and outs of this data structure better.

# Errors and changes for the future

I feel that the GUI element of the calculator is pretty good apart for the placement of the decimal point. I my attempt to get the gridbag layout working well I didn’t plan out the button placements in an ergonomic fashion. Also after looking at some of the others projects an element like a history screen is both pleasant to look at and very functional.

In an attempt at avoiding errors my calculator starts off with a “0” on screen, this will then resolve to become part of the equation but initially it looks kind of out of place with the user inputs appending onto it. In a future implementation this is something I would change.

Also once a result is given it is in double format regardless of if it needs to be. It is the correct result but from an aesthetical point of view it is not pleasant.

I did not document my errors with screenshot throughout the development process and thus have none to show here, so in future I will keep track of those in a picture folder during development. Most of it would be repetitive empty stack exception errors as I was learning this data structure and they were very helpful.

One of the glaring oversights of my calculator is its ability to preform brackets multiplication. Upon future implementations of this application it would be on top of the features I would like to implement.

# Appendicies

## Infix to Postfix code

public static String[] infixToPostfix(String[] exp)

{

// initializing empty String for result

String[] result = new String[50];

int count = 0 ;

// initializing empty stack

Stack<String> stack = new Stack<>();

//loop for the length of the array

for (int i = 0; i<exp.length; ++i)

{

// but only carry out actions while there is an element in the array

if (exp[i] != null) {

// take the element in the array position i and store it in variable s(String)

String s = exp[i];

// if its a number write it to the array(do this by checking precedence)

if(*Prec*(s) == -1) {

result[count] = s;

count++;

}

// opening bracket push to stack

else if(*Prec*(s) == -2) {

stack.push(s);

}

// if its a closing bracket pop the stack into the array until a "(" is found.

//also do nothing with the ")", do not put it anywhere

else if(*Prec*(s) == -3) {

while(*Prec*(stack.peek()) != -2) {

result[count] = stack.pop();

count++;

}

// when open bracket is found pop it off the stack and save it no where

if(*Prec*(stack.peek()) == -2) {

stack.pop();

}

}

// if the stack is empty or the element on the stack has a lower precedence to s, push s to the stack

else if(stack.isEmpty() || *Prec*(s) > *Prec*(stack.peek())) {

stack.push(s);

}

else {

// in the end if the precedence of the element in the stack is greater than s, pop everything from the stack to the array until

// the stack is empty or s is greater precedence to whats on the stack

while (!stack.isEmpty() && *Prec*(stack.peek()) >= *Prec*(s)) {

result[count] = stack.pop();

count++;

}

//then push on the stack its self after that

stack.push(s);

}

}

}

// final house cleaning, if there is anything left in the stack pop it to the array

while(!stack.isEmpty()) {

result[count] = stack.pop();

}

// send the post fix array to the method call

return result;

}

## Post fix resolution code

public class Postfix

{

// Method to evaluate value of a postfix expression

static double evaluate(String[] exp)

{

//create a stack

Stack<Double> stack=new Stack<>();

// Scan all characters one by one

for(int i=0;i<exp.length;i++)

{

if (exp[i] != null) {

String s = exp[i];

double d = 0;

// If the scanned character is an operand (number here),

// push it to the stack.

if(*isNumeric*(s)) {

d = Double.*parseDouble*(s);

stack.push(d);

System.***out***.println(d);

}

// If the scanned character is an operator, pop two

// elements from stack apply the operator

else

{

double val1 = stack.pop();

double val2 = stack.pop();

switch(s)

{

case "+":

stack.push(val2+val1);

break;

case "\u2013":

stack.push(val2- val1);

break;

case "\u00F7":

try {

Verifier.*verifyNum*(val1);

stack.push(val2/val1);

}

catch(DivZero e) {

JOptionPane.*showMessageDialog*(null, "You can't divide by 0 stoopid head!)","Insulting Calculator", JOptionPane.***ERROR\_MESSAGE***);

}

break;

case "\*":

stack.push(val2\*val1);

break;

}

}

}

}

if (!stack.empty())

return stack.pop();

else

return 0;

}

public static boolean isNumeric(String str)

{

return str.matches("-?\\d+(\\.\\d+)?"); //match a number with optional '-' and decimal.

}

}