

Project Calculator

Theodora Tataru C00231174

Ana Griga C00231441

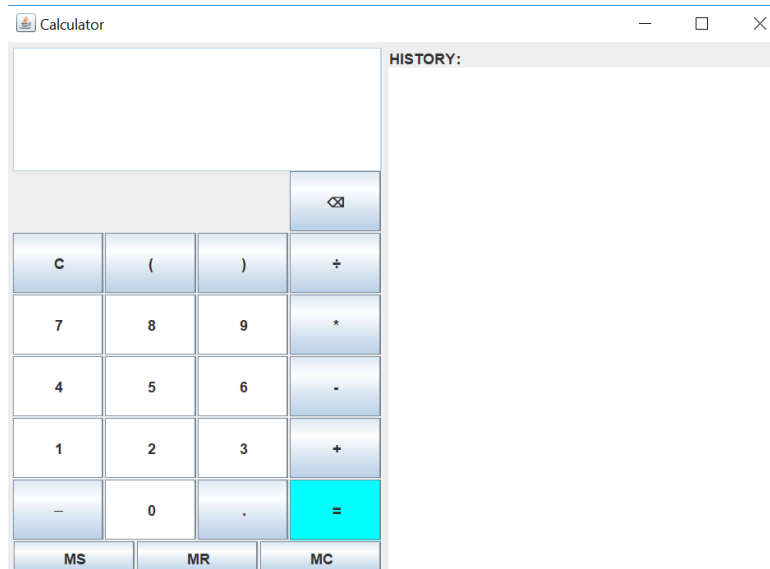
Software Development

Tutor: Dr. Jason Barron

Contents

1.INTRODUCTION	4
2.REQUIREMENTS.....	4
3.GRAPHICAL USER INTERFACE.....	5
4.FLOWCHARTS LOGIC	11
4.FUNCTIONALITY	12
5.CODE SNIPPETS.....	12
6.TESTING.....	16
7.ERROR LOG AND IMPROVEMENTS	18
8.ERROR HANDLING CLASS	20
9.CONCLUSION.....	21

Calculator Project Report



1.INTRODUCTION

Dr. Jason Barrow and Ms. Aine Byrne from IT Carlow requested to develop a calculator application in Java using the Graphical User Interface (GUI) components and all the knowledge gained over the past two years.

We have created the application based on our understanding of the object oriented concepts of inheritance, polymorphism, aggregation and we used the algorithms studied in our Discrete Structures classes.

The Calculator application performs basic mathematical calculations and is presented to the user in a friendly and easy to use way and provides appropriate messages when wrong input is given by the user.

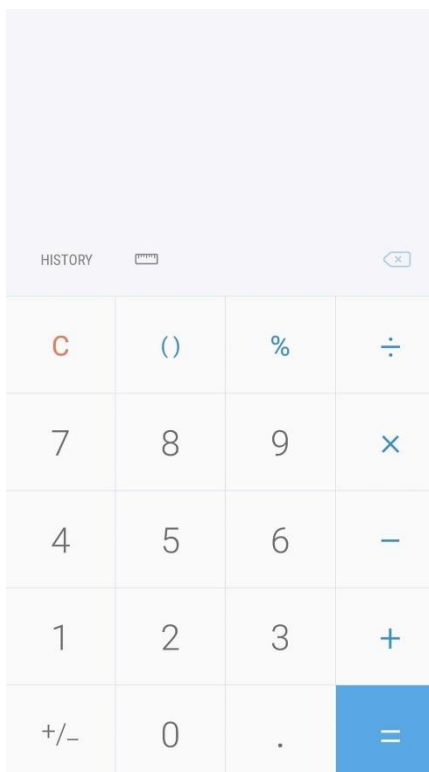
2.REQUIREMENTS

The application requests input from the user, process the input received, provides desired output by performing the following actions:

- Addition
- Subtraction
- Multiplication
- Division
- Clear Function
- Backspace Function
- Memory functions
- History Function
- Exception Handling

3.GRAPHICAL USER INTERFACE

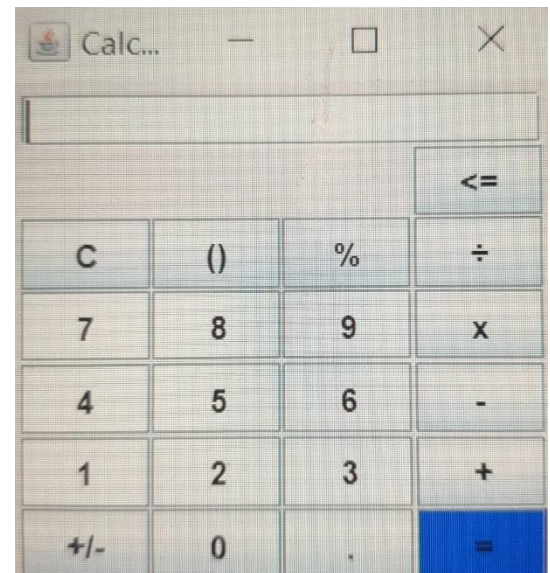
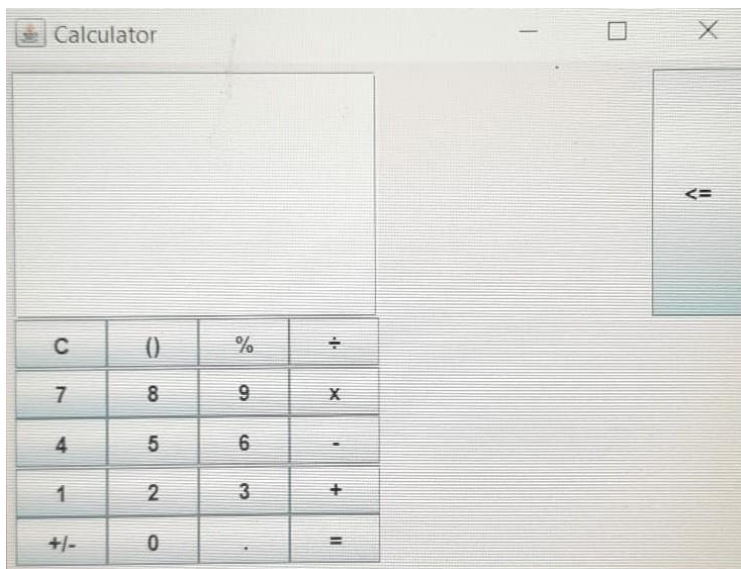
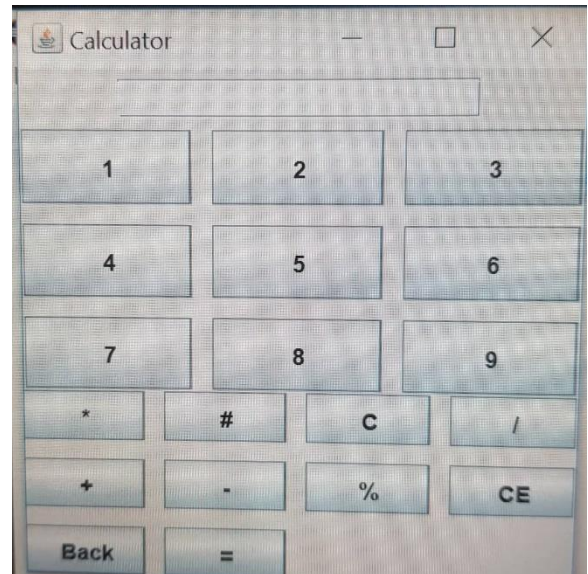
Inspiration Source



Our source of inspiration came after the Samsung Calculator showed below.

We tried to keep the same layout and colors but the memory buttons and the history panel are our own design.

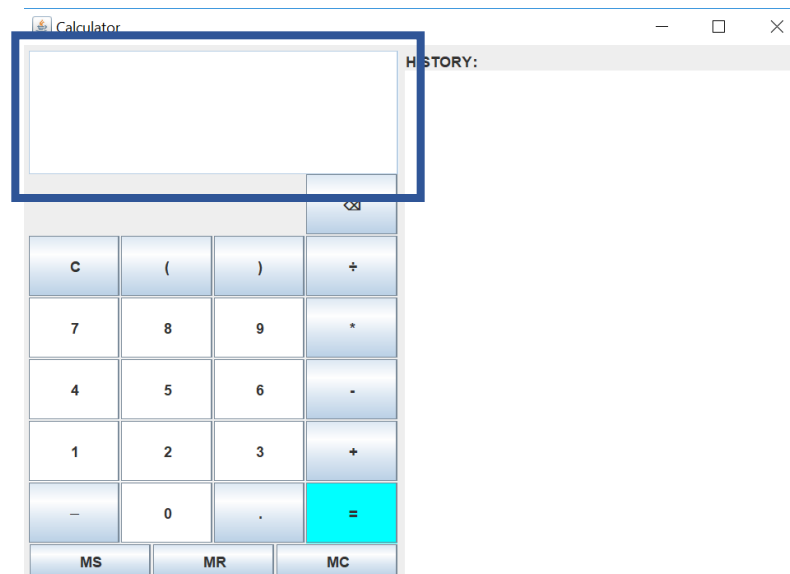
First we have created Calculator.java class where we have all the SWING GUI components and the Action Listeners. Since we have started we have played around with different ideas of how we wanted the Calculator to look like. Please see below some of the stages :



Our interface is made out of 7 panels:

- **TEXT FIELD**

A panel for the text field where the user can see his input and also the output. The text field is editable only by pressing the calculator's buttons.

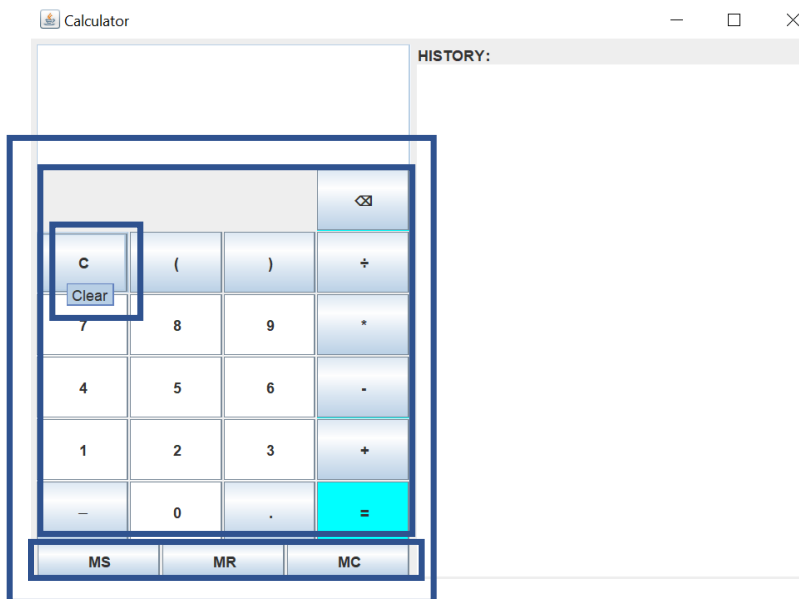


- ALL BUTTONS

There is an *allButtons* panel that is formed from two other panels: *mainButtons* panel and *memoryButtons* panel. The *mainButtons* panel contains all the operators, operands, parenthesis, clear button and backspace button. In front of the backspace button there are three empty panels used for design purpose, having the backspace button on the right of the first row.

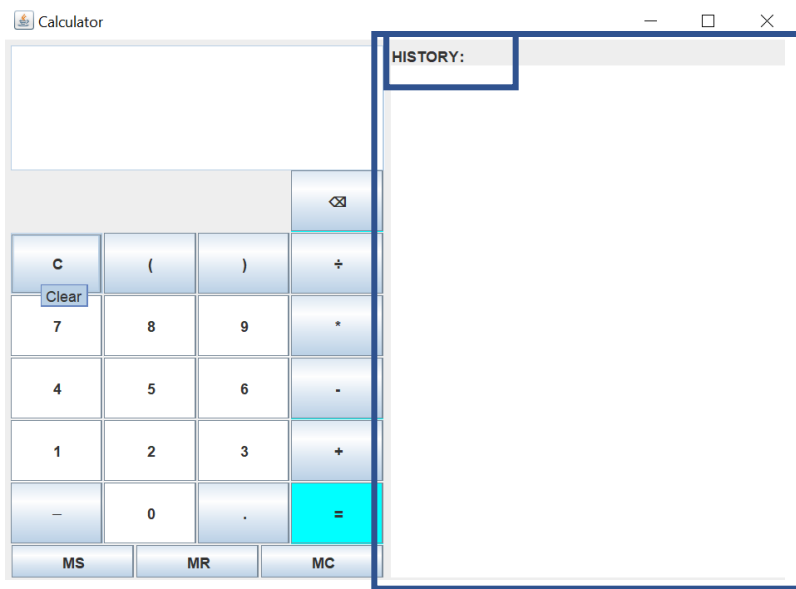
In the *memoryButtons* panel we added only the memory buttons. We took the decision to add them in a different panel for design purpose.

Hoovering with the mouse over the function buttons a help text will be visible for the user.



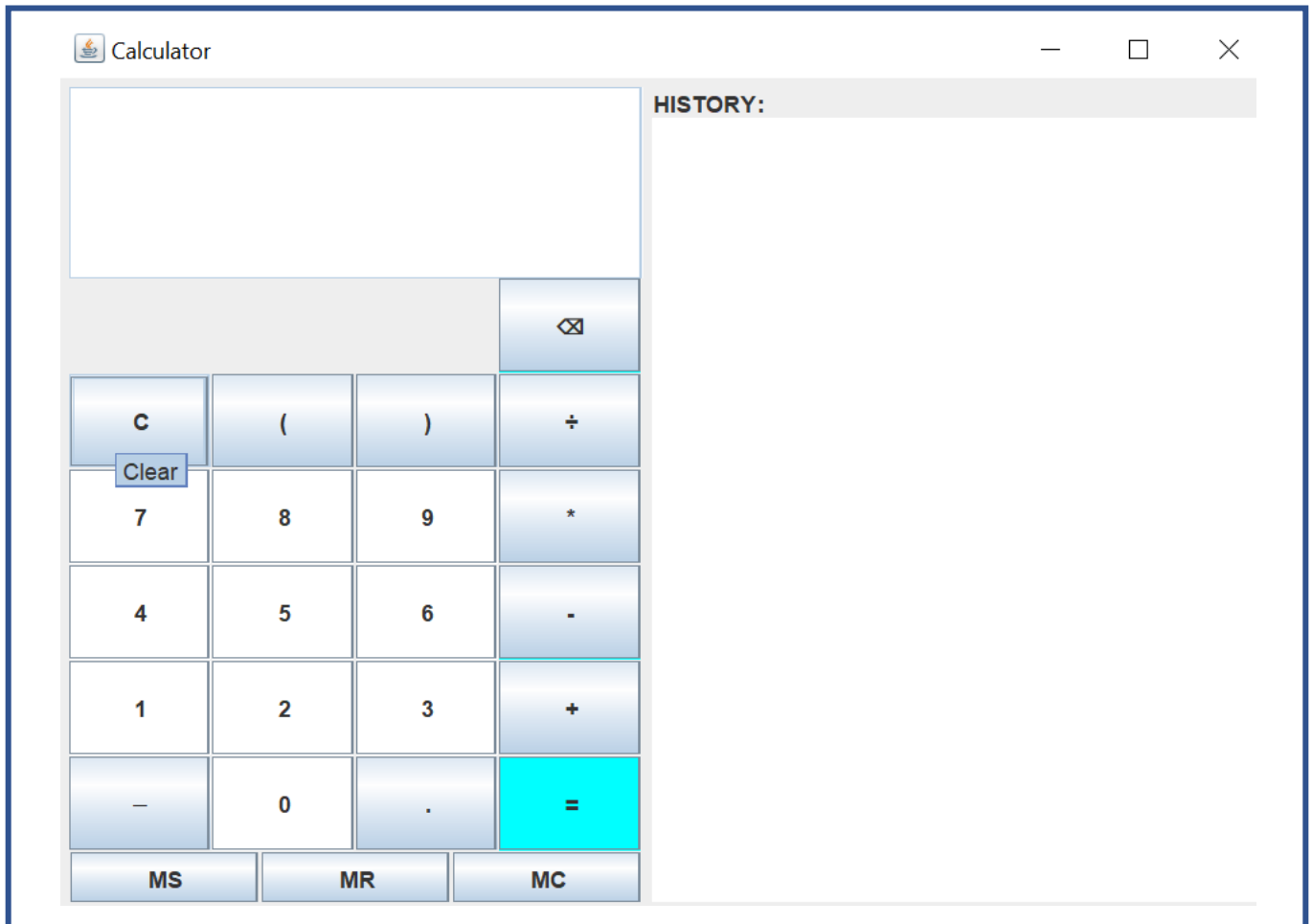
- HISTORY PANEL

The *history* panel contains the text area for the history which is not editable, it is read-only. We also added at the top of the panel a label called HISTORY.

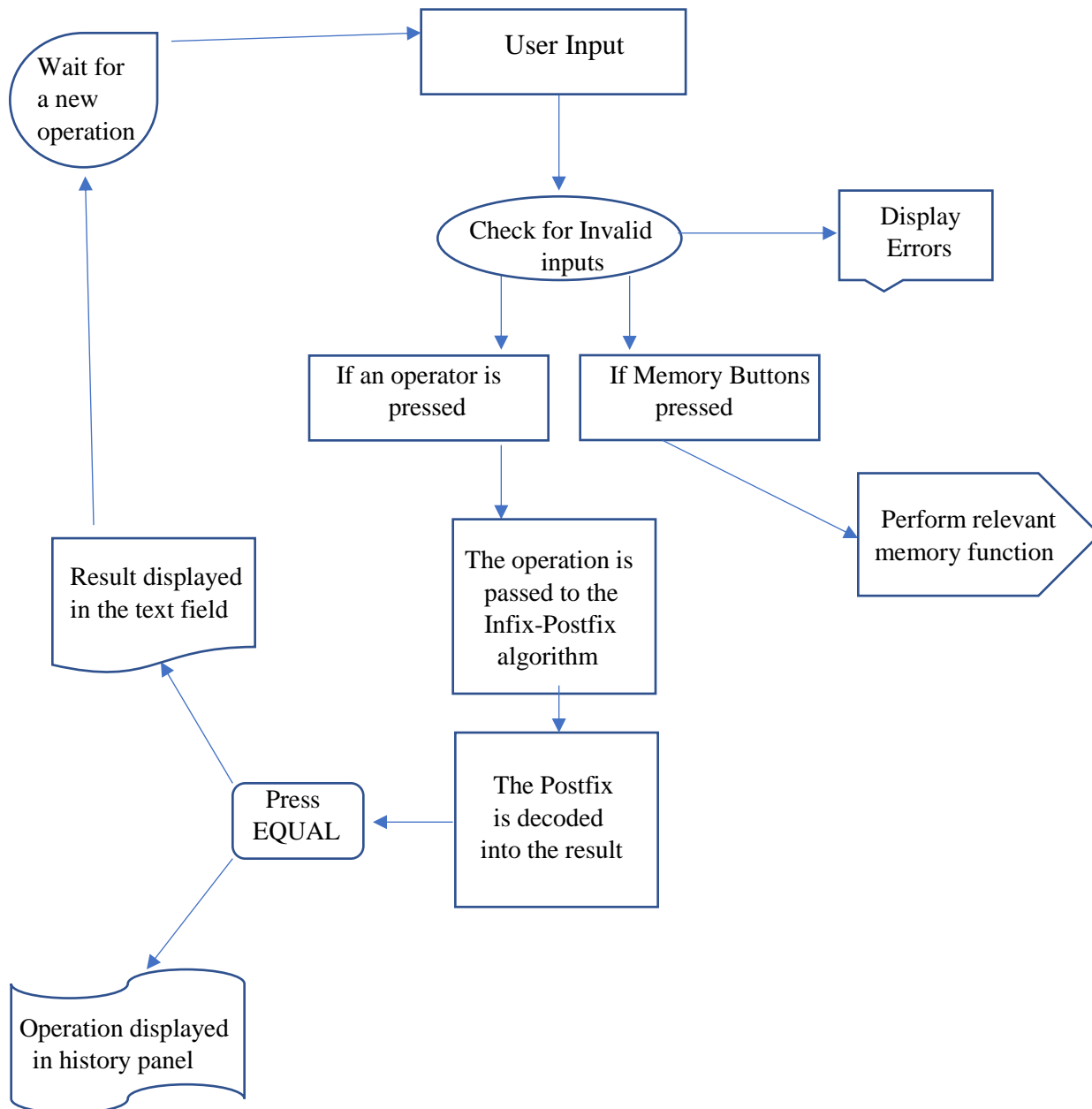


- BIG PANEL

The *allButtons* panel and the *history* panel are added together with a flow layout and placed in the main frame.



4.FLOWCHARTS LOGIC



4.FUNCTIONALITY

A class called `ButtonFunction.java` was created in which is coded the functionality of the following buttons:

- Clear Button
- Backspace Button
- Operands Buttons
- Decimal Point Button
- Negative Sign Button
- Operators Buttons
- Memory Store Button
- Memory Recall Button
- Memory Clear Button
- Equal Button

5.CODE SNIPPETS

The calculator is using the reverse polish notation: the infix and the postfix to resolve mathematical computations.

Using this two algorithms the user is allowed to use brackets, multiplication, division, addition, subtraction, operations with negative numbers and decimal points.

The main data structure used for this algorithms is the *Stack()*.

When we designed the calculator we have created 2 methods: *operandButton()* and *operatorButton()* and the data structure *infixArray* to work together as a team.

We have created a string called *elements* that is part of the *operandButton()* method. As long as the user press a number, decimal point or negative sign, the pressed digit is appended at the end of the string. The moment the user presses an operator, the *operatorButton()* method will pass the string *elements* into the *infixArray*, as a string and the operand is inserted into the same array into the next position. The *elements* is then reset as an empty string and ready for a fresh operator or operand.

```

public static void operandButton()
{
    Calculator.elements += Calculator.buttonLabel;
    Calculator.infixArray[Calculator.infixArrayCount] = Calculator.elements;
    // the pressed DIGIT is added at the end (append) to the ELEMENTS string
    // the new String is added into an array
    // we do not increase the infixArrayCount, as on this position we might append another digit
} // end OPERAND BUTTON

```

```

public static void operatorButton()
{
    Calculator.infixArrayCount++; // Until an OPERATOR is pressed, the number(s) string is
                                // added to the previous index.
    Calculator.infixArray[Calculator.infixArrayCount] = Calculator.buttonLabel;

    // we add at that index the pressed OPERAND
    Calculator.elements = ""; // we clear the elements, to add new number or the new operand
    // and not be mixed with previous digits or operands
    Calculator.infixArrayCount++; // we increase the position of the next available index
} // end OPERATORS BUTTONS METHOD

```

The `negativeSign()` method is design to allow the negative sign symbol to be added only in front of a number. The negative sign cannot be used for subtraction.

```

public static void negativeSign()
{
    if(Calculator.elements.equals("")) // if the ELEMENTS is empty because NEGATIVE sign can be
    added just at the beginning of a number
    {
        Calculator.elements += "-";
        Calculator.infixArray[Calculator.infixArrayCount] = Calculator.elements;
    }
    else // if the NEGATIVE sign is added in the middle of the number, ERROR
    {
        // alert, because we cannot use the negative sign as a minus!
        JOptionPane.showMessageDialog(new JFrame(), "You can not use the NEGATIVE
sign for substraction!", "Calculation Error", JOptionPane.ERROR_MESSAGE);
        // clear the text field if the error pops up
        clear();
    }
} // end NEGATIVE SIGN

```


The *theNumberIsADouble()* method checks if the number is a double or an integer.

After pressing the equal button, we divide the result to modular 1 and if the remainder is 0 we display the result as an integer, otherwise the result is displayed as a double.

```
public static boolean theNumberIsADouble(String result)
{
    // by default we accept that the number is a double
    boolean answer = true;
    double theResult = Double.parseDouble(result);
    if(theResult%1==0) // if the remainder is 0, the number is an integer
        answer = false;
    return answer;
} // end THE NUMBER IS A DOUBLE
```

The *checkDecimalPoint()* method checks if the decimal point is placed in the right position (has at least one digit in front of it) and that there is only one decimal point per number.

```
public static void checkDecimalPoint()
{
    if(Calculator.infixArray[Calculator.infixArrayCount]!=null) // if the DECIMAL POINT IS NOT THE first CHARACTER
    {
        if(Calculator.infixArray[Calculator.infixArrayCount].contains("."))
        {
            // if there is already a decimal point in the number, ignore it
            String theText = Calculator.textField.getText();
            Calculator.textField.setText(theText.substring(0, theText.length()-1));
            // System.out.println("Decimal point ignored"); // TEST - CONSOLE READING
        }
        else // if there is no decimal point in the number, add it to the number
            operandButton();
    }
    else // if the DECIMAL point starts the number, we ignore it
    {
        String theText = Calculator.textField.getText();
        Calculator.textField.setText(theText.substring(0, theText.length()-1));
        System.out.println("Decimal point ignored"); // TEST - CONSOLE READING
    }
} // end CHECK DECIMAL POINT
```


The `memoryStore()` method allows to store a number in the calculator memory as long as is a valid number and as long as there is no other number already saved.

public static void memoryStore()

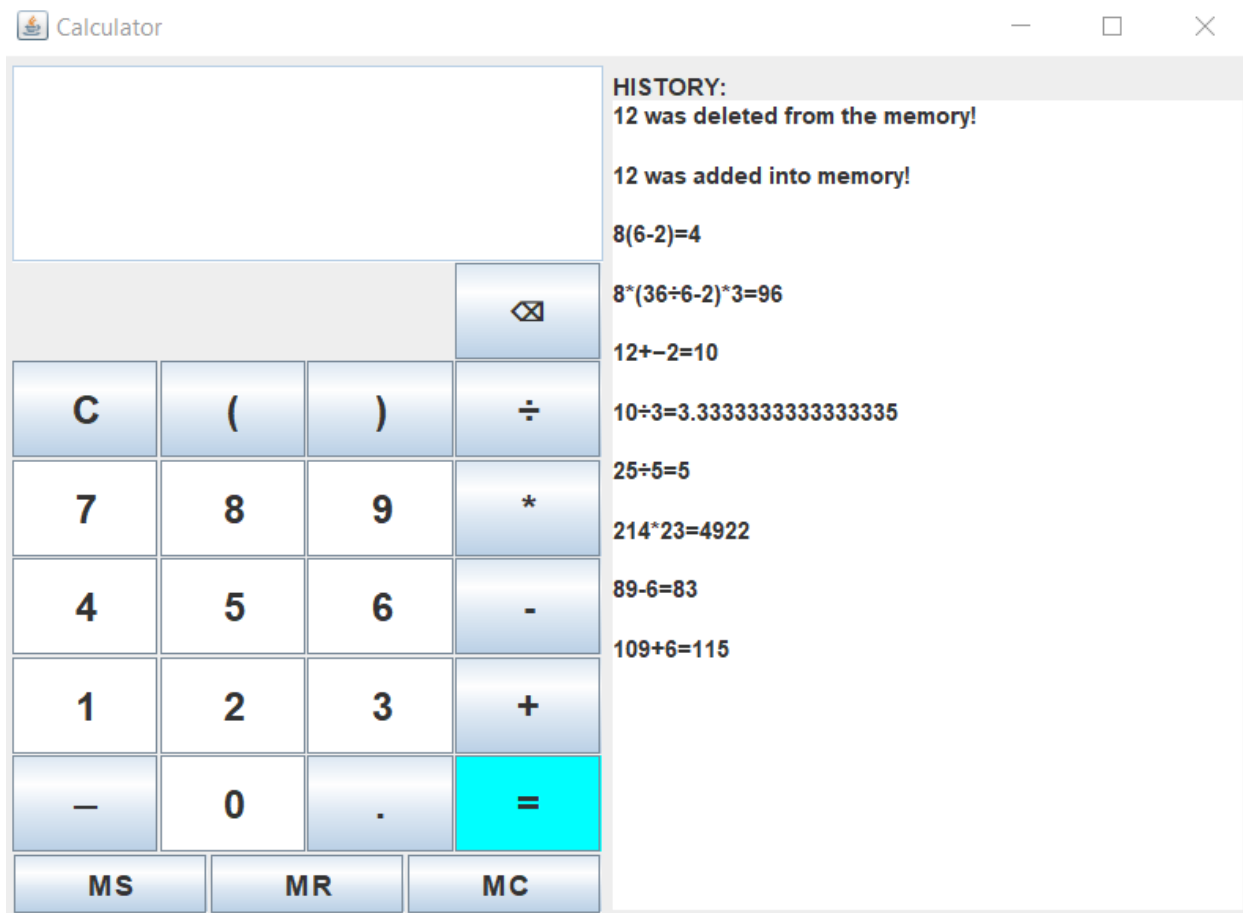
```
// Deleting from the TEXTFIELD the label of the button MS
String theText = Calculator.textField.getText();
Calculator.textField.setText(theText.substring(0, theText.length() - 2));

//System.out.println(theText.substring(0, theText.length() - 2));// test only - CONSOLE READING
If (Calculator.infixArrayCount==0 && Calculator.MS.length()==0 &&
Calculator.infixArray[Calculator.infixArrayCount]!=null)
    { // if the number is valid to be added into the memory (not followed by an operand or a null index)
        // if there is no element saved into MS, the element can be saved into it
        Calculator.MS = Calculator.infixArray[Calculator.infixArrayCount];
        // add the operation into history text area
        System.out.println("The " + Calculator.MS + " was added into memory!"); // test - CONSOLE
        Calculator.history.setText(Calculator.MS + " was added into memory!" + "\n\n" +
Calculator.history.getText());
    }
else if(Calculator.infixArrayCount==0 && Calculator.MS.length()>0)
    { // if there is already an element saved into MEMORY, give the user an error
        JOptionPane.showMessageDialog(new JFrame(), "Clear the memory first!", "Error",
JOptionPane.ERROR_MESSAGE);
        // Display error in history
        Calculator.history.setText(Calculator.MS + " is already in the memory!" + "\n\n" +
Calculator.history.getText());
    }
else
    { // if the element is not valid for the memory (if it's not a number)
        JOptionPane.showMessageDialog(new JFrame(), "Please insert a valid number!", "Error",
JOptionPane.ERROR_MESSAGE);
        // Display error in history
        Calculator.history.setText(Calculator.textField.getText() + " is not a valid number to be added in
memory!" + "\n\n" + Calculator.history.getText());
    }
}
// end MEMORY STORE
```

6.TESTING

Test No	Test Date	Test Scenario	Input	Expected Output	Actual Output	Status
01	03.01.2019	User clicks a number button	7	7	7	Success
02	03.01.2019	Using the Clear button	C	Clear the screen	The screen is cleared	Success
04	07.01.2019	Addition	109+6	115	115	Success
06	07.01.2019	Subtraction	89-6	83	83	Success
07	07.01.2019	Multiplication	214*23	4922	4922	Success
08	07.01.2019	Division	25/5 10/3	5 3.3333333333333335	5 3.3333333333333335	Success Success
09	08.01.2019	Using the Backspace button		Deleting one character at the time	Characters deleted one by one	Success
10	08.01.2019	Addition with negative numbers	12+-2	10	10	Success
11	08.01.2019	Mathematical Expressions (using parenthesis)	8*(36/6-2)*3	96	96	Success
12	08.01.2019	Mathematical Expressions	8(6-2)	32	4	Failed
13	11.01.2019	Memory Store button	Saving 12 into MS	12 was saved Into memory	12 was saved Into memory	Success
14	11.01.2019	Recall Memory	Display no. 12 from memory	12	12	Success
15	11.01.2019	Clear Memory	Clear 12 form the memory	12 was deleted from the memory	12 was deleted from the memory	Success

Calculator screenshot according with the testing table.

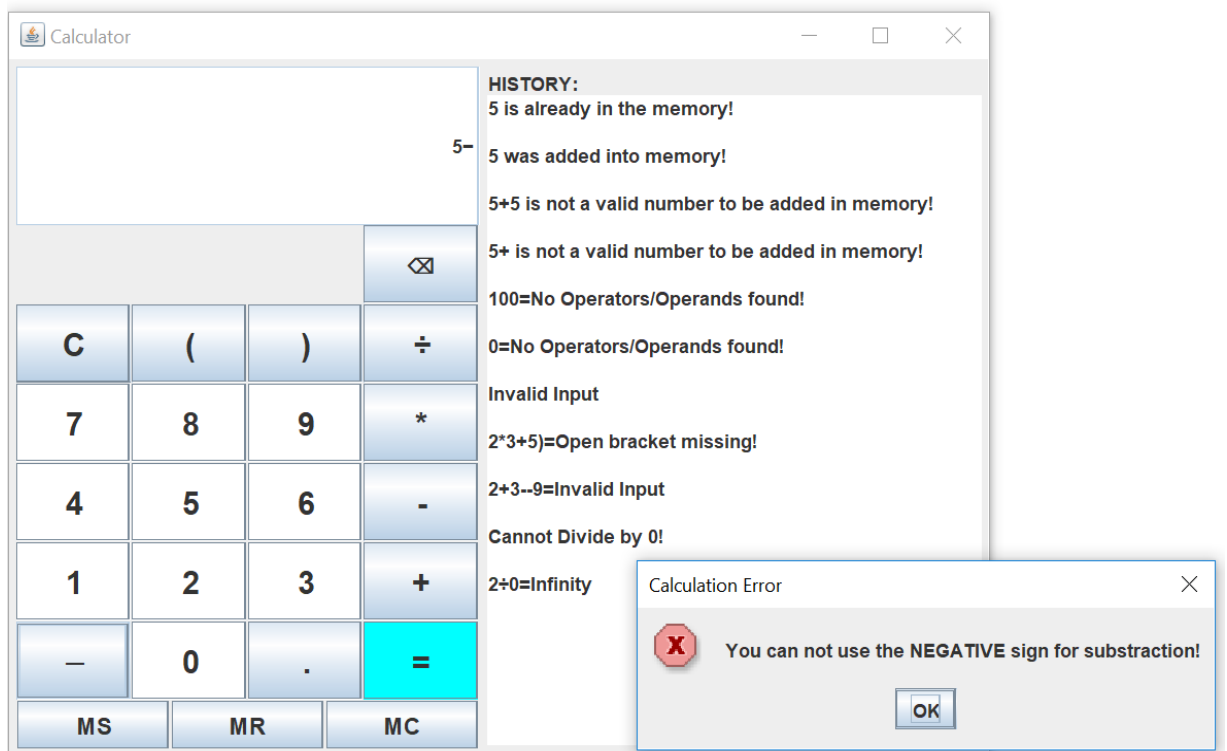


7.ERROR LOG AND IMPROVEMENTS

During the time of development of the Calculator we encounter many challenges.

Most of the errors were corrected:

- Invalid Input
- Multiple Decimal points or in the wrong position
- Deleting from the array when the backspace was pressed
- Division by zero
- Starting the mathematical expressions with an operator
- Pressing equal with no expression to be solved
- Overwriting in the memory store



Many of these errors were solved very elegantly but others could have been improved. For example one of the most important errors to be handled is when the user presses two operators one after the other (eg. $12 + *$) our Calculator will display an error informing the user about the invalid input. A better way, more elegant would be to replace the first operator with the following one assuming that the user made a mistake.

We could have also added more functions such as square root, power, modular, etc but for the time we had available we could not come up with a proper postfix to deal with this kind of calculations.

8.ERROR HANDLING CLASS

Our Calculator has one class for handling errors `CalculatorExceptions` which deals with the division by zero.

```
public class CalculatorExceptions extends Exception
{
    public CalculatorExceptions()
    {
        super();
        // clear everything!
        ButtonFunction.clear();
        // display the error to the history area
        Calculator.history.setText(Calculator.textField.getText() + "Cannot Divide by 0!" + "\n\n" +
        Calculator.history.getText());
        // set the result as null
        Calculator.infixArray[Calculator.infixArrayCount]=null;
        // display the error to the user
        JOptionPane.showMessageDialog(new JFrame(), "The denominator cannot be 0!", "Calculation
        Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

During the development there were more errors to be handled but there are all solved in the methods body.

9.CONCLUSION

We started this project thinking that developing the Calculator will be an easy process but every step was a challenge which taught us important lessons. This was our first project that showed us how programming is used in the real world and gave us the opportunity to work with different data structures and to use the algorithms studied this year.

Developing this project made us feel more confident on our skills in programming and algorithms.