Table of Contents:

1. Logic Part of Calculator.

1. **Algorithm used.**
2. **Implemented functions.**
3. **Short example.**

Algorithm used.

Before we dive into the magic of library that can calculate expressions with brackets and different operations let's discuss about one problem we had while developing it. The biggest problem with expressions with brackets is that we need to know in what order we should execute different operations and what operations have higher priority than others for ex. Multiplication has higher priority compared to addition however it losses effect if the addition is in brackets before multiplication. In order to solve this problem without using some parsers that will make our brain melt we found a nice algorithm. The algorithm is called Reverse Polish notation or postfix notation. And what it does basically is transforming an expression with brackets to one without.

For example:

(A + B) * (C + D)

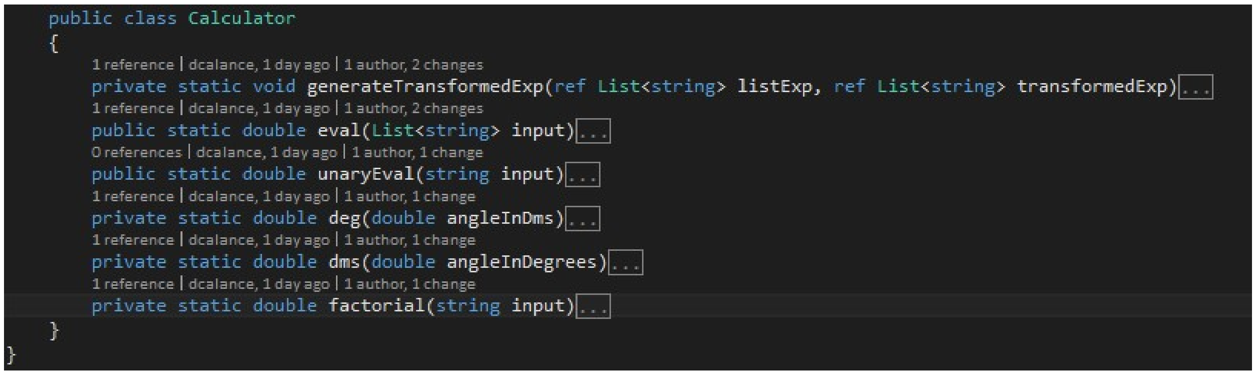
In postfix notation will look like this:

A B + C D + *

In this expression we can see clearly in what order we can execute operations, without even knowing how it works we can observe that addition between A and B comes first, then we take next 2 operands near the next operator what are C and D and add them, after that we obtain operand that is result of A+B and operand C+D that are followed by multiplication. This algorithm evaluates expressions very fast and it is easy to implement.

Implemented functions.

In order to divide our logic of calculator and interface we created a separate class that we can use for evaluation of expressions and unary operators or functions. Let's take a look at our class:



```
    public class Calculator
    {
        1 reference | dcalance, 1 day ago | 1 author, 2 changes
        private static void generateTransformedExp(ref List<string> listExp, ref List<string> transformedExp)[...]
        1 reference | dcalance, 1 day ago | 1 author, 2 changes
        public static double eval(List<string> input)[...]
        0 references | dcalance, 1 day ago | 1 author, 1 change
        public static double unaryEval(string input)[...]
        1 reference | dcalance, 1 day ago | 1 author, 1 change
        private static double deg(double angleInDms)[...]
        1 reference | dcalance, 1 day ago | 1 author, 1 change
        private static double dms(double angleInDegrees)[...]
        1 reference | dcalance, 1 day ago | 1 author, 1 change
        private static double factorial(string input)[...]
    }
}
```

The class contains 2 functions accessible by user:

- **eval**
- **unaryEval**

eval takes as parameter a List of strings that is the input, in each element we have either operator, operand or bracket. We used list because it is very easy to process and we don't have to handle problems that will occur with processing a string. It supports the following operations:

- **Addition (+)**
- **Subtraction (-)**
- **Multiplication (*)**
- **Division (/)**
- **Modulo (%)**
- **Power (^)**
- **Exp (Exp)**
- **Root of order (y√x)**

unaryEval takes as parameter an input string of the form **unaryOp:number**. It has implementation for operations:

- **Factorial (fact)**
- **E to power (exp)**
- **X to cube (cube)**
- **1/X (invert)**
- **Natural log (ln)**
- **Asin (asin)**
- **Acos (acos)**
- **Atan (atan)**
- **Transform to degree, minutes, seconds (dms)**
- **Transform to degree decimal (dec)**
- **Sin (sin)**

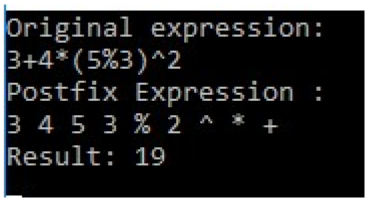
- **Cos (cos)**
- **Square (sqr)**
- **Square root (sqrt)**
- **Tan (tan)**
- **Ten to power (10to)**
- **Base 10 log (lg)**

Short example.

An example of function input would be **fact:3.14** (yes, factorial of floating point number, special for this option I implemented Stirling's approximation formula to approximate factorial using a function).

The functions **dms, deg, factorial** are implementation of those operations for function **unaryEval**.

Function **generateTransformedExp** is called by **eval** and in it is implemented the algorithm for transforming the input expression with brackets into postfix expression. The nice fact about it is that we can set custom priorities to operations, meaning we can add new operators and set their priorities and it will generate the correct postfix expression. Since our class can be called by all kind of interfaces let's take a look at an example in console:

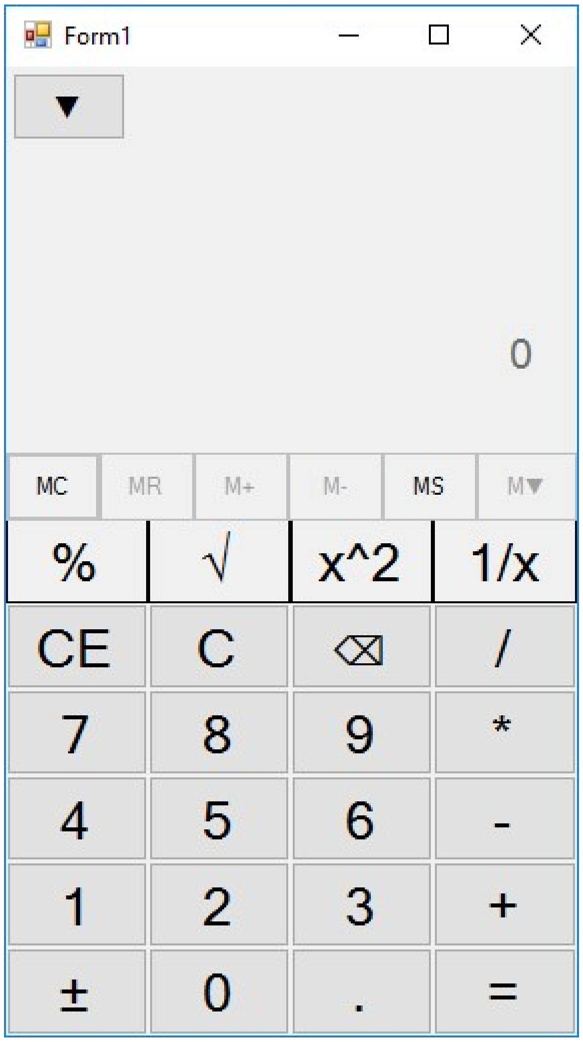


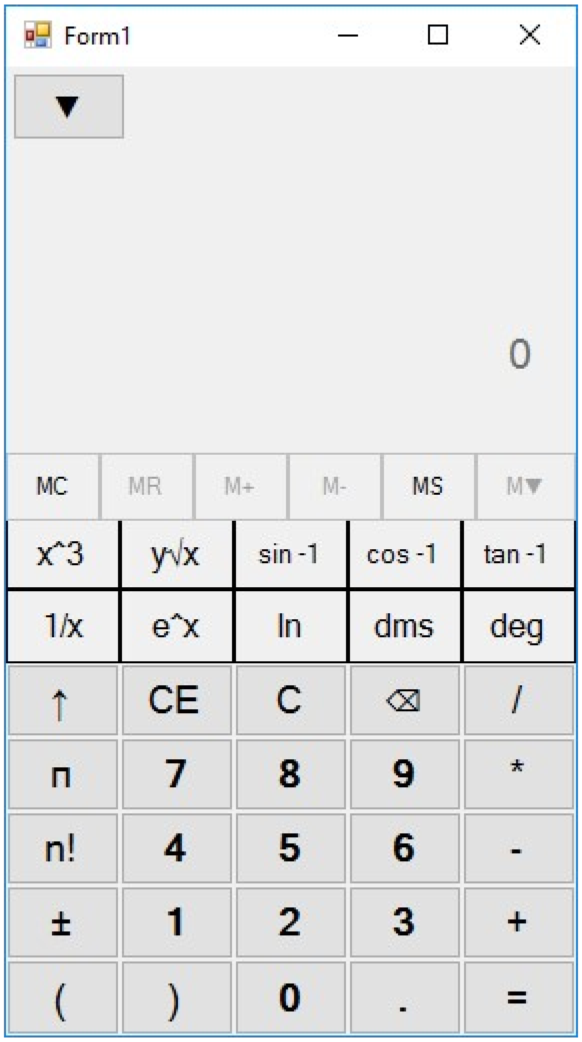
```
Original expression:
3+4*(5%3)^2
Postfix Expression :
3 4 5 3 % 2 ^ * +
Result: 19
```

2. Calculator interface and functionality

1. **Screens of final product.**
2. **Short about IDE and framework.**
3. **Structure of program.**
4. **Modes of operations.**
5. **Logic behind input.**
6. **Logic behind buttons.**

Screens:





Short about IDE and framework.

The GUI of calculator is implemented in Windows Forms using IDE Visual Studio 2015 in C#. The windows forms framework allows us to create interface from **Design Mode** and add elements to it and also we can write them manually if we want something specific. There are a lot of elements we can use in our windows form application like buttons, panels, textLabel and a lot other that have specific proprieties we can set.

Structure of program.

Our program is divided in 2 parts:

- **Elements generation and setting.**
- **Handling functions of controls subscribed to events.**

If we look in our project, we see those files as separate however if we look more closely we see the class as `public partial class Form1 : Form` and on other side it's the same. This means that we divided only logically this function and at compile time there is only one function.

Modes of operation.

We have 2 modes of operation of this calculator:

- **Classic mode.**
- **Scientific mode.**

The logic behind both calculators it's the same the only difference is the number of available operations.

Logic behind input.

We have 2 elements that take care of input:

- String that is the current input (We can see its value down)
- List that records all the operations and numbers we input.

The string is responsive to the current input we make, when we add numbers, change sign, add point. Once we press one of the binary operators or equal we input the string in the List along with the operator. One exception is when we use unary operator, in that case we call the corresponding function for evaluation and the string will be updated with the result, what we can add to the list without any problems.

The list has elements of string type and consists of all inputs, each in separate element. The list is always adding elements when binary operators are pressed or brackets, and once you hit equal button the expression from list is evaluated and the list becomes empty, result is transferred to input string.

Logic behind buttons.

The buttons are the most important thing in this construction. Since our application is event-driven we have every button subscribed to certain events and we handle them. A typical event handle function looks like this:

```
private void scientificPanel_Paint(object sender, PaintEventArgs e)
        {
        }
```

Where sender is the control that sends message and e is the variable of event.

We didn't create new event handler for every button because it would be a waste of time and a lot of useless code since we can use easily buttons that have same property together, such buttons are grouped as:

- **Digits.**
- **Binary operators.**

All other buttons are handled separately. We could make all unary operators be handled together, maybe another time.

3. Graph plotting.

1. **Library used.**
2. **Generation of points.**
3. **Functionality.**

Library used.

To plot a graph, we had to use some tools already implemented to make our work easier. We used the library **OXYPlot** that is available on Windows Forms as well as other platforms. The library is open source and has a lot of tools, the only problem is the documentation but if you study how it works you will have a very powerful tool for making and manipulating graphs.

Generation of points.

In order to make a graph we need input data. In this library we can input the data in two ways:

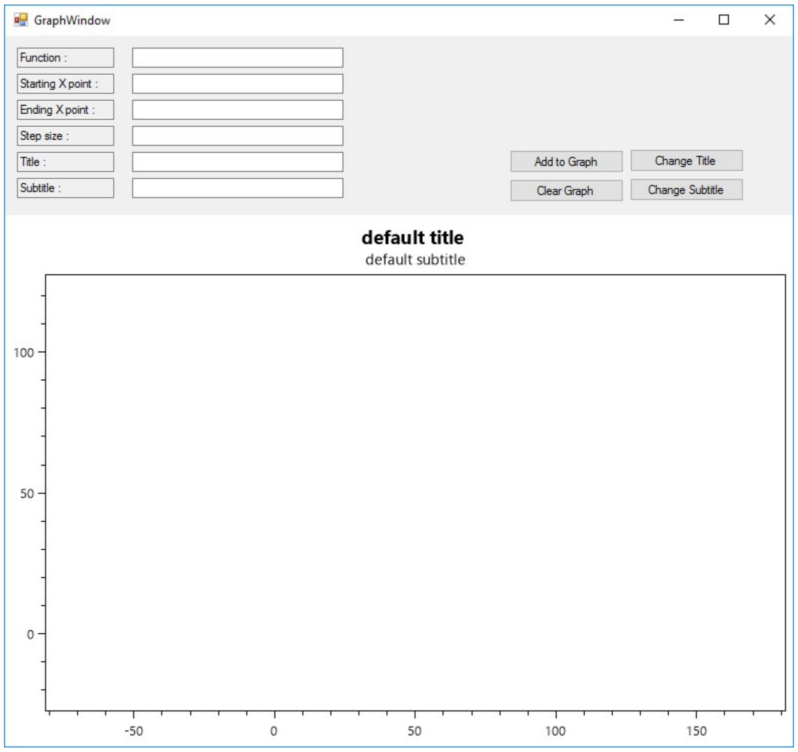
- **Inputting coordinates manually.**
- **Inputting a lambda as function.**

The second option is very nice however generating lambda expression from user input is not very nice. The complexity of making a parser that will generate a lambda function from input string is bigger than that of making this calculator work, that is why we use first option. Basically we use our calculator library that we implemented to generate points from an interval with a certain step.

We use the calculator library in 2 steps: first we evaluate the unary expressions and transform them into numbers then we evaluate all expression, and this is happening for every value in that interval. Our tool for parsing string is not very powerful that is why the input should respect:

- Spaces between every operand and operator (except unary operators)
- In case of unary functions the value should be inside brackets without any spaces (expressions inside not allowed, only constant or variable).
- Only 1 variable. (The library supports plotting of functions of multiple variables)
- Only functions and operators included in Calculator library.

Functionality.



We observe that we have 6 text fields:

- **Function input.**
- **Starting X point.**
- **Ending X point.**
- **Step size.**
- **Title.**
- **Subtitle.**

We need only first 4 elements to be able to plot a function, and also we can plot multiple graphs on same page. We also have options of changing **title**, **subtitle** and to **add to graph** and **clear graph**. This option can be selected from the interfaces of calculator button located on left-top of calculator window. Calculator window and graph are not dependent in work; the only dependency is that graph window is child of calculator window.

Conclusion:

This work is already something more appropriate to a functional application that can be used by users. Windows Forms is not a good framework for usage because is heavy platform dependent, all controls and default events are straight from WinApi , the windows api. A similar tool would be wpf forms, if you take a look most of the tools are the same however they are implemented to work independent, without WinApi. Even though Windows Forms is not a pretty good tool it is very good to learn the basics, because it is very easy to use and at the same time you have a lot of control and functionality.