**SODV 2202**

**Object Oriented Programming**

**Assignment 1**

**Calculator**

**Assignment 1: Calculator**

**Overview**

Design and implement a program to evaluate mathematical expressions typed by the user. For example, if the user typed:

5 + (((12 – 7) \* 10) / 2)

Your program would print the result:

30

**Directions**

Start by creating a UML diagram outlining your planned design for the assignment.

Your program should be able to:

1. Handle addition, subtraction, multiplication, and division.
2. Evaluate expressions with a single operation (ie: 1+2).
3. Evaluate larger expressions with multiple operations (ie: 1\*2-3+4).
4. Perform operations in the correct order of operations (ie 2+3\*4 is 14 not20).
5. Ignore whitespace in the expression.
6. Handle negative numbers.
7. Handle numbers with a decimal.
8. Handle brackets (ie: (2+3)\*4 is 20).
9. Handle nested brackets.

When you are finished, update your UML diagram to reflect your final program.

**Evaluation**

This assignment will be graded for both design and functionality.

The design component will be evaluated as follows:

|  |  |
| --- | --- |
| **Design Task** | **Marks** |
| **Diagram and design**   * Does the chosen design match the given problem? * Is the design scalable? * Does the UML diagram include all relevant components? | 20 |
| **Code quality**   * Is the code clear and easy to read? * Is the style consistent? * Are comments used when necessary? | 20 |
| **Total** | 40 |

Design tasks are worth 20 marks each, but will be assigned a grade of 20, 15, 10, 5, or 0.

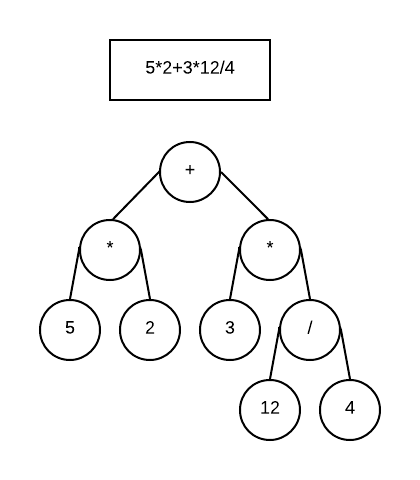
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *20 - Excellent* | *15 - Good* | *10 - Fair* | *5 - Poor* |
| *Diagram and design* | Chosen design matches the problem and is scalable. UML includes all relevant elements. | Either design could use some minor adjustments or UML is lacking some elements. | Design does not keep scalability in mind. Any new features would be difficult to add. | Little effort put into planning the project. |
| *Code quality* | Code is clear and easy to read. Style is consistent throughout and comments are used where needed. | Code is mostly clear. A few inconsistencies in style, or locations where comments would help with readability. | Code could use some refactoring. Style is inconsistent with little to no documentation. | Code is very difficult to read. |

The UML diagram is primarily for helping communicate your program’s design. It does not necessarily need to list every single element of your code, but should include enough to show how your program is organized.

**Hint**

The first task is to parse the input string and identify the components of the expression that the user typed. In C#, you can access an individual character in a string using the [] operator like you would access an element in an array. There is also a handy standard function called Char.IsDigit() to check if a given character is a number.

There are 2 common approaches to evaluating a math expression. One is to build an expression tree, which is a data structure that makes evaluating the expression extremely easy once the tree is built. Each node in the expression tree represents either an operator or a value. An example of an expression tree can be seen below:



Another approach is to first convert the expression to reverse polish notation. We normally write math using infix operators, like in the case of “3 + 4”. Reverse polish notation uses post-fix operators, so “3 + 4” would be written “3 4 +”. Writing it this way makes it easy use a stack to evaluate the expression.