Arithmetic Expression Trees (Part 1)
Cpt S 321 Homework Assignment
Washington State University

Submission Instructions:

- Create a branch called "Branch_HW5" and work in this branch for this assignment.
- **When you are done, merge the branch back to the master.**
- Once you are done and before the deadline, tag the version that you would want us to grade with the assignment number (for example, "HW5").
- On Canvas -> Assignments -> Submit the link to your repository (a link to the tag or the branch works) by the HW deadline.
- **IMPORTANT: The HW must be tagged by the due date and a link to that tag needs to be submitted via Canvas in order to receive a grade.**

Assignment Instructions:

Read each step's instructions *carefully* before you write any code.

In this assignment you will create the ExpressionTree class in your logic engine DLL. You will create a standalone console application as well, to demo its functionality. The ExpressionTree class will implement an arithmetic expression parser that builds a tree for the expression. This tree can then be used for evaluation of the expression. The parsing aspect of this assignment is simplified and you will extend the parser in the next homework. But the entire evaluation functionality, including setting variable values, will be implemented in this assignment.

Create a class named **ExpressionTree** in the **CptS321** namespace. (Do not create a class named expTree, EXPTREE, etc. Make a class name that is an EXACT match including casing. Part of this assignment is likely to be tested automatically and it will not give you credit if it cannot find the class.) Create this class in your logic engine DLL. The console app for this assignment just references your engine DLL and provides a simple menu for testing. The console application, the logic engine DLL, and your test project are to be submitted for grading. Include the following functions in the ExpressionTree class, again with EXACT signatures matching:

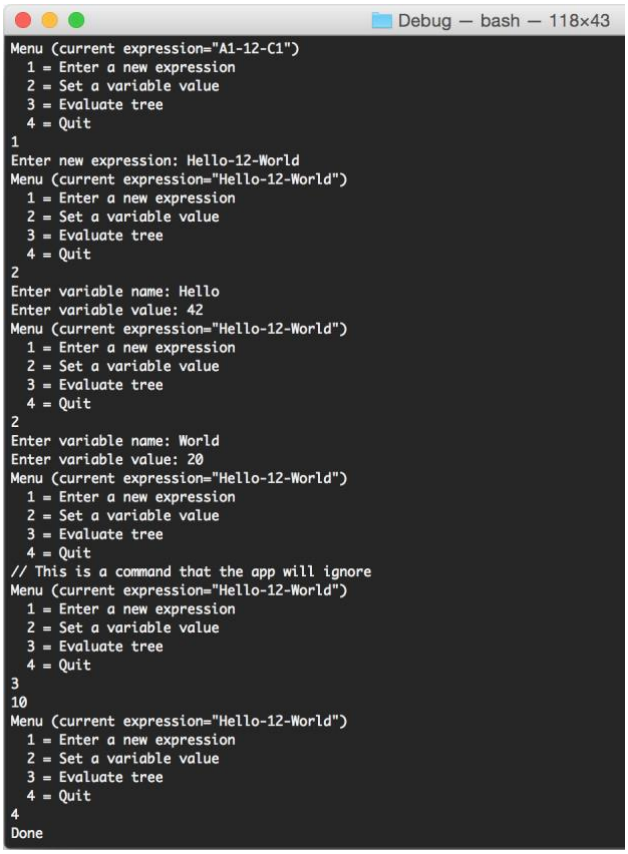
- public ExpressionTree(string expression)
    - Implement this constructor to construct the tree from the specific expression
- public void SetVariable(string variableName, double variableValue)
    - Sets the specified variable within the ExpressionTree variables dictionary

- public double Evaluate()
    - Implement this method with no parameters that evaluates the expression to a double value

Create a console application that presents a menu when run. As another reminder, this app references your engine DLL. This menu must contain the following options:

1. The option to enter an expression string. You may assume that only valid expressions will be entered with no whitespaces. Simplified expressions are used for this assignment and the assumptions you are allowed to make are discussed later on.
2. The option to set a variable value in the expression. This must prompt for both the variable name and then the variable value.
3. The option to evaluate to the expression to a numerical value.
4. The option to quit.
5. Should the user enter an "option" that isn't one of these 4, simply ignore it. As trivial as this may seem it is vital should the assignment be tested with an automated grading app.
    - On that note, also avoid use of Console.ReadKey() as it can be problematic when grading with an automated app. Simply fall through to the end of main after the quit option is selected.

The screenshot below shows what this might look like:



Requirement Details:

Support simplified expressions:

- For this assignment all expressions will be simplified next to what you'll have to support in the next homework. As stated earlier, this assignment is primarily about getting the correct evaluation logic in place for the tree.
- Assume expressions will NOT have any parentheses
- Assume expressions will only have a single *type* of operator, but can have any number of instances of that operator in the expression
    - Example 1: expression could be "A+B+C1+Hello+6"
    - Example 2: expression could be "C2-9-B2-27"
    - Example 3: expression could NOT be "X+Y-Z" because that has two different types of operators: + and –
- (In the next assignment you'll have to support all valid arithmetic expressions)

- Support operators +,-,*, and / for addition, subtraction, multiplication, and division, respectively. Again, only one type will be present in an expression that the user enters.
- Parse the expression that the user enters and build the appropriate tree in memory

Tree Construction and Evaluation:

- Build the expression tree correctly internally. Non-expression-tree-based implementations will not be worth any points.
- Each node in the tree will be in one of three categories:
    - Node representing a constant numerical value
    - Node representing a variable
    - Node representing a binary operator
- The above three types should match nicely with a design of a base class for the node and then 3 inheriting classes.
    - No node should store any more data than it needs to. As just one example of what this means, the constant value and variable nodes never have children so their class declarations shouldn't have references to children (nor should they be inheriting such declarations from a parent class).
- Your ExpressionTree class must have a public evaluation function that takes no parameters and returns the value of the expression as a double.
    - public double Evaluate()

Support for Variables:

- Support correct functionality of variables including multi-character values (like "A2").
- Variables will start with an alphabet character, upper or lower-case, and be followed by any number of alphabet characters and numerical digits (0-9).
- A set of variables is stored per-expression, so creating a new expression will clear out the old set of variables.
- Have a default expression, something like "A1+B1+C1" would be fine, so that if setting variables is the first action that the user chooses then you have an expression object to work with.
- If variables are not set by the user, they can be default to 0 for this HW. In later HWs, once we learn how to deal with exceptions, we will change this.

Clean, REUSABLE code:

- Do not have things like the variable dictionary declared as a local variable in Main(). It must be a member of the ExpressionTree class. Such implementations would not allow you to easily bring your code into the WinForms app for the later homework assignment.
- Follow the principles discussed in class!
- Have clean, well-documented code and an easy to use interface for grading.

Point breakdown (the assignment is worth 10 points):

- 5 points for implementing the correct functionality

And as usual:

- 1 point: For a "healthy" version control history, i.e., 1) the HW assignment should be built iteratively, 2) every commit should be a cohesive functionality, 3) the commit message should concisely describe what is being committed, 4) you should follow TDD – i.e., write and commit tests first and then implement and commit the functionality.
- 1 point: Code is clean, efficient and well organized.
- 1 point: Quality of identifiers.
- 1 point: Existence and quality of comments.
- 1 point: Existence and quality of test cases.

General Homework Requirements	
Quality of Version Control	● Homework should be built iteratively (i.e., one feature at a time, not in one huge commit). ● Each commit should have cohesive functionality. ● Commit messages should concisely describe what is being committed. ● TDD should be used (i.e, write and commit tests first and then implement and commit functionality). ● Include "TDD" in all commit messages with tests that are written before the functionality is implemented. ● Use of a .gitignore. ● Commenting is done as the homework is built (i.e, there is commenting added in each commit, not done all at once at the end).
Quality of Code	● Each file should only contain one public class. ● Correct use of access modifiers. ● Classes are cohesive. ● Namespaces make sense. ● Code is easy to follow. ● StyleCop is installed and configured correctly for all projects in the solution and all warnings are resolved. If any warnings are suppressed, a good reason must be provided. ● Use of appropriate design patterns and software principles seen in class.
Quality of Identifiers	● No underscores in names of classes, attributes, and properties. ● No numbers in names of classes or tests. ● Identifiers should be descriptive. ● Project names should make sense.

	● Class names and method names use PascalCasing. ● Method arguments and local variables use camelCasing. ● No Linguistic Antipatterns or Lexicon Bad Smells.
Existence and Quality of Comments	● Every method, attribute, type, and test case has a comment block with a minimum of <summary>, <returns>, <param>, and <exception> filled in as applicable. ● All comment blocks use the format that is generated when typing "///" on the line above each entity. ● There is useful inline commenting in addition to comment blocks that explains how the algorithm is implemented.
Existence and Quality of Tests	● Normal, boundary, and overflow/error cases should be tested for each feature. ● Test cases should be modularized (i.e, you should have a separate test case for each thing you test - do not combine them into one large test case). ● *Note: In assignments with a GUI, we do not require testing of the GUI itself.*