Arithmetic Expression Trees (Part 2)
Cpt S 321 Homework Assignment
Washington State University

Submission Instructions:

- Create a branch called "Branch_HW6" and work in this branch for this assignment.
- **When you are done, merge the branch back to the master.**
- Tag the version that you would want us to correct with the assignment number. For example, "HW6". **IMPORTANT REMINDER: If a HW does not have a tag by the due date, it will NOT be graded and it will receive automatically 0. If you think you will be late with a HW do the following:**
    - **1) send an email to all TAs and cc the instructor BEFORE THE DUE DATE and tell us that you will be late**
    - **2) send us another email when you finish and tag the assignment so that we can grade**

Assignment Instructions:

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

In this assignment you will finish what you started in the previous homework assignment by implementing an arithmetic expression parser that builds a tree for an expression. This tree can then be used for evaluation of the expression. You may want to refer back to the instructions for the first half of this assignment to make sure that everything was properly implemented there and that you are following the required naming conventions.

Recall that your expression tree class is implemented in your engine DLL and demoed in this assignment through a standalone console application that references the DLL. Integration into the spreadsheet will happen in a future homework assignment.

Requirement Details:

Support parentheses and operators with proper precedence:

- Support the addition and subtraction operators: + and -
    - You may assume that all instances of the minus character are for subtraction. In other words, you don't have to support negation.
- Support the multiplication and division operators: * and /
- Each operator must be implemented with proper precedence.
- Parentheses must be supported and obeyed. Implementations without support for parentheses will result in a 50% deduction on all working operators.

Tree Construction:

- Build the expression tree correctly internally.
- You must use the Factory pattern seen in class.

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).
- As you build the tree, every time you encounter a new variable add it to the dictionary of variable values with a default value of 0. This will be needed for integration with the spreadsheet application in a future assignment.
- Variables are stored per-expression, so creating a new expression should clear out the previous set of variable values. However, all variable values must persist as long as the expression is NOT being changed.
    - Ex: You shouldn’t reset any variables after an evaluation or anything like that

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 <u>in addition to comment blocks</u> 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.*