**CS 220 – Data Abstraction**

**PEX 3 – Expression Calculator**

**PEX 3 Preliminary Submission Due: 2200, Lesson 27, Thursday, 22 March**

**PEX 3 Final Submission Due: 2200, Lesson 32, Thursday, 12 April**

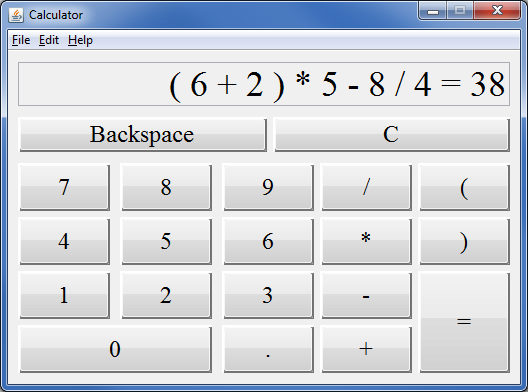
|  |
| --- |
| Help Policy: **AUTHORIZED RESOURCES:** Any, except another cadet’s program.  **NOTE:**   * Never copy another person’s work and submit it as your own. * Do not jointly create a program unless explicitly allowed. * You must document all help received from sources other than your instructor. * **DFCS will recommend a course grade of F for any cadet who egregiously violates this Help Policy or contributes to a violation by others.**  Documentation Policy:  * You must document all help received from any source other than your instructor. * The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance. * If no help was received on this assignment, the documentation statement must state “NONE.” * If you checked answers with anyone, you must document with whom on which problems. You must document whether or not you made any changes, and if you did make changes you must document the problems you changed and the reasons why.  Vague documentation statements must be corrected before the assignment will be graded, and will result in a 5% deduction on the assignment.Turn-in Policies:  * On-time turn-in is at 2200 on the due date, same day for both M and T day sections. * Late turn-ins will receive a 25% penalty per 24 hours late unless prior arrangements have been made with your instructor. * There is no early turn-in bonus or extra credit for this assignment. |

1. Objectives

* Be able to create and traverse a tree
* Be able to use dynamically allocated objects
* Be able to gracefully handle exceptions
* Be able to create and use a GUI
* Be able to create a thorough set of JUnit tests
* Be able to document code using Javadoc

1. Pair Programming

You will again use the [Pair Programming](http://en.wikipedia.org/wiki/Pair_programming) approach for this exercise. Both students must work together ***on all parts*** of the solution and will receive the same grade. You must inform your instructor of your partner selection by Lesson 24. *You cannot work with the same partner as PEX 2*.

1. Background

Your task for this exercise is to create a calculator similar to the Windows calculator that allows the user to create and evaluate expressions with numeric values; addition, subtraction, multiplication, and division operations; and parentheses to specify order of operations. For example, the user should be able to enter the expression, “( 6 + 2 ) \* 5 - 8 / 4”, and have it evaluate to 38.

Your program will evaluate the expression by converting the infix expression created by the user into prefix notation, then creating an expression tree from the prefix notation, and finally evaluating the expression tree. Note: This is ***three separate steps***!

[Prefix notation](http://en.wikipedia.org/wiki/Prefix_notation) places operators to the left of their operands. In prefix notation, the expression “4 + 2” would be written as “+ 4 2”. If all operators are of fixed arity, the notation does not require parentheses to be parsed unambiguously. The algorithm for converting an infix expression to a postfix expression is described later in this document.

1. Preliminary Exercise

For the preliminary exercise you will create a fully functional GUI with the exception of the “=” button. The GUI should include all buttons shown in the example earlier in this document. The File menu should have an Exit option that closes the GUI and terminates the program. The Edit menu should have Copy and Paste options that copy the contents of the display to a clipboard and paste the contents of the clipboard to the display, respectively. The Help menu should have an About option that shows information about the application (names of authors, date, etc.)

The constructor of the ExpressionTree will be passed a string in which each token is separated by whitespace so it can be read with a Scanner. Thus, as the user clicks buttons on the GUI, it will need to create expressions with appropriate spaces between tokens. ***This is a non-trivial task!***

Spaces before and after operators and parentheses are easy, but striving for excellence in all it does, the GUI will ensure there is only a *single space between these tokens*.

The GUI must allow users to enter numbers with more than one digit. Thus, it will need to recognize when a digit should and should not be preceded or followed by a space. Users should also be able to type a *single* decimal point within a number. If the user clicks a decimal point before any other digits, a leading zero should precede the decimal point.

Unary negation of entire expressions is particularly tricky and is not required for this exercise. However, the GUI can allow unary negation of numeric values by ensuring there is no space between the minus sign and the number. This means the scanner will parse the minus sign as part of the value rather than as a separate operator.

For up to **5% extra credit on the prelim**, include unary negation of numeric values in your GUI’s functionality.

You also will create a thorough set of JUnit tests for your infix to prefix converter and your expression tree. In addition to testing if the evaluate method of the tree calculates the correct numeric result, you will also need to ensure your tree is properly constructed. An effective way to do this is to check the in-order, pre-order, and post-order traversals of the tree.

Note: For the preliminary exercise submission, all of your JUnit tests will fail as you have not yet written the infix to prefix converter or completed the expression tree. The goal here is that you have a carefully written set of JUnit tests to help in the development of the rest of the exercise.

1. Preliminary Submission Requirements

* Fill in your name at the top of all provided source files!
* Fill in your documentation statement at the top of ExpressionTree.java!
* Fill in your pair programming log in the PairProgrammingLog.txt file!
* Your NetBeans project name must be LastnamesPEX3. (Right-click on the project in NetBeans and change the name!) Zip this entire folder to LastnamesPEX3.zip.
* Using the Webpost link on the left side of the course web page, submit only one file for the preliminary exercise, a zip file containing your entire NetBeans project.

1. Programming Exercise

For the programming exercise you will implement the ExpressionTree class and add the final functionality to the “=” button in your GUI. This means when the “=” button is pressed, the text in the display area is passed to the constructor in your ExpressionTree class. Within the code provided are several public methods that must be completed. You also will need to add some private methods to fully solve the problem.

You also must revise your JUnit tests based on any feedback you are given by your instructor after the preliminary exercise submission and ensure your infix to prefix converter and expression tree passes all JUnit tests.

1. Programming Exercise Submission Requirements

* Fill in your name at the top of all provided source files!
* Fill in your documentation statement at the top of ExpressionTree.java!
* Fill in your pair programming log in the PairProgrammingLog.txt file!
* Your NetBeans project name must be LastnamesPEX3. (Right-click on the project in NetBeans and change the name!)
* Your ExpressionTree must have the exact class name and public method names provided so your instructor can run their own set of JUnit tests.
* Your NetBeans project name must be LastnamesPEX3. (Right-click on the project in NetBeans and change the name!) Zip this entire folder to LastnamesPEX3.zip.
* Using the Webpost link on the left side of the course web page, submit only one file for the preliminary exercise, a zip file containing your entire NetBeans project.

1. Helpful hints

**Get started early!** Creating the GUI may seem like a trivial task, but there are very subtle issues regarding the proper spacing of the tokens in the expression.

The algorithm for converting an infix expression to a prefix expression **uses two stacks**, one for operators and one for operands, both of which contain Strings. The algorithm is as follows:

* While there are more tokens in the infix expression,
  + Get the next token from the infix expression.
  + If the token is a left parenthesis, push it onto the operator stack.
  + Else if the token is a right parenthesis,
    - While there are operators on top of the operator stack,
      * Pop the top operand off the operand stack into variable *x*.
      * Pop the next operand off the operand stack into variable *y*.
      * Pop the top operator off the operator stack into a variable *op*.
      * Concatenate *op*, *y*, *x* (with appropriate spacing) and push the result onto the operand stack.
    - Pop and discard the left parenthesis from the top of the operator stack.
  + Else if the token is an operator,
    - If the operator stack is empty or the top item on the operator stack is a left parenthesis, push the token onto the operator stack.
    - Else
      * While there are operators on top of the operator stack and the token has lower or equal precedence than the top operator on the operator stack,
        + Pop the top operand off the operand stack into variable *x*.
        + Pop the next operand off the operand stack into variable *y*.
        + Pop the top operator off the operator stack into a variable *op*.
        + Concatenate *op*, *y*, *x* (with appropriate spacing) and push the result onto the operand stack.
      * Push the token onto the operator stack.
  + Else // Token must be a number.
    - Push the token onto the operand stack (still as a string).
* While the operator stack is not empty,
  + Pop the top operand off the operand stack into variable *x*.
  + Pop the next operand off the operand stack into variable *y*.
  + Pop the top operator off the operator stack into a variable *op*.
  + Concatenate *op*, *y*, *x* (with appropriate spacing) and push the result onto the operand stack.
* The top (and only) item on the operand stack is the result.

**CS 220 – PEX 3 Prelim – Grade Sheet Name:**

Points

Criteria Earned Available

|  |  |  |  |
| --- | --- | --- | --- |
| GUI includes necessary buttons (see example in this document) | |  | **4** |
| GUI includes menu bar with appropriate functionality | |  | **4** |
| GUI properly spaces operators, parentheses, and single-digit numbers | |  | **4** |
| GUI properly handles multiple digit numbers and numbers with decimal points | |  | **4** |
| GUI components have appropriate variable names | |  | **4** |
| JUnit tests[[1]](#footnote-1) of expressions with a mixture of + and - | |  | **4** |
| JUnit tests of expressions with a mixture of \* and / | |  | **4** |
| JUnit tests of expressions with a mixture of +, -, \*, / | |  | **4** |
| JUnit tests of expressions with a mixture of +, -, \*, /, and parentheses | |  | **4** |
| JUnit tests of invalid expressions | |  | **4** |
| **Subtotal:** | |  | **40** |
| Extra Credit: GUI allows unary negation of numeric values | |  | **+ 2** |
| **Adjustments** | **All Java code meets specified standards:** |  | **− 4** |
| **Vague/Missing Documentation:** |  | **− 2** |
| **Submission Requirements Not Followed:** |  | **− 2** |
| **Late Penalties:** |  | **25/50/75%** |
| **Total w/adjustments:** |  |  |

Comments from Instructor:

**CS 220 – PEX 3 – Grade Sheet Name:**

Points

Criteria Earned Available

|  |  |  |  |
| --- | --- | --- | --- |
| JUnit tests properly updated after preliminary feedback | |  | **4** |
| Infix to prefix converter passes all JUnit tests (yours and ours) | |  | **8** |
| Expression tree passes all JUnit tests (yours and ours) | |  | **8** |
| GUI creates and evaluates an ExpressionTree object and displays the result | |  | **8** |
| GUI gracefully handles exceptions and displays appropriate user feedback | |  | **8** |
| ExpressionTree parses input string and recursively builds tree | |  | **8** |
| ExpressionTree recursively evaluated after being built | |  | **8** |
| ExpressionTree includes correct inOrder, preOrder, and postOrder methods | |  | **12** |
| All methods have adequate Javadoc comments | |  | **4** |
| In line comments present and accurately describe what the code does | |  | **4** |
| Pair programming log provided demonstrating roughly equal drive time | |  | **8** |
| **Subtotal:** | |  | **80** |
| Extra Credit: toString method re-creates required parentheses | |  | **+ 4** |
| **Adjustments** | **All Java code meets specified standards:** |  | **− 8** |
| **Vague/Missing Documentation:** |  | **− 4** |
| **Submission Requirements Not Followed:** |  | **− 4** |
| **Late Penalties:** |  | **25/50/75%** |
| **Total w/adjustments:** |  |  |

Comments from Instructor:

1. JUnit test categories listed on this grade sheet apply both to the infix to prefix converter and the expression tree. [↑](#footnote-ref-1)