CSC115 Assignment 3: Calculator

Objectives

Upon completion of this assignment, you need to be able to:

- Create a simple referenced based stack..
- Use a stack and a simple list to solve a common problem.
- Begin developing generic programming tools.
- Parse String objects.
- Create, throw, propagate, and handle Java Exceptions.
- Continue accumulating good programming skills.
- For interest only: to have been introduced to a little *pattern matching*.
- For interest only: to have been introduces to Java GUIs (Graphical User Interface).

References:

- CSC115 Java coding conventions.
- Textbook Chapters 3 and 7.
- Textbook Chapter 6: The section called Algebraic Expressions.
- (optional) An introductory tutorial to regular expressions.

Introduction

Our current assignment is to provide the source code for a very basic calculator. Before we even think about the graphical design that will capture user input, we need to build the *engine* that converts a standard arithmetic infix expression into its numerical solution. We prefer to calculate a solution by parsing an expression from left to right. However, the infix expression is not calculated this way: we need to follow operator precedence and parentheses. Luckily, the equivalent postfix version will allow us to parse an expression from left to right and easily evaluate the solution.

We break this system into two parts:

Part 1:: Convert an infix expression into its equivalent postfix expression.

Part 2:: Evaluate the postfix expression to calculate the numerical result.

Quick Start

(1) Create a fresh directory to contain this assignment: `CSC115/assn3` is a recommended name. Download this pdf file and the `.java` files to this directory.

(2) Download and extract the `docs.zip` and `javafiles.zip` in the same directory. A `docs` directory will be created to store the specifications for the `public` classes. The following links will work if the documents are stored in the `docs` directory local to this pdf file.

- StringStack specifications
- ArithExpression specifications

(3) Most of the classes are complete. Some are supplementary, and others are just for fun.

(4) Create two simple classes first: `StackEmptyException` and `Node`.

(5) Complete the `StringStack` class.

(6) Examine the individual methods of `ArithExpression`. Read the textbook, the specification document, and the notes inside the source code. Read the detailed instructions in the next section of this document.

Detailed Instructions

There are several files supplied in this assignment. A lot of the code has already been written. Read through these segments and the documentation links to understand how they fit together and how one class *helps* another. We provide details of every class in the next sections. The following files are completely done and need no additions:

- `Calculator.java`
- `InvalidExpressionException.java`
- `Memory.java`
- `TokenList.java`
- `Tools.java`

The Node and StackEmptyException classes:

The `Node` class needs to be created. The `InvalidExpressionException` class is provided as a model for the `StackEmptyException` that you must also create. Once you have complete these two classes, then start the `StringStack` class.

The StringStack class:

The shell of the `StringStack` class is provided. Use the specification document to fill in the necessary methods. Note that the underlying data structure must be a singly-linked list that is initialized by the `head` data field. Since a Stack ADT is not concerned with the number of elements, a `count` or `size` data field is not required. You may add any private methods that you find helpful; for example, a printout of the contents is always recommended for debugging and testing purposes. You must provide a test harness in the `main` method that tests each of the methods.

The Tools class:

Every programmer should have a `Tools` class to store helpful methods that are not necessarily attached to a particular object. An indication that a method is a candidate for the toolbox is that it needn't be as restrictive as is required within the class it is being written for. For example, the method `isBalancedBy` originated in the `ArithExpression` class, since very good infix expression must have balanced parentheses. During the writing of

the header comments, it occurred to us that it didn't matter that the string was an arithmetic expression.

We thought that finding balanced parentheses would be useful for any string, or for other types of parentheses. So we gave the method a little more flexibility and put int into the toolbox. There it can be used not only for arithmetic expressions, but also for parsing html meta-tags or source code braces. This code also uses *recursion*, a subject covered during lectures.

The TokenList class:

This completed class is the very simplest of lists. It holds only `String` objects, only appends to the end of a list, and does not allow for the removal of an object. We use the `TokenList` for each of the data fields in the `ArithExpression` class, one field holds the infix expression as a set of tokens (numbers or operators) and the other holds the postfix expression as a set of tokens. Having a set of tokens allows us to parse the individual items in an expression.

The ArithExpression class:

The ArithExpression class is the engine of the calculator and the most intensive part of this assignment. Before completing the methods of this class, make sure that `StringStack` has been thoroughly tested and you understand the purpose of each of the supporting classes.

The constructor is complete: it checks for balanced parentheses and tokenizes the incoming string by calling the completed `tokenizeInfix` method. When the constructor is finished, the data field `infixTokens` will be full of *tokens*, which are differentiated by the following:

- an operator; one of the following:

  `^,*,/,+,-`

- a round parenthesis; one of the following:

  `(,)`

- a string that isn't identified as one of the above items. Note that if the expression is valid, it should be a legitimate number, but we are not checking for that level of accuracy in this method.

The two simple methods `getInfixExpression` and `getPostfixExpression` are useful for debugging; finish these first.

The method `infixToPostfix` needs to be completed. The difficult part is developing a complete algorithm that can take any infix expression and convert it to a postfix expression. Fortunately there is an algorithm in the textbook: you may choose this one, or develop one of your own. It is fairly complex, so we recommend you modify the textbook's, which uses a stack to parse the infix and create the postfix. It is also possible to do this using recursion, but if you use the textbook's algorithm, then use the `StringStack`.

The `evaluate` method parses the postfix expression contained in the `postfixTokens` data field after `infixToPostfix` is done. The algorithm for this is also discussed in the textbook; it uses a stack. You are to use the `StringStack` for this. HINT: look at the `Double` class in Java for a means to turn a `String` object into a `double` data typle.

Both the `infixToPostfix` and `evaluate` methods will need to call smaller private methods. The completed `isOperator` is an example of such a method. You are welcome to use and / or modify it to suit your programming needs.

Requirements for both StringStack and ArithExpression:

Follow the instructions within the incomplete source code provided. All methods must meet the specifications in the documentation files. Both `StringStack` and `ArithExpression` must contain a set of test cases in the appropriate `main` method, demonstrating that thorough testing was done. Only when everything was tested, should you use the graphical calculator to input a variety of tests.

Using the GUI Calculator

The `Calculator` is provided for your enjoyment, and as inspiration for anyone wishing to investigate computer graphics using Java. Enjoy using this as a tester for a number of expressions once you have verified some tests in `main`. However, if you find an error, use this information as an indicator that you need to go back to the internal tester, to find the source of the error.

To play with the Calculator, simply compile it and run it in the directory. If `ArithExpression.java` compiles, then the calculator will run. Note that this a GUI; it does not *do* any work. It delegates all the work to the `Memory` class and to your `ArithExpression` class. Initially, hitting an `ENTER` key or clicking the = button will print out the value -1.0. Eventually, it should produce the correct answer. If the input is incorrect, and ArithExpression works as it should, you will see `INPUT ERROR` in the textbox.

Submission

Submit the following completed file to the Assignment folder on conneX.

- `Node.java`
- `StackEmptyException.java`
- `StringStack.java`
- `ArithExpression.java`

Please make sure you have submitted the required file(s) and conneX has sent you a confirmation email. Do not send [`.class`] (the byte code) files. Also, make sure you *submit* your assignment, not just save a draft. Draft copies are not made available to the instructors, so they are not collected with the other submissions. We *can* find your draft submission, but only if we know that it's there.

A note about academic integrity

It is OK to talk about your assignment with your classmates, and you are encouraged to design solutions together, but each student must implement their own solution.

Grading

Marks are allocated for the following:

- Proper programming style is demonstrated, as per the coding conventions on CSC115 conneX Resources.
- Use of private helper methods where applicable: if similar code is used in more than one method, then a separate private method should be created that is then called by the methods using similar operations. See the array based version of the list as an example.
- Internal testing in both `StringStack` and `ArithExpression` main methods of all public methods that you complete.

You will receive no marks for any Java files that do not compile.