University of Victoria
Faculty of Engineering: Computer Science

CSC 115: Fundamentals of Programming II

Spring 2016

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 an Exception, throw, propagate and handle Java Exceptions.
- Continue to apply good programming practice in
  - designing programs
  - proper documentation,
  - testing and debugging problems with code.
- Understand a little about pattern matching.

Resources:

- CSC 115 Java Coding Conventions.pdf (found on conneX, under Resources)
- Textbook Chapter 3
- Textbook Chapter 6: The section called Algebraic Expressions.
- Textbook Chapter 7.
- (optional) An introductory tutorial to regular expressions.

Introduction:

We are 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) If you haven't done so already, download all the necessary documents for this assignment from conneX into a specific folder for CSC115 assignment three, `assn3` is a recommended name. You may want to separate the documentation files into a separate directory.
2) All of the specification documentation for public class and public methods can be found in the appropriate `*.html` files.
3) The following classes are complete: `InvalidExpressionException`, `TokenList`, and `Tools`.
4) Create the simple classes first: `StackEmptyException` and `Node`.
5) Complete the `StringStack`. It should be easy.
6) Look at the individual methods of `ArithExpression`. Read the textbook, the specification document, and the notes inside the `ArithExpression.java` file. Read the detailed instructions that follow.

Detailed Instructions:

There are several files supplied in this assignment. Some of the code has already been written. Take advantage of these segments by understanding how they work and how they fit with the rest of the files. These are small gifts to you, to help you develop good programming skills.

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. Only after completing these, complete 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` 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 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 every 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 perhaps 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 it into the toolbox. There it can be used not only for arithmetic expressions, but also for parsing html meta-tags or source code braces.

The TokenList class:
This completed class is the very simplest of lists. It contains strings, only appends to the end of the list, and does not remove anything. This works very well as a simple storage for each of the infix and postfix expressions.

The StringStack class:
The Stack ADT is not concerned with implementation. However, for this assignment, the (to be completed) `StringStack` is a singly-linked data structure, specifically designed to contain `String` objects. A single data field is provided. As in the previous assignments, any references to the `java.util` package are unacceptable.

The ArithExpression class:
This class is the engine of the calculator and the most intensive part of this assignment. Before starting this, 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 (also complete) and tokenizes the incoming string into `infixTokens` (also complete). The method `infixToPostfix` needs to be completed. Before starting this, you are encouraged to complete the two methods that return the string representations of both `infixTokens` and `postfixTokens`. They are extremely useful during implementation and debugging. The textbook describes an algorithm for converting a valid infix expression into a postfix expression, using a stack. In the labs during week 5, we will demonstrate another strictly recursive solution. You may use either algorithm.

The `evaluate` method parses the `postfixTokens`. The algorithm for this is also discussed in the textbook and 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 type.

Both the `infixToPostfix` and evaluate methods should call smaller private methods when possible. The completed `isOperator` is an example of such a method. Smaller methods are much easier to test and often reusable.

> If you are interested in further study, pattern matching using regular expressions is a very powerful tool used in word processing, web search engines, natural language processing, and computational molecular biology. We use some of the Java pattern matching and regular expression tools inside the `tokenize` method.

Requirements for both StringStack and ArithExpression:
Follow the instructions within the incomplete source code provided. All methods must meet the specifications in the html files.. Both `StringStack` and `ArithExpression` must contain a set of test cases in the appropriate `main` method, demonstrting that thorough testing was done. We recommend using a `Scanner` object in `ArithExpression.main` that allows you to input any number of arithmetic expressions as a final test.

Submission

Submit the following completed files to the Assignment folder on conneX:

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

Please make sure that conneX has sent you a confirmation email. Do not send [`.class`] (the byte code) file, or any another file for this assignment. Also make sure you *submit* your assignment, not just save a draft. All draft copies are not available to the instructors, so we cannot mark them.

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 (code) their own solution.
We will be using plagiarism detection software on your assignment submissions.

Grading

All submitted java files must compile.
The marker will be looking for:

- Proper programming style as per the code conventions on CSC115 conneX Resources.
- Source code follows the specifications and instructions.
- Evidence of private helper methods where applicable.
- Testing of the StringStack and ArithExpression, using the main method as a test harness.