Assignment 1: Stacks and Reverse Polish Notation

Deadline for Submission: Monday 3rd February, 2020 2pm.

Hand in Method: Submit on MyDundee. The submission link will not be made available until Thursday 30th January. Note that this is an individual assignment (not groupwork!).

Date Feedback will be Received by: Latest date to receive feedback will be Monday 24th February.

Penalty for Late Submission: One grade point per day late (meaning if a submission is one day late and marked as a C2 it will receive a C3 grade) A day is defined as each 24-hour period following the submission deadline including weekends and holidays. Assignments submitted more than 5 days after the agreed deadline will receive a zero mark (AB).

Percentage of Module: This assignment is worth 10% of this module for AC12001 and 7% for AC22007.

Overview of Assignment

- Stacks
- Reverse Polish Notation
- Exceptions, String splitting

Assessment Details

Everyone should attempt the first four requirements below (out of 80%); you should only attempt the optional requirements if you feel confident that you have finished all of the others. In your report please state which requirements were met and what problems, if any, you encountered.

Aim

The aim of this assignment is to learn how to design and code a Java program that uses a stack data structure, implemented from first principles, using a linked list. You will also need to understand the concept of reverse polish notation. Note: DO NOT IMPLEMENT YOUR STACK USING AN ARRAY FOR THIS ASSIGNMENT, AND DO NOT USE A LIBRARY CLASS FOR YOUR LIST OR STACK! Implementing from first principles means that you must code the stack class yourself and not use any library classes to do this. We will use the library collection classes in another assignment.

Background

In order for a program in a high-level language like C, C++ or Java to be run on a computer, it must first be translated into machine code (or bytecode), or compiled in some way. A compiler must perform many different tasks, including error checking, interpretation of loops and if statements, etc. One of the most basic operations is the translation of simple arithmetic instructions into machine code.

In order to calculate arithmetic expressions, a compiler will convert the expression from “infix” notation into “reverse polish” notation (RPN). This allows calculations in low-level assembler code.
Example: The infix expression “4 * 2 + 6” would be compiled into RPN as “4 2 * 6 +”.

Requirements

You are to design and write a calculator program which uses reverse polish notation. You only need to consider the operators: +, -, *. The full list of requirements are:

1) Your program should accept a complete postfix equation of tokens (numbers and operators) from the user as a single String of characters, e.g. "3, 5, *, 6, 4, +, 2, *, +"

2) The program should then process the postfix equation and calculate the answer using a stack data structure implemented with a linked list created from first principles (not using a pre-written library class).

3) Your program should include a Menu with options to show the instructions on how to work the calculator, the calculator itself and an exit option.

4) Documentation should include a Class design, Pseudo code (or activity diagram) for the main methods, Test plans for common scenarios (e.g. pushing, popping on empty and non-empty stacks, correct and incorrect RPN input, different RPN operators), a completed test sheets (results of tests specified in the test plan), documentation and a self-evaluation report. Please ask before submitting if you are unsure about what documentation to include in your submission.

5) *[Optional]* Add data validation to check input for invalid characters and invalid postfix. Handle runtime errors and invalid postfix using exceptions.

6) *[Optional]* Implement the '/' operator method using floating-point data to handle the non-integer division results/

7) *[Optional: Advanced]* As well as outputting the correct answer to an RPN input equation, show the infix version of the equation, using brackets as appropriate.

If you feel confident then please design, implement and test your program to meet the above requirements in any sensible manner. However, to help you, I have given a set of suggested instructions below.

Method

You should reuse the code for your **List** and **ListNode** classes from our previous Lab (use the sample answers if you did not get your own version to work) – these already provide the general structure required to add items from a List (or a Stack in this case). You can then adapt and/or 'wrap' the additional Stack related functionality around these existing classes. Remember: please make use of the assistance available in the labs to help you.

In attempting this lab, the following steps could be followed:

1) Produce your test plan for the basic **List** methods ready for running and recording your tests after each major change to your program.
2) Create a new Java project with the **List** and **ListNode** classes and any other useful code you used for Lab 1.
3) Modify the **ListNode** class to store an **int** instead of the Student data we used previously (you will be pushing and popping integers onto the Stack, each node in the list will contain the next number encountered in the postfix expression being evaluated). Operators such as '+', '_', etc. do not need to be stored as nodes on the stack. Operators will be evaluated at the point when they are encountered during the parsing of the postfix equation and only the result of the operation will be stored onto the stack.
4) Implement a `deleteFromStart()` method in your List class. Note that this was an optional extra in the lab practical from week 1 so you may already have done this. It was not included in the sample solution.
5) Consider the design of a new class, **Stack.** Decide whether to create a new class (**Stack**) which contains a field of type **List** (I recommend this approach) or modify the **List** class to be a stack. You can use

existing methods on the List class (such as 'add' and 'delete') to achieve the 'push' and 'pop' functionality required.Design the following methods:

- Check if stack is empty
- Push a number onto the stack
- Pop a number from stack

6) Produce a test plan for the basic **Stack** methods above.

7) Implement the **Stack** class and its methods.

8) Design and code a **Tester** class for your program, write an automated `stackTest` method to create an instance of your Stack and call methods to push and pop numbers to and from the stack.

9) Now, add an **RPNCalculator** class to your Java project. This class will use an instance of your Stack to help it process a reverse polish notation expression. A Stack object can be declared as one of the fields in the RPNCalculator class or as a local variable within one of its methods. To process your postfix expression, you will create an instance of the Stack and then call its methods to push and pop items from the stack as required during the processing steps of the expression.

10) Add a method to your RPNCalculator class which uses a stack and its operations (push and pop) to evaluate a reverse polish notation expression. Your method should receive two strings as input parameters as follows: one string which will be the reverse polish expression to evaluate (e.g. "4,2,+") and another String containing the delimiting character that separates the tokens within the expression (in this case a comma ","). Inside your method you will need to split the contents of the first String into the individual tokens that it contains so you can process them one at a time in order. One option is to use the **split** method of the String class which we have seen previously but there are other possible techniques. Feel free to use whatever technique you are most comfortable with. *Hint: Look up the String split() method in the Java class library or see the "parseInput" sample code in the Week 2 folder on My Dundee.*

11) Thoroughly test your RPN calculator by writing more automated test code to do this in your Tester class. For example, in an `RPNTester`() method in your Tester class, create an instance of your `RPNCalculator` class and then call the above method with some test inputs such as the following: "4,2,+" [answer should be 6!]. Remember to write your test plan (before coding) and then document your test results (after coding) before submitting.

12) Test the `RPNCalculator` class more thoroughly to show that it can handle more complex calculations too. Do some RPN calculations by hand and see if your program gets the same results. Record these details in your Test plan to show the different tests you have performed.

13) Write a Menu in your tester class with options to show the instructions on how to work the calculator, the calculator itself, run the automated test methods and an exit option.

14) Ensure that your report is completed, execute your Test Plan and record the results, generate the html JavaDoc and a runnable JAR file that you have tested.

Consider attempting some of the optional extra requirements once you are happy that you have completed all the other requirements.

Marking Scheme

Requirements 1-5	Report & Designs (pseudo code and/or flowcharts), self-evaluation.	out 10%
	Test plan and completed test sheets	out of 10%
	Code	
	Listnode and List classes (including List delete method):	out of 10%
	Stack: Push, Pop and other stack method(s)	out of 20%
	Manipulation of strings, i.e. splitting into tokens	out of 5%
	Process RPN expressions	out of 15%
	User interface, including input validation	out of 10%
Requirement 6	Optional requirements	out of 20%

Please note that well-written code, commenting and a runnable JAR file have not been specifically allocated marks. This is because it is expected that you will include these in all assignments and that marks will be deducted for any type of bad practice (e.g. poor use of static keyword, lack of commenting or Javadoc comments, badly written code or no main method), no JAR file or JAR file that doesn't run. As always, please ask for help if you have any problems.

Submission

You must submit your lab by the deadline given in a single zip file via the link on My Dundee which can be found under Assignments→1.RPN Stack.

Your .zip file should be labelled with your surname, first name and the assignment number, e.g.
smith_john_assignment1.zip

This file will include:

- ONE written report which will include:
  - An introduction stating the problem
  - A summary of the requirements – saying which you were able to tackle and which were successful
  - Your designs using including pseudo code (don't forget your class design)
  - Your test plans & results
  - Your self-evaluation – detailing problems and how you tried to solve them, stating anything you couldn't fix
- A runnable JAR file of your program.
- Your full Eclipse or BlueJ project folder which must include your Java source code and the html Javadoc documentation. If you don't use Eclipse of BlueJ, then please ensure that you include your Java source code files and your Javadoc html. We need your source code to mark your assignments.

This lab contributes 10 % of your coursework marks for AC12001 and 7% for AC22007

Remember to SUBMIT whatever you have done by the deadline, rather than over-running and handing it in late - this should avoid you getting behind.