CSC 115 A01 (Java)
Fall 2012 L.Jackson
Credits: R.Brown

Assignment #4

Due

Milestone (Submit Part 1): before 3:30 pm on Friday, November 2, 2012
Final Submission (Submit Part 2): before 3:30pm on Friday, November 9, 2012

Learning Outcomes: Upon successful completion of Assignment #4 you will be able to:

- Implement a generic collection that is capable of holding a variety of value types.
- Implement and use a stack ADT.
- Transition between `int` values and `Integer` objects when using a generic collection, and also why it is necessary to do so. This is an example of what is known as "boxing" or "auto-boxing".
- Adapt a given pseudo-code algorithm, designed for one context, to work under a different context and set of requirements.
- Throw and catch exceptions, produce a meaningful error message, and exit a program normally.

Overview

Interpreting the meaning of expressions in many languages often requires a form of temporary memory to record partially understood clauses. A stack is often the ideal form of memory for such tasks.

In this assignment, your task is to complete the implementation of a simple arithmetic expression interpreter. The interpreter is capable of handling two forms of expressions: *infix* and *postfix* (see "Algebraic Expressions" under section 6.2 of Carrano & Prichard). Here are some examples,

Infix	**Postfix**	**Result**
`5 + 4`	`5 4 +`	`9`
`5 - 4`	`5 4 -`	`1`
`5 * 4`	`5 4 *`	`20`
`123 / 45`	`123 45 /`	`2`
`1 + 2 * 3 - 4`	`1 2 3 * + 4 -`	`3`
`1 + 2 * 3 / 4 - 5`	`1 2 3 4 * + 5 / -`	`-3`
`( 4 + 5 ) * ( 6 / 3 )`	`4 5 + 6 3 / *`	`18`
`2 + ( 3 + ( 4 + 5 ) )`	`2 3 4 5 + + +`	`14`

The expressions contain integers and the operators + (addition), – (subtraction), * (multiplication) and / (division), and in the case of infix expressions, the nesting operators (and).

Postfix expressions can be evaluated directly using a stack of integers to keep track of partial results. To evaluate an infix expression it needs to be translated to an equivalent postfix version, then evaluate the postfix form.

Part 1

Problem statement: Create a generic implementation of the Stack ADT interfacc using a linked-list collection of values.

Specifications:

- Download `Stack.java` and `EmptyStackException.java`.
- Create a class called `LinkedStack` that implements the `Stack` interface using a linked-list collection of values. Your stack must operate as detailed in the given `Stack` interface. (In particular, the methods must all operate in O(1) time and must throw the correct exception types.)
- Your `LinkedStack` must be a generic implementation. In other words, the same code must be capable of representing a stack of integers (`LinkedStack<Integer>`) and a stack of strings (`LinkedStack<String>`).
- Your `LinkedStack` should have a "default" public constructor that requires no parameters.
- Be sure to develop (and submit!) a test program that checks whether your `LinkedStack` works as required.

Submitting your Solution:

When complete submit your `LinkedStack.java` file to the CSc 115 Connex Site using the Assignments: Assignment 4 Milestone link before 3:30 on Friday, November 2, 2012.

This milestone is a formative exercise: It will not be graded but its completion will inform you and the instructor of your progress through this assignment.

Part 2 (a) Postfix Calculator

Problem statement:

Implement of an arithmetic interpreter that evaluates *infix* expressions.

Specifications:

- Download `Expression.java` and `ExpressionFormatException.java`.
- Follow the algorithm in section 7.4 of Carrano & Prichard to complete the code for the `evaluateAsPostfix()` method. Note, the pseudo-code in 7.4 operates on individual characters whereas your code should operate on *tokens*—multi-character words that may represent a number or an operator. The provided constructor of `Expression` will divide an input expression string into a list of tokens for you.
- Use your `LinkedStack` from Part 1 as a `Stack<Integer>` of partially computed results.
- Your code should throw an `ExpressionFormatException` if the tokens do not represent a valid postfix expression. The list of tokens should not be modified.

- Develop a test program that checks whether your `Expression` code translates and evaluates correctly, and also rejects invalid inputs by throwing an `ExpressionFormatException`.

Part 2 (b) **Infix-to-Postfix Translator**

Problem statement:

Implement of an arithmetic interpreter that evaluates *infix* expressions.

Specifications:

- Follow the second algorithm in section 7.4 of Carrano & Prichard to complete the code for `toPostfix()`. Again, your code must operate on tokens rather than characters. This means you will not be able to use a `switch` statement as shown in the pseudo-code in the text.
- Again, use your `LinkedStack`, but this time make it a `Stack<String>` of tokens.
- Your code should throw an `ExpressionFormatException` if the tokens do not represent a valid infix expression. The list of tokens should not be modified, instead create a new `Expression`.
- Develop a test program that checks whether your `Expression` code translates and evaluates correctly, and also rejects invalid inputs by throwing an `ExpressionFormatException`.

Submitting your Solution:

- When complete submit All .java files that you used, modified, or created to complete this assignment to the CSc 115 Connex Site using the Assignments: Assignment 4 Submission link before 3:30 on Friday, November 9, 2012.
- Any file that you modified or created must contain a comment at the top that includes your name and student ID.
- If you adopted or adapted code from other sources, you must include an appropriate crediting reference to the original author or source.
- Be sure to include any testers that you wrote.