CSE 214 – Homework II

Instructor: Dr. Ritwik Banerjee

This homework document consists of 3 pages. Carefully read the entire document before you start coding. This homework requires you to implement the `Stack` data structure without using any readily available classes or interfaces in Java.

---

1. Overview

In this homework, some code is already given to you. Your task is to write additional code based on what is given. The code given to you consists of two packages. The contents are as follows, with interfaces in green, and classes in blue (with enums italicized):

package datastructures.sequential
- `LIFOQueue`
- `SNode`

package applications.arithmetic
- `Converter`
- `DyckWord`
- `ArithmeticExpression`
- *`Brackets`*
- *`Operator`*

Your tasks consist of implementing a generic `Stack` class, and then using it to evaluate simple arithmetic expressions involving addition, subtraction, multiplication, and division.

2. Tasks

In this section, we will step through these tasks one by one. The homework is worth a total of 50 points, and the points for each task or subtask are provided in the right margin within parentheses.

1. **Implementing the stack data structure:** (5)

   Based on the `datastructures.sequential` package, you must implement your own `Stack` class in that same package. This class should implement the interface `LIFOQueue`. In order to do this, you will need to use the `SNode` class as well. Note that what you are being asked to do here is essentially build a stack using nodes and pointers instead of an array.

2. **Dyck Words:** (5)

   Dyck words are words that use balanced parentheses. The idea here is to check if an arithmetic expression is a valid Dyck word. This is an important step, because if an expression is not a valid Dyck word, then we cannot evaluate it correctly. For example, "()", "(2)", "$2 \times \{5 + (7 - 1)\}$" are valid Dyck words, while "$1 - 4)$" is not.

   A majority of the code for this class is provided to you. All you need to do is complete the body of the method `isDyckWord` in the `DyckWord` class. You are required to use the `enum` types in the `Brackets` class for this.

3. **Evaluating arithmetic expressions:**

   Note that "$(2 - 3+)$" is a valid Dyck word, but not a valid arithmetic expression. This task, however, is beyond the scope of this homework. For the purposes of this homework, you can safely assume that *if* an expression is a valid Dyck word, *then* it is also a valid arithmetic expression. This third, and largest, component of the homework consists broadly of two steps:

   - convert a given infix expression to its corresponding postfix expression, and
   - evaluate the postfix expression to obtain the final value.

The conversion algorithm:

To convert an infix into a postfix expression, we use a stack, which is used to hold operators rather than numbers. The stack is needed is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared. The algorithm rules are as follows, as you read an infix expression from left-to-right, one character at a time:

1. Initialize an empty stack.
2. Print operands as they arrive.
3. Push operators and left parenthesis on the stack as they arrive.
4. If we see a right parenthesis, pop the stack and print its symbols until we see a left parenthesis. The left parenthesis is popped, but not printed as output.
5. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
6. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
7. At the end of the expression, pop and print all operators on the stack.

(a) **Convert from infix to postfix:** For this, you are already provided with an interface called `Converter` in the `applications.arithmetic` package. This interface has a feature that may be new to you: it has a static nested class called `TokenBuilder`. It is directly in the interface because we want to build our code in a way such that if, in future, we want to add a different type of conversion (say, from infix to prefix), we can use the same interface without having to duplicate the `TokenBuilder` class[1]. (20)

The `TokenBuilder` class should be used to identify multi-digit as well as decimal numbers as single tokens. Note that the examples we saw in class only had single-digit numbers, so taking a single character as a token was enough. But for expressions like "$12.5 * (17 - 15)$", we must consider "12.5", "17" and "15" as single tokens in the main algorithm for conversion.

Your task is to write the `ToPostfixConverter` class in the `applications.arithmetic` package. This class must implement the `Converter` interface. Rules to follow:

1. The conversion algorithm must be coded in the `ToPostfixConverter#convert` method.
2. The conversion code must use your `Stack` class in the `datastructures.sequential` package, and **NOT** the `Stack` class provided by Java.
3. The `ToPostfixConverter` class must utilize the `Brackets` and `Operator` enums. Any use of hard-coded parentheses or operators like '+', '*' in the code will be penalized.

(b) **Evaluating postfix expressions:** Now that we have the code to obtain postfix expressions, the final component of this homework is to evaluate them. In this last part, you are required to mimic the design of the conversion process above.

i. First, you must create an interface called `Evaluator` in the `applications.arithmetic` package. Just like the `Converter` interface has the `convert` method, this must have the following: (3)
   - `String evaluate(String expressionString)`

   [*You may have additional methods as well, but that is entirely up to you.*]

ii. Second, you must create a `PostfixEvaluator` class that implements the `Evaluator` interface. This step, too, mirrors what we did in the conversion process. The postfix evaluation algorithm we studied in class, using stacks, must be coded in the `PostfixEvaluator#evaluate` method. (10)

4. **Documentation** (7)

The final component of this homework is not about coding, but about making your code easier to understand. This is done by means of Java documentation (*i.e.*, "Javadoc"). The interfaces, classes and methods already provided to you serve as examples of how to write Javadoc. Any interface, class, or method that you write should have similar documentation.

1. Methods should have:
   - a very brief description of what the method actually does,
   - one line explanations of its input parameters by means of the `@param` tags,
   - one line explanation of what it returns by means of the `@return` tag, and
   - if the method throws an error or exception, then a brief explanation of when and why such it is thrown (done using either the `@throws` or `@exception` tags).

[1] Another way of doing this is by creating an abstract class implementing `Converter` and then making other classes extending that abstract class.

2. Classes and interfaces (including inner classes) should have a description of why the class (or interface) is there, what purpose it serves, and how it can be used. This description should typically be no more than 5-6 lines. (More fundamental classes such as the ones you may find in Java source code sometimes have very large Javadocs, but we are not writing code anywhere nearly that complex, hence the 5-6 line limit.)

3. Running the code

You must have noticed that there is one class, `ArithmeticExpression`, which is provided, but not used directly. This is a very simple class that acts as a wrapper around `DyckWord`. Similar to how we have wrapper classes like `Integer` around `int`. The `ArithmeticExpression` class contains the `main` method to run the entire code. This, too, is already provided. The main method simply reads a text file that contains one infix expression per line, and prints out their values. If, however, the infix expression is not a valid Dyck word, it prints out an error message.

Guidelines

Can I change the given code?

No, and that includes the package lines at the top of the Java files. You can add your own code, but you cannot modify any existing code.

What can I assume about the test inputs?

1. Your code will be tested with numerical expressions only (not variables like 'x, y, z').
2. The infix expressions will not have any whitespaces. That is, your code is expected to run properly with "12.5*(7+0.25)" but you don't have to handle "12.5 * (7 + 0.25)".
3. The infix expressions will not require you deal with negative numbers as a single token. However, you should be able to handle expressions that finally evaluate to a negative number. For example, $1 - 8/2$ is something your code should be able to deal with, but you don't have to worry about expressions like $-4 + 2$.

How can I create postfix expressions involving decimals and multi-digit numbers?

You should use a single whitespace to separate individual tokens. E.g., "12.5*(7+0.25)" should become "12.5 7 0.25 + *", and not "12.570.25+*".

How detailed should the documentation be?

The documentation section already answers this. But, in addition to that, you may safely ignore documentation of standard getter and setter methods.

Can I add my name under the author tag?

Yes, but only if you have actually added some significant code in that class. On one hand, you should not add yourself as an author if you only add, say, some tiny bit of code to a file. On the other hand, if you add anything significant (like the convert and evaluate methods implementations), you SHOULD add yourself as an author using the `@author` tag in Javadoc.

What should I submit?

A single .zip archive consisting of only those .java files you created or where you added your own code. The archive must NOT have folders or subfolders within itself.[2]

What expressions will the code be tested with?

Test cases cannot be disclosed ahead of time! There is a file called "examples.txt" (provided with the code) with some example inputs in it, but these are by no means all the test cases.

Some additional points to be strictly enforced:

- **My code compiles on my laptop but ...** Then you have changed some code that you were not supposed to change, or submitted something that is not in line with the guidelines provided. **If your code does not compile, it will not be graded**.
- **I submitted the class files by mistake.** Class files are not code. They cannot be graded.

Submissions are due on Blackboard by Friday Oct 30, 2020.

[2] Extract your archive and double-check its structure before submitting to be sure of this.