Data Structures (CSC 151) **Winter 2023**
Project 4: Converting Infix to Postfix **Due: Tuesday, 02/28/2023 (11:59pm)**

> Unless specifically told otherwise, you may not import anything into your projects. That means you may not use any built-in data structures like ArrayList or Vector. This rule applies to all projects in this course.

Note: You are permitted to use the java.lang.Object class versions of primitive data types in this and future projects (e.g. Integer). For this project you are also allowed to use the Iterator and Iterable interfaces, which can be imported from java.util.Iterator. None of these are required to complete the project, but you are permitted to use them if you find it useful to do so.

Learning Objectives:

1. Use the Stack ADT
2. Practice with postfix expressions
3. Practice with generics, Java interfaces, and object casting

Reminder: Programming assignments are *individual* projects. I encourage you to talk to others about the general nature of the project and ways to approach it; however, the technical work (the code writing, and inspiration behind it) must be substantially your own. If any person besides you contributes in any way to the project, you must credit their work on your project. Similarly, if you include information that you gleaned from other published sources, you must cite them as references. Looking at and/or copying other people's programs or written work is inappropriate and will be considered cheating.

1 Introduction

A long time ago (often in elementary school), you learned to evaluate mathematical expressions using parentheses. For example, the the parentheses in the expression:

(9-4)*5

told you that the subtraction operation should be performed before the multiplication. If the parentheses were not there, you would follow the precedence order of operations which says:

1. Exponents are evaluated from left to right
2. Multiplication and division operations are performed next, left to right
3. Addition and subtraction operations are performed last, left to right

So for example:

$32 / 2^3 - 1$

The order of operations says that the exponent (2^3) should be done first, followed by the division, and finally the subtraction, giving the answer of 3. Without this order of operations, an expression like this one would be ambiguous without parentheses. This "normal" way of writing mathematical expressions is called *infix notation*.

On the other hand, a mathematical expression written in *postfix notation* doesn't need an order of operations or parentheses. Postfix expressions are never ambiguous. In postfix notation, the operator is written *after* the operants instead of in between them. Here are some examples (make sure you understand these before you try to implement the project):

infix	postfix
7+1	7 1 +
(6-2)*8	6 2 - 8 *
(6-2)*8	24 2 3 ^ / 1 –

2 Your Mission

Your assignment is to write a computer program that will convert parenthesized or unparenthesized infix expressions into their equivalent postfix expression. You'll be reading a list of infix expressions from a data file. You should display (to System.out) each infix expression along with its equivalent postfix expression in the following format:

```
7+1 --> 7 1 +
(6-2)*8 --> 6 2 - 8 *
```

In this project, we'll use capital letters to stand for numbers. So the output of the output your program produces should look like the following:

```
A+B --> AB+
A+B-C --> AB+C-
```

3 The Algorithm

Converting infix to postfix expressions can be done using a stack. The stack will keep track of the parentheses and operators (plus (+), minus (–), multiply (*), divide (/), exponent (^)). To convert an infix expression into postfix, follow these rules.

1. Read each token (each operand, operator, or parenthesis) from left to right. Only one pass is needed for this.
2. If the next token is an operand (represented by a capital letter), immediately append it to the postfix string.
3. If the next token is an operator, pop and append to the postfix string every operator on stack until one of the following occurs:
    a. the stack is empty
    b. the top of the stack is a left parenthesis (which stays on the stack)
    c. the operator on top of the stack has a lower precedence than the current operator (i.e. the one that was just popped off the stack)

4. If the next token is a left parenthesis, push it onto the stack.
5. If the next token is a right parenthesis, pop and append to the postfix string all operators on the stack down to the most recently scanned left parenthesis. Then discard this pair of parentheses.
6. If the next token is a semicolon, then the infix expression has been completely scanned. However, the stack may still contain some operators. (Can you explain why?) All remaining operators should be popped and appended to the postfix string.

Here's how the algorithm works on the expression `(A+B*(C-D))/E;` the rule number at the end of each line indicates which rule listed above was used to reach the current state from that of the previous line.

Next Token	Stack	Postfix string	Rule
(	(		4
A	(	A	2
+	(+	A	3
B	(+	AB	2
*	(+ *	AB	3
(	(+ * (	AB	4
C	(+ * (	ABC	2
–	(+ * (–	ABC	3
D	(+ * (–	ABCD	2
)	(+ *	ABCD–	5
)		ABCD–*+	5
/	/	ABCD–*+	3
E	/	ABCD–*+E	2
;		ABCD–*+E/	6

4 The Details

You'll be using generics and interfaces in this project, so while you're free to add to the minimum requirements described below, you shouldn't deviate too far from the general paradigm. I've provided starter files for you on Nexus. Most of the classes are blank, but you should name them in the same way they're given in the starter code in order for the Gradescope tests to run correctly. Here are the ones that aren't blank:

- `proj4_input.txt`: This text file contains some infix expressions to help you get started. The expressions are delimited by semicolons (it's a type of character-separated-value file!). You should still come up with some infix expressions of your own to test. Work through these examples by hand so you understand how it's supposed to work and what the expected output should be.
- `Token.java`: This is a Java interface from which you'll implement classes that represent various tokens (like Plus.java below). Each implementing class will have a toString() method to return the Token in String format and a handle() method to dictate how to process the Token when it is encountered. The Javadocs explain further.
- `Plus.java`: This is a skeletal example of one of the token classes.
- `Stack.java`: This is a skeleton of the Stack ADT.
- `StackTest.java`: a few JUnit tests to get you started. Note how `setUp` and `tearDown` methods create and destroy the Stack before and after each test.
- `Converter.java`: This class contains some sample code to help you figure out how to read input from a file.
- `Client.java`: All `main()` should need to do is create a new Converter and tell it to `convert()`.

Here are some details about the classes you'll need to write.

- The Stack class will hold tokens to be processed. You may use either an array-based or linked-list-based implementation. If you use a linked-list version, I would **strongly recommend** that you reuse your LinkedList class from Project 3. The stack should not deal with ListNodes directly.

  Your Stack should use a generic parameter so that it works with any kind of data. When you use it, you'll make a Stack<Token> so that this project's stack will only be able to hold objects that implement the Token interface. You **must** write your own Stack class. You may not use Java's built-in version.
- There is one class for each of the tokens to be processed. Each of these will be an implementation of the Token interface described above. There are eight tokens that need to be processed: "+", "-", "*", "/", "^", "(", ")", and ";" so there's one class for each of those. As mentioned above, each of these classes represents a token whose `handle()` method will describe how it should be processed according to the rules given above.
- The Converter class contains an instance method, convert(), that runs the algorithm detailed earlier in this document. That is, it reads in a token, identifies it, makes a new Token type object from it, then calls the item's handle() method to take care of it. It repeats this process until all

tokens have been read and processed. Having each item's behavior internalized via a method like this is called encapsulation, another object-oriented design goal.

Feel free to add other (appropriate) methods to any of the classes above.

Note about precedence: When dealing with operators, you will need a way of comparing them to see which has a higher precedence. To do this, use an int variable for each of the operators. A higher int means a higher precedence for that operator. Use the following precedence values:

- Exponent: 3
- Multiplication and division: 2
- Addition and subtraction: 1

Then, comparing two tokens is just a matter of comparing their int values.

Why do it this way? As you get into the project, it may seem quicker to code this in a straightforward manner without using the interface and all the implementing classes. In fact, it probably **would** be quicker, but not very well-designed. By using the interface, you control exactly what the stack can (and cannot) hold. Instead of holding just Strings or Objects, the stack can only contain Tokens. By limiting the stack in this way, you add robustness to the code by preventing the stack from being misused in ways that you can't foresee.

It also allows for the individual tokens to encapsulate their own behavior within their own classes. That way, if you modify the program by, say, adding the modulo operator (%), it can be done without touching the code for how division works.

Where to start: There are a lot of things to do in this assignment. I would recommend using a top-down design to help manage all of it. Here's a good way to approach it:

1. I've given you a bunch of code to start with. Studying and testing it should be your first priority. Understand the interface. Check out the input. Process the input on paper (or otherwise by hand) so you'll (1) know what the correct answers are and (2) have a better grasp on how the algorithm works. Try out the example code for reading from the file so you know how to use it.
2. You'll need a Stack class that can be used to store Tokens. Write and **test** the Stack class early. Use the testing skills you've been developing over the term so far to make sure your Stack **really works** before you do things that rely on it working correctly.
3. The primary algorithm at the top of this document is handled by Converter. Get a good understanding of exactly what this class is doing on its own and what tasks it is handing off to methods from other classes. Converter is the one that's repeatedly getting tokens, figuring out which one it is, and calling the appropriate `handle()` method. Think of `handle()` this way: instead of you writing all of the code in Converter (which would be very messy), you'll let each token "handle" itself. So once you've determined that a token is, for example, "+", you'll turn it into a Plus object, then tell it to `handle()` itself. And it will be each token's version of handle() that does the work of pushing on the stack, popping it, or whatever else. That makes each token responsible for doing the right thing and makes your code much more modular. This is a key part

of the algorithm, so you should make sure you understand what this means before moving forward (and come talk to me about it if you don't).

4. Do one Token at a time. Don't try to take care of all the tokens all at once. Concentrate on being able to read a "+" and get it on the stack. Now can you pop it off and get it back? Can you correctly convert an infix expression that just uses "+" like A+B+C?

5 Submission

Be sure to include the honor code affirmation in the comments of one of the classes:
I affirm that I have carried out the attached academic endeavors with full academic honesty, in accordance with the Union College Honor Code and the course syllabus.

Before you submit, check to make sure that your code satisfies the following requirements:

1. All files are properly commented
2. All files are formatted neatly and consistently
3. Your code has been cleaned up. Your code does not contain snippets that don't contribute to the purpose of the program (e.g. commented out code from earlier attempts or comments that aren't helpful).
4. Your variable and method names are informative.
5. There are no magic numbers (or other magic values).
6. Your code practices good information hiding
7. Your code makes use of already existing public methods or private helper methods to make complex tasks more modular and to minimize the number of methods that access instance variables directly.
8.

Once you've checked all these requirements, submit all of your code to *Project 4: Infix to Postfix* on Gradescope.

6 Grading guidelines

- Correctness: programs do what the problems require. (Based on Gradescope tests)
- Testing: The evidence submitted shows that the program was tested thoroughly. The tests run every line of code at least once. Different input scenarios, and especially border cases, were tested.
- Documentation: Every public method and class is documented in the appropriate format (Javadoc) and provides the necessary information. The information provided would enable a user to use the class/method effectively in their code.
  ADT invariants should be documented.
- Programming techniques: Programming constructions have been used appropriately in a way that makes the code efficient and easy to read and maintain. If the project required use of a specific technique or algorithm, this has been correctly implemented. For this project, I'll be paying special attention to good information hiding, modularity, robustness , and that the stated invariants are coherently implemented.
- Style: The program is written and formatted to ensure readability. For example, naming conventions are obeyed, whitespace (blanks, line breaks, indentation) is used to help structure the code, formatting is consistent, and the code is well organized.