Data Structures (CSC 151)
Fall 2024
Project 4: The Infix to Postfix Converter **Due: Tuesday, 10/29/2024**

Learning Objectives:

1. Use the Stack ADT
2. Practice with postfix expressions
3. Practice with generics, Java interfaces, and object casting
4. Learn how to enforce the design goal of robustness

1 Introduction

Way back when, you learned to evaluate mathematical expressions using parentheses. For example, the parentheses in the expression

$$(6 - 2) * 8$$

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

So, for example, in the expression

$$24/2^3 - 1$$

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 2. 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 operands instead of in between the operands. Here are some examples (make sure you understand these before you try to implement the project):

infix	postfix
$7 + 1$	$71+$
$(6–2) * 8$	$62–8*$
$24/2^3 - 1$	$2423 \wedge /1-$

2 Your Mission

Your assignment is to write a computer program that will convert parenthesized or unparenthesized infix expressions into their equivalent postfix expressions. You will 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 --> 71+
(6-2)*8 --> 62-8*
```

In the project, we will use capital letters to stand for numbers. So the output your program should produce will look as follows:

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

3 The Algorithm

Infix to postfix conversion can be accomplished with 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 required.
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 the stack until one of the following conditions 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

   Then push the current operator onto 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. (Why?) All remaining operators should be popped and appended to the postfix string.

Here's how the algorithm works on the infix 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. I'm supplying you with the starter files on Nexus. Most of the classes are blank – I just need you to have them named in the way Gradescope expects them. 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. Of course, you should be testing your code on your own input too! Perform the algorithm on these by hand so you understand how it is supposed to work.
- `Token.java`: This is a Java interface from which you will 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`: Skeleton of the Stack ADT.
- `StackTest.java`: JUnit tests for Stack. I've written a few to get you started. Note how I use the `setUp` and `tearDown` methods to 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, be sure you reuse your LinkedList class from Project 3. The stack should not deal with ListNodes directly.

Your Stack should use a generic type 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 performs the algorithm detailed at the top of this document. That is, it reads in a token, identifies it, makes a new Token type from it, and 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.

A word 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 this project, it may seem that it would be quicker to code this algorithm 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 you cannot 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 do I start??? There are a lot of pieces to this assignment. Use top-down design to help you manage it all. Here's a good way to approach this:

1. I'm giving you a bunch of code to begin with. So studying and testing it should be your first priority. Understand the interface. Check out the input. Process the input on paper first so you'll (1) know what the right 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 your Stack class early. Use the testing skills you've been developing in the last two assignments to make sure your Stack really works before you do things that rely on it.

3. The primary algorithm at the top of this document is handled by Converter. Get a good understanding on exactly what this class is doing on its own and what tasks it is subcontracting out to other methods. 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 simply let each token "handle" itself. So once you've determined that a token is, say, "+", you'll turn it into a Plus object, and 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. 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 going on (and come see me 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 Submit

Be sure to include the honor code affirmation in the comments of one of the Client class: *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.*

Submit *all of your .java* files to Gradescope. That includes all of the .java files that were part of the starter code. You don't have to submit the input file.

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

1. Are all of your files properly commented?

2. Are your files formatted neatly and consistently?

3. Did you clean up your code once you got it to work? It should not contain any code snippets that don't contribute to the purpose of the program, commented out code from earlier attempts, or unnecessary comments.

4. Do you use variable and method names that are informative?

5. Did you get rid of all magic numbers?

6. Does your code practice good information hiding?

7. Does your code make use of already existing public methods or private helper methods to modularize complex tasks and to minimize the number of methods that have to access instance variables?

Finally, submit your files to Gradescope (*Project 4: Infix to Postfix Converter*).

6 Gentle Reminder

Programming assignments are *individual* projects. I encourage you to talk to others about the general nature of the project and ideas about how to pursue it. However, the technical work, the writing, and the inspiration behind these 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 have 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.

7 Grading guidelines

- Correctness: Your program does what the problem specifications require. (Based on Gradescope tests.)
- Testing: The evidence submitted shows that the program was tested thoroughly. The tests run every line of the code at least once. Different input scenarios, and especially border cases, were tested. **A note on testing:** I am expecting at least a completed version of StackTest.java. If you write tests for the token classes, please submit those, too.
- 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. **The ADTs' invariants are documented.**
- Programming techniques: Programming constructions have been used appropriately in such a way that make the code efficient and easy to read and maintain. If the project required the use of a specific technique or algorithm, this has been correctly implemented. For this project, I will particularly look for good information hiding and modularity, robustness, and that the stated invariant is 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 help structure the code, formatting is consistent and the code is well organized.