CSC 115: Fundamentals of Programming II
Assignment #3: Stacks

Due date

Monday, July 10, 2017 at 9:00 am via submission to conneX.

How to hand in your work

Submit the requested files through the Assignment #3 link on the CSC 115 conneX site. Please make sure you follow all the required steps for submission – *including confirming your submission!*

Learning outcomes

When you have completed this assignment, you will have learned:

- How to create a generic implementation of ~~*StackLink*~~ *StackLinked* (i.e., linked-list implementation of the *Stack* ADT) using *reference-based nodes*.
- How to use your stack implementation to evaluate a postfix expression.
- How to use your stack implementation – plus Java's built-in *LinkedList* class – to write an infix-to-postfix expression converter.

Task 1

Ref-based Stack implementation

During lectures and labs we have examined the Stack ADT and the principle operations it defines (e.g., *push*, *pop*, *isEmpty*, etc.). An ADT definition itself does not provide an implementation, so in this assignment you will actually implement a *Stack* using the ref-based nodes approach we have seen earlier for linked lists.

One further improvement, however, is that we will also write ~~*StackLink*~~ *StackLinked* such that it will be a generic type. That is, when you write code instantiating a ~~*StackLink*~~ *StackLinked*, you will also be able to indicate what element type is to be stored in the stack instance. We have provided four Java files:

- *Stack.java*: The definition of the *Stack* ADT interface (not be modified);
- *StackNode.java*: A helper type that will be used to implement *StackLink* (not to be modified);
- *StackEmptyException.java*: Needed for some operations in *Stack*, i.e., *pop* and *peek* (not to be modified); and

- ***~~StackLink.java~~ StackLinked.java*: The file you will complete for Task 1.**

Your task is to complete the implementation of ~~*StackLink*~~ *StackLinked* using a ref-based node approach. Tests 1 through 22 in *A3test.java* are given to guide your implementation effort. To simplify reasoning about your implementation these tests use *Integer* when instantiating a ~~*StackLink*~~ *StackLinked*.

Note: You **are not permitted** to use the JCF's *Stack* implementations to complete your program.

Task 2

Using ~~StackLink~~ StackLinked to implement a postfix-expression evaluator

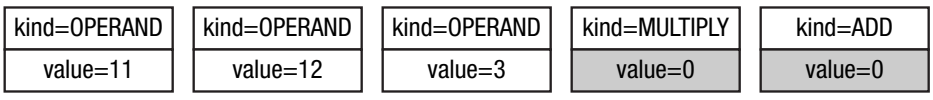
Another topic in lectures about stacks is their use in problem solving. One of these problems involves the evaluation of postfix expressions. In the solution to this problem, a stack is used to store operands and the results of operations, with successful expression evaluation resulting in a single integer on the stack. (Please review the lecture slides for details on how this algorithm works.)

In what follows three classes are mentioned (and these are provided to you):

- *PostfixMachine.java* (completed in tasks 2 and 3)
- *ExpressionToken.java* (not to be modified)
- *SyntaxErrorException.java* (not to be modified)

In this task and the next, you will need to implement the methods *eval()* and *infix2postfix()*, and we strongly recommend you first implement *eval()* (i.e., complete task 2 before task 3).

You are provided a complete method named *tokenize()* in *PostfixMachine*. This method accepts a string as a parameter (e.g., "11 12 3 * +") and returns a *LinkedList* that is a sequence of *ExpressionToken* instances. The sequence in the list for the "11 12 3 * +" is shown below (where the left-most token is at the head of the list):



The implementation of *eval()* has one parameter that is a *LinkedList*, itself created by some earlier call to *tokenize()*, and *eval()* evaluates the expression represented by that incoming list. Code may directly access the *kind* and *value* fields in an *ExpressionToken*. (If a token is not an operand, then the value stored in that token should be ignored as is hinted at in the figure above where the *value* area is greyed-out.)

Note *tokenize()* ***does not*** check for expression validity. It is possible that poorly-formed postfix expressions may be constructed (e.g., "31 29 + -" where there are too few operands; "3 7 8 -" where there are too many operands; "210 67 ^" where an unrecognized operator is given; and "123 0 45 * /" where a division-by-zero occurs).

Your goal for task 2 is to complete the implementation of the *eval()* method in the class *PostfixMachine.java*. Note all of the methods in this class are marked *static*, and also there is no *main()* method in the class. We can use methods in this class by naming the class and then using dot notation just as if we called a method on some object. (See the code in *testPostfixEvaluatorBasic()* within *A3test.java* for examples of this syntax.) Tests 23 to 29 check behavior on well-formed postfix expressions.

Read the "Purpose" and "Example" comments before the body of *eval()* for more information on the errors to be detected by your implementation and how your code must behave. Tests 30 through 33 check for this error-detection behavior.

Note: You **must** use your ~~*StackLink*~~ *StackLinked* implementation to store operands and operation results. However, you are free to decide what is to be stored in the stack (an *Integer*? an *ExpressionToken* of kind=OPERAND? etc.)

Task 3

Infix to postfix conversion

Your last task is to apply the use of your *StackLinked* and a *List* to convert an infix expression into postfix. This code must be located in the *infix2postfix*() method within *PostfixMachine.java*. The method has one parameter which is a *LinkedList* of *ExpressionToken* representing the infix expression, and returns a *LinkedList* of *ExpressionToken* representing the resulting postfix translation.

The algorithm for this conversion has been covered in lectures, and so assembling the details from the slides into suitable loops, selection statements and variables may seem a bit confusing at first. However, the *Purpose* and *Examples* comments with *infix2postfix* have been given in such a way as to hint towards an implementation strategy.

As in Task 2, the *tokenize()* method does not check input strings for validity, and therefore your implementation should check for errors. (Error checking here is much simpler than when processing a postfix expression.) Again, please read the comments placed just before *infix2postfix* for error-detection details.

Tests 34 through 38 check for correctness of *infix2postfix* operation. Tests 39 to 43 combine this conversion with evaluation (i.e., going from a infix expression all the way to completed postfix evaluation.)

Note: You **must** use your implementation of *StackLinked* to store expression tokens. You **must** use the JCF's *LinkedList* class when building up the postfix expression. Working with the built-in *LinkedList* is actually quite straightforward. For example,

here is some code that iterates through the *ExpressionToken* instances returned from some call to *tokenize()*:

```
LinkedList<ExpressionToken> aList = PostfixExpression.tokenize("2 + 2");
for (ExpressionToken et : someList) {
    System.out.println(et);
}
```

which has the same effect as this expressed using the old-school for-loop.

```
LinkedList<ExpressionToken> aList = PostfixExpression.tokenize("2 + 2");
for (int i = 0; i < someList.size(); i++) {
    et = aList.get(i);
    System.out.println(et);
}
```

File to submit:

- ~~*StackLink*~~ *StackLinked*
- *PostfixMachine.java, i.e., completed implementations for eval() and infix2postfix()*

Grading scheme

Requirement	Marks
Submitted files compile without errors or warnings	2
Code passes the first set of test cases (1 to 22)	4
Code passes the second set of test cases (23 to 29)	4
Code passes the third set of test cases (30 to 33)	3
Code passes the fourth set of test cases (34 to 38)	4
Code passes the fifth set of test cases (39 to 43)	1
Code uses follows the posted coding conventions on commenting.	2
Total	**20**

Note: Code submitted must be written using techniques in keeping with the goals and stated restrictions of the assignment. Therefore passing a test is not automatically a guarantee of getting marks for a test (i.e., your solution must not be written such that it hardcodes results for specific tests in *A3test.java* yet would be unable to work with similar tests when different data is used).

In order to obtain a passing grade for the assignment, you must satisfy at least the first three requirements by passing all of their test cases.