SYSC 2100 Algorithms and Data Structures
Winter 2019
Assignment 3: Recursion and Stacks
Due: March 8th, 2019

Name your classes and methods strictly as specified (case sensitive).

1. Design a class named `LanguageRecognizerG` to implement a language recognizer. The `LanguageRecognizerG` class **must** accept strings from the user, and determine recursively (method **`recursiveRecogG`**) whether the string is a word of the *G* language.

   The *G* language has the following grammar:
   <G> = empty string | <E> | <V> <E> | <E> <G> <V>
   <E> = & | #
   <V> = W | A

   The client program (exterior to your class) will read the word from the keyboard as follows:

   ```
   Enter the G-language word to check:
   ```

   Suppose that the user enters the word:

   ```
   ###
   ```

   The client program will then proceed to create an object of your class with the user-entered word and check with one simple call of a method. The client program **should not** implement any result printing at all. That is the responsibility of your class via its methods. **A client program is provided on Page 4. Feel free to use it for your tests!**

   The output should appear as follows:

   ```
   Recursion: Word “###” is NOT a word of the G language
   ```

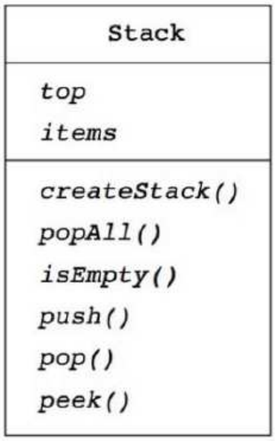
   If the entered word is **`#A`** instead, the output would be:

   ```
   Recursion: Word “#A” IS a word of the G language
   ```

   CAUTION: If you take care of the printing inside **`recursiveRecogG`** you will run into a *multiple printing* problem. To eliminate this, have a second method **`recursivePrintG`** that takes care of the printing for recursion. That is the only method that the client program will call for the language check. It then becomes the job of **`recursivePrintG`** to make use of **`recursiveRecogG`**.

   Bonus Question *(no marks):* Try solving the same problem as above non-recursively **using the Java Collections Framework class *Stack*** (method **`stackRecogG`**).

2. Implement your own ADT-list-based stack class named **`StackListBased`.** **Use the ADT *LinkedList* of the Java Collections Framework.** Your Stack implementation should be capable of performing the operations shown in the following UML diagram.



```
Stack
top
items
createStack()
popAll()
isEmpty()
push()
pop()
peek()
```

Design another class named **`InfixCalculator`** to implement an infix calculator using your previously implemented class **`StackListBased`**. The **`InfixCalculator`** class **must** accept infix expressions from the user and evaluate them with method **`evaluateInfix`**. This method will first convert the infix expression to postfix expression (method **`convertPostfix`**), and then evaluate the resulting postfix expression (method **`getPostfix`**). Use only the operators +, -, *, and /. You can assume that the infix expression is syntactically correct and that the unary operators are illegal. However, the infix expression should

a. allow for any type of spacing between operands, operators, and parentheses
b. allow for multi-digit integer operands

The client program (exterior to your class) will read the infix expression to evaluate from the keyboard as follows:

```
Enter the infix expression to evaluate:
```

Suppose that the user enters the expression:

```
(10 + 3 * 4 / 6)
```

The client program will then proceed to create an object of your class with the user-entered expression and evaluate it the method `evaluateInfix()`.

The output for some example infix operations should appear as follows:

```
infix: (10 + 3 * 4 / 6)
postfix: 10 3 4 * 6 / +
result: 12
```

```
infix: 12*3 - 4 + (18 / 6)
postfix: 12 3 * 4 - 18 6 / +
result: 35

infix: 35 - 42* 17 /2 + 10
postfix: 35 42 17 * 2 / - 10 +
result: -312

infix: 3 * (4 + 5)
postfix: 3 4 5 + *
result: 27

infix: 3 * ( 17 - (5+2))/(2+3)
postfix: 3 17 5 2 + - * 2 3 + /
result: 6
```

Submission Requirements: Submit your assignment (3 source files**: LanguageRecognizerG.java, StackListBased.java, and InfixCalculator.java)** using **cuLearn**. Your program should compile and run as is in the default lab environment, and the code should be well documented. Submit all the files individually **without using any archive or compression**.

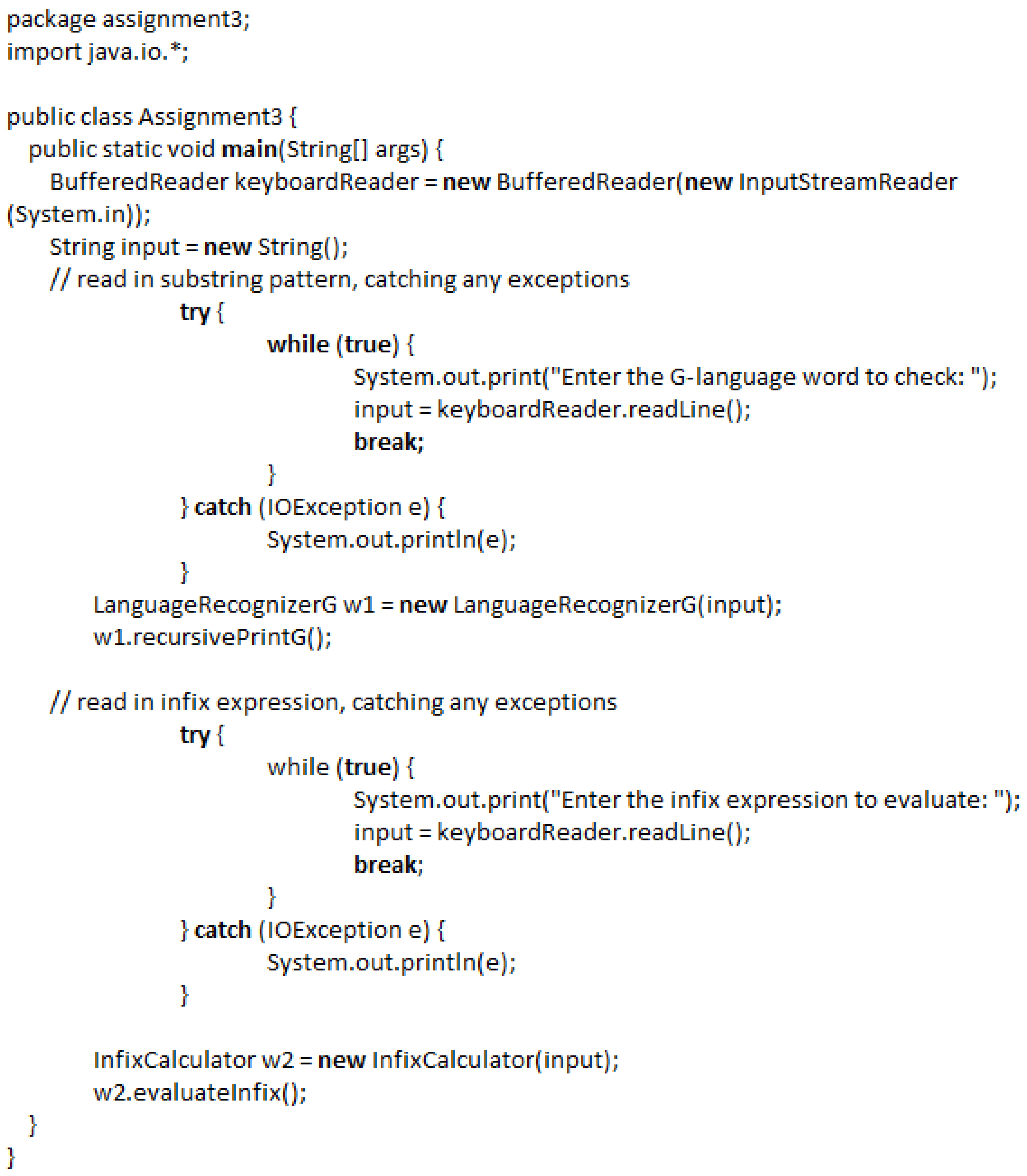
Marks will be based on:

- Completeness of your submission
- Correct solution to the problem
- Following good coding style
- Sufficient and high-quality in-line comments
- Adhering to the submission requirements (in particular the naming convention and the submission of uncompressed source files only)

The due date is based on the time of the **cuLearn** server and will be strictly enforced. If you are concerned about missing the deadline, here is a tip: multiple submissions are allowed. So you can always submit a (partial) solution early, and resubmit an improved solution later. This way, you will reduce the risk of running late, for whatever reason (slow computers/networks, unsynchronized clocks, failure of the Internet connection at home, etc.).

In **cuLearn**, you can manage the submission until the deadline, taking it back, deleting/adding files, etc, and resubmitting it. The system also provides online feedback whether you submitted something for an assignment. It may take a while to learn the submission process, so I would encourage you to experiment with it early and contact the TA(s) in case you have problems, as only assignments properly and timely submitted using **cuLearn** will be marked and will earn you assignment credits.

The Client Program:



```
package assignment3;
import java.io.*;

public class Assignment3 {
  public static void main(String[] args) {
    BufferedReader keyboardReader = new BufferedReader(new InputStreamReader
(System.in));
    String input = new String();
    // read in substring pattern, catching any exceptions
                try {
                        while (true) {
                                System.out.print("Enter the G-language word to check: ");
                                input = keyboardReader.readLine();
                                break;
                        }
                } catch (IOException e) {
                        System.out.println(e);
                }
        LanguageRecognizerG w1 = new LanguageRecognizerG(input);
        w1.recursivePrintG();

    // read in infix expression, catching any exceptions
                try {
                        while (true) {
                                System.out.print("Enter the infix expression to evaluate: ");
                                input = keyboardReader.readLine();
                                break;
                        }
                } catch (IOException e) {
                        System.out.println(e);
                }

        InfixCalculator w2 = new InfixCalculator(input);
        w2.evaluateInfix();
  }
}
```