CSC 212 Programming Assignment # 2
Implementing and Using Queues and Stacks
Due date: 11/07/2019

Guidelines:	This is an **individual** assignment. The assignment must be submitted to **Web-CAT**

In this programming assignment, we will be using queues and stacks to represent and evaluate postfix and infix expressions.

1 Implementing stack and queue

The first part consists in the implementation of the ADT Queue and Stack.

1. Given the following specification of the ADT Queue, implement this data structure using linked representation. You should write the class `LinkedQueue` that implements the interface `Queue`.

   - enqueue (Type e): **requires**: Queue Q is not full. **input**: Type e. **results**: Element e is added to the queue at its tail. **output**: none.
   - serve (Type e): **requires**: Queue Q is not empty. **input**: none. **results**: the element at the head of Q is removed and its value assigned to e. **output**: Type e.
   - length (int l): **requires**: none. **input**: none. **results**: The number of elements in the Queue Q is returned. **output**: l.
   - full (boolean flag): **requires**: none. **input**: none. **results**: If Q is full then flag is set to true, otherwise flag is set to false. **output**: flag.

2. Given the following specification of the ADT Stack, implement this data structure using linked representation. You should write the class `LinkedStack` that implements the interface `Stack`.

   - push(Type e): **requires**: Stack S is not full. **input**: Type e. **results**: Element e is added to the stack as its most recently added elements. **output**: none.
   - pop(Type e): **requires**: Stack S is not empty. **input**: **results**: the most recently arrived element in S is removed and its value assigned to e. **output**: Type e.
   - empty(boolean flag): **requires**: none. **input**: none. **results**: If Stack S is empty then flag is true, otherwise false. **output**: flag.
   - full(boolean flag): **requires**: none. **input**: none. **results**: If S is full then Full is true, otherwise Full is false. **output**: flag.
   - reverse(): **requires**: none. **input**: none. **results**: The entire stack S is reversed **output**: none.

2 Evaluating expressions

1. Implement the interface `PostFixExp` using the class `PostFixExpImp`. This interface represents a postfix expression involving *non-negative* integers. Expressions can include any of the following binary operators : `*,/,%,+,-,<,>,>=,<=,==,!=`.

   Remark 1.

   - *"/" represents integer division and "%" represents the modulo operation on integers.*
   - *The tokens of the expression are separated by a space, for example: `5 3 - 4 2 + *`.*
   - *The comparison operators should return the integer values 0 and 1 to represent false and true.*

   A postfix expression is considered to be *valid* if and only if:

   (a) Each operator is preceded by two postfix expressions. The smallest postfix expression is a singular integer.

   (b) The expression contains no division by or modulo 0.

```
public interface PostFixExp {

  // Set a new expression written in postfix notation. Tokens are
     separated by a space.
  void setExp(String exp);

  // Return the expression in postfix notation. Tokens are separated by
     a space.
  String getExp();

  // Return the operands stack after parsing k tokens. If an error is
     encountered before k tokens are parsed, null is returned. An error
      can be a syntax error or division by zero. Assume that k is valid
     .
  Stack<Integer> evaluate(int k);

  // Check if the expression is valid (no syntax error and no division
     by zero). If the expression is invalid, the index of the token
     that caused the error is returned. If the expression is valid, the
      method returns -1.
  int validate();

  // Return the expression in infix notation. If a syntax error is
     encountered, null is returned (notice that division by zero is not
      considered an error here). Parentheses must be put around every
     operation, like "( ( 3 * 2 ) + ( 6 / 3 ) )". Tokens must be
     separated by a space.
  InFixExp toInFix();
}
```

2. Implement the interface `InFixExp` using the class `InFixExpImp`. This interface represents an infix expression on *non-negative* integers. Expressions can include any of the following operators: `*,/,%,+,-,<,>,<=,>=,==,!=` in addition to the parentheses: `(,)`.

   Remark 2.

   - *"/" represents integer division and "%" represents the modulo operation on integers.*

- *The tokens of the expression are separated by a space, for example:* `6 / 2 * ( 5 - 2 )`.
- *The comparison operators should return the integer values 0 and 1 to represent false and true.*

An infix expression is considered to be valid if and only if:

(a) Each operator is preceded by a valid infix expression and succeeded by a valid infix expression. The smallest infix expression is a singular number.

(b) The expression contains no division by or modulo 0.

(c) The parentheses in the expression are in a valid configuration.

```
public interface InFixExp {

  // Set a new expression written in infix notation. Tokens are
     separated by a space.
  void setExp(String exp);

  // Return the expression in infix notation (without $).
  String getExp();

  // Return the number of tokens in the expression (including $).
  int getNbTokens();

  // Return a pair containing respectively operands and operations
     stacks after parsing k tokens ($ is considered as a token). If an
     error is encountered before k tokens are parsed, null is returned.
      An error can be a syntax error or division by zero. Assume that k
      is valid.
  Pair<Stack<Integer>, Stack<String>> evaluate(int k);

  // Check if the expression is valid (no syntax error and no division
     by zero). If the expression is invalid, the index of the token
     that caused the error is returned. If the expression is valid, the
      method returns -1.
  int validate();

  // Return the expression in postfix notation. If a syntax error is
     encountered, null is returned (notice that division by zero is not
      considered an error here. Tokens must be separated by a space.
  PostFixExp toPostFix();
}
```

Hint 1. *To handle parentheses in the evaluation algorithm:*

- *When you see a '(', push it to the stack.*
- *When you see a ')', pop and perform all operations untill you find a matching '('*

3 Deliverable and rules

You must deliver:

1. Source code submission to Web-CAT. You have to upload the following class in a zipped file:

- `LinkedQueue.java`
- `LinkedStack.java`
- `PostFixExpImp.java`
- `InFixExpImp.java`

Notice that you should **not upload** the interfaces.

The submission **deadline** is: **11/07/2019**.

You have to read and follow the following rules:

1. The specification given in the assignment (**class and interface names, and method signatures**) must not be modified. Any change to the specification results in compilation errors and consequently the mark zero.
2. All data structures used in this assignment **must be implemented** by the student. The use of Java collections or any other data structures library is strictly forbidden.
3. This assignment is an individual assignment. Sharing code with other students will result in harsh penalties.
4. Posting the code of the assignment or a link to it on public servers, social platforms or any communication media including but not limited to Facebook, Twitter or WhatsApp will result in disciplinary measures against any involved parties.
5. The submitted software will be evaluated automatically using Web-Cat.
6. All submitted code will be automatically checked for similarity, and if plagiarism is confirmed penalties will apply.
7. You may be selected for discussing your code with an examiner at the discretion of the teaching team. If the examiner concludes plagiarism has taken place, penalties will apply.