Project 1: Implementation of the Stack ADT and Its Application

**Deadlines:** submit your files via handin by midnight (end of the day) on **Tuesday, 10/07/14.**

**Late submission:** submit your files via handin by midnight (end of the day) on Wednesday, 10/08/14. **Note:** 50 point penalty will be applied.

**IMPORTANT:** To speed up the grading process (the source code inspection of your program), in each class definition you are required to **PLACE all class members** (nested classes, instance variables, methods) **IN THE SAME ORDER as listed in the assignment.**

**DON'T** change anything in the assignment – FOLLOW requirements **precisely.**

I. **Design, implement, and test a generic Stack ADT.**

1) Define a **generic** class **MyStack<T>** to represent a stack implemented as a **linked list.**

*MyStack* has a **private** nested class **Node** with two **public** instance variables - one for the list element and other for the link to the next node in the collection. You may add a constructor if you want to.

*MyStack* has only one **private** instance variable of **Node** type to hold the address of the first node in the linked list.

*MyStack* has one constructor that creates an empty stack.

*MyStack* has the following **public** methods:

- **push:** a void method with a **T** type parameter; the parameter object is added on the top of the stack (to the beginning of the liked list).

- **pop:** has no parameters and returns a **T** type value; the top element is removed from the stack (the first node of the linked list is deleted) and is returned by the method.

  If the stack is empty, throws an **EmptyStackException** type exception1.

- **peek:** has no parameters and returns a **T** type value – the value of the top element (the first element of the list) – without removing it from the list.

  If the stack is empty, throws an **EmptyStackException** type exception1.

- **isEmpty:** has no parameters and returns a **boolean** value – **true** if the stack is empty, and **false** otherwise.

**Note:** this is the exact content of *MyStack* class – it should **NOT** contain additional members except **private** support methods if needed (in this class there won’t be such need).

1 The **EmptyStackException** class is in **java.util** package, so you need to either import the package or refer to the class by its full name.
ATTENTION:
In all following parts of this assignment, as well as in future assignments, if you are using a switch statement, make sure that the switch-expression and consequently the switch-cases are of char type. Note that switch statement with String type expression has been added in Java 7; however, the compilers installed in our labs (and used in grading) do NOT support that option.

2) Define a client class (a driver) StackTest to test the performance of your MyStack ADT

This class has only one member – the main method. In the main method you need to:

1. Create a MyStack object to hold String type values (it will be created as an empty stack).

2. Define a scanner object for keyboard input (i.e. connect it to System.in). You must make a scanner for System.in only once here and use it for all keyboard inputs in this main method. DON’T re-define a new scanner for System.in – you will be penalized if you do.

   Note: re-defining a new scanner for System.in will cause problems in grading (when your program is run with piped input), so DON’T do it.

3. Output the following menu that lists all operations provided by MyStack class:

   Choose one of the following operations:
   - push/add (enter the letter a)
   - pop/delete (enter the letter d)
   - peek (enter the letter p)
   - check if the list is empty (enter the letter e)
   - Quit (enter the letter q)

   Important: do NOT change letters for menu choices: use a, d, p, e, q as assigned.

4. For as long as the user does not choose to quit, do the following:
   - Prompt the user to enter a menu choice (do NOT print the menu here; remember, the menu should be output ONLY ONCE, in step 3, before entering this loop).
   - Input user’s choice. Although you need to input only one letter (a, d, p, e, q), read in the whole line and use the first letter only; this will eliminate any future issues if the user accidentally entered additional symbols here.
   - Analyze user’s choice (use a switch statement with default case which will execute for invalid menu choices). Arrange the execution of the requested operation; if necessary (push operation), ask for an input value and input it from a separate line (DON’T enter the input value on the same line as the user’s choice). After the execution of each request, output one of the following messages respectively: “5 pushed in”, “5 popped out”, “5 on the top”, “empty”, “not empty”, “quitting” (the value 5 is used just for an example). Remember that your stack is defined to hold strings so you should not differentiate numeric values from other strings.

   Important:
   1. If the operation throws an exception (pop and peek), don’t let the program crush or terminate, handle it by outputting a message “Invalid operation on an empty stack”.
   2. The input value for the push method is read as one whole line.
   3. If the user entered a wrong letter for a menu choice, output a message “Invalid choice” and finish the execution of that request (do NOT try to loop in asking to correct it).

5. After the user chooses to quit, finish the program with a farewell message.
II. Design, implement, and test a client class for Stack ADT created in section I.

1) Define a class `Converter` containing the following 2 static methods. This class may NOT contain any additional members except `private` support methods if needed.

**Method1.** The prototype of this method is the following:

```
public static String infixToPostfix (String expression)
```

This method converts the given arithmetic expression in the standard form (i.e. in the `infix` notation) into `postfix` notation. The `infix` expression is given as a parameter and it is a string. The `postfix` expression is the return value and it is a string.

**Precondition:** the parameter’s value is a **valid** infix expression. This means that you should **not** do any validity check for the parameter.

**IMPORTANT:**
1) For the infix to postfix conversion **you are required to use the algorithm** presented in section 3.6.3 of your textbook on pages 87-90 (pages 86-89 in the 2-nd edition).

**Advice:** first read about postfix expressions on pages 85-87 (pp. 84-86 in 2-nd edition).

2) For a Stack ADT **you are required to use the MyStack class** defined in this handout.

**Requirements and clarifications:** in the string parameter containing the `infix` expression:

a. You should expect **FOUR** arithmetic operations: +, –, *, / and also `parenthesis` ( and ).
   i. The precedence of + and – is the same.
   ii. The precedence of * and / is the same and it is **higher** than the precedence of + and –.
   iii. Parentheses are treated as a special case (see the textbook).

b. The operands of the `infix` expression **can be anything** – any number (integer or float), as well as any `variable` name (any “word”) can be an operand. You should **NOT** care about the type of the operand though – **all operands should be treated as strings**.

c. For simplicity, let’s agree that you will always expect to have **at least one space between neighbor terms** (operands, operations, and parenthesis). Thus, each term of the `infix` expression is one token/word of the parameter string (this will make the line breaking/tokenizing very easy – use an object of `Scanner` or `StringTokenizer` class to break the expression into tokens).

**Attention:** there may be **more than 1** space separating neighbor terms.

d. In the return value of the method, which is a string containing the `postfix` expression, all terms (operands and operations) **must be separated by at least one space** (each term is one separate token of the resulting string).

e. There **should NOT be** any parentheses in the resulting `postfix` expression.
**Method2.** The prototype of this method is the following:

```java
public static double postfixValue (String expression)
```

This method evaluates the given postfix expression (calculates its value). The postfix expression is given as a parameter, and its value (double type) is returned by the method.

**Precondition:** the parameter’s value is a valid postfix expression. This means that you should **not** do any validity check for the parameter.

**IMPORTANT:**

1) For the evaluation of the postfix expression **you are required to use the algorithm** presented in section 3.6.3 of your textbook on pages 85-87 (pages 84-86 in 2-nd edition).

2) For a Stack ADT **you are required to use the MyStack class** defined in this handout.

**Requirements and clarifications:** in the string parameter containing postfix expression:

a. As operands in the postfix expression you can have ONLY numbers (integers and/or floats); thus in a postfix expression you can have only numbers and operation signs. **Hint:** you need to treat all numbers as double (every integer can be viewed as a floating point number and can be parsed into double).

b. For simplicity, let’s agree that you will always expect to have at least one space between neighbor terms (operands and operations). So, each term is one token/word of the parameter string (this will make the line breaking/tokenizing very easy – use an object of Scanner or StringTokenizer class to break the expression into tokens). **Attention:** there may be more than 1 space separating neighbor terms.

**Note:** These 2 methods are all that Converter has – it should NOT contain additional members except private support methods if needed.

**ADVICE:**
To get the whole picture, I suggest that you first read everything about both of the abovementioned algorithms on pages 85-90 (pages 84-89 in the second/older edition) in THE order they are presented in the textbook (first read about what a postfix expressions is and how it is evaluated, and then about converting the infix expression to postfix). And only AFTER that, start working on implementing those algorithms.
2) Define a client class (a driver) called *ConTest* for the *Converter*.

The goal of *ConTest* (short for *Converter Test*) is to test if the two methods of *Converter* class work properly. This class contains only main method.

Here is the work of the main method:

1. Define a scanner object for keyboard input (i.e. connect it to System.in). You must make a scanner for System.in only once here and use it for all keyboard inputs in this main method. **DON’T** re-define a new scanner for System.in – **you will be penalized if you do**. 
   **Note:** re-defining a new scanner for System.in will cause problems in grading (when your program is run with piped input), so DON’T do it.

2. Output the following menu:

   Choose one of the following operations:
   - Infix to postfix conversion (enter the letter i)
   - Postfix expression evaluation (enter the letter p),
   - Arithmetic expression evaluation (enter the letter a)
   - Quit the program (enter the letter q)

   **Important:** do NOT change letters for menu choices: use i, p, a, q as assigned.

   **Explanation:** In the third option of the menu (i.e. “Arithmetic expression evaluation”) we need to do the following two steps back to back: (i) convert the given infix expression to postfix, and then (ii) evaluate the resulting postfix expression (the one obtained in step (i)).
   **Note:** to be able to evaluate the arithmetic expression, it should contain only numeric operands.

3. For as long as the user does not choose to quit, do the following:
   - Prompt the user to enter a menu choice (do NOT output the whole menu here).
   - Input user’s choice. Although you need to input only one letter (i, p, a, q), read in the whole line and use the first letter only; this will eliminate any future issues if the user accidentally entered additional symbols here.
   - Analyze user’s choice (use a switch statement with default case which will execute for invalid menu choices). Arrange the execution of the requested operation: ask for the appropriate input expression, input it, and invoke the appropriate method(s) of the *Converter* class. After the execution of each request output the result using one of the following messages respectively: “the postfix expression is: ….”, “the value of the postfix expression is: ….”, and “the value of the arithmetic expression is: ….” (note that in your message the “….” should be replaced with the appropriate result).

   **Important:**
   1. For each request you need to input the expression from ONE separate line (DON’T input it on the same line with the user’s choice).
   2. If the user entered a wrong letter for a menu choice, output a message “Invalid choice” and finish the execution of that request (DON’T try to loop in asking to correct it).
   3. Let’s agree that the user will always give a valid expression for the required operation, so there is no need to perform any validity check on the inputted expression.

5. After the user chooses to quit, finish the program with a farewell message.
TEST YOUR PROGRAM THOROUGHLY: the testing of project assignments is completely on you – test cases will not be provided.

Make sure every operation works properly for different valid expressions – test MANY expression, test relatively LONG expressions containing MANY terms.
- Make sure to give expressions that contain – and / operations in addition to + and *.
- Make sure to give expressions that contain multiple sets of parenthesis.
- Make sure to give expressions that contain operands that are integers and/or floats.
- Make sure to do infix-to-postfix conversion for expressions with non-numeric operands.

**Important:**
Your code should be self-documented:
-- Start each file with a comment-header. The header should start at the first line (do NOT leave empty lines at the beginning of a file) and should include the author’s name and id (as in id@calpoly.edu), as well as the date and the assignment name (e.g. Project 1).

**NOTE:** If you are working in a pair, include the names and ids of BOTH partners.
-- Leave at least one empty line after the header; put one empty line between class members.
-- Provide a description for each class (including nested classes).
-- Provide a description for every method and every constructor.
-- Include a comment explanation next to instance/class variable definitions.

Your source code will be inspected and poor documentation may be noted.

**Submitting your work:**
Turn in ALL your source files (four .java files) electronically via “handin” procedure by the deadline (see the top of the first page of this handout for the due date and time).

The account id is: hg-cpe103

The name of the assignment is: Project1

The name of the assignment for late submissions is: Project1_late.

**Important:** make sure that your programs compile and execute on vogon or on unixX (i.e. unix1, inix2, unix3, etc). So, before submitting, compile and run your program on vogon or unixX machines.