Programming Assignment 5
Introduction to Stacks

1 Objectives

1. Implement stack data structure
2. Implement an application using stacks.
3. Convert from infix to postfix notations.
4. Evaluate the postfix expression.

2 Stack Implementation

Organize your code under package with name

`eg.edu.alexu.csd.datastructure.stack.cs<your-two-digits-class-number>`

and you need to implement the following interface. Your class should inherit from this interface and supply all it's method with the exact signature. No modification is permitted.

You should provide any user interface (UI) for testing the implementation of the Stack. Your user interface should ask for input and process it properly. The UI should include all the above mentioned functionalities. UI should include options for the 5 operations listed before. It can be as follows:
1: Push
2: Pop
3: Peek
4: Get size
5: Check if empty

```java
package eg.edu.alexu.csd.datastructure.stack;

public interface IStack {
    /**
     * Removes the element at the top of stack and returns that element.
     *
     * @return top of stack element, or through exception if empty
     */
    public Object pop();

    /**
     * Get the element at the top of stack without removing it from stack.
     *
     * @return top of stack element, or through exception if empty
     */
    public Object peek();

    /**
     * Pushes an item onto the top of this stack.
     *
     * @param object
     *            to insert
     */
    public void push(Object element);

    /**
     * Tests if this stack is empty
     *
     * @return true if stack empty
     */
    public boolean isEmpty();

    /**
     * Returns the number of elements in the stack.
     *
     * @return number of elements in the stack
     */
    public int size();
}
```

3 Application

You are required to write a JAVA program that takes a symbolic infix expression as input and converts it to postfix notation, enter the values of each symbol, evaluates the postfix expression and outputs the resulting expression and its value. All operands are integer values, but intermediate results could be floating points. Your program should support the operators +, -, * and /. Expressions can contain parentheses. You should use the stack implementation you did in part (A) for any stack data structure needed in the program.

3.1 Postfix Vs. Infix Notations

The standard way of writing expressions is known as infix notation because the binary operator is placed in-between its two operands. Although infix notation is the most common way of writing expressions, it is not the one used by compilers to parse expressions. Instead, compilers typically use a parenthesis-free notation referred to as postfix. In this notation, each operator appears after its operands. The following table shows some infix expressions and their corresponding postfix expressions.

Infix	Postfix
2 + 3 * 4	2 3 4 * +
a * b + 5	a b * 5 +
(1 + 2) * 7	1 2 + 7 *
a * b / c	a b * c /
(a / (b - c + d)) * (e - a) * c	a b c - d + / e a - * c *
a / b - c + d * e - a * c	a b / c - d e * + a c * -

3.2 Evaluating Postfix Experessions

The postfix expression is scanned left-to-right. Operands are placed on a stack until an operator is found. Operands are then removed from the stack, the operation is performed on the operands and the result is pushed on to the stack. The process continues until the end of expression is reached. The following table shows the evaluation of the expression 6 2 / 3 - 4 2 * +.

Next token	Action	Stack
6	Push operand	6
2	Push operand	6 2
/	Pop 2 operands, perform operation and push result	3
3	Push operand	3 3
-	Pop 2 operands, perform operation and push result	0
4	Push operand	0 4
2	Push operand	0 4 2
*	Pop 2 operands, perform operation and push result	0 8
+	Pop 2 operands, perform operation and push result	8

3.3 Conversion between Notations

All variables, constants or numerals are added to the output expression. Left parentheses are always pushed onto the stack. When a right parenthesis is encountered, symbols on top of the stack are popped and copied to the resulting expression until the symbol at the top of the stack is a left parenthesis. At that time, parentheses are discarded.
If the symbol being scanned has a higher precedence than the symbol at the top of the stack, the symbol being scanned is pushed onto the stack and the scan pointer is advanced.
If the precedence of the symbol being scanned is lower than or equal to the precedence of the symbol at the top of the stack, one element of the stack is popped to the output and the scan pointer is not advanced. So, the symbol being scanned will be compared with the new top element on the stack.
When the end of the expression is encountered, the stack is popped to the output.
If the top of the stack is a left parenthesis and the end of the expression is encountered, or a right parenthesis is scanned when there is no left one in the stack, the parentheses of the original expression were unbalanced and an unrecoverable error has occurred.

The following table indicates the conversion of the expression a * (b + c) * d

Next Token	Action	Stack	Expression
a	Output variable		a
*	Push operation	*	a
(	Push left parenthesis	* (	a
b	Output variable	* (	a b
+	Push operation	* (+	a b
c	Output variable	* (+	a b c
)	Pop the stack until left parenthesis is found	*	a b c +
*	Pop operation and push the new operation	*	a b c + *
d	Output variable	*	a b c + * d
	Pop all stack elements		a b c + * d *

3.4 Integration

The core of your expression evaluator application must implement the following interface

```
package eg.edu.alexu.csd.datastructure.stack;

public interface IExpressionEvaluator {
    /**
     * Takes a symbolic/numeric infix expression as input and converts it to
     * postfix notation. There is no assumption on spaces between terms or the
     * length of the term (e.g., two digits symbolic or numeric term)
     *
     * @param expression
     *           infix expression
     * @return postfix expression
     */
    public String infixToPostfix(String expression);

    /**
     * Evaluate a postfix numeric expression, with a single space separator
     *
     * @param expression
     *           postfix expression
     * @return the expression evaluated value
     */
    public int evaluate(String expression);
}
```

4 Deliverables

- You should use your own data structures which were implemented in the previous assignments. Don't use any built-in data structure.
- Take into consideration that your implementation will be used later in the project (if there is), so it has to be fully functional, well documented and reusable. Try very hard to clean up your implementation. Remove all unused variables. Do not write redundant and repeated code.
- You should **VALIDATE** every single input in part (B). Some marks will be deduced if your program crashed on invalid input.
- Write appropriate **JavaDoc** comments for your types (classes), methods and variables. Generate JavaDoc using the **JDK javadoc utility**, and deliver it as **compressed file**

under a folder called "**Materials/InfixPostfix**" besides the **src** folder in your repository. You can read more from this link, and this link.

- Deliver an executable JAR (JRE 1.8) under a folder called "**Materials/InfixPostfix**" besides the **src** folder in your repository.
- Try out your code using http://onlinetester.tk/
- You should work individually.
- Late submission is accepted for only one week.
- Delivering a copy will be severely penalized for both parties, so delivering nothing is so much better than delivering a copy.

Good Luck :)