COSC 1437: Programming Lab & Assignment 05 (Fall 2014)

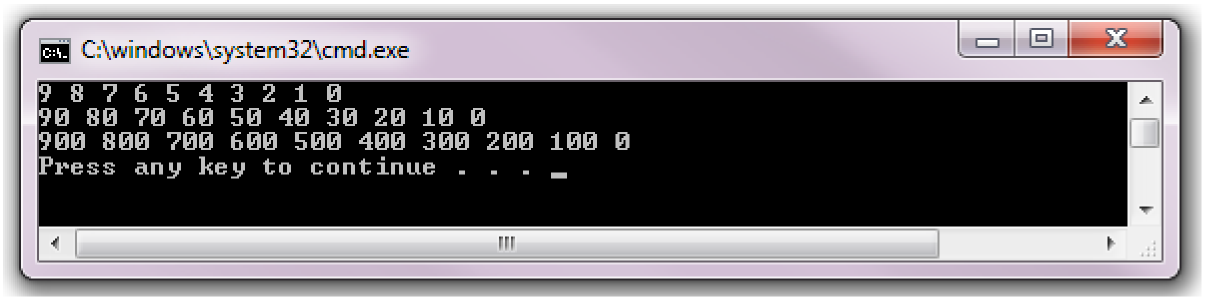
Evaluation of Postfix Expression (using Stack)

Topics

- To do design and implement a calculator for postfix expressions. Your interpreter should read a sequence of postfix expression and print the evaluation result of the expression.
- You are free to use one of the following three stack ADT implementations (in Ch. 7), which include array-based, reference-based, and list-based.

Prerequisite Activities

- Three stack implementations (i.e., "StackArrayBased.java", "StackReferenceBased.java", and "StackListBased.java") and their required files are provided with one simple driver, "TestStack.java" in the zipped file. You don't need to modify code(s) in "TestStack.java"; however, the three Stack implementations are not correct. Therefore, your first activity is to correct the errors in the stack implementation(s), referring to "handout_chap7.pdf".
- If you successfully correct all the errors and run "TestStack.java", you should get the following output:



```
9 8 7 6 5 4 3 2 1 0
90 80 70 60 50 40 30 20 10 0
900 800 700 600 500 400 300 200 100 0
Press any key to continue . . .
```

Main Activities

- A *postfix* expression is an expression in which the operands are followed by the operator. An *infix* expression is an expression in the familiar format, where the operator is in between the operands. Similarly, a *prefix* expression is an expression where the operator comes before the operands.

  For example, the infix expression **`6.3 * (5 - 2 + 4)`** can be written in postfix form as follows: **`6.3 5 2 - 4 + *`**

- Hewlett-Packard pocket calculators used postfix input expression. The programming languages Lisp and Scheme use prefix expressions.

- Your calculator should recognize positive double operands and the four binary operators **`+`**, **`-`**, **`*`**, and **`/`**. For each expression, your calculator should calculate and print its value.

- Your calculator should read from **`System.in`** and write to **`System.out`**. Any error messages for incorrect input or division by zero should be printed to **`System.err`**.

- For example, the input – corresponding to the infix expression **6.3 * (5 – 2 + 4):**
  `6.3 5 2 - 4 + *`
  should result in the following output:
  `6.3 5 2 - 4 + * = 44.1`
  Similarly,
  `12 16 + 4 2 / * 5 +`
  should result in the following output:
  `12 16 + 4 2 / * 5 + = 61.0`

- For reading input from the keyboard, one way is to use the **`nextLine`**() method of the **`Scanner`** class. This will read in the entire line of text. The input may then be parsed (separated) into the individual tokens using the **`split`**() method of the **`String`** class. Allow any white space as delimiter, i.e., one or more spaces, tabs, etc. The expression will be terminated by an enter key.
- The following example illustrates how the `String.split()` method can be used to break up a string into its basic tokens:

```
String test =  "this is a test";
String[] result = test.split("\\s"); // \s is for whitespace characters
for (int x=0; x<result.length; x++)
       System.out.println(result[x]);
```

  prints the following output:

```
this
is
a
test
```

- Another way, if you so desire, is to use the StringTokenizer class. As an example:

```
String test = "this is a test";
StringTokenizer st = new StringTokenizer(test);
while (st.hasMoreTokens()) {
       System.out.println(st.nextToken());
}
```

  prints the following output:

```
this
is
a
test
```

- The evaluation algorithm has been covered in class, but to refresh your memory: When you encounter a number, you push its value on the stack. When you encounter an operator, you pop twice, assign them to operand2 and operand1, respectively, perform the operation to the two operands, and push the result on top of the stack. When you encounter the = sign, you pop the value of the expression off the stack and print it.

- The stack you use for evaluation can be one of the three stack implementations that we discussed in Ch. 7: an array-based; a reference-based; and list-based stack. You need to print an error message if you exhaust the stack space, just in case you are using array-based stack implementation. Stack underflow can occur if the expression is not well-formed. If there is more than one value on the stack when an = is encountered, that is also indicative of a non-well-formed expression.

- Print all error messages to **`System.err`**. Please start all error messages with the string "**`Error:`** ". You may have the following two cases of errors: when you try to pop while stack is already empty (i.e., you have extra operator(s) or missing operand(s)) or when the stack is not empty at the end of all operations done (i.e., you have extra operand(s) or missing operator(s)).

  For example, the following is an error message for a missing operator:

  ```
  Type your postfix arithmetic expression (e.g., 10 20 30 / *): 10 20 30 /
  Error: Invalid postfix expression...
  10 20 30 /
  ```

  Also, the following is an error message for a missing operand:

  ```
  Type your postfix arithmetic expression (e.g., 10 20 30 / *): 10 20 / *
  Error: Invalid postfix expression...
  10 20 / *
  ```

- Note that your "EvaluatePostfix.java" in the zipped file will not be compiled; thus, you make sure you have corrected all the errors of your stack implementation(s). Now, you are asked to implement "isOperator()", "evaluateOperation()", and "evaluateExpression()" methods (shown in the following UML). Please check the correctness of your implementation with the example computation and error conditions given above.

EvaluatePostfix
+ **getExpressionFromKeyboard()**: String + **isOperator(**s: String**)**: boolean + **evaluateOperation(**operand1: double, operand2: double, operator: String**):** double + **evaluateExpression(**expression: String**)**: Double + **main(**args: String[]): void

What to Hand in

- Turn in your program (i.e., "EvaluatePostfix.java") via Blackboard.