The concept of stack is extremely important in computer science and is used in a wide variety of problems. This assignment requires you to write a program that can be used to evaluate ordinary arithmetic expressions that contains any of the five arithmetic operators (+, -, \*, /, %).

This exercise requires three distinct steps, namely:-

1. Verify that the infix arithmetic expression (the original expression), that may contain regular parentheses, **is properly formed as far as parentheses are concerned**.
2. If the parenthesized expression is properly formed, convert the expression from an infix expression to its equivalent postfix expression, called Reverse Polish Notation (RPN) named after the Polish Mathematician J. Lukasiewics.
3. Evaluate the postfix expression, and print the result.

**Step 1 - Verify that the expression**

Given an arithmetic expression, called an infixed expression, to verify that it is properly formed as far as parentheses are concerned, do the following:

* Create an empty stack to hold left parenthesis **ONLY**.
* Scanned the arithmetic expression from left to right, one character at a time.
* While there are more characters in the arithmetic expression

{

If the character is a left parenthesis ‘(‘, push it on to the stack. However if the character is a right parenthesis, ‘)’, visit the stack and pop the top element from off the stack.

}

* If the stack contains any element at the end of reading the arithmetic expression, then the expression was not properly formed.

**Step 2 - Convert infixed expression to postfix**

Given that an arithmetic expression is properly form with respect to parentheses, do the following:

* Create an empty stack to hold any arithmetic operators and left parenthesis, **ONLY**.
* A string to contain the postfix expression – the output from this conversion.
* Scan the arithmetic expression from left to right.
* While there are more symbols in the arithmetic expression,

{

After a symbol is scanned, there are four (4) basic rules to observed and apply accordingly:

1. If the symbol is an operand (a number), write it to the output string.
2. If the symbol is an operator and if the stack is empty, push the symbol on the stack.

Otherwise, if the symbol is either ‘**(**‘ or ‘**)**’, check for the following conditions:

If the symbol is ‘**(**‘, push on to the stack,

Otherwise

If the symbol is ‘**)**’

{

Pop everything from the operator stack down to the **first** ‘(‘. Write each item

popped from the stack to the output string. Do not write the item ‘**)**’. Discard it.

}

1. If the symbol scanned is an arithmetic operator, check for the following and apply accordingly:

If the operator on the top of the stack has higher or equal precedence, that operator is popped from off the stack, and is written to the to the output string. This process is continues until one of two things happen:

* + 1. Either the first ‘(‘ is encountered. When this occurs, the ‘(‘ is removed from the stack and is discarded, and the recently scanned symbol is placed on the stack

**OR**

* + 1. The operator on the stack has lower precedence than the one just scanned, then the recently scanned symbol is pushed onto the stack.

}

4. After the arithmetic expression is exhausted, any operator is remaining on the stack must be popped from off and is written to the output string.

**Step 3 - Evaluate the post fixed expression, if possible**

* Initialize an empty stack to hold operands only.

While there are more symbols in the postfix string

{

* If the token is an operand, push it onto the stack.
* If the token is an operator

{

* Pop the two topmost values from the stack, and store them in the order **t1**, the topmost, and **t2** the second value.
* Calculate the partial result in the following order **t2** **operator t1**
* Push this partial result on the stack.

NO3TE: If the stack does not have two operands, a malformed postfix expression has occurred, and evaluation should be terminated.

}

}

* When the end of the input string is encountered, the result of the expression is popped from the stack.

NOTE: If the stack is empty or if it has more than one operand remaining, the result is unreliable.

Implement the above two algorithms for the following binary operators: addition +, subtraction -, multiplication \*, division /, and modulus operation %. **All operations are integer operations**. To keep things simple, place at least one blank space between each token in the input string.

|  |
| --- |
| An incomplete copy of what the test class could be liked.  class RPN  {  public static void main(String[] arg)  {  String s[] = {"5 + ) \* ( 2",  " 2 + ( - 3 \* 5 ) ",  "(( 2 + 3 ) \* 5 ) \* 8 ",  "5 \* 10 + ( 15 - 20 ) ) - 25",  " 5 + ( 5 \* 10 + ( 15 - 20 ) - 25 ) \* 9"  };  for (int i = 0; i < s.length; i++)  {  Arithmetic a = new Arithmetic(s[i]);    if (a.isBalance())  {  System.out.println("Expression " + s[i] + " is balanced\n");  a.postfixExpression();  System.out.println("The post fixed expression is " + a.getPostfix());  a.evaluateRPN();    **/\***  **Supply the necessary codes to determine the**  **validity of your answers**  **\*/**  }  else  System.out.println("Expression is not balanced\n");  }  }  } |

Define the following the class Arithmetic with principal methods:

1. **isBalance()** determines if the parentheses are balanced
2. **postFixExpression()** that converts the infix expression to postfix expression
3. **evaluateRPN()** that evaluates the postfix expression