Expression Evaluation

Evaluate an expression represented by a String. Expression can contain parentheses, you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, *, and /. Arithmetic Expressions can be written in one of three forms:

Infix Notation: Operators are written between the operands they operate on, e.g. 3 + 4 .

Prefix Notation: Operators are written before the operands, e.g + 3 4

Postfix Notation: Operators are written after operands.

Infix Expressions are harder for Computers to evaluate because of the addional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms. A very well known algorithm for converting an infix notation to a postfix notation is Shunting Yard Algorithm by Edgar Dijkstra. This algorithm takes as input an Infix Expression and produces a queue that has this expression converted to a postfix notation. Same algorithm can be modified so that it outputs result of evaluation of expression instead of a queue. Trick is using two stacks instead of one, one for operands and one for operators. Algorithm was described succinctly on http://www.cis.upenn.edu/ matuszek/cit594-2002/Assignments/5-expressions.htm, and is re-produced here. (Note that credit for succinctness goes to author of said page)

```
1. While there are still tokens to be read in,
   1.1 Get the next token.
   1.2 If the token is:
       1.2.1 A number: push it onto the value stack.
       1.2.2 A variable: get its value, and push onto the value stack.
       1.2.3 A left parenthesis: push it onto the operator stack.
       1.2.4 A right parenthesis:
         1 While the thing on top of the operator stack is not a
           left parenthesis,
             1 Pop the operator from the operator stack.
             2 Pop the value stack twice, getting two operands.
             3 Apply the operator to the operands, in the correct order.
             4 Push the result onto the value stack.
         2 Pop the left parenthesis from the operator stack, and discard it.
       1.2.5 An operator (call it thisOp):
         1 While the operator stack is not empty, and the top thing on the
           operator stack has the same or greater precedence as thisOp,
           1 Pop the operator from the operator stack.
           2 Pop the value stack twice, getting two operands.
           3 Apply the operator to the operands, in the correct order.
           4 Push the result onto the value stack.
         2 Push thisOp onto the operator stack.
2. While the operator stack is not empty,
    1 Pop the operator from the operator stack.
    2 Pop the value stack twice, getting two operands.
    3 Apply the operator to the operands, in the correct order.
    4 Push the result onto the value stack.
3. At this point the operator stack should be empty, and the value
   stack should have only one value in it, which is the final result.
```

It should be clear that this algorithm runs in linear time – each number or operator is pushed onto and popped from Stack only once. Also see http://www2.lawrence.edu/fast/GREGGJ/CMSC270/Infix.html, http://faculty.cs.niu.edu/~hutchins/csci241/eval.htm.

Following is Java implementation of above algorithm.

```
/* A Java program to evaluate a given expression where tokens are separated
   by space.
   Test Cases:
     "10 + 2 * 6"            ---> 22
     "100 * 2 + 12"          ---> 212
     "100 * ( 2 + 12 )"      ---> 1400
     "100 * ( 2 + 12 ) / 14" ---> 100
```

```
*/
import java.util.Stack;

public class EvaluateString
{
    public static int evaluate(String expression)
    {
        char[] tokens = expression.toCharArray();

         // Stack for numbers: 'values'
        Stack<Integer> values = new Stack<Integer>();

        // Stack for Operators: 'ops'
        Stack<Character> ops = new Stack<Character>();

        for (int i = 0; i < tokens.length; i++)
        {
             // Current token is a whitespace, skip it
            if (tokens[i] == ' ')
                continue;

            // Current token is a number, push it to stack for numbers
            if (tokens[i] >= '0' && tokens[i] <= '9')
            {
                StringBuffer sbuf = new StringBuffer();
                // There may be more than one digits in number
                while (i < tokens.length && tokens[i] >= '0' && tokens[i] <= '9')
                    sbuf.append(tokens[i++]);
                values.push(Integer.parseInt(sbuf.toString()));
            }

            // Current token is an opening brace, push it to 'ops'
            else if (tokens[i] == '(')
                ops.push(tokens[i]);

            // Closing brace encountered, solve entire brace
            else if (tokens[i] == ')')
            {
                while (ops.peek() != '(')
                  values.push(applyOp(ops.pop(), values.pop(), values.pop()));
                ops.pop();
            }

            // Current token is an operator.
            else if (tokens[i] == '+' || tokens[i] == '-' ||
                     tokens[i] == '*' || tokens[i] == '/')
            {
                // While top of 'ops' has same or greater precedence to current
                // token, which is an operator. Apply operator on top of 'ops'
                // to top two elements in values stack
                while (!ops.empty() && hasPrecedence(tokens[i], ops.peek()))
                  values.push(applyOp(ops.pop(), values.pop(), values.pop()));

                // Push current token to 'ops'.
                ops.push(tokens[i]);
            }
        }

        // Entire expression has been parsed at this point, apply remaining
        // ops to remaining values
        while (!ops.empty())
            values.push(applyOp(ops.pop(), values.pop(), values.pop()));

        // Top of 'values' contains result, return it
        return values.pop();
    }

    // Returns true if 'op2' has higher or same precedence as 'op1',
    // otherwise returns false.
    public static boolean hasPrecedence(char op1, char op2)
    {
        if (op2 == '(' || op2 == ')')
            return false;
```

```
        if ((op1 == '*' || op1 == '/') && (op2 == '+' || op2 == '-'))
            return false;
        else
            return true;
    }

    // A utility method to apply an operator 'op' on operands 'a'
    // and 'b'. Return the result.
    public static int applyOp(char op, int b, int a)
    {
        switch (op)
        {
        case '+':
            return a + b;
        case '-':
            return a - b;
        case '*':
            return a * b;
        case '/':
            if (b == 0)
                throw new
                UnsupportedOperationException("Cannot divide by zero");
            return a / b;
        }
        return 0;
    }

    // Driver method to test above methods
    public static void main(String[] args)
    {
        System.out.println(EvaluateString.evaluate("10 + 2 * 6"));
        System.out.println(EvaluateString.evaluate("100 * 2 + 12"));
        System.out.println(EvaluateString.evaluate("100 * ( 2 + 12 )"));
        System.out.println(EvaluateString.evaluate("100 * ( 2 + 12 ) / 14"));
    }
}
```

Output:

```
22
212
1400
100
```

See this for a sample run with more test cases.