CS 202 Lab/Programming Assignment 3
Due: Before the class on February 18, 2015

It is a non-trivial task for a computer system to evaluate complex expressions in infix form, such as 12+34*(10+5)-3*4, because the precedence rules would require certain operations to be performed before others. (That is why prefix and postfix expressions are preferable by computer.) A simple method to resolve this complexity is to require all operations to be fully parenthesized. That is, every pair of operands and their operator must be enclosed by parentheses. The above expression would be written ((12+(34*(10+5)))-(3*4)).

But how do you evaluate a fully parenthesized expression? You can use several **stacks**. Your program should scan an input string, from the left to right. If the next character is a blank space, just ignore it. If the next character is a digit, it must be an operand. We should get the number (which may consist of one or more digit characters) and save its value, so push it onto an operand stack. (But, why do we use a stack here? Can we simply put it in a variable, or into an array?) If the next character is an operator, we need to save it somewhere because an operator can be performed only when we have two operands. So, push it onto an operator stack. (Again, why do we use a stack here?) If the next character is a left bracket ("(", "[" or "{"), save it onto the third stack until the matching right bracket is read. If the next character is a right bracket ")", "]" or "}", then we have reached the innermost parentheses, so pop an element from the bracket stack to see whether there is a match. If there is a match, pop two elements of the operand stack, pop the operator stack, perform the operation, and push the result onto the operand stack. At the end of the expression, the bracket stack and the operator stack should be empty and the operand stack should contain a single number which is the final value of the expression.

The input to your program consists of an unknown number of expressions, each consists of blank spaces, digits, operators (+, -, *, and /) and characters representing the there types of brackets (), [], and {}. Only integers are used in this program and / is the integer division.

Your program should interactively read in the input data file, read and process each statement from the data file. Run your program using the file **Lab3.data**, which available on Kodiak.

For each line of the input, your program should either display the result of that expression, or a message "Syntax Error" if that expression is invalid.

A sample input and the corresponding correct output are shown as follows:

Sample Input

```
( 100 * 20 )
[12345-345]
(55  -  22))
([1- (20 / 3])*4)
([1-(20/3)]*4)
50+60
((  50 +  60 ) *  123)
```

Sample Output

```
( 100 * 20 ) = 2000
[12345-345] = 12000
(55  -  22)) = Syntax Error
([1- (20 / 3])*4) = Syntax Error
([1-(20/3)]*4) = -20
50+60 = Syntax Error
((  50 +  60 ) *  123) = 13530
```

Submit your program source code **.java** to Kodiak. Be sure to **comment your code** and put **your name** and a **brief description** at the beginning of your program.

Hand in the **output** and printout of source code **.java file**. Staple the output on top of Lab 3 source code printout. On the output, be sure to put **your name**, and "**Output from Lab3**" on top of the printout.

Since your program will need a stack of integers, and two stacks of characters, you will simply use the **generic Stack** class defined in the **JCF** (**Java Collections Framework**). Note that this class does not have a depth() method. Instead, it has a **size**() method inherited from the **Vector** class.

Here is a skeleton of the program:

```
import java.util.Stack;  // to use the generic stack class

public class Lab3
{
     public static void main(String[] args)
     {
          ...  // declaration of Strings and other variables

          read in data file name;

          open the data file;

          while there is more string to read
          {
               read next expression;
               print the expression;
               pass that expression to the method process
          }
     }


     static void process (String exp)
     {
          Stack<Integer> operands = new Stack<Integer>();
          declare a stack of characters to store operators;
          declare a stack of characters to store left brackets;
          declare other necessary variables

          set a boolean variable(let's call it "valid") to true;
          start from the beginning of the exp string;

          use a while loop or a for loop to step through
               each character of the expression
          {
               get the next character of the string;

               if the next character is a space,
                    just continue the next iteration;
               else if the next character is a left bracket,
                    push it onto the left bracket stack;
```

```
        else if the next character is a digit
        {
             it must be an operand, we need to continue
                  to read the remaining digit characters
                  and convert them to an integer until
                  the end of string is reached or next
                  character is not a digit;
                  (How?)
             push the operand (its integer value)
                  onto the operand stack;
        }

        else if the next character is an operator
             push it onto the operator stack;

        else // it must be a right bracket
        {    // we now reach the innermost parentheses,
             // so we now need to check the left bracket
             // that was saved

             if the left bracket stack is not empty
             {
                  pop the stack;

                  if the brackets match, then
                  {
                       if the operand stack has 2 or more
                            elements, then
                            pop top two elements of the
                            operand stack;
                       else
                            set valid to false and exit the
                            loop;

                       if the operator stack is not empty
                            pop the operator stack;
                       else
                            set valid to false and exit the
                            loop;

                       perform the operation;
                       push result to the operand stack;
                  }

                  else
                       set valid to false and exit
                            the loop;
             }
             else
                  set valid to false and exit the loop;
        }
   }
```

```
            if the expression is valid, the left bracket stack and
                 the operator stack are both empty, and the operand
                 stack contains only one element, then
            {
                 pop the operand stack;
                 print the value;
            }
            else
                 print an error message;
       }


       static boolean isLeftBracket (char c)
       {
            return true or false depending whether c is a
            left      bracket '(', '[' or '{';
       }


       static boolean isOperator (char c)
       {
            return true or false depending whether c is an
            operator '+, '-', '*' or '/';
       }
  }
```

Note that in the process function you also need to check whether or not a character is a digit. You do not need to write a function for it because the Character class already has such a **static** method, which can be used as follows:

```
if (Character.isDigit(c))

     ...
```

The Character class also has other similar methods, such as isUpperCase, isLowerCase, isLetter, etc.