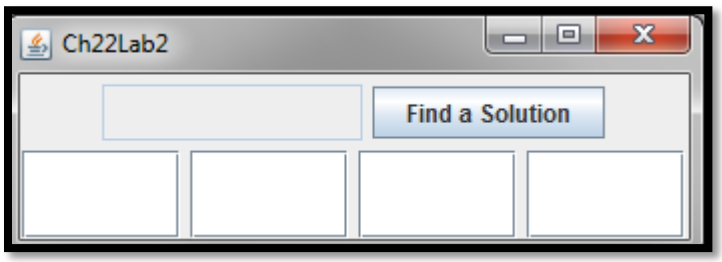
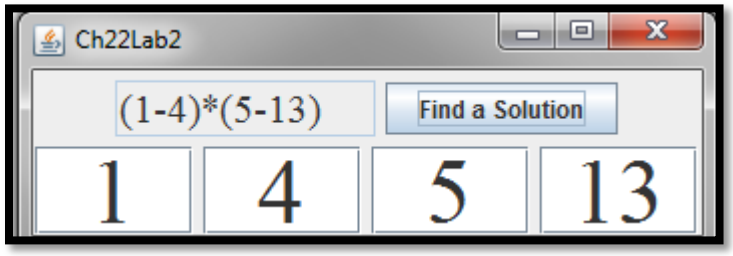
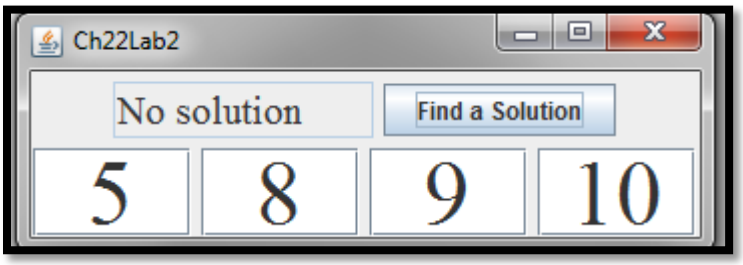
CIT 249
Chapter 22 Lab2

For this lab we will complete #17 on page 827. Instead of our typing in an expression we type in numbers and click the "Find a Solution" and a solution is displayed if there is a combination that equals 24, otherwise it will display no solution. The following are figures demonstrating this:







Let's get started.

1. Open a new document and save the file as Ch22Lab2.java.

2. First we need to import several predefined classes by typing:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
```

3. Type the class header and opening brace having the class extend the JApplet class. This application will be set to run as either an applet or a standalone application.
4. Next we'll construct or text fields. Type:

```
private JTextField jcCard1 = new JTextField();
private JTextField jcCard2 = new JTextField();
private JTextField jcCard3 = new JTextField();
private JTextField jcCard4 = new JTextField();

private JTextField jtfSolution = new JTextField();
```

   - Since no size was designated the text fields will be adjusted according to the width of the frame.

5. Type the constructor method header and opening brace.
6. Constructor our button by typing:

```
JButton jbtFindSolution = new JButton("Find a Solution");
```

   - This will set the text to display on the button as well.

7. Construct two panels. Type:

```
// Creates Grouping Panels
JPanel topPanel = new JPanel();
JPanel cardPanel = new JPanel(new GridLayout(1, 4, 5, 5));
```

   - The topPanel will maintain its default layout of FlowLayout, left to right,.
   - The cardPanel has its layout set to GridLayout with 1 row, 4 columns and 5 horizontal and vertical pixels between components added to the panel.

8. We make some adjustments to one of our text fields. Type:

```
// Sets interface variables
jtfSolution.setEditable(false);
jtfSolution.setHorizontalAlignment(SwingConstants.LEFT);
jtfSolution.setColumns(9);
jtfSolution.setFont(new Font("Times New Roman", Font.PLAIN, 20));
```

- The text field is set so that it cannot be edited.
- Its horizontal alignment has been set to left alignment.
- It has been set to 9 columns.
- Its font has been set to "Times New Roman", normal of 20 points.

9. We add components to the panels by typing:

```
// Adds all interface items to panels
topPanel.add(jtfSolution);
topPanel.add(jbtFindSolution);
cardPanel.add(jcCard1);
cardPanel.add(jcCard2);
cardPanel.add(jcCard3);
cardPanel.add(jcCard4);
```

10. We set the horizontal alignment for several text fields by typing:

```
jcCard1.setHorizontalAlignment(JTextField.CENTER);
jcCard2.setHorizontalAlignment(JTextField.CENTER);
jcCard3.setHorizontalAlignment(JTextField.CENTER);
jcCard4.setHorizontalAlignment(JTextField.CENTER);
```

11. We construct a new Font object and set this font to 4 of the text fields. Type:

```
Font font = new Font("Times New Roman", Font.PLAIN, 45);
jcCard1.setFont(font);
jcCard2.setFont(font);
jcCard3.setFont(font);
jcCard4.setFont(font);
```

12. We add the panels to the applet/frame. Type:

```
// Adds panels to applet
add(topPanel, BorderLayout.NORTH);
add(cardPanel, BorderLayout.CENTER);
```

13. We add a listener to the button. Type:

```
jbtFindSolution.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        jtfSolution.setText(findSolution());
    }
});
```

- Here, when the button is pressed the findSolution() method is invoked adn the jtfSolution text field is set to display the solution.

14. Close the constructor method.
15. For the main method I will simply give you the code. By this time you should be able to understand everything. Type:

```
/** main method */
public static void main(String[] args)
{
    JFrame frame = new JFrame();
    Ch22Lab2 applet = new Ch22Lab2();

    frame.add(applet);
    frame.setLocationRelativeTo(null);

    frame.setTitle("Ch22Lab2");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(350, 120);
    frame.setVisible(true);
}
```

16. The findSolution() method is responsible for all the work. First type the method header and opening brace of:

```
public String findSolution()
{
```

17. Construct 4 integers that acquire their values from the 4 text fields. Type:

```
int a = Integer.parseInt(jcCard1.getText().trim());
int b = Integer.parseInt(jcCard2.getText().trim());
int c = Integer.parseInt(jcCard3.getText().trim());
int d = Integer.parseInt(jcCard4.getText().trim());
```

- Here we use the trim() method to trim away any white space.

18. Create three String variables, the last one is a String array. Type:

```
String noSolution = "No solution";
String solution;
String[] operators = {"+", "-", "*", "/"};
```

- The String array holds the different operators.

19. We create a multi-dimensional array that has every possible combination. Type:

```
int[][] allCombinations = { { a, b, c, d }, { d, a, b, c },
{ c, d, a, b }, { b, c, d, a }, { a, b, d, c }, { c, a, b, d },
{ d, c, a, b }, { b, d, c, a }, { a, d, c, b }, { b, a, d, c },
{ c, b, a, d }, { d, c, b, a }, { a, c, b, d }, { d, a, c, b },
{ b, d, a, c }, { c, b, d, a }, { b, a, c, d }, { d, b, a, c },
{ c, d, b, a }, { a, c, d, b }, { a, d, b, c }, { c, a, d, b },
{ b, c, a, d }, { d, b, c, a } };
```

20. We have three for-each loops. Since none of them have opening braces they will run only the line of code that follows the loop header. Type:

```
for (String firstOp : operators)
    for (String secondOp : operators)
        for (String thirdOp : operators)
```

- Notice that this is for the operators.

21. We also have another for-each loop and two regular for loops. Type:

```
for (int[] cardNums : allCombinations)
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 5; j++)
        {
```

22. We continue with an if statement and a nested if statement. Type:

```
if (i == 0)
{
     if (j == 0)
     {
        solution = cardNums[0] + firstOp + cardNums[1] + secondOp + cardNums[2] +
                    thirdOp + cardNums[3];  //place on single line
            if (EvaluateExpression.evaluateExpression(solution) == 24)
                return solution;
    }
```

- if *i* equals zero we will first check to see if *j* also equals zero. If *j* equals zero then the variable *solution* is set to equal the first element in the *cardNums* array(in for-each loop), plus the *firstOp* and so on.
- if what is returned by the EvaluateExpression class' evaluteExpression() method equals 24 we return the *solution*. Notice that we pass the value of the variable *solution* to this method.

23. We continue with several else-if statement before closing the other if statement. Type:

```
else if (j == 1)
   {
    solution = "(" + cardNums[0] + firstOp
       + cardNums[1] + ")" + secondOp
       + cardNums[2] + thirdOp
       + cardNums[3];
    if (EvaluateExpression
```

```
      .evaluateExpression(solution) == 24)
    return solution;

}
else if (j == 2)
{
  solution = cardNums[0] + firstOp + "("
      + cardNums[1] + secondOp
      + cardNums[2] + ")" + thirdOp
      + cardNums[3];
  if (EvaluateExpression
      .evaluateExpression(solution) == 24)
    return solution;

}
else if (j == 3)
{
  solution = cardNums[0] + firstOp
      + cardNums[1] + secondOp + "("
      + cardNums[2] + thirdOp
      + cardNums[3] + ")";
  if (EvaluateExpression.evaluateExpression(solution) == 24)
    return solution;

}
```

```
else if (j == 4)
{
  solution = "(" + cardNums[0] + firstOp
      + cardNums[1] + ")" + secondOp
      + "(" + cardNums[2] + thirdOp
      + cardNums[3] + ")";
  if (EvaluateExpression.evaluateExpression(solution) == 24)
    return solution;

}
}
```

24. Our else-if statements also have nested if statement and are similar to the previous one. Type:

```
else if (i == 1)
 {
  if (j == 0)
  {
   solution = "(" + cardNums[0] + firstOp
       + cardNums[1] + secondOp
       + cardNums[2] + ")" + thirdOp
       + cardNums[3];
   if (EvaluateExpression.evaluateExpression(solution) == 24)
     return solution;

  }
```

```
  else if (j == 1)
  {
    solution = "((" + cardNums[0] + firstOp
        + cardNums[1] + ")" + secondOp
        + cardNums[2] + ")" + thirdOp
        + cardNums[3];
    if (EvaluateExpression.evaluateExpression(solution) == 24)
      return solution;

  }
  else if (j == 2)
  {
    solution = "(" + cardNums[0] + firstOp
        + "(" + cardNums[1] + secondOp
        + cardNums[2] + "))" + thirdOp
        + cardNums[3];
    if (EvaluateExpression.evaluateExpression(solution) == 24)
      return solution;
  }
}
else if (i == 2)
{
  if (j == 0) {
    solution = cardNums[0] + firstOp + "("
        + cardNums[1] + secondOp
```

```
      + cardNums[2] + thirdOp
      + cardNums[3] + ")";
  if (EvaluateExpression.evaluateExpression(solution) == 24)
    return solution;

}
else if (j == 1)
{
  solution = cardNums[0] + firstOp + "(("
      + cardNums[1] + secondOp
      + cardNums[2] + ")" + thirdOp
      + cardNums[3] + ")";
  if (EvaluateExpression.evaluateExpression(solution) == 24)
    return solution;

}
else if (j == 2)
{
  solution = cardNums[0] + firstOp + "("
      + cardNums[1] + secondOp + "("
      + cardNums[2] + thirdOp
      + cardNums[3] + "))";
  if (EvaluateExpression.evaluateExpression(solution) == 24)
    return solution;
```

```
            }
          }
        }
  return noSolution;
}
```

25. Our final two methods are very similar, but not the same as in the first lab, so I will not go into any explanations. Do not copy these methods from the first lab. Instead of the GenericStack class we create a Stack object. Type:

```
    /* Class to evaluate card Expression */
  public static class EvaluateExpression
  {
    /** Evaluate an expression */
    public static double evaluateExpression(String expression)
    {
      // Create operandStack to store operands
      Stack<Double> operandStack = new Stack<Double>();

      // Create operatorStack to store operators
      Stack<Character> operatorStack = new Stack<Character>();

      // Extract operands and operators
      java.util.StringTokenizer tokens = new java.util.StringTokenizer(expression, "()+-/*",
  true);

      // Phase 1: Scan tokens
```

```
while (tokens.hasMoreTokens())
{
  String token = tokens.nextToken().trim(); // Extract a token
  if (token.length() == 0) // Blank space
    continue; // Back to the while loop to extract the next
  // token
  else if (token.charAt(0) == '+' || token.charAt(0) == '-')
  {
    // Process all +, -, *, / in the top of the operator stack
    while (!operatorStack.isEmpty()
        && (operatorStack.peek().equals('+')
          || operatorStack.peek().equals('-')
          || operatorStack.peek().equals('*') || operatorStack
          .peek().equals('/')))
    {
      processAnOperator(operandStack, operatorStack);
    }

    // Push the + or - operator into the operator stack
    operatorStack.push(new Character(token.charAt(0)));
  }
  else if (token.charAt(0) == '*' || token.charAt(0) == '/')
  {
    // Process all *, / in the top of the operator stack
    while (!operatorStack.isEmpty()
```

```
      && (operatorStack.peek().equals('*') || operatorStack
          .peek().equals('/')))
  {
    processAnOperator(operandStack, operatorStack);
  }

  // Push the * or / operator into the operator stack
  operatorStack.push(new Character(token.charAt(0)));
}
else if (token.trim().charAt(0) == '(')
{
  operatorStack.push(new Character('(')); // Push '(' to stack
}
else if (token.trim().charAt(0) == ')')
{
  // Process all the operators in the stack until seeing '('
  while (!operatorStack.peek().equals('('))
  {
    processAnOperator(operandStack, operatorStack);
  }

  operatorStack.pop(); // Pop the '(' symbol from the stack
}
else
{ // An operand scanned
```

```
      // Push an operand to the stack
      operandStack.push(new Double(token));
    }
  }

  // Phase 2: process all the remaining operators in the stack
  while (!operatorStack.isEmpty())
  {
    processAnOperator(operandStack, operatorStack);
  }

  // Return the result
  return ((Double) (operandStack.pop())).doubleValue();
}

/**
 * Process one operator: Take an operator from operatorStack and apply it
 * on the operands in the operandStack
 */
public static void processAnOperator(Stack<Double> operandStack,
    Stack<Character> operatorStack)
    {
  if (operatorStack.peek().equals('+'))
  {
    operatorStack.pop();
```

```
  double op1 = ((Double) (operandStack.pop())).doubleValue();
  double op2 = ((Double) (operandStack.pop())).doubleValue();
  operandStack.push(new Double(op2 + op1));
}
else if (operatorStack.peek().equals('-'))
{
  operatorStack.pop();
  double op1 = ((Double) (operandStack.pop())).doubleValue();
  double op2 = ((Double) (operandStack.pop())).doubleValue();
  operandStack.push(new Double(op2 - op1));
}
else if (operatorStack.peek().equals('*'))
{
  operatorStack.pop();
  double op1 = ((Double) (operandStack.pop())).doubleValue();
  double op2 = ((Double) (operandStack.pop())).doubleValue();
  operandStack.push(new Double(op2 * op1));
}
else if (operatorStack.peek().equals('/'))
{
  operatorStack.pop();
  double op1 = ((Double) (operandStack.pop())).doubleValue();
  double op2 = ((Double) (operandStack.pop())).doubleValue();
  operandStack.push(new Double(op2 / op1));
}
```

```
    }
  }
}
```

26. Compile the class and fix any errors.
27. Run the program.
28. Compress all files into a single zip file and submit to the drop box.