```
/*  -----------------------------------------------------
 *  Node.java
 *  -----------------------------------------------------
 *  Author:  Matthew Ferlaino
 *  Course:        COSC2006A
 *  ID:      169657520
 *  Email:   mferlaino@algomau.ca
 *  Date:           Nov 5, 2018
 *  ----------------------------------------------------- */

public class Node {
        // Variables
        private Object item;
        private Node next;

        // Default Constructor
        public Node (Object newItem, Node next) {
                item = newItem;
                this.next = next;
        }

        // Getters
        public Object getItem() {
                return item;
        }

        public Object getNext() {
                return next;
        }

        // Setters
        public void setItem(Object newItem) {
                item = newItem;
        }
}

/*  -----------------------------------------------------
 *  StackInterface.java
 *  -----------------------------------------------------
 *  Author:  Matthew Ferlaino
 *  Course:        COSC2006A
 *  ID:      169657520
 *  Email:   mferlaino@algomau.ca
 *  Date:           Nov 5, 2018
 *  ----------------------------------------------------- */

public interface StackInterface {
	public boolean isEmpty();
	public void push(Object item);
	public Object pop() throws StackException;
	public void popAll();
	public Object peek() throws StackException;
}
/*  -----------------------------------------------------
 *  StackException.java
 *  -----------------------------------------------------
 *  Author:  Matthew Ferlaino
 *  Course:		COSC2006A
 *  ID:	169657520
 *  Email:	mferlaino@algomau.ca
 *  Date:		Nov 13, 2018
 *  ----------------------------------------------------- */

public class StackException extends RuntimeException{
	public StackException(String s) {
		super(s);
	}
}

/*  -----------------------------------------------------
 *  Stack.java
 *  -----------------------------------------------------
 *  Author:  Matthew Ferlaino
 *  Course:		COSC2006A
 *  ID:	169657520
 *  Email:	mferlaino@algomau.ca
 *  Date:		Nov 5, 2018
 *  ----------------------------------------------------- */
public class Stack implements StackInterface{
	// Variables
	private Node top;

	// Constructor
	public Stack() {
		top = null;
	}

	// Methods
```

```
        // isEmpty()
        public boolean isEmpty() {
                return top == null;
        }

        // push()
        public void push(Object item) {
                top = new Node(item, top);
        }

        // pop()
        public Object pop() throws StackException{
                if (!isEmpty()) {
                        Object temp = top.getItem();
                        top = (Node)top.getNext();
                        return temp;
                }
                else throw new StackException("StackException on pop: stack empty");
        }

        // popAll()
        public void popAll() {
                top = null;
        }

        // peek()
        public Object peek() throws StackException{
                if (!isEmpty()) return top.getItem();
                else throw new StackException("StackException on pop: stack empty");
        }
}
/*  -------------------------------------------------------
 *  ArthimeticConversions.java
 *  -------------------------------------------------------
 *  Author:  Matthew Ferlaino
 *  Course:        COSC2006A
 *  ID:     169657520
 *  Email:   mferlaino@algomau.ca
 *  Date:           Nov 13, 2018
 *  ------------------------------------------------------- */
// Imports
import java.util.StringTokenizer;
import javax.swing.JFrame;
```

```
@SuppressWarnings("serial")
public class Convertor extends JFrame{
    /*
     * 'Convertor' contains the following methods:
     *                  1. precedence(String op1, String op2)
     *
     *                  2. isOperand(String ch)
     *
     *                  3. addSpaces(String str)
     *
     *              4. result(double op1, double op2, String operator)
     *
     *                  5. verifyInfix(String equation)
     *
     *                  6. convertToPostfix(String equation)
     *
     *      7. evalPostfix(String equation)
     */

    /****************************************************** precedence()
method ******************************************************/
    public static boolean precedence(String op1, String op2) {
        /*
         * This is a method which will return 'true' if the second operator
         * is greater than the first, else 'false'.
         *
         * Preconditions: receives two items popped off the stack,
         *                          both of which are operators
         *
         * Postconditions: returns 'true' if op1 < op2 or else it returns 'false'
         *                          if op1 > op2 || op1 == op2
         */

        // Precedence Order (high to low): (,),^,x/,+-


        // if peek() < current : pop until peek() > current || isEmpty()

        switch (op1) {
      case "+":
      case "-":
        return !(op2.equals("+") || op2.equals("-"));

      case "*":
            case "/":
               return op2.equals("^") || op2.equals("(");

            case "^":
               return op2.equals("(");

            case "(":
               return true;

            case ")":
                return false;

            default:
                return false;
         }
      }

      /******************************************************* isOperand()
method *******************************************************/
      public static boolean isOperand(String ch) {
            /*
             * This method will return 'true' if the string passed in
             * is an operand, meaning it is a number not any of the operators.
             * We can use this method to prove the opposite, proving that a string
             * passed in is an operator by negating the boolean it returns:
             * ----------------------------------------------------------------------
             * String ch = "+";
             * if (!isOperand(ch)); // if 'ch' IS NOT and operand it must be an operator
             * ----------------------------------------------------------------------
             *
             * Preconditions: receives a string 'ch' from an equation string
             *
             * Postconditions: returns 'true' if 'ch' is an operand or else it returns 'false'
             *
             */

            // if 'ch' IS NOT equal to any of the following return true, else return false
            if (      !ch.equals("+")
                        && !ch.equals("-")
                        && !ch.equals("*")
                        && !ch.equals("/")
                        && !ch.equals("^")
                        && !ch.equals("(")
                        && !ch.equals(")")
                && !ch.equals(".")
            ) return true;
        return false;
    }

    /***************************************************************** result() method
******************************************************************/
    public static double result(double op1, double op2, String operator) {
        /*
         * This method will return a solution corresponding to statement
         * that is hit in the the switch statement.
         *
         * Preconditions: receives two doubles, op1, op2 (two operands) and string
operator (an operator)
         *
         * Postconditions: returns the evaluation of the operands based on the operator
passed in
         *
         */

        // Variable
        double solution = 0.0;

        // Switch statement which checks which operator we have and performs the
operation
        switch (operator) {
            case "+":
                solution = op1 + op2;
                break;
            case "-":
                solution = op1 - op2;
                break;
            case "*":
                solution = op1 * op2;
                break;
            case "/":
                solution = op1 / op2;
                break;
            case "^":
                solution = Math.pow(op1, op2);
                break;
            case ".":
                break;
        }
        return solution;
    }

    /******************************************************* addSpaces()
method *******************************************************/
    public static String addSpaces(String equation){
        /*
         * In order to evaluate a postfix expression or to convert
         * an infix expression to a postfix expression we must have
         * whitespace between the operators and operands so we
         * can use StringTokenizer to grab all the tokens
         * from the string. Single char indexing using charAt(i)
         * is not effective for double digit numbers.
         *
         * Ex: (2+22)*3 --> ( 2 + 22 ) * 3
         *
         * Preconditions: receives an equation as a string with no
         *                white spaces. Ex: (2+22)*3
         *
         * Postconditions: returns the equation string with added white spaces
         *                 between all operands and operators, making sure
         *                 it doesn't add white spaces between a double
digit number like 22.
         *                 Ex: ( 2 + 22 ) * 3
         *
         */

        String finalEqn = "";

     // Loop for the length of 'equation'
     for (int i = 0; i < equation.length(); i++) {
        // Scope Variables
        int index;
        String temp = "";

        // If the char is a digit
      if(Character.isDigit(equation.charAt(i))){

        // Save the index of i
       index = i;

       while(Character.isDigit(equation.charAt(index))) {
         temp += equation.charAt(index);
         index++;
```

```
               // If our index is pointing to the end of the equation, break
               if (index == equation.length()) break;
             }

             // Concatenate to finalEqn
             finalEqn += temp + " ";

             // Decrement index and assign back to i
             i = --index;
           }

           else if(!isOperand(equation.charAt(i) + "")) finalEqn += equation.charAt(i) + " ";
         }

         return finalEqn;
       }

       /************************************************** verifyInfix()
method **************************************************/
       public static boolean verifyInfix(String equation) {
              /*
               * This method will use a count algorithm to determine
               * if the string the user enters (via the calculator GUI)
               * is a valid infix expression.
               *
               * Preconditions: takes in a string 'equation' which is
               *                           a string the user entered via our calculatorGUI
               *
               * Postconditions: returns a boolean, true if equation is a valid infix
               *                           or false if infix is not valid
               */
              // count
              int count = 0;
              int bracketCount = 0;

              // Add spaces to 'equation' so tokenizer will work
              equation = addSpaces(equation);

              StringTokenizer token = new StringTokenizer(equation);
              while (token.hasMoreTokens()) {
                     String character = token.nextToken();

                     switch(character) {
```

```
                case "(":
                    bracketCount++;
                    break;
                case ")":
                    bracketCount--;
                    break;

                case "+":
                case "-":
                case "/":
                case "*":
                case "^":
                case ".":
                    count--;
                    break;

                default:
                    count++;
            }
            if (bracketCount < 0) return false;
            if (count > 1 || count < 0) return false;
        }
        if (count == 1 && bracketCount == 0) return true;
        return false;
    }

    /********************************************************
convertToPostfix() method
********************************************************/
    public static String convertToPostfix(String equation) {
        /*
         * This method will convert infix notation to postfix notation.
         * Method uses A LOT of string manipulation.
         *
         * Preconditions: takes in a string 'equation' which is a verified
         *                        infix expression Ex: (3*22)+2
         *
         * Postconditions: returns a string 'postfixEqn' which is the
         *                         infix expression converted to a postfix expression
         */


        /*
         * How do we convert from infix to postfix?
         * 1) Every time an operand is encountered in 'equation', append to postfixStr
         *
         * 2) When we encounter a bracket, check to see if its is '(' or ')'
         *  - If we have a front bracket we want to push()
         *  - If we have a close bracket we want to pop() and append to string until we
encounter a '('
         *    on the stack or until we encounter an empty stack
         *
         * 3) When operator is encountered we want to peek()
         *  - If current >= peek() operator push() current
         *  - If current < peek(), pop() and append then push current
         *  * We continue until we find an operator of lower precedence or until stack is
empty *
         *
         * 4) When end of string is reached pop() and append the rest of the stack if the
stack !isEmpty()
         *
         */

        // Variables
        String postfixEqn = "";
        Stack operatorStack = new Stack();


        // Add spaces to 'equation' so StringTokenizer will work
        equation = addSpaces(equation);

        StringTokenizer token = new StringTokenizer(equation);
        while (token.hasMoreTokens()) {
            // Grabs first item from the equation
            String item = token.nextToken();

            // If we encounter an operand concatenate to string 'postfix'
            if (isOperand(item) == true) postfixEqn += item + " ";

            // If we encounter an operator
            else {
                switch (item) {
                    case ")":
                        String peek = operatorStack.peek().toString();

                        // while top of stack is not '(' pop and concatenate
                        while (!peek.equals("(")) {
                            postfixEqn += operatorStack.pop() + " ";
                            peek = operatorStack.peek().toString();
                        }

                        // The while loop will break when a '(' is
encountered, we then need to pop once to discard '(' from the stack
                        operatorStack.pop();
                        break;

                    case "(":
                        // If stack is empty we should push our item
                        if (operatorStack.isEmpty())
operatorStack.push(item);

                        else {
                            peek = operatorStack.peek().toString(); //
top of stack

                            // If the top of the stack has a lower
precedence than item, push item onto the stack
                            if (precedence(peek, item) == true)
operatorStack.push(item);

                            // If top of stack has higher precedence
than item pop until isEmpty() or lower priority operator is found on the stack
                            else {
                                // Pop from stack and append to
output
                                postfixEqn += operatorStack.pop() +
" ";

                                while (!operatorStack.isEmpty()) {
                                    // See the top of the stack
                                    peek =
operatorStack.peek().toString();

                                    // If the top of the stack is
smaller than item
                                    if(precedence(peek, item) ==
true) {

    operatorStack.push(item);
                                        break;
                                    }
                                    //postfixEqn +=
(operatorStack.pop() + " ");
                                }

                                if (operatorStack.isEmpty())
operatorStack.push(item);
                            }
                        }
                        break;

                    // If we hit default then we have one of the following
operators +,-,/,*, ^
                    default:
                        // If stack is empty we should push our item
                        if (operatorStack.isEmpty())
operatorStack.push(item);

                        else {
                            peek = operatorStack.peek().toString(); //
top of stack

                            // If the top of the stack has a lower
precedence than item, push item onto the stack
                            if (precedence(peek, item) == true)
operatorStack.push(item);

                            // If top of stack has higher precedence pop
until isEmpty() or lower priority operator is found on the stack
                            else {
                                // Pop from stack and append to
output

                                postfixEqn += operatorStack.pop() +
" ";

                                while (!operatorStack.isEmpty()) {
                                    // See the top of the stack
                                    peek =
operatorStack.peek().toString();

                                    // If the top of the stack is
smaller than the character
                                    if(precedence(peek, item) ==
true) {
    operatorStack.push(item);
                                        break;
                                    }
                                    //postfixEqn += 
(operatorStack.pop() + " ");
                                }

                                if (operatorStack.isEmpty()) 
operatorStack.push(item);;
                            }
                        }
                        break;
                }
            }

        }

        // After we finish comparing stack items with chars in 'equation', if stack isn't 
empty, pop and append the rest of the stack items to the postfixEqn string
        while (!operatorStack.isEmpty()) postfixEqn += (operatorStack.pop() + " ");

        return postfixEqn;
    }

    /********************************************************* evalPostfix() 
method *********************************************************/
    public static double evalPostfix(String equation) {
        /*
         * This method will convert evaluate our converted postfix expression
         * 
         * Preconditions: takes in a string 'equation' which is a postfix expressions
         *                              containing proper spaces
         *                              Ex: ( 3 * 22 ) + 2
         * 
         * Postconditions: returns a double which is the evaluated
         *                              postfixEqn
         * 
         */

        /*
         * Postfix Evaluation Algorithm
         * 1) Every time an operand is encountered in 'equation', push()
         * 
         * 2) When we encounter a operator, pop 2 items off the stack and evaluate the
expression
         *
         * 3) Push the evaluation onto the stack
         *
         * 4) Once done, pop the final item off the stack, this is the solution
         *
         */

        // Create an instance of ADT Stack
        Stack stack = new Stack();

        // StringTokenizer
        StringTokenizer token = new StringTokenizer(equation);

        while (token.hasMoreTokens()) {
            String item = token.nextToken();

            // 1. Check to see if the char is an operand or not, if it is push it onto the
stack
            if (isOperand(item) == true) stack.push(item);

            // 2. If it is not, pop two operands off the stack, parse them and pass
them into our result() method along with 'item' which is the operator that evaluates them
            else {
                // Operands
                double op2 = Double.parseDouble(stack.pop().toString());   //
second operand
                double op1 = Double.parseDouble(stack.pop().toString());  // first
operand

                // Push the method's result onto the stack
                stack.push(result(op1, op2, item));
            }
        }

        // Return the last item on the stack which is the solution, parse it as a double
        return Double.parseDouble(stack.pop().toString());
    }
}
/*  -----------------------------------------------------
 *  calculatorGUI.java
 *  -----------------------------------------------------
 *  Author:  Matthew Ferlaino
 *  Course:        COSC2006A
 *  ID:     169657520
 *  Email:   mferlaino@algomau.ca
 *  Date:          Nov 13, 2018
 *  ------------------------------------------------------- */
// Imports
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;

public class Calculator extends Convertor{
        // Buttons For Calculator GUI
        /* Operators */
        private JButton addButton = new JButton("+");
        private JButton subButton = new JButton("-");
        private JButton mulButton = new JButton("x");
        private JButton divButton = new JButton("÷");
        private JButton equalsButton = new JButton("=");

        /* Special Cases*/
        private JButton modButton = new JButton("%");
        private JButton decimalButton = new JButton(".");
        private JButton openBracButton = new JButton("(");
        private JButton closeBracButton = new JButton(")");
        private JButton exponentButton = new JButton("x^y");
        private JButton clearButton = new JButton("AC");
        private JButton backButton = new JButton("<--");

        /* Operands */
        private JButton oneButton = new JButton("1");
        private JButton twoButton = new JButton("2");
        private JButton threeButton = new JButton("3");
        private JButton fourButton = new JButton("4");
        private JButton fiveButton = new JButton("5");
        private JButton sixButton = new JButton("6");
        private JButton sevenButton = new JButton("7");
        private JButton eightButton = new JButton("8");
        private JButton nineButton = new JButton("9");
        private JButton zeroButton = new JButton("0");

        // Text Fields
        private JTextField field = new JTextField();
        private JTextField field2 = new JTextField();
```

```
// JPanels
private JPanel p1 = new JPanel(); // will hold our calculator screen
private JPanel p2 = new JPanel(); // will hold a row of calculator buttons
private JPanel p3 = new JPanel(); // will hold a row of calculator buttons
private JPanel p4 = new JPanel(); // will hold a row of calculator buttons
private JPanel p5 = new JPanel(); // will hold a row of calculator buttons
private JPanel p6 = new JPanel(); // will hold our equal button
private JPanel p7 = new JPanel(); // will hold notation screen

// No-arg constructor
public Calculator() {
	// Set the Layout
	setLayout(new GridLayout(7,1));

	// Add panels to the frame
	add(p1);
	add(p2);
	add(p3);
	add(p4);
	add(p5);
	add(p6);
	add(p7);

	// Add to p1
	p1.setLayout(new GridLayout(1,1));
	p1.add(field);

	// Add to p2 the buttons for that row
	p2.setLayout(new GridLayout(1, 5));
	p2.add(sevenButton);
	p2.add(eightButton);
	p2.add(nineButton);
	p2.add(backButton);
	p2.add(divButton);

	// Add to p3 the buttons for that row
	p3.setLayout(new GridLayout(1, 5));
	p3.add(fourButton);
	p3.add(fiveButton);
	p3.add(sixButton);
	p3.add(mulButton);
	p3.add(openBracButton);

	// Add to p4 the buttons for that row
```

```
p4.setLayout(new GridLayout(1, 5));
p4.add(oneButton);
p4.add(twoButton);
p4.add(threeButton);
p4.add(addButton);
p4.add(closeBracButton);

// Add to p5 the buttons for that row
p5.setLayout(new GridLayout(1, 5));
p5.add(zeroButton);
p5.add(decimalButton);
p5.add(clearButton);
p5.add(subButton);
p5.add(exponentButton);

// Add to p6
p6.setLayout(new GridLayout(1,1));
p6.add(equalsButton);

// Add to p7
p7.setLayout(new GridLayout(1,1));
p7.add(field2);


// Add the ActionListener to all buttons
/* Operands */
zeroButton.addActionListener(new ButtonListener());
oneButton.addActionListener(new ButtonListener());
twoButton.addActionListener(new ButtonListener());
threeButton.addActionListener(new ButtonListener());
fourButton.addActionListener(new ButtonListener());
fiveButton.addActionListener(new ButtonListener());
sixButton.addActionListener(new ButtonListener());
sevenButton.addActionListener(new ButtonListener());
eightButton.addActionListener(new ButtonListener());
nineButton.addActionListener(new ButtonListener());

/* Operators */
backButton.addActionListener(new ButtonListener());
mulButton.addActionListener(new ButtonListener());
addButton.addActionListener(new ButtonListener());
subButton.addActionListener(new ButtonListener());
divButton.addActionListener(new ButtonListener());
```

```
        /* Special Cases */
        openBracButton.addActionListener(new ButtonListener());
        closeBracButton.addActionListener(new ButtonListener());
        equalsButton.addActionListener(new ButtonListener());
        decimalButton.addActionListener(new ButtonListener());
        clearButton.addActionListener(new ButtonListener());
        exponentButton.addActionListener(new ButtonListener());

        // Set both textfields to uneditable so user can only enter in data via the GUI
buttons
        field.setEditable(false);
        field2.setEditable(false);
    }

    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String currEqn = field.getText();
            String command = e.getActionCommand();

            switch(command) {
                case "1":
                case "2":
                case "3":
                case "4":
                case "5":
                case "6":
                case "7":
                case "8":
                case "9":
                case "0":
                case "+":
                case "-":
                case "(":
                case ")":
                    field.setText(currEqn + command);
                    break;
                //case ".":

                case "÷":
                    field.setText(currEqn + "/");
                    break;

                case "x":
                        field.setText(currEqn + "*");
                        break;

                    case "x^y":
                        field.setText(currEqn + "^");
                        break;

                    case "AC":
                        // Reset the textfields
                        field.setText("");
                        field2.setText("");
                        break;

                    case "<--":
                        // Backspace
                        if (field.getText() == null || field.getText().equals(""))
break;

                        else {
                            // Remove last the char from the string
                            String newEquation = field.getText().substring(0,
field.getText().length() - 1);
                            field.setText(newEquation);
                            break;
                        }
                    case "=":
                        if (verifyInfix(field.getText()) == true) {
                            // We have a valid verified infix expression, call
convertToPostifx()
                            String postFix = convertToPostfix(field.getText());
                            field.setText("" + evalPostfix(postFix));
                            field2.setText("" + postFix);
                        }

                        else {
                            field.setText("");
                            field2.setText("Error");
                        }
                        break;
                }
            }
        }
}
```