Lab Report for Exercise 7: Fun with Calculators 2

Authors: Pavel Tsvyatkov (**Matr. Nr.:** 559632) & Lukas Glasmacher (**Matr. Nr.:** 571104) | **Date:** 3.12.2019

Lab exercises

1. **Make a copy of one of your Calculators. Make sure that it works before you begin! If neither of you got a Calculator to work, ask colleagues for permission to use theirs. Give them credit in your report!**

We both had a working calculator from the previous exercise, so our first step was to make a copy of it. Then we opened the the project in BlueJ, as the program offers an advantageous perspective on the interconnectedness of the classes. Before having a closer look at the code, we tested it in both decimal and hexadecimal mode to ensure it works correctly.

2. **Rework it to accept a long String of single digits separated by operators that have precedence. The bored may use multi-digit numbers and floating-point or scientific notation, if they please.**

To solve this exercise first we decided to create a new field of type String in the CalcEngine class and called it displayInput.

```
 7 public class CalcEngine {
 8
 9     // The current value (to be) shown in the display.
10     protected String displayInput;
```

We had to rework most of the methods in the class, because in the previous exercise we worked mostly with integers. Now we decided to change the getDisplayValue method. We renamed it to getDisplayInput and it now has String as return type.

```
16     /**
17      * @return The value that should currently be displayed
18      * on the calculator display.
19      */
20     public String getDisplayInput() {
21         return displayInput;
22     }
```

We also didn't need the numberPressed method anymore, since our calculator accepts a String now. We created a new method buttonPressed that takes something of type String as a parameter so when a button is pressed, it gets saved in the displayInput string.
Our method looked like this.

```
24    /*
25     * Pressing a button
26     * Either start a new operand, or incorporate this number as
27     * the least significant digit of an existing one.
28     */
29    public void buttonPressed(String button) {
30        displayInput += button;
31    }
```

We also changed the clear() method and set the displayInput to an empty String.

```
192    public void clear(){
193        displayInput = "";
194    }
```

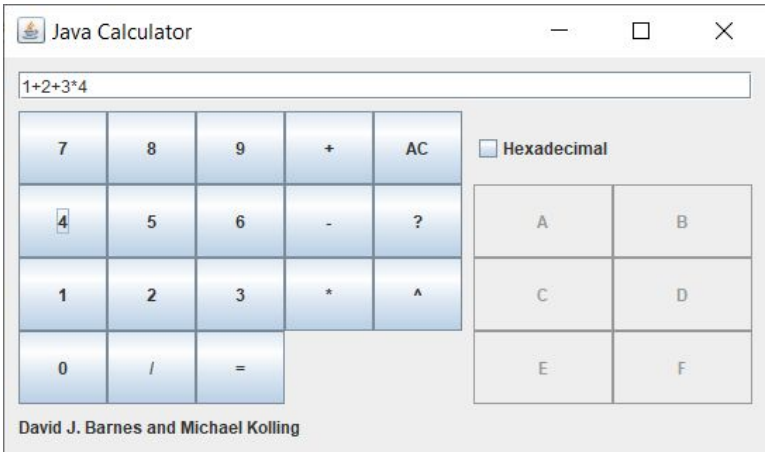
The next thing we had to do is to change our actionPerformed since our calculator doesn't need the plus(), minus(), multiply(), divide() methods. We asked Timo's group and they gave us the idea to use a switch statement for the command. We decided to use it like they proposed. Now we had the commands that we need only and the default case was that a button was pressed. In case equals button was pressed we call the equals() method in the CalcEngine class and we had to make sure we can catch an UnderflowException, if there was one.
Our method looked like this.

```
104    public void actionPerformed(ActionEvent event) {
105        String command = event.getActionCommand();
106
107        switch(command) {
108        case "AC":
109            calc.clear();
110            break;
111        case "?":
112            showInfo();
113            break;
114        case "=":
115            try {
116                calc.equals();
117            } catch (StackUnderflow stackUnderflow) {
118                stackUnderflow.printStackTrace();
119            }
120            break;
121        default:
122            calc.buttonPressed(command);
123        }
```

The last change that we needed to do was in the redisplay method. Instead of calling getDisplayValue, we were now calling our getDisplayInput method.

```
129    /*
130     * Update the interface display to show the current value of the
131     * calculator.
132     */
133    protected void redisplay(){
134        display.setText(calc.getDisplayInput());
135    }
```

After the changes, we went on to try the calculator and saw that it now accepts a String input.



3. **Once you get a String input, add in calls to convert this expression to postfix and evaluate the postfix when = is pressed. Presto, your calculator now takes care of operator priorities, like magic! Just a little bit of mathematical thought, and you can introduce new functionality!**

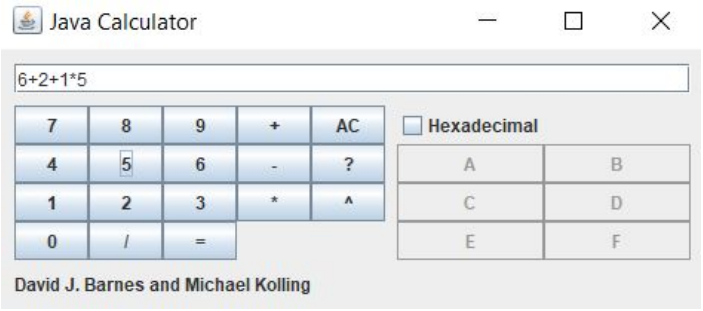
To do this exercise first we had to use the methods from our last Lab exercise. We added the methods **`evaluate`**, **`infixToPostfix`** and **`getPrecedence`** from the **`Postfix`** class into **`CalcEngine`**. We also kept the classes **`StackAsList`**, **`Node`** and **`StackUnderflow`** in addition to the interface **`Stack`**; all of which were required for the needed methods to run properly.

Now we had to change our equals() method in the CalcEngine class, so that it converts an expression to postfix and then evaluates that expression.

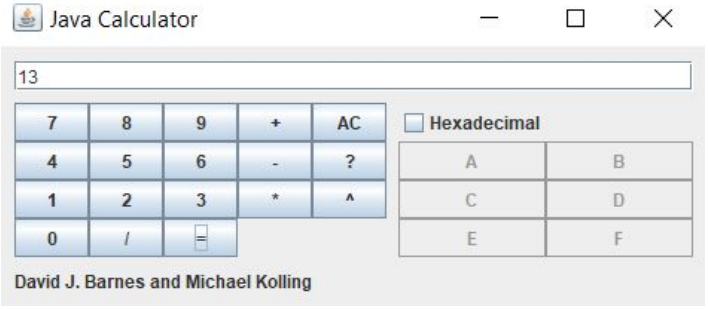
To do this we first call the method infixToPostfix and pass our displayInput as parameter. Then we call the evaluate method to evaluate the expression and also use the toString method, because our displayInput is now of type String. The result is now stored in our displayInput. We made sure to add that our method can throw a StackUnderflow exception.

```
36    public void equals() throws StackUnderflow {
37        displayInput = Integer.toString(evaluate(infixToPostfix(displayInput)));
38    }
```

After that we decided to test our calculator and see if we get the correct result.



After pressing equals, we got the correct result.



4. **What will be good test cases for the precedence of * over +? Find 15 good test cases and try them out, documenting the results.**

To do this exercise we worked in Eclipse and used JUnit. First we created a new Test class and called it CalcEngineTest. First we created a new field CalcEngine that we called calc and then we created a constructor. After that in the setUp method that gets called before each test case we were creating a new

CalcEngine.
The setUp method looked like that.

```
25  @Before
26  public void setUp()
27  {
28      calc = new CalcEngine();
29  }
```

Then we started creating new test cases. We tried to think of different examples, to make sure the precedence of * over + works correctly.

For our first test case we decided to have one assertion before we call the equals method and another assertion after that. It looked like this.

```
40  public void test1() throws StackUnderflow
41  {
42      calc.buttonPressed("1");
43      calc.buttonPressed("+");
44      calc.buttonPressed("2");
45      calc.buttonPressed("+");
46      calc.buttonPressed("3");
47      calc.buttonPressed("*");
48      calc.buttonPressed("4");
49      // Making sure that the getDisplayInput
50      // correctly displays the given input
51      assertEquals("1+2+3*4", calc.getDisplayInput());
52      calc.equals();
53      assertEquals("15", calc.getDisplayInput());
54  }
```

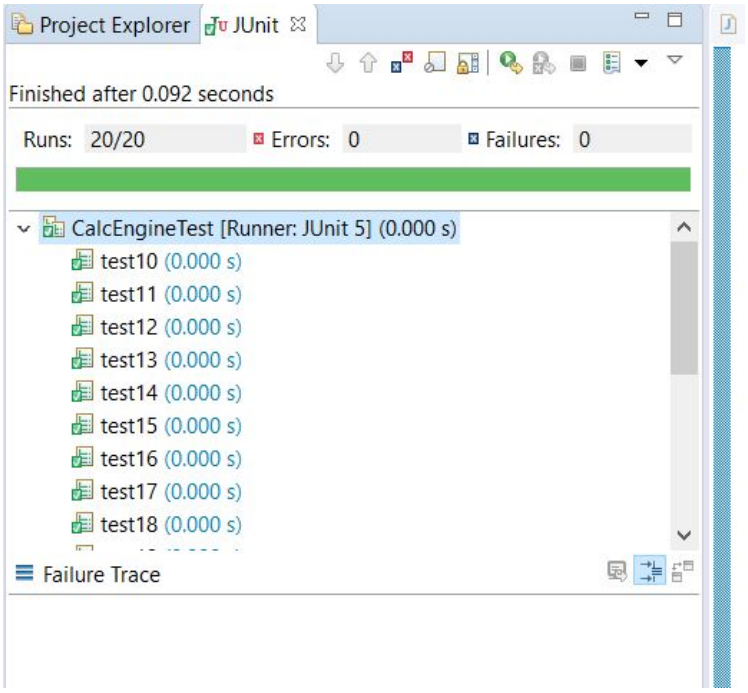
In most of our test cases we used the buttonPressed method to enter the input, but we also decided to have some cases where we set the input directly and call the equals method. We also used division in some of the test cases.

```
320 @Test
321 /*
322  * Testing without buttonPressed
323  * just setting the displayInput to something
324  */
325 public void test15() throws StackUnderflow
326 {
327     calc.displayInput = "6/3+2+3*9+9*9+1*5-9/9";
328     calc.equals();
329     assertEquals("116", calc.getDisplayInput());
330 }
```

We also decided to have some additional test cases where we have exponents and see if our calculator can deal with that correctly.

```
332    @Test
333    /*
334     * Since our calculator also has a button
335     * to set a number to a power we
336     * decided to test how it works
337     */
338    public void test16() throws StackUnderflow
339    {
340        calc.displayInput = "2^3+2*5+7";
341        calc.equals();
342        assertEquals("25", calc.getDisplayInput());
343    }
```

After running the test class we got the following result.



All test cases were passed correctly.

5. **Check that this still works with hexadecimal numbers.**

We noticed that our calculator doesn't work correctly with hexadecimals when calling the evaluate method on our displayString. We took a look at the classes, but at first we were not able to see what had to be changed in order to make the calculator work when we are in hexadecimal mode. We tried different things, but we couldn't find a working solution. We thought that this is because we need to be able to work with multidigits and our calculator was not able to deal with that yet. We decided to ask Timo's group if they were able to solve this exercise. After discussing with them about the exercise, they explained to us what changes they implemented and how they got the calculator to work in hexadecimal mode. We then tried to incorporate the changes as they did.

To be able to work in hexadecimal mode, we had to create a new method called evaluateHex in the HexCalcEngine class. This method is different than the evaluate method that we have in CalcEngine, which is used when we are in decimal mode.

In the HexCalcEngine class, in the evaluateHex method we declared and initialized two new int variables. Now we need to check if the scanned character is a letter. In case it's a letter, we assign the according value to n.

```
41              } else {
42                  //the scanned character is a digit
43                  int num = 0;
44                  int n = 0;
45                  // check if c is a letter
46                  // and set the according value
47                  while(Character.isLetter(c)) {
48                      switch (c) {
49                          case 'A':
50                              n = 10;
51                              break;
52                          case 'B':
53                              n = 11;
54                              break;
55                          case 'C':
56                              n = 12;
57                              break;
58                          case 'D':
59                              n = 13;
60                              break;
61                          case 'E':
62                              n = 14;
63                              break;
64                          case 'F':
65                              n = 15;
66                              break;
67                      }
68
```

Since we are in hexadecimal mode, now we had to multiply the number by 16.

```
69                      num = num * 16 + n;
70                      i++;
71                      c = pfx.charAt(i);
72                  }
```

After that we needed another while loop that checks if the scanned character is a digit. In case it is, we convert the character to an int in the same way we learned how to do it in our previous lab exercise.

```
74                  while(Character.isDigit(c)) {
75                      num = num*16 + (c - '0');
76                      i++;
77                      c = pfx.charAt(i);
78                  }
```

Now the result is the top of the Stack and we had to return it and use the Integer.toHexString() method. Since our letters are in uppercase we also call the toUpperCase() method.

```
86          int result = (int) s.pop();
87          return Integer.toHexString(result).toUpperCase();
88      }
```

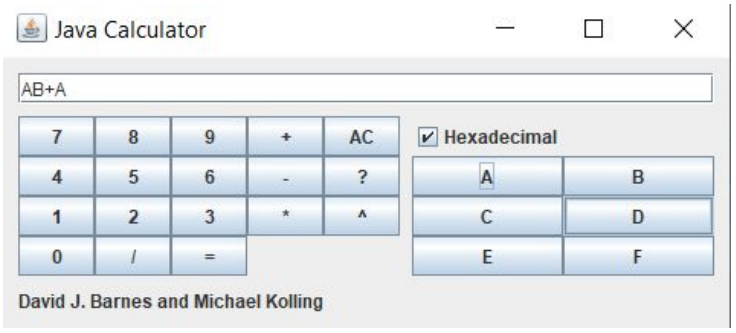
To be able to get the correct result, the equals method also has to be changed when working in hexadecimal mode. Here we first check if we are in hexadecimal mode and if we are, then we call the reworked evaluateHex method after converting the expression from infix to postfix.

```
90      public void equals() throws StackUnderflow {
91          if (inHexMode)
92              displayInput = evaluateHex(infixToPostfix(displayInput));
93          else
94              super.equals();
95      }
```

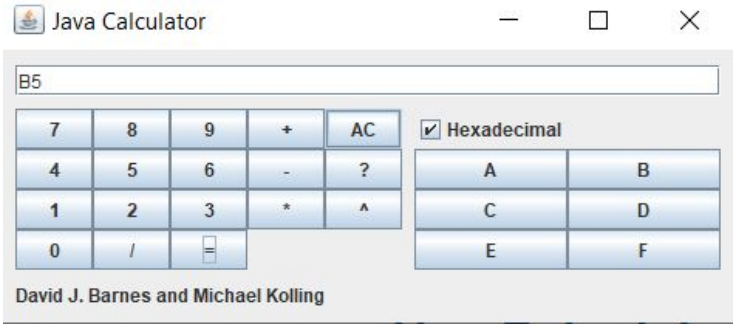
The other important change that we didn't know about was that to make hexadecimals work, we need to be able to separate the numbers. Since Timo's group was able to deal with that, they told us about this change and we were able to add it to our infixToPostfix method in the CalcEngine class.

```
    // Added to separate numbers
    if (i + 1 >= ifx.length() || !Character.isLetterOrDigit(ifx.charAt(i + 1)))
        result += ' ';
}
```

We decided to see how the calculator now works in hexadecimal mode. We tested with the following expression.



We got the following result.



Evaluation

It took us the 90 minutes of the lab to finish up to task 3, while we did the rest of the work at home. For this exercise, we used the material from the finished projects from lab 5 and lab 6 - worked together on with our lab partners. We also exchanged information with Sophia Stölzle and Timo Schmidt from our course. On the last day of work, we overhauled our entire project by implementing the changes done to have the calculator work with a String in CalcEngine, instead of UserInterface and HexUI only - This took most of the time we spent on this lab.

Pavel

Working on this exercise was interesting, because our calculator was now able to take a String, do its calculations and return a correct value. While working on the test cases I learned how to do the setUp method in Eclipse that has an annotation @Before, so that we can call that method before each test case.

Lukas

During my work on the lab, i could learn more in-depth about the calculator and how a postfix conversion/evaluation can have advantages in making calculations with Java. Although the beginning of the lab was easy to me, as it didn't take too much to fuse the classes from my previous labs together, we later had to change almost everything, because the basework wasn't right - to calculate hexadecimal numbers, the engine had to be able to deal with multi-digit numbers. I spent hours trying to figure out a solution to this problem, at least learning some new coding from it. Luckily, Pavel was able to sort out a solution and we adjusted our code and report accordingly.

Appendix

Code for Calculator

```
public class Calculator
{
    private CalcEngine engine;
    private UserInterface gui;

    public static void main (String [] args) {
       Calculator calculator = new Calculator();
    }

    /**
     * Create a new calculator and show it.
     */
    public Calculator()
    {
        engine = new CalcEngine();
        gui = new HexUI(engine);
    }

    /**
     * In case the window was closed, show it again.
     */
```

```
    public void show()
    {
        gui.setVisible(true);
    }
}
```

Code for CalcEngine

```
/**
 * The main part of the calculator doing the calculations.
 *
 * @author  David J. Barnes and Michael Kolling
 * @version 2008.03.30
 */

public class CalcEngine
{
    // The calculator's state is maintained in three fields:
    //     buildingDisplayValue, haveLeftOperand, and lastOperator.

    // Are we already building a value in the display, or will the
    // next digit be the first of a new one?
    private boolean buildingDisplayValue;
    // Has a left operand already been entered (or calculated)?
```

```
private boolean haveLeftOperand;

// The most recent operator that was entered.

private char lastOperator;



// The current value (to be) shown in the display.

private int displayValue;

// The value of an existing left operand.

private int leftOperand;









/**

 * Create a CalcEngine.

 */



public CalcEngine()

{

    clear();

}



/**

 * @return The value that should currently be displayed

 * on the calculator display.

 */
```

```
public int getDisplayValue()
{
    return displayValue;
}

/**
 * A number button was pressed.
 * Either start a new operand, or incorporate this number as
 * the least significant digit of an existing one.
 * @param number The number pressed on the calculator.
 */

public void numberPressed(int number)
{

    if(buildingDisplayValue) {
        // Incorporate this digit.
        displayValue = displayValue*10 + number;
    }
    else {
        // Start building a new number.
        displayValue = number;
        buildingDisplayValue = true;
    }
```

```
}


/**
 * The 'plus' button was pressed.
 */


public void plus()
{
    applyOperator('+');


}


/**
 * The 'minus' button was pressed.
 */


public void minus()
{
    applyOperator('-');


}


/**
 * The 'multiply' button was pressed.
 */
```

```
public void multiply()
{
    applyOperator('*');
}

/**
 * The '=' button was pressed.
 */

public void equals()
{
    // This should completes the building of a second operand,
    // so ensure that we really have a left operand, an operator
    // and a right operand.
    if(haveLeftOperand &&
            lastOperator != '?' &&
            buildingDisplayValue) {
        calculateResult();
        lastOperator = '?';
        buildingDisplayValue = false;
    }
    else {
        keySequenceError();
    }
```

```
}

/**
 * The 'C' (clear) button was pressed.
 * Reset everything to a starting state.
 * Update: change to small 'c' to avoid problems with hex. 'C'
 */

public void clear()
{
    lastOperator = '?';
    haveLeftOperand = false;
    buildingDisplayValue = false;
    leftOperand = 0;
    displayValue = 0;
    //EXERCISE7

}

/**
 * @return The title of this calculation engine.
 */

public String getTitle()
{
```

```
    return "Java Calculator";
}

/**
 * @return The author of this engine.
 */

public String getAuthor()
{
    return "David J. Barnes and Michael Kolling";
}

/**
 * @return The version number of this engine.
 */

public String getVersion()
{
   return "Version 1.0";
}

/**
 * Combine leftOperand, lastOperator, and the
 * current display value.
 * The result becomes both the leftOperand and
```

```
     * the new display value.
     */

    private void calculateResult()
    {
        switch(lastOperator) {
            case '+':
                displayValue = leftOperand + displayValue;
                haveLeftOperand = true;
                leftOperand = displayValue;
                break;
            case '-':
                displayValue = leftOperand - displayValue;
                haveLeftOperand = true;
                leftOperand = displayValue;
                break;
            case '*':
                displayValue = leftOperand * displayValue;
                haveLeftOperand = true;
                leftOperand = displayValue;
                break;
            default:
                keySequenceError();
                break;
        }
```

```
}


/**
 * Apply an operator.
 * @param operator The operator to apply.
 */


private void applyOperator(char operator)
{
    // If we are not in the process of building a new operand
    // then it is an error, unless we have just calculated a
    // result using '='.
    if(!buildingDisplayValue &&
                !(haveLeftOperand && lastOperator == '?')) {
        keySequenceError();
        return;
    }


    if(lastOperator != '?') {
        // First apply the previous operator.
        calculateResult();
    }
    else {
        // The displayValue now becomes the left operand of this
        // new operator.
```

```
            haveLeftOperand = true;
            leftOperand = displayValue;
        }
        lastOperator = operator;
        buildingDisplayValue = false;
    }

    /**
     * Report an error in the sequence of keys that was pressed.
     */

    private void keySequenceError()
    {
        System.out.println("A key sequence error has occurred.");
        // Reset everything.
        clear();
    }

    public String evaluateHex(String pfx) throws UnderflowException{

        // Create new Stack
        StackAsList s = new StackAsList();

        // Iterate through all characters of the pfx String
        for (int i = 0; i < pfx.length(); i++) {
```

```
            char c = pfx.charAt(i);
            //check if it is a space (separator)
            if(c == ' ')
                continue;
            if (c == '*' || c == '/' || c == '^' || c == '+' || c == '-') {
                int a = (int) s.pop();
                int b = (int) s.pop();
                switch(c) {
                    case '+':
                        s.push(b + a);
                        break;
                    case '-':
                        s.push(b - a);
                        break;
                    case '*':
                        s.push(b * a);
                        break;
                    case '/':
                        if (a == 0)
                            throw new UnsupportedOperationException("Cannot
divide by zero");
                        s.push(b / a);
                        break;
                    case '^':
                        s.push((int) Math.pow(b, a));
                        break;
```

```
        }


    } else {
        //if here means, its a digit
        int num = 0;
        int n = 0;
        //extract the characters and store it in num


        while(Character.isLetter(c)) {
            switch (c) {
                case 'A':
                    n = 10;
                    break;
                case 'B':
                    n = 11;
                    break;
                case 'C':
                    n = 12;
                    break;
                case 'D':
                    n = 13;
                    break;
                case 'E':
                    n = 14;
                    break;
```

```
                    case 'F':
                        n = 15;
                        break;
                }

                num = num * 16 + n;
                i++;
                c = pfx.charAt(i);
            }

            while(Character.isDigit(c)) {
                num = num*16 + (c - '0');
                i++;
                c = pfx.charAt(i);
            }

            i--;
            //push the number in stack
            s.push(num);
        }
    }

    int result = (int) s.pop();
    return Integer.toHexString(result).toUpperCase();
}
```

```
    public int evaluate (String pfx) throws UnderflowException

       {

            StackAsList<Integer> pfxList = new StackAsList<>(); //initializes a
list to store integers

            pfx = pfx.replaceAll(" ",""); //remove possible empty spaces from
the String

            int result; //stores the value being pushed to the list

            for (int i = 0; i < pfx.length(); i++) //iterates through the
characters of pfx

            {

                  char c = pfx.charAt(i); //stores the characters of pfx

                  if(Character.isDigit(c)) //if t is an operand

                        pfxList.push(Character.getNumericValue(c)); //push the
value of the char to the list

                  else //if t is an operator

                  {

                        int rhs = pfxList.pop(); //put the top into rhs, pop
it

                        int lhs = pfxList.pop(); //put the top into lhs, pop
it

                 switch(c) //calculate lhs t rhs and push the result

                 {

            case '+':

             result = lhs + rhs;

                pfxList.push(result);

                break;

            case '-':

             result = lhs - rhs;

                pfxList.push(result);
```

```
                break;
            case '/':
             result = lhs / rhs;
                pfxList.push(result);
                break;
            case '*':
             result = lhs * rhs;
                pfxList.push(result);
                break;
            case '^':
             result = (int) Math.pow(lhs, rhs);
             pfxList.push(result);
             }
                 }
                   }
            return pfxList.top();
            }
```

```
      public String infixToPostfix(String ifx) throws UnderflowException {
            StackAsList<Character> stack = new StackAsList<>();
            ifx = ifx.replaceAll(" ",""); // remove possible empty spaces from
the String
            String result = "";
```

```
            try {
                for (int i = 0; i < ifx.length(); i++) { // until input
string is empty
                    char t = ifx.charAt(i); // grab token


                    if (Character.isDigit(t)) {
                        result += t;
                    }
                  switch (t) {
                    case 'A':
                        result+= 10;
                        break;
                    case 'B':
                        result+= 11;
                        break;
                    case 'C':
                        result+= 12;
                        break;
                    case 'D':
                        result+= 13;
                        break;
                    case 'E':
                        result+= 14;
                        break;
                    case 'F':
                        result+= 15;
```

```
                        break;

                    }

                        if (isOperator(t)) {

                                    while (!stack.isEmpty()

                                            && isOperator(stack.top())
                                            && !rightAss(t)
                                            && ((prec(t, stack.top()) == -1)
                                    || (prec(t, stack.top()) == 0))) {

                                            result += stack.pop();

                                    }

                                    stack.push(t);

                        }



                        if (t == '(')

                                stack.push(t);

                        if (t == ')') {

                                while (stack.top() != '('){

                                        if (stack.isEmpty()){

                                                System.out.println("PARSE ERROR:
\' ( \' missing!"); // error

                                                break;

                                        } else {

                                                result += stack.pop();

                                        }

                                }

                                stack.pop();

                        }

                }
```

```
                    while(!stack.isEmpty()){
                            if (stack.top() == '('){
                                    System.out.println("PARSE ERROR: to many
parentheses!"); // error
                                    break;
                            } else {
                                    result += stack.pop();
                            }
                    }

            } catch (UnderflowException uex){
                    uex.printStackTrace(); // stack's empty, dude
            }

            return result;
      }

      // precedence evaluation
      public int prec (char opr1, char opr2){

            // returns : -1 lower, 0 equal, 1 higher than opr2

            if (opr1 == '+'){
                    if (opr2 == '+'){
                            return 0;
                    } else
```

```
                return -1;
        }


        if (opr1 == '-'){
               if (opr2 == '+'){
                       return 1;
               }
               if (opr2 == '-'){
                       return 0;
               } else return -1;
        }


        if (opr1 == '*'){
               if (opr2 == '+'){
                       return 1;
               }
               if (opr2 == '-'){
                       return 1;
               }
               if (opr2 == '*'){
                       return 0;
               }
               else return -1;
        }
        if (opr1 == '/'){
```

```
                if (opr2 == '+'){
                        return 1;
                }
                if (opr2 == '-'){
                        return 1;
                }
                if (opr2 == '*'){
                        return 1;
                }
                if (opr2 == '/'){
                        return 0;
                } else return -1;
        }

        if (opr1 == '^'){
                return 1; // highest precedence
        }
        return 0;
}

        // right associative?
boolean rightAss(char opr){
        switch (opr){
                case '+':
                        return false;
```

```
                    case '-':
                           return false;
                    case '*':
                           return false;
                    case '/':
                           return false;
                    case '^':
                           return true;
                    default: return false;
             }
      }

      boolean isOperator(char token){
             if(token =='+'||token =='-'||token =='*'||token =='/'|| token=='^'){
                    return true;
             } else return false;
      }
}
```

Code for UserInterface

```
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;
import javax.swing.border.*;

/**
 * A graphical user interface for the calculator. No calculation is being
 * done here. This class is responsible just for putting up the display on
 * screen. It then refers to the "CalcEngine" to do all the real work.
 *
 * @author David J. Barnes and Michael Kolling
 * @version 2008.03.30
 */
public class UserInterface
    implements ActionListener
{
    protected CalcEngine calc;
    private boolean showingAuthor;

    protected JFrame frame;
    protected JTextField display;
    protected JLabel status;

    protected String input = "";

    /**
     * Create a user interface.
```

```
     * @param engine The calculator engine.
     */
    public UserInterface(CalcEngine engine)
    {
        calc = engine;
        showingAuthor = true;
        makeFrame();
        frame.setVisible(true);
    }

    /**
     * Set the visibility of the interface.
     * @param visible true if the interface is to be made visible, false
otherwise.
     */
    public void setVisible(boolean visible)
    {
        frame.setVisible(visible);
    }

    /**
     * Make the frame for the user interface.
     */
    protected void makeFrame()
    {
        frame = new JFrame(calc.getTitle());
```

```
JPanel contentPane = (JPanel)frame.getContentPane();
contentPane.setLayout(new BorderLayout(8, 8));
contentPane.setBorder(new EmptyBorder( 10, 10, 10, 10));

display = new JTextField();
contentPane.add(display, BorderLayout.NORTH);

JPanel buttonPanel = new JPanel(new GridLayout(4, 4));
    addButton(buttonPanel, "7");
    addButton(buttonPanel, "8");
    addButton(buttonPanel, "9");
    addButton(buttonPanel, "c");

    addButton(buttonPanel, "4");
    addButton(buttonPanel, "5");
    addButton(buttonPanel, "6");
    addButton(buttonPanel, "?");

    addButton(buttonPanel, "1");
    addButton(buttonPanel, "2");
    addButton(buttonPanel, "3");
    addButton(buttonPanel, "*");
    //buttonPanel.add(new JLabel(" "));
```

```
        addButton(buttonPanel, "0");
        addButton(buttonPanel, "+");
        addButton(buttonPanel, "-");
        addButton(buttonPanel, "=");

    contentPane.add(buttonPanel, BorderLayout.CENTER);

    status = new JLabel(calc.getAuthor());
    contentPane.add(status, BorderLayout.SOUTH);

    frame.pack();
}

/**
 * Add a button to the button panel.
 * @param panel The panel to receive the button.
 * @param buttonText The text for the button.
 */
protected void addButton(Container panel, String buttonText)
{
    JButton button = new JButton(buttonText);
    button.addActionListener(this);
    panel.add(button);
}
```

```
/**
 * An interface action has been performed.
 * Find out what it was and handle it.
 * @param event The event that has occured.
 */
public void actionPerformed(ActionEvent event)
{
    String command = event.getActionCommand();

    if(command.equals("0") ||
       command.equals("1") ||
       command.equals("2") ||
       command.equals("3") ||
       command.equals("4") ||
       command.equals("5") ||
       command.equals("6") ||
       command.equals("7") ||
       command.equals("8") ||
       command.equals("9")) {
        int number = Integer.parseInt(command);
        input = input + number;
        calc.numberPressed(number);
    }
    else if(command.equals("+")) {
        calc.plus();
```

```
            input = input + "+";
        }
        else if(command.equals("-")) {
            calc.minus();
            input = input + "-";
        }
        else if(command.equals("*")) {
            calc.multiply();
            input = input + "*";
        }
        else if(command.equals("=")) {
            calc.equals();
            //old code: input = input + "=" + calc.getDisplayValue();
            input = input + "=";
        }
        else if(command.equals("c")) {
            calc.clear();
            input = "";
        }
        else if(command.equals("?")) {
            showInfo();
        }
        // else unknown command.

        redisplay();
```

```
}


/**
 * Update the interface display to show the current value of the
 * calculator.
 */
protected void redisplay()
{
    if (input.contains("="))
       {   try{
           input = input + calc.evaluate(calc.infixToPostfix(input));
           } catch (UnderflowException uex){
        uex.printStackTrace();}
       }
   display.setText(input);

    //old code:display.setText("" + calc.getDisplayValue());
}


/**
 * Toggle the info display in the calculator's status area between the
 * author and version information.
 */
private void showInfo()
{
```

```
        if(showingAuthor)
            status.setText(calc.getVersion());
        else
            status.setText(calc.getAuthor());

        showingAuthor = !showingAuthor;
    }
}
```

Code for HexUI

```
import java.awt.BorderLayout;
import java.awt.Checkbox;
import java.awt.Container;
//import java.awt.GridBagLayout;
import java.awt.GridLayout;
//import java.awt.Panel;
import java.awt.event.ActionEvent;
//import java.awt.event.ItemListener;

import javax.swing.AbstractButton;
//import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
//import javax.swing.JLabel;
```

```
import javax.swing.JPanel;
//import javax.swing.JTextField;
import javax.swing.border.EmptyBorder;

public class HexUI extends UserInterface {
	Checkbox checkbox;
	JPanel hexButtons;
	JFrame hexFrame;
	boolean stateCheckbox = true;
	String hexNumber;

	public HexUI(CalcEngine engine) {
		super(engine);
		hexFrame.setVisible(true);
		frame.setVisible(false);
	}

	protected void makeFrame() {
		hexFrame = new JFrame(calc.getTitle());
		JPanel content = (JPanel) hexFrame.getContentPane();
		content.setLayout(new BorderLayout(8, 8));
		content.setBorder(new EmptyBorder(10, 10, 10, 10));

		hexButtons = new JPanel(new GridLayout(6, 1));
		addButton(hexButtons, "A");
```

```
        addButton(hexButtons, "B");

        addButton(hexButtons, "C");

        addButton(hexButtons, "D");

        addButton(hexButtons, "E");

        addButton(hexButtons, "F");


        content.add(hexButtons, BorderLayout.EAST);


        super.makeFrame();

        content.add(super.frame.getContentPane(), BorderLayout.CENTER);


        JPanel checkPanel = new JPanel(new GridLayout(1, 1));

        addCheckbox(checkPanel, "Hexadecimal", true);

        // checkbox.addActionListener();

        content.add(checkPanel, BorderLayout.SOUTH);


        hexButtons.setVisible(true);


        /*status = new JLabel(calc.getAuthor());

        content.add(status, BorderLayout.SOUTH);*/


        hexFrame.pack();


    }
```

```
public void setVisible(boolean visible) {
	hexFrame.setVisible(visible);
}

public void actionPerformed(ActionEvent event) {
	String command = event.getActionCommand();
	AbstractButton hexDeCheckBox = (AbstractButton) event.getSource();
	boolean selected = hexDeCheckBox.getModel().isSelected();

	//System.out.println(command);
	if (command.equals("A")) {
		calc.numberPressed(10);
		input = input + "A";
	} else if (command.equals("B")) {
		calc.numberPressed(1);
		calc.numberPressed(1);
		input = input + "B";
	} else if (command.equals("C")) {
		calc.numberPressed(1);
		calc.numberPressed(2);
		input = input + "C";
	} else if (command.equals("D")) {
		calc.numberPressed(1);
		calc.numberPressed(3);
		input = input + "D";
```

```
        } else if (command.equals("E")) {
                calc.numberPressed(1);
                calc.numberPressed(4);
                input = input + "E";
        } else if (command.equals("F")) {
                calc.numberPressed(1);
                calc.numberPressed(5);
                input = input + "F";
        } else {
                super.actionPerformed(event);
        }
        System.out.println(input);
        if (command.equals("Hexadecimal")) {
                hexButtons.setVisible(selected);
                stateCheckbox = selected;
        }
        if (stateCheckbox) {
                redisplay();
        }
        else {
                super.redisplay();
        }

}
```

```
      protected void redisplay() {

          if (input.contains("="))

          {{   try{

              //int decimalDisplay = calc.getDisplayValue();

              int decimalDisplay = calc.evaluate(calc.infixToPostfix(input));

              input = Integer.toHexString(decimalDisplay);

              } catch (UnderflowException uex){

           uex.printStackTrace();}

          }

          }

          display.setText(input);


          /*char[] hexArray = { '0', '1', '2', '3', '4', '5', '6', '7', '8',
'9', 'A', 'B', 'C', 'D', 'E', 'F' };

             int decimalDisplay = calc.getDisplayValue();

             String hexDisplayString = "";

             int hexDisplay;

             if (decimalDisplay == 0) {

                    hexDisplayString = "0";

             }

             while (decimalDisplay > 0) {

                    hexDisplay = decimalDisplay / 16;

                    hexDisplayString += hexArray[decimalDisplay % 16];

                    decimalDisplay = hexDisplay;

             }
```

```
            String hexReverse = mirror(hexDisplayString);
            */









      }


      private String mirror(String str) {
            String mirrorString = "";
            for (int i = str.length() - 1; i >= 0; i--) {
                  mirrorString += str.charAt(i);
            }
            hexNumber = mirrorString;
            return mirrorString;
      }


      private void addCheckbox(Container panel, String checkboxString, boolean
state) {
            JCheckBox checkbox = new JCheckBox(checkboxString, state);
            checkbox.addActionListener(this);
            panel.add(checkbox);
      }


      protected String returnHexNumber() {
            return hexNumber;
```

```
        }



}
```

Code for Stack

```
public interface Stack<T> {

    void push(T t); // pushing values

    T pop () throws UnderflowException; // fetching values

    T top () throws UnderflowException; // looking up most recent value

    boolean isEmpty(); // checking whether stack is empty

    void empty();

}

// in her examples she uses Object as parameter types for pop and top
```

Code for StackAsList

```
public class StackAsList<T> implements Stack<T> {



    private LinkedList <T>list = new LinkedList<>();



    @Override
```

```
public void push(T t) {
    list.add(t);
}

@Override
public T pop() throws UnderflowException {
    try {
        T pop = list.current();
        list.remove();
        return pop;
    }catch (UnderflowException uex) {
        uex.printStackTrace();
        return null;
    }
}

@Override
public T top() throws UnderflowException {
    try {
        return list.current();
    } catch (UnderflowException uex) {
        uex.printStackTrace();
        return null;
    }
}
```

```
    @Override
    public boolean isEmpty() {
        return list.isEmpty();
    }

    @Override
    public void empty() {
        list.reset();
    }
}
```

Code for LinkedList

```
public class LinkedList<T> {

    private Node head = null;
    private Node curr = null;



    // class for our list elements / nodes
    private class Node {
        // generic types for generic data:
```

```
        private T data;
        private Node next;

        // constructor
        Node(T data, Node x) {
            this.data = data;
            this.next = x;
        }
    }

    public void add(T data) {

        Node tmp = new Node(data, head);
        head = tmp;
        curr = head;
    }

    public void remove() {
        // wenn die Liste leer ist oder der ggw. node nicht angelegt worden ist, nichts zurückgeben
        if (this.isEmpty() || curr == null) {
            return;
        } else {
            // das das ggw. element immer das erste ist und damit keinen vorgänger hat, wird sein nächstes element zum neuen head beim löschen
            head = curr.next;
```

```
            curr = head;
        }
    }

    // abchecken ob die Liste elemente besitzt
    public boolean isEmpty() {
        return head == null;
    }

    // liste resetten
    public void reset(){
        head = null;
        curr = null;
    }

    // etwas eleganter mit exception:
    public T current() throws UnderflowException {
        if (isEmpty()) throw new UnderflowException("no current element: list is
empty");
        return curr.data;
    }

}
```

Code for UnderflowException

```
public class UnderflowException extends Exception {

    UnderflowException(String msg){

        super(msg);

    }

}
```

Code for CalcEngineTest

```
import static org.junit.Assert.*;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;


/*

 * In this class we are going to test the precedence

 * of + over *. We need to find 15 good test cases

 * and see the result.

 */

public class CalcEngineTest

{


    private CalcEngine calc;


    public CalcEngineTest()
```

```
{



}


/*
 * The setUp method gets called before
 * every test case
 */
@Before
public void setUp()
{
    calc = new CalcEngine();
}


@After
public void tearDown()
{
}


@Test
/**
 * Testing 1+2+3*4
 */
public void test1() throws StackUnderflow
{
```

```
        calc.buttonPressed("1");
        calc.buttonPressed("+");
        calc.buttonPressed("2");
        calc.buttonPressed("+");
        calc.buttonPressed("3");
        calc.buttonPressed("*");
        calc.buttonPressed("4");
        // Making sure that the getDisplayInput
        // correctly displays the given input
        assertEquals("1+2+3*4", calc.getDisplayInput());
        calc.equals();
        assertEquals("15", calc.getDisplayInput());
    }

    @Test
    /**
     * Testing 1+2*5
     */
    public void test2() throws StackUnderflow
    {
        calc.buttonPressed("1");
        calc.buttonPressed("+");
        calc.buttonPressed("2");
        calc.buttonPressed("*");
        calc.buttonPressed("5");
```

```
        calc.equals();
        assertEquals("11", calc.getDisplayInput());
    }

    @Test
    /**
     * Testing 1*2+3*4
     */
    public void test3() throws StackUnderflow
    {
        calc.buttonPressed("1");
        calc.buttonPressed("*");
        calc.buttonPressed("2");
        calc.buttonPressed("+");
        calc.buttonPressed("3");
        calc.buttonPressed("*");
        calc.buttonPressed("4");
        calc.equals();
        assertEquals("14", calc.getDisplayInput());
    }

    @Test
    /**
     * Testing 5*6+1*3+2
     */
```

```
public void test4() throws StackUnderflow
{
    calc.buttonPressed("5");
    calc.buttonPressed("*");
    calc.buttonPressed("6");
    calc.buttonPressed("+");
    calc.buttonPressed("1");
    calc.buttonPressed("*");
    calc.buttonPressed("3");
    calc.buttonPressed("+");
    calc.buttonPressed("2");
    calc.equals();
    assertEquals("35", calc.getDisplayInput());
}

@Test
/**
 * Testing 1*9+2*8+3*7
 */
public void test5() throws StackUnderflow
{
    calc.buttonPressed("1");
    calc.buttonPressed("*");
    calc.buttonPressed("9");
    calc.buttonPressed("+");
```

```
        calc.buttonPressed("2");
        calc.buttonPressed("*");
        calc.buttonPressed("8");
        calc.buttonPressed("+");
        calc.buttonPressed("3");
        calc.buttonPressed("*");
        calc.buttonPressed("7");
        calc.equals();
        assertEquals("46", calc.getDisplayInput());
    }

    @Test
    /**
     * Testing 2*9-1*9+1
     */
    public void test6() throws StackUnderflow
    {
        calc.buttonPressed("2");
        calc.buttonPressed("*");
        calc.buttonPressed("9");
        calc.buttonPressed("-");
        calc.buttonPressed("1");
        calc.buttonPressed("*");
        calc.buttonPressed("9");
        calc.buttonPressed("+");
```

```
        calc.buttonPressed("1");

        calc.equals();

        assertEquals("10", calc.getDisplayInput());

    }


    @Test

    /**

     * Testing 3-9-1*3+9

     */

    public void test7() throws StackUnderflow

    {

        calc.buttonPressed("3");

        calc.buttonPressed("-");

        calc.buttonPressed("9");

        calc.buttonPressed("-");

        calc.buttonPressed("1");

        calc.buttonPressed("*");

        calc.buttonPressed("3");

        calc.buttonPressed("+");

        calc.buttonPressed("9");

        calc.equals();

        assertEquals("0", calc.getDisplayInput());

    }


    @Test
```

```
/**
 * Testing 9*4-8*4+4
 */
public void test8() throws StackUnderflow
{
    calc.buttonPressed("9");
    calc.buttonPressed("*");
    calc.buttonPressed("4");
    calc.buttonPressed("-");
    calc.buttonPressed("8");
    calc.buttonPressed("*");
    calc.buttonPressed("4");
    calc.buttonPressed("+");
    calc.buttonPressed("4");
    calc.equals();
    assertEquals("8", calc.getDisplayInput());
}

@Test
/**
 * Testing 5*4+9-4*5-9
 */
public void test9() throws StackUnderflow
{
    calc.buttonPressed("5");
```

```
        calc.buttonPressed("*");
        calc.buttonPressed("4");
        calc.buttonPressed("+");
        calc.buttonPressed("9");
        calc.buttonPressed("-");
        calc.buttonPressed("4");
        calc.buttonPressed("*");
        calc.buttonPressed("5");
        calc.buttonPressed("-");
        calc.buttonPressed("9");
        calc.equals();
        assertEquals("0", calc.getDisplayInput());
    }

    @Test
    /**
     * Testing 1+2+3*4+5*6-7*7+1
     */
    public void test10() throws StackUnderflow
    {
        calc.buttonPressed("1");
        calc.buttonPressed("+");
        calc.buttonPressed("2");
        calc.buttonPressed("+");
        calc.buttonPressed("3");
```

```
        calc.buttonPressed("*");
        calc.buttonPressed("4");
        calc.buttonPressed("+");
        calc.buttonPressed("5");
        calc.buttonPressed("*");
        calc.buttonPressed("6");
        calc.buttonPressed("-");
        calc.buttonPressed("7");
        calc.buttonPressed("*");
        calc.buttonPressed("7");
        calc.buttonPressed("+");
        calc.buttonPressed("1");
        calc.equals();
        assertEquals("-3", calc.getDisplayInput());
    }

    @Test
    /**
     * Testing 3*9+1+2+6*8+2*5-9*8
     */
    public void test11() throws StackUnderflow
    {
       calc.buttonPressed("3");
       calc.buttonPressed("*");
       calc.buttonPressed("9");
```

```
    calc.buttonPressed("+");
    calc.buttonPressed("1");
    calc.buttonPressed("+");
    calc.buttonPressed("2");
    calc.buttonPressed("+");
    calc.buttonPressed("6");
    calc.buttonPressed("*");
    calc.buttonPressed("8");
    calc.buttonPressed("+");
    calc.buttonPressed("2");
    calc.buttonPressed("*");
    calc.buttonPressed("5");
    calc.buttonPressed("-");
    calc.buttonPressed("9");
    calc.buttonPressed("*");
    calc.buttonPressed("8");
    calc.equals();
    assertEquals("16", calc.getDisplayInput());

}

@Test
/**
 * Testing 9/3+2*6+5
 */
```

```
public void test12() throws StackUnderflow
{
   calc.buttonPressed("9");
   calc.buttonPressed("/");
   calc.buttonPressed("3");
   calc.buttonPressed("+");
   calc.buttonPressed("2");
   calc.buttonPressed("*");
   calc.buttonPressed("6");
   calc.buttonPressed("+");
   calc.buttonPressed("5");
   calc.equals();
   assertEquals("20", calc.getDisplayInput());
}

@Test
/**
 * Testing 8/4+2*4+3*3+6/3
 */
public void test13() throws StackUnderflow
{
   calc.buttonPressed("8");
   calc.buttonPressed("/");
   calc.buttonPressed("4");
   calc.buttonPressed("+");
```

```
    calc.buttonPressed("2");
    calc.buttonPressed("*");
    calc.buttonPressed("4");
    calc.buttonPressed("+");
    calc.buttonPressed("3");
    calc.buttonPressed("*");
    calc.buttonPressed("3");
    calc.buttonPressed("+");
    calc.buttonPressed("6");
    calc.buttonPressed("/");
    calc.buttonPressed("3");
    calc.equals();
    assertEquals("21", calc.getDisplayInput());
}

@Test
/**
 * Testing without buttonPressed
 * just setting the displayInput
 * to something
 */
public void test14() throws StackUnderflow
{
    calc.displayInput = "9/3+2*4+1+9+3*5";
    calc.equals();
```

```
    assertEquals("36", calc.getDisplayInput());
}

@Test
/*
 * Testing without buttonPressed
 * just setting the displayInput to something
 */
public void test15() throws StackUnderflow
{
    calc.displayInput = "6/3+2+3*9+9*9+1*5-9/9";
     calc.equals();
     assertEquals("116", calc.getDisplayInput());
}

@Test
/*
 * Since our calculator also has a button
 * to set a number to the power
 * we also decided to test how it works
 * using ^,*,+,/
 */
public void test16() throws StackUnderflow
{
   calc.displayInput = "2^3+2*5+7";
```

```
      calc.equals();
      assertEquals("25", calc.getDisplayInput());
   }

   @Test
   public void test17() throws StackUnderflow
   {
      calc.displayInput = "3^2+1+7*2";
      calc.equals();
      assertEquals("24", calc.getDisplayInput());
   }

   @Test
   public void test18() throws StackUnderflow
   {
      calc.displayInput = "5^2+1*3+1*2+5";
      calc.equals();
      assertEquals("35", calc.getDisplayInput());
   }

   @Test
   public void test19() throws StackUnderflow
   {
      calc.displayInput = "5^2+1*3+1*2+5";
      calc.equals();
```

```
        assertEquals("35", calc.getDisplayInput());
    }

    @Test
    public void test20() throws StackUnderflow
    {
        calc.displayInput = "2^5-6*5+8+6/2";
        calc.equals();
        assertEquals("13", calc.getDisplayInput());
    }
}
```