

File - Main.java

```
1 import calculator.Calculator;
2 import calculator.JCalculator;
3
4 public class Main
5 {
6     public static void main(String ... args) {
7         if(args.length > 0 && args[0].equals("-t")) {
8             new Calculator().run();
9         } else new JCalculator();
10    }
11 }
12
```

File - Node.java

```
1 package util;
2 /**
3  * This represents a generic node in a linked structure.
4  * It is designed to hold data of a specified type, along
5  * with a reference to the next node in the structure.
6  *
7  * @param <T> The type of the data to be stored within the node.
8  */
9 public class Node<T> {
10
11     T data;
12     Node<T> next;
13
14     /**
15      * Creates a new node with the provided data and the reference
16      * to the next node.
17      *
18      * @param data The data to be stored within the node.
19      * @param next The next node in the linked structure or
20      *             {@code null} if this is the last node.
21      */
22     public Node(T data, Node<T> next){
23         this.data = data;
24         this.next = next;
25     }
26
27 }
28
```

File - Stack.java

```
1 package util;
2
3 import java.util.EmptyStackException;
4
5 /**
6  * Generic stack data structure implementation.
7  * @param <T> the type of elements in the stack
8 */
9 public class Stack<T> {
10
11     private Node<T> stackTop;
12     private int size;
13
14     /**
15      * Constructs an empty stack.
16      */
17     public Stack() {
18         stackTop = null;
19         size = 0;
20     }
21
22     /**
23      * Checks if the stack is empty.
24      * @return true if the stack is empty, false otherwise.
25      */
26     public boolean empty() {
27         return stackTop == null;
28     }
29
30     /**
31      * Pushes an element onto the top of the stack.
32      * @param data the element to be pushed.
33      */
34     public void push(T data) {
35         stackTop = new Node<>(data, stackTop);
36         size++;
37     }
38
39     /**
40      * Removes and returns the element at the top of the stack.
41      * @return the element at the top of the stack
42      * @throws EmptyStackException if the stack is empty
43      */
44     public T pop() throws EmptyStackException {
45         if (this.empty()) {
46             throw new EmptyStackException();
47         }
48         T tmp = stackTop.data;
49         stackTop = stackTop.next;
50         size--;
51         return tmp;
52     }
53
54     /**
55      * Removes all elements from the stack.
56      */
57     public void clear() {
58         stackTop = null;
59         size = 0;
60         //Garbage collector will do the rest...
61     }
62
63     /**
64      * @return a string representation of the stack
65      */
66     public String toString() {
67         if (stackTop == null) {
```

File - Stack.java

```
68         return "";
69     }
70
71     StackIterator<T> it = new StackIterator<T>(stackTop);
72     StringBuilder res = new StringBuilder(stackTop.data.toString() + " ");
73
74     while (it.hasNext()) {
75         res.append(it.next().toString());
76         if (it.hasNext()) res.append(" ");
77     }
78     return res.toString();
79 }
80
81 /**
82 * @return an array containing all elements in the stack from top to bottom.
83 */
84 public Object[] array() {
85     if (this.empty()) {
86         return new Object[size];
87     }
88
89     Object[] tmp = new Object[size];
90     tmp[0] = stackTop.data;
91     StackIterator<T> it = new StackIterator<>(stackTop);
92     int ind = 1;
93
94     while (it.hasNext()) {
95         tmp[ind] = it.next();
96         ind++;
97     }
98     return tmp;
99 }
100 }
```

File - StackIterator.java

```
1 package util;
2
3 /**
4  * Generic stack iterator, used for the generic stack implementation.
5  * @param <T> the type of elements in the stack
6 */
7 public class StackIterator<T> {
8
9     private Node<T> current;
10
11    /**
12     * Checks if there is a next element in the iterator.
13     * @return true if there is a next element, false otherwise.
14     */
15    public boolean hasNext() {
16        return (current.next != null);
17    }
18
19    /**
20     * Returns the data of the next element of the iterator and advances it.
21     * @return the data of the next element of the iterator.
22     */
23    public T next() {
24        current = current.next;
25        return current.data;
26    }
27
28    /**
29     * Constructs a StackIterator object with the given node.
30     * @param current the node
31     */
32    public StackIterator(Node<T> current) {
33        this.current = current;
34    }
35 }
```

```

File - StackTest.java
1 package util.test;
2
3 import org.junit.Test;
4 import java.util.EmptyStackException;
5
6 import static org.junit.Assert.*;
7
8 import util.*;
9
10 /**
11 * This tests the functions of the class Stack using junit. {@link Stack}
12 *
13 * @author Arthur Junod
14 * @author Edwin Haeffner
15 * Date : 13/11/2023
16 */
17 public class StackTest {
18     /**
19      * Test the empty() method of the implemented stack
20      */
21     @Test
22     public void testEmptyStack() {
23         Stack<Integer> stack = new Stack<Integer>();
24         assertTrue(stack.empty());
25     }
26
27     /**
28      * Test the push() and pop() methods of the implemented stack
29      */
30     @Test
31     public void testPushAndPop() {
32         Stack<Integer> stack = new Stack<Integer>();
33
34         stack.push(1);
35         stack.push(2);
36         stack.push(3);
37
38         assertFalse(stack.empty());
39         assertEquals((Integer)3, stack.pop());
40         assertEquals((Integer)2, stack.pop());
41         assertEquals((Integer)1, stack.pop());
42         assertTrue(stack.empty());
43     }
44
45     /**
46      * Test if the Stack throw an exception when using pop() on an empty one
47      */
48     @Test
49     public void testPopEmptyStack() {
50         Stack<Integer> stack = new Stack<Integer>();
51         assertThrows(EmptyStackException.class, stack::pop);
52     }
53
54     /**
55      * Test if the print of the stack is correct
56      */
57     @Test
58     public void testToString() {
59         Stack<Integer> stack = new Stack<Integer>();
60         assertEquals("",stack.toString());
61
62         stack.push(1);
63         stack.push(2);
64         stack.push(3);
65
66         String expected = "3 2 1";
67         assertEquals(expected, stack.toString());

```

File - StackTest.java

```
68     }
69
70     /**
71      * Test the transformation to an array of the Stack
72      */
73     @Test
74     public void testArray() {
75         Stack<Integer> stack = new Stack<Integer>();
76         stack.push(1);
77         stack.push(2);
78         stack.push(3);
79
80         Object[] expected = {3, 2, 1};
81         assertEquals(expected, stack.array());
82     }
83
84 }
```

File - Clear.java

```
1 package calculator;
2 /**
3 * Clear everything that ClearError does but also empties the stack.
4 * Extends Operator and implements execute().
5 * {@link ClearError}
6 *
7 * @author Arthur Junod
8 * @author Edwin Haeffner
9 * Date : 22/11/2023
10 */
11 public class Clear extends ClearError {
12     public Clear(State calcState) {
13         super(calcState);
14     }
15
16     /**
17      * Clears the current value of the calculator and removes errors if any.
18      */
19     @Override
20     void execute() {
21
22         super.execute();
23         calcState.stack.clear();
24
25     }
26 }
27
```

File - State.java

```
1 package calculator;
2
3 import util.Stack;
4
5 /**
6  * This class holds the current state of a calculator session.
7  * This includes the current stack of numbers, the current input or
8  * result as a string, and flags indicating if the current input is
9  * a result or if there has been an error.
10 */
11 * @author Arthur Junod
12 * @author Edwin Haeffner
13 * Date : 22/11/2023
14 */
15 public class State {
16
17     Stack<Double> stack;
18     String current;
19     boolean isResult;
20     boolean error;
21
22     /**
23      * Constructs a State object.
24      */
25     public State() {
26         stack = new Stack<>();
27         current = "0";
28         isResult = false;
29         error = false;
30     }
31
32     /**
33      * @return the current value as a double.
34      * @throws NumberFormatException if current isn't a double.
35      * @throws NullPointerException if the string is null
36      */
37     double getCurrentDouble() {
38         return Double.parseDouble(current);
39     }
40 }
41
```

File - UnaryOp.java

```
1 package calculator;
2
3 /**
4 * This class is used as a backbone for all the unary operators.
5 * It is then required to implement the operate method with the desired operator.
6 * {@link Operator}
7 *
8 * @author Arthur Junod
9 * @author Edwin Haeffner
10 * Date : 22/11/2023
11 */
12 public abstract class UnaryOp extends Operator {
13
14     /**
15      * Constructs a UnaryOp object.
16      * @param calcState the calculator state
17      */
18     public UnaryOp(State calcState) {
19         super(calcState);
20     }
21
22     /**
23      * Executes the unary operation on the current double value.
24      * The result is stored in the current String from calcState,
25      * and the isResult flag is set to true.
26      */
27     protected void execute() {
28         if(!calcState.current.isEmpty() && !calcState.error) {
29             calcState.current = String.valueOf(operate(calcState.getCurrentDouble()));
30             calcState.isResult = true;
31         }
32     }
33
34     protected void setError(){
35         calcState.error = true;
36     }
37
38     /**
39      * Performs the unary operation on the given value.
40      *
41      * @param value the value to operate on
42      * @return the result of the operation
43      */
44     protected abstract double operate(double value);
45 }
```

File - BinaryOp.java

```
1 package calculator;
2
3 /**
4 * This class is used as a backbone for all the binary operators.
5 * It is then required to implement the operate method with the desired operator.
6 *
7 * @author Arthur Junod
8 * @author Edwin Haeffner
9 * Date : 28/11/2023
10 */
11 public abstract class BinaryOp extends Operator {
12     public BinaryOp(State calcState) {
13         super(calcState);
14     }
15
16     /**
17      * Executes the binary operation by popping an operand from the stack
18      * and using it with the current. The result is then put back in the current.
19      * If the stack is empty the error flag is set to true.
20      */
21     @Override
22     protected void execute() {
23
24         if (calcState.stack.empty() || calcState.error) {
25             calcState.error = true;
26         } else {
27
28             double d = calcState.stack.pop();
29
30             //To cast the double to String
31             calcState.current = String.valueOf(operate(d, calcState.getCurrentDouble()));
32             calcState.isResult = true;
33         }
34     }
35
36     /**
37      * Performs the binary operation on the given values.
38      * @param a the first value.
39      * @param b the second value.
40      * @return the result of the operation.
41      */
42     protected abstract double operate(double a, double b);
43
44 }
45
46 }
```

File - Operator.java

```
1 package calculator;
2
3
4 /**
5  * This class is used as a backbone for all the operators and is mainly used to
6  * store the current state of the calculator and to give an abstract method to
7  * execute the operator.
8 *
9  * @author Arthur Junod
10 * @author Edwin Haeffner
11 * Date : 22/11/2023
12 */
13 public abstract class Operator {
14     State calcState;
15
16     /**
17      * Executes the operator.
18      */
19     abstract void execute();
20
21     /**
22      * Constructs an Operator object.
23      * @param calcState the calculator state.
24      */
25     Operator(State calcState) {
26         this.calcState = calcState;
27     }
28
29 }
30
```

File - Opposite.java

```
1 package calculator;
2
3
4 /**
5  * Transform the value of State.current into it's opposite.
6  * Extends Operator and implements execute().
7  * {@link Operator}
8 *
9  * @author Arthur Junod
10 * @author Edwin Haeffner
11 * Date : 22/11/2023
12 */
13 public class Opposite extends Operator {
14
15     public Opposite(State calcState){super(calcState);}
16
17     @Override
18     void execute() {
19         if(!calcState.error && !calcState.current.equals("0")
20             && !calcState.current.equals("NaN")
21             && !calcState.current.equals("Infinity")) {
22             if(calcState.current.charAt(0) == '-'){
23                 calcState.current = calcState.current.substring(1);
24             }else{
25                 calcState.current = "-" + calcState.current;
26             }
27         }
28     }
29 }
30
```

File - Backspace.java

```
1 package calculator;
2
3 /**
4 * Erase the char at the far right of State.current.
5 * Extends Operator and implements execute().
6 * {@link Operator}
7 *
8 * @author Arthur Junod
9 * @author Edwin Haeffner
10 * Date : 22/11/2023
11 */
12 public class Backspace extends Operator {
13
14     /**
15      * Constructs a Backspace object.
16      * @param calcState the calculator state
17      */
18     public Backspace(State calcState) {super(calcState);}
19
20     /**
21      * Removes the last character from the current value of the calculator.
22      */
23     @Override
24     void execute() {
25         if(!calcState.error && !calcState.current.equals("Infinity")
26             && !calcState.current.equals("NaN")){
27             if (calcState.current.length() > 1) {
28                 String cur = calcState.current;
29                 calcState.current = cur.substring(0, cur.length() - 1);
30             } else {
31                 calcState.current = "0";
32             }
33         }
34     }
35
36
37 }
38
```

File - Calculator.java

```
1 package calculator;
2
3 import calculator.binaryop.Addition;
4 import calculator.binaryop.Division;
5 import calculator.binaryop.Multiplication;
6 import calculator.binaryop.Subtraction;
7 import calculator.unaryop.Inverse;
8 import calculator.unaryop.Square;
9 import calculator.unaryop.SquareRoot;
10
11 import java.util.HashMap;
12 import java.util.Scanner;
13
14 /**
15 * This class allow us to launch the calculator in the terminal.
16 *
17 * @author Arthur Junod
18 * @author Edwin Haeffner
19 * Date : 28/11/2023
20 */
21 public class Calculator {
22
23     private final State calcState = new State();
24     private final Scanner sin = new Scanner(System.in);
25     private final HashMap<String, Operator> operations;
26
27     public Calculator(){
28         operations = new HashMap<>();
29         operations.put("+", new Addition(calcState));
30         operations.put("-", new Subtraction(calcState));
31         operations.put("*", new Multiplication(calcState));
32         operations.put("/", new Division(calcState));
33         operations.put("sqrt", new SquareRoot(calcState));
34         operations.put("inv", new Inverse(calcState));
35         operations.put("sq", new Square(calcState));
36         operations.put("c", new Clear(calcState));
37         operations.put("ce", new ClearError(calcState));
38         operations.put("ms", new MemoryStore(calcState));
39         operations.put("mr", new MemoryRecall(calcState));
40     }
41
42     /**
43      * This function runs the calculator in the terminal
44      */
45     public void run() {
46         System.out.println("Java Calculator");
47
48         while (true) {
49             String in = sin.nextLine();
50             if (in.equals("exit")) break;
51
52             // Verify if the user input is an operation
53             if (operations.containsKey(in)) {
54                 operations.get(in).execute();
55             } else {
56                 // The try{}catch{} verifies if the user input is a double
57                 try {
58                     // Append the entier user input in the State.current
59                     // and then set it as a result so that it will
60                     // be pushed to the State.stack next time we enter a number.
61                     new AddToCurrent(calcState,
62                         String.valueOf(Double.parseDouble(in))).execute();
63                     calcState.isResult = true;
64                 } catch (NumberFormatException ignore) {
65                     System.out.println("Invalid input");
66                     continue; // Ask an input again if it's invalid
67                 }
68             }
69         }
70     }
71 }
```

File - Calculator.java

```
68         }
69
70         display();
71     }
72 }
73
74 /**
75  * Display the calculator in the terminal.
76 */
77 private void display()
78 {
79     String res;
80     if (calcState.error) res = "ERROR :3";
81     else res = calcState.current;
82     System.out.println(res + " " + calcState.stack.toString());
83 }
84
85 }
86
```

File - ClearError.java

```
1 package calculator;
2
3 /**
4 * Put the 2 flag (error, isResult) to false and put the State.current back to "0".
5 * Extends Operator and implements execute().
6 * {@link Operator}
7 *
8 * @author Arthur Junod
9 * @author Edwin Haeffner
10 * Date : 22/11/2023
11 */
12 public class ClearError extends Operator {
13
14     public ClearError(State calcState) {
15         super(calcState);
16     }
17
18     @Override
19     void execute() {
20
21         calcState.current = "0";
22         calcState.error = false;
23         calcState.isResult = false;
24
25     }
26 }
27
```

File - JCalculator.java

```
1 package calculator;
2
3 import calculator.binaryop.*;
4 import calculator.unaryop.Inverse;
5 import calculator.unaryop.Square;
6 import calculator.unaryop.SquareRoot;
7
8 import java.awt.Color;
9 import java.awt.Font;
10 import java.awt.GridBagConstraints;
11 import java.awt.GridBagLayout;
12 import java.awt.Insets;
13
14 import javax.swing.JButton;
15 import javax.swing.JFrame;
16 import javax.swing.JLabel;
17 import javax.swing.JList;
18 import javax.swing.JScrollPane;
19 import javax.swing.JTextField;
20
21 //import java.awt.event.*;
22
23 public class JCalculator extends JFrame
24 {
25     // Tableau representant une pile vide
26     private static final String[] empty = { "< empty stack >" };
27
28     // Zone de texte contenant la valeur introduite ou resultat courant
29     private final JTextField jNumber = new JTextField("0");
30
31     // Composant liste representant le contenu de la pile
32     private final JList<String> jStack = new JList<>(empty);
33
34     // Contraintes pour le placement des composants graphiques
35     private final GridBagConstraints constraints = new GridBagConstraints();
36
37     private final State calcState = new State();
38
39     // Mise à jour de l'interface apres une operation (jList et jStack)
40     private void update()
41     {
42         // Modifier une zone de texte, JTextField.setText(string nom)
43         // Modifier un composant liste, JList.setListData(Object[] tableau)
44         if(calcState.error) {
45             jNumber.setText("ERROR :3");
46         } else {
47             jNumber.setText(calcState.current);
48         }
49         if(!calcState.stack.empty()){
50             Object[] objets = calcState.stack.array();
51             String[] chaines = new String[objets.length];
52
53             for (int i = 0; i < objets.length; i++) {
54                 chaines[i] = String.valueOf(objets[i]);
55             }
56             jStack.setListData(chaines);
57
58         } else {
59             jStack.setListData(empty);
60         }
61     }
62
63     // Ajout d'un bouton dans l'interface et de l'operation associee,
64     // instance de la classe Operation, possedeant une methode execute()
65     private void addOperatorButton(String name, int x, int y, Color color,
66                                   final Operator operator)
67     {
```

File - JCalculator.java

```
68     JButton b = new JButton(name);
69     b.setForeground(color);
70     constraints.gridx = x;
71     constraints.gridy = y;
72     getContentPane().add(b, constraints);
73     b.addActionListener((e) -> {
74         operator.execute();
75         update();
76     });
77 }
78
79 public JCalculator()
80 {
81     super("JCalculator");
82     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
83     getContentPane().setLayout(new GridBagLayout());
84
85     // Contraintes des composants graphiques
86     constraints.insets = new Insets(3, 3, 3, 3);
87     constraints.fill = GridBagConstraints.HORIZONTAL;
88
89     // Nombre courant
90     jNumber.setEditable(false);
91     jNumber.setBackground(Color.WHITE);
92     jNumber.setHorizontalAlignment(JTextField.RIGHT);
93     constraints.gridx = 0;
94     constraints.gridy = 0;
95     constraints.gridwidth = 5;
96     getContentPane().add(jNumber, constraints);
97     constraints.gridwidth = 1; // reset width
98
99     // Rappel de la valeur en memoire
100    addOperatorButton("MR", 0, 1, Color.RED, new MemoryRecall(calcState));
101
102    // Stockage d'une valeur en memoire
103    addOperatorButton("MS", 1, 1, Color.RED, new MemoryStore(calcState));
104
105    // Backspace
106    addOperatorButton("<=", 2, 1, Color.RED, new Backspace(calcState));
107
108    // Mise a zero de la valeur courante + suppression des erreurs
109    addOperatorButton("CE", 3, 1, Color.RED, new ClearError(calcState));
110
111    // Comme CE + vide la pile
112    addOperatorButton("C", 4, 1, Color.RED, new Clear(calcState));
113
114    // Boutons 1-9
115    for (int i = 1; i < 10; i++)
116        addOperatorButton(String.valueOf(i), (i - 1) % 3, 4 - (i - 1) / 3,
117                          Color.BLUE, new AddToCurrent(calcState, String.valueOf(i)));
118
119    // Bouton 0
120    addOperatorButton("0", 0, 5, Color.BLUE, new AddToCurrent(calcState, "0"));
121
122    // Changement de signe de la valeur courante
123    addOperatorButton("+/-", 1, 5, Color.BLUE, new Opposite(calcState));
124
125    // Operateur point (chiffres apres la virgule ensuite)
126    addOperatorButton(".", 2, 5, Color.BLUE, new AddToCurrent(calcState, "."));
127
128    // Operateurs arithmetiques a deux operande: /, *, -, +
129    addOperatorButton("/", 3, 2, Color.RED, new Division(calcState));
130    addOperatorButton("*", 3, 3, Color.RED, new Multiplication(calcState));
131    addOperatorButton("-", 3, 4, Color.RED, new Subtraction(calcState));
132    addOperatorButton("+", 3, 5, Color.RED, new Addition(calcState));
133
134    // Operateurs arithmetiques a un operande: 1/x, x^2, Sqrt
135    addOperatorButton("1/x", 4, 2, Color.RED, new Inverse(calcState));
```

File - JCalculator.java

```
135     addOperatorButton("x2", 4, 3, Color.RED, new Square(calcState));
136     addOperatorButton("Sqrt", 4, 4, Color.RED, new SquareRoot(calcState));
137
138     // Entrée: met la valeur courante sur le sommet de la pile
139     addOperatorButton("Ent", 4, 5, Color.RED, new AddCurrToStack(calcState));
140
141     // Affichage de la pile
142     JLabel jLabel = new JLabel("Stack");
143     jLabel.setFont(new Font("Dialog", 0, 12));
144     jLabel.setHorizontalAlignment(JLabel.CENTER);
145     constraints.gridx = 5;
146     constraints.gridy = 0;
147     getContentPane().add(jLabel, constraints);
148
149     jStack.setFont(new Font("Dialog", 0, 12));
150     jStack.setVisibleRowCount(8);
151     JScrollPane scrollPane = new JScrollPane(jStack);
152     constraints.gridx = 5;
153     constraints.gridy = 1;
154     constraints.gridheight = 5;
155     getContentPane().add(scrollPane, constraints);
156     constraints.gridheight = 1; // reset height
157
158     setResizable(false);
159     pack();
160     setVisible(true);
161 }
162 }
163 }
```

File - MemoryStore.java

```
1 package calculator;
2
3 /**
4 * Store the State.current in a memory inside this class so that it can be called later.
5 * Extends Operator and implements execute().
6 * {@link Operator}
7 *
8 * @author Arthur Junod
9 * @author Edwin Haeffner
10 * Date : 22/11/2023
11 */
12 public class MemoryStore extends Operator {
13
14     static private double mem;
15
16     /**
17      * Constructs a MemoryStore object.
18      * @param calcState the calculator state
19      */
20     public MemoryStore(State calcState) {
21         super(calcState);
22     }
23
24     /**
25      * Stores the current value of the calculator in a static variable.
26      */
27     @Override
28     void execute() {
29         if(!calcState.current.isEmpty() && !calcState.error) {
30             mem = calcState.getCurrentDouble();
31         }
32     }
33
34     /**
35      * @return the value stored in the static variable.
36      */
37     static double getMem() {
38         return mem;
39     }
40
41
42 }
43
```

File - UnaryOpTest.java

```
1 package calculator;
2
3 import calculator.unaryop.*;
4 import org.junit.Test;
5
6 import static org.junit.Assert.*;
7
8 /**
9  * This tests the functions, using junit 4.13,
10 * of the class Inverse, Square and SquareRoot
11 * inheriting the abstract class UnaryOp.
12 * {@link UnaryOp}
13 * {@link Inverse}
14 * {@link Square}
15 * {@link SquareRoot}
16 *
17 * @author Arthur Junod
18 * @author Edwin Haeffner
19 * Date : 04/12/2023
20 */
21 public class UnaryOpTest {
22     private final State state = new State();
23
24     /**
25      * Call execute() on the unary operation given.
26      *
27      * @param state The state which is the same the unary op initialized with
28      * @param op The unary operation we want to execute on the state
29      * @param curr Value that we want to give to the State.current
30      *             (execute() is used on it)
31      * @return The result of the unary operation in Double
32     */
33     private double testUOpExecute(State state, UnaryOp op, String curr){
34         state.current = curr;
35         op.execute();
36         return state.getCurrentDouble();
37     }
38
39     /**
40      * Test if the inverse works normally
41      */
42     @Test
43     public void inverseTest(){
44         assertEquals(0.25, testUOpExecute(state, new Inverse(state), "4"), 0.001);
45     }
46
47     /**
48      * Test if the square operation works
49      */
50     @Test
51     public void squareTest(){
52         assertEquals(4, testUOpExecute(state, new Square(state), "2"), 0.001);
53     }
54
55     /**
56      * Test if the square root operation works
57      */
58     @Test
59     public void squareRootTest(){
60         assertEquals(4, testUOpExecute(state, new SquareRoot(state), "16"), 0.001);
61     }
62
63     /**
64      * Test if that we do not execute() anything if the error flag is true in the state
65      */
66     @Test
67     public void testErrorExecute(){
```

File - UnaryOpTest.java

```
68     State state = new State();
69     state.error = true;
70     // We don't execute anything so the state will stay with current = 1.
71     assertEquals(1, testUOpExecute(state, new Inverse(state), "1"), 0.001);
72     assertEquals(1, testUOpExecute(state, new Square(state), "1"), 0.001);
73     assertEquals(1, testUOpExecute(state, new SquareRoot(state), "1"), 0.001);
74 }
75
76 /**
77 * Test that we return "Infinity if we do 1/0
78 */
79 @Test
80 public void testZeroInverse(){
81     new Inverse(state).execute();
82     assertEquals("Infinity", state.current);
83 }
84
85 /**
86 * Test that if we take the square root of a negative number
87 * the error flag is set to true.
88 */
89 @Test
90 public void testNegSquareRoot(){
91     state.current = "-3";
92     new SquareRoot(state).execute();
93     assertTrue(state.error);
94 }
95 }
96
97
```

File - AddToCurrent.java

```
1 package calculator;
2 /**
3  * Append the toAdd attribute to the State.current.
4  * Extends Operator and implements execute().
5  * {@link Operator}
6 *
7  * @author Arthur Junod
8  * @author Edwin Haeffner
9  * Date : 22/11/2023
10 */
11 public class AddToCurrent extends Operator {
12
13     private final String toAdd;
14     /**
15      * Constructs a AddToCurrent object.
16      * @param calcState the calculator state
17      * @param toAdd the number to add to the current value of the calculator
18      */
19     public AddToCurrent(State calcState, String toAdd) {
20         super(calcState);
21         this.toAdd = toAdd;
22     }
23
24     /**
25      * Append a number or a dot to the current value of the calculator.
26      */
27     @Override
28     void execute() {
29         if(calcState.error) return;
30
31         if(calcState.isResult){
32             calcState.isResult = false;
33             calcState.stack.push(calcState.getCurrentDouble());
34             calcState.current = "";
35         }
36         //Remove the 0 if adding number, else keep it
37         if(calcState.current.equals("0") && !toAdd.equals(".")){
38             calcState.current = "";
39         }
40         //If the current value already contains a dot, don't add another one
41         if(!calcState.current.contains(".")) {
42             calcState.current += toAdd;
43         }
44     }
45 }
46 }
```

File - BinaryOpTest.java

```
1 package calculator;
2
3 import calculator.binaryop.*;
4 import org.junit.Test;
5
6 import static org.junit.Assert.*;
7
8 /**
9  * This tests the functions, using junit 4.13,
10 * of the class Addition, Division, Subtraction and Multiplication
11 * inheriting the abstract class BinaryOp.
12 * {@link BinaryOp}
13 * {@link Addition}
14 * {@link Subtraction}
15 * {@link Division}
16 * {@link Multiplication}
17 *
18 * @author Arthur Junod
19 * @author Edwin Haeffner
20 * Date : 04/12/2023
21 */
22 public class BinaryOpTest {
23
24     private final State state = new State();
25
26     /**
27      * Call execute() on the binary operation given.
28      *
29      * @param state State of the binary operation.
30      * @param op Binary operation given.
31      * @param curr Value the State.current will be set (second operand of operation).
32      * @param stackTop Value of the top of the State.stack
33      *                  (first operand of the operation).
34      * @return The result of the binary operation in Double
35      */
36     private double testB0pExecute(State state, BinaryOp op,
37                                   String curr, String stackTop){
38         state.current = curr;
39         state.stack.push(Double.parseDouble(stackTop));
40         op.execute();
41         return state.getCurrentDouble();
42     }
43
44
45     /**
46      * Test the Addition with valid inputs
47      */
48     @Test
49     public void testAddition(){
50         assertEquals(17, testB0pExecute(state, new Addition(state), "9", "8"), 0.001);
51     }
52
53     /**
54      * Test the Division with valid inputs
55      */
56     @Test
57     public void testDivision(){
58         assertEquals(4, testB0pExecute(state, new Division(state), "4", "16"), 0.001);
59     }
60
61     /**
62      * Test the Multiplication with valid inputs
63      */
64     @Test
65     public void testMultiplication(){
66         assertEquals(56,
67                     testB0pExecute(state, new Multiplication(state), "7", "8"), 0.001);
68     }
69 }
```

File - BinaryOpTest.java

```
68     }
69
70     /**
71      * Test the Subtraction with valid inputs
72      */
73     @Test
74     public void testSubtraction(){
75         assertEquals(3, testB0pExecute(state, new Subtraction(state), "4", "7"), 0.001);
76     }
77
78     /**
79      * Test that the operation do nothing when error flag of State is true
80      */
81     @Test
82     public void testExecuteError(){
83         state.error = true;
84         assertEquals(1, testB0pExecute(state, new Subtraction(state), "1", "1"), 0.001);
85         assertEquals(1, testB0pExecute(state, new Addition(state), "1", "1"), 0.001);
86         assertEquals(1,
87             testB0pExecute(state, new Multiplication(state), "1", "1"), 0.001);
88         assertEquals(1, testB0pExecute(state, new Division(state), "1", "1"), 0.001);
89     }
90
91     /**
92      * Test the error flag of State is put to true when using
93      * a binary operation on an empty stack
94      */
95     @Test
96     public void testStackEmptyError(){
97         new Subtraction(state).execute();
98         assertTrue(state.error);
99     }
100
101    /**
102     * Test that we get "Infinity" when dividing by 0 >:( 
103     */
104    @Test
105    public void testDivideByZero(){
106        state.stack.push(14.);
107        new Division(state).execute();
108        assertEquals("Infinity", state.current);
109    }
110
111    /**
112     * Test we get "NaN" when dividing two zeros together
113     */
114    @Test
115    public void testDivideZeros(){
116        state.stack.push(0.);
117        new Division(state).execute();
118        assertEquals("NaN", state.current);
119    }
120 }
121
```

File - MemoryRecall.java

```
1 package calculator;
2
3 /**
4 * Recall the memory stored when MemoryStore is called.
5 * Extends Operator and implements execute().
6 * {@link Operator}
7 * {@link MemoryStore}
8 *
9 * @author Arthur Junod
10 * @author Edwin Haeffner
11 * Date : 22/11/2023
12 */
13 public class MemoryRecall extends Operator{
14
15     public MemoryRecall(State calcState) {super(calcState);}
16
17     /**
18      * Recalls the stored value by the MemoryStore class and puts it
19      * in the current value of the calculator.
20      */
21     @Override
22     void execute(){
23         if(!calcState.error) {
24             calcState.current = String.valueOf(MemoryStore.getMem());
25             if (calcState.current.equals("0")) calcState.isResult = false;
26         }
27     }
28
29
30
31
32 }
33
34
```

File - OppositeTest.java

```
1 package calculator;
2
3 import org.junit.Test;
4 import static org.junit.Assert.*;
5
6 /**
7 * This tests the functions, using junit 4.13, of the class Opposite
8 * {@link Opposite}
9 *
10 * @author Arthur Junod
11 * @author Edwin Haeffner
12 * Date : 04/12/2023
13 */
14 public class OppositeTest {
15     State state = new State();
16
17     /**
18      * Test that Opposite works
19      */
20     @Test
21     public void testOpposite(){
22         state.current = "-1267";
23         new Opposite(state).execute();
24         assertEquals("1267", state.current);
25     }
26
27     /**
28      * Test that it does nothing when there's a zero in State.current
29      */
30     @Test
31     public void testZero(){
32         new Opposite(state).execute();
33         assertEquals("0", state.current);
34     }
35
36     /**
37      * Test that it does nothing when the error flag is true
38      */
39     @Test
40     public void testError(){
41         state.current = "589";
42         state.error = true;
43         new Opposite(state).execute();
44         assertEquals("589", state.current);
45     }
46
47     /**
48      * Test that it does nothing when the State.current = "NaN"
49      */
50     @Test
51     public void testNaN(){
52         state.current = "NaN";
53         new Opposite(state).execute();
54         assertEquals("NaN", state.current);
55     }
56
57     /**
58      * Test that it does nothing when the State.current = "Infinity"
59      */
60     @Test
61     public void testInfinity(){
62         state.current = "Infinity";
63         new Opposite(state).execute();
64         assertEquals("Infinity", state.current);
65     }
66 }
```

File - BackspaceTest.java

```
1 package calculator;
2
3 import org.junit.Test;
4
5 import static org.junit.Assert.*;
6 /**
7 * This tests the functions, using junit 4.13, of the class Backspace
8 * {@link Backspace}
9 *
10 * @author Arthur Junod
11 * @author Edwin Haeffner
12 * Date : 04/12/2023
13 */
14 public class BackspaceTest {
15     State state = new State();
16
17     /**
18      * Test the backspace
19      */
20     @Test
21     public void testBackspace(){
22         state.current = "123.45";
23         new Backspace(state).execute();
24         assertEquals("123.4", state.current);
25     }
26
27     /**
28      * Test that if we backspace when at State.current.length = 1 it just put 0
29      */
30     @Test
31     public void testLengthAt1(){
32         state.current = "1";
33         new Backspace(state).execute();
34         new Backspace(state).execute();
35         new Backspace(state).execute();
36         assertEquals("0", state.current);
37     }
38
39     /**
40      * Test that we cannot backspace when the error flag is true
41      */
42     @Test
43     public void testError(){
44         state.current = "31";
45         state.error = true;
46         new Backspace(state).execute();
47         assertEquals("31", state.current);
48     }
49
50     /**
51      * Test that we cannot backspace when there's "Infinity" in State.current
52      * so that we can't have invalid values
53      */
54     @Test
55     public void testInfinity(){
56         state.current = "Infinity";
57         new Backspace(state).execute();
58         assertEquals("Infinity", state.current);
59     }
60
61     /**
62      * Test that we cannot backspace when there's "NaN" in State.current
63      * so that we can't have invalid values
64      */
65     @Test
66     public void testNaN(){
67         state.current = "NaN";
```

File - BackspaceTest.java

```
68         new Backspace(state).execute();
69         assertEquals("NaN", state.current);
70     }
71 }
72
```

File - AddCurrToStack.java

```
1 package calculator;
2
3 /**
4 * Push the State.current string into the State.stack.
5 * Extends Operator and implements execute().
6 * {@link Operator}
7 *
8 * @author Arthur Junod
9 * @author Edwin Haeffner
10 * Date : 22/11/2023
11 */
12 public class AddCurrToStack extends Operator {
13
14     public AddCurrToStack(State calcState){super(calcState);}
15
16     /**
17      * Adds the current value of the calculator to the stack.
18      */
19     @Override
20     void execute() {
21         if(!calcState.error) {
22             calcState.stack.push(calcState.getCurrentDouble());
23             calcState.current = "0";
24             calcState.isResult = false;
25         }
26     }
27 }
28
```

File - AddToCurrentTest.java

```
1 package calculator;
2
3 import org.junit.Test;
4
5 import static org.junit.Assert.assertEquals;
6
7 /**
8 * This tests the functions, using junit 4.13, of the class AddToCurrent
9 * {@link AddToCurrent}
10 *
11 * @author Arthur Junod
12 * @author Edwin Haeffner
13 * Date : 04/12/2023
14 */
15 public class AddToCurrentTest {
16     State state = new State();
17
18     /**
19      * Test if the execute() of AddToCurrent works
20      */
21     @Test
22     public void testAddToCurrent(){
23         new AddToCurrent(state, "5").execute();
24         assertEquals("5", state.current);
25     }
26
27     /**
28      * Test if we can make 0.* numbers
29      */
30     @Test
31     public void testDoubleWithZeroAtStart(){
32         new AddToCurrent(state, ".").execute();
33         new AddToCurrent(state, "5").execute();
34         assertEquals("0.5", state.current);
35     }
36
37     /**
38      * Test if we prohibit the addition of multiple points
39      */
40     @Test
41     public void testMultiplePoints(){
42         new AddToCurrent(state, "4").execute();
43         new AddToCurrent(state, ".").execute();
44         new AddToCurrent(state, "3").execute();
45         new AddToCurrent(state, ".").execute();
46         assertEquals("4.3", state.current);
47     }
48
49     /**
50      * Test if we prohibit the addition of multiple zeros
51      */
52     @Test
53     public void testAddMultipleZeros(){
54         new AddToCurrent(state, "0").execute();
55         new AddToCurrent(state, "0").execute();
56         new AddToCurrent(state, "0").execute();
57         assertEquals("0", state.current);
58     }
59
60     /**
61      * Test that we can't change current when error flag of State is true
62      */
63     @Test
64     public void testAddError(){
65         state.error = true;
66         new AddToCurrent(state, "5").execute();
67         assertEquals("0", state.current);
```

File - AddToCurrentTest.java

```
68     }
69
70     /**
71      * Test that we put the current into the stack when it's a result,
72      * and we add a new number
73      */
74     @Test
75     public void testAddWhenResult(){
76         state.current = "134.5";
77         state.isResult = true;
78         new AddToCurrent(state, "8").execute();
79         assertEquals("8", state.current);
80         assertEquals(134.5, state.stack.pop(), 0.001);
81     }
82 }
83
```

File - AddCurrToStackTest.java

```
1 package calculator;
2
3 import org.junit.Test;
4
5 import static org.junit.Assert.*;
6 /**
7 * This tests the functions, using junit 4.13, of the class AddCurrToStack
8 * {@link AddToCurrent}
9 *
10 * @author Arthur Junod
11 * @author Edwin Haeffner
12 * Date : 04/12/2023
13 */
14 public class AddCurrToStackTest {
15
16     State state = new State();
17
18     /**
19      * Test that AddCurrToStack works
20     */
21     @Test
22     public void testAddCurrToStack(){
23         state.current = "5.12";
24         new AddCurrToStack(state).execute();
25         assertEquals("0", state.current);
26         assertEquals(5.12, state.stack.pop(), 0.001);
27     }
28
29     /**
30      * Test that we can't add something to the stack when the error flag is true
31     */
32     @Test
33     public void testError(){
34         state.error = true;
35         state.current = "6.78";
36         new AddCurrToStack(state).execute();
37         assertTrue(state.stack.empty());
38     }
39
40 }
41
```

File - ClearErrorAndClearTest.java

```
1 package calculator;
2
3 import org.junit.Test;
4
5 import static org.junit.Assert.*;
6
7 /**
8 * This tests the functions, using junit 4.13, of the class Clear and ClearError
9 * {@link Clear}
10 * {@link ClearError}
11 *
12 * @author Arthur Junod
13 * @author Edwin Haeffner
14 * Date : 04/12/2023
15 */
16 public class ClearErrorAndClearTest {
17
18     State state = new State();
19
20     /**
21      * Test that ClearError works and does nothing to the stack
22      */
23     @Test
24     public void testClearError(){
25         state.current = "1235";
26         state.error = true;
27         state.isResult = true;
28         state.stack.push(234.);
29         new ClearError(state).execute();
30         assertFalse(state.error);
31         assertFalse(state.isResult);
32         assertEquals("0", state.current);
33         assertEquals(234, state.stack.pop(), 0.001);
34     }
35
36     /**
37      * Test that Clear works
38      */
39     @Test
40     public void testClear(){
41         state.current = "124";
42         state.error = true;
43         state.isResult = true;
44         state.stack.push(2.67);
45         new Clear(state).execute();
46         assertFalse(state.error);
47         assertFalse(state.isResult);
48         assertEquals("0", state.current);
49         assertTrue(state.stack.empty());
50     }
51 }
52 }
```

File - MemoryStoreAndRecallTest.java

```
1 package calculator;
2
3 import org.junit.Test;
4
5 import static org.junit.Assert.*;
6
7 /**
8 * This tests the functions, using junit 4.13, of the class MemoryStore and MemoryRecall
9 * {@link MemoryStore}
10 * {@link MemoryRecall}
11 *
12 * @author Arthur Junod
13 * @author Edwin Haeffner
14 * Date : 04/12/2023
15 */
16 public class MemoryStoreAndRecallTest {
17     State state = new State();
18
19     /**
20      * Test that MemoryStore and MemoryRecall work properly
21      */
22     @Test
23     public void testMemoryStoreAndRecall(){
24         state.current = "43";
25         new MemoryStore(state).execute();
26         assertEquals("43", state.current);
27         assertEquals(43, MemoryStore.getMem(), 0.001);
28         state.current = "0";
29         new MemoryRecall(state).execute();
30         assertEquals("43.0", state.current);
31     }
32
33     /**
34      * Test that neither MemoryStore nor MemoryRecall does nothing
35      * when the error flag is true
36      */
37     @Test
38     public void testError(){
39         state.current = "53";
40         state.error = true;
41         new MemoryStore(state).execute();
42         assertEquals(0, MemoryStore.getMem(), 0.001);
43         new MemoryRecall(state).execute();
44         assertEquals("53", state.current);
45     }
46
47
48
49 }
50
```

File - Square.java

```
1 package calculator.unaryop;
2
3 import calculator.State;
4 import calculator.UnaryOp;
5
6 /**
7 * The class implementing the square unary operation.
8 * {@link UnaryOp}
9 *
10 * @author Arthur Junod
11 * @author Edwin Haeffner
12 * Date : 22/11/2023
13 */
14 public class Square extends UnaryOp {
15     public Square(State calcState) {
16         super(calcState);
17     }
18     /**
19      * @return the square of the given number.
20      */
21     protected double operate(double a) {
22
23         return a * a;
24     }
25 }
```

File - Inverse.java

```
1 package calculator.unaryop;
2
3 import calculator.UnaryOp;
4 import calculator.State;
5
6 /**
7 * The class implementing the inverse unary operation.
8 * {@link UnaryOp}
9 *
10 * @author Arthur Junod
11 * @author Edwin Haeffner
12 * Date : 04/12/2023
13 */
14 public class Inverse extends UnaryOp {
15     public Inverse(State calcState){super(calcState);}
16     /**
17      * @return the inverse of the given number.
18      */
19     protected double operate(double a) {
20         return 1/a;
21     }
22 }
23
```

File - SquareRoot.java

```
1 package calculator.unaryop;
2
3 import calculator.State;
4 import calculator.UnaryOp;
5
6 import static java.lang.Math.sqrt;
7
8 /**
9  * The class implementing the square root unary operation.
10 * {@link UnaryOp}
11 *
12 * @author Arthur Junod
13 * @author Edwin Haeffner
14 * Date : 22/11/2023
15 */
16 public class SquareRoot extends UnaryOp {
17     public SquareRoot(State calcState) {
18         super(calcState);
19     }
20
21     /**
22      * @return the square root of the given number.
23      */
24     protected double operate(double a) {
25         if(a < 0) setError();
26         return sqrt(a);
27     }
28 }
```

File - Addition.java

```
1 package calculator.binaryop;
2
3 import calculator.State;
4 import calculator.BinaryOp;
5
6 /**
7  * Addition class that extends the abstract class BinaryOp and implements operate(),
8  * it can execute an addition.
9  * {@link BinaryOp}
10 *
11 * @author Arthur Junod
12 * @author Edwin Haeffner
13 * Date : 22/11/2023
14 */
15 public class Addition extends BinaryOp {
16
17     public Addition(State calcState){super(calcState);}
18
19     /**
20      * @return the sum of the two given numbers.
21      */
22     protected double operate(double a, double b){
23         return a + b;
24     }
25 }
26
```

File - Division.java

```
1 package calculator.binaryop;
2
3 import calculator.State;
4 import calculator.BinaryOp;
5
6 /**
7  * Division class that extends the abstract class BinaryOp and implements operate(),
8  * it can execute a division.
9  * {@link BinaryOp}
10 *
11 * @author Arthur Junod
12 * @author Edwin Haeffner
13 * Date : 22/11/2023
14 */
15 public class Division extends BinaryOp {
16
17     public Division(State calcState){super(calcState);}
18
19     /**
20      * @return the division of the two given numbers.
21      */
22     protected double operate(double a, double b){
23         return a / b;
24     }
25 }
```

File - Subtraction.java

```
1 package calculator.binaryop;
2
3 import calculator.State;
4 import calculator.BinaryOp;
5
6 /**
7  * Subtraction class that extends the abstract class BinaryOp and implements operate(),
8  * it can execute a subtraction.
9  * {@link BinaryOp}
10 *
11 * @author Arthur Junod
12 * @author Edwin Haeffner
13 * Date : 22/11/2023
14 */
15 public class Subtraction extends BinaryOp {
16
17     public Subtraction(State calcState){super(calcState);}
18     /**
19      * @return the difference of the two given numbers.
20      */
21     protected double operate(double a, double b){
22         return a - b;
23     }
24 }
```

File - Multiplication.java

```
1 package calculator.binaryop;
2
3 import calculator.State;
4 import calculator.BinaryOp;
5
6 /**
7 * Multiplication class that extends the abstract class BinaryOp
8 * and implements operate(), it can execute a multiplication.
9 * {@link BinaryOp}
10 *
11 * @author Arthur Junod
12 * @author Edwin Haeffner
13 * Date : 22/11/2023
14 */
15 public class Multiplication extends BinaryOp {
16
17     public Multiplication(State calcState){super(calcState);}
18
19     /**
20      * @return the multiplication of the two given numbers.
21      */
22     protected double operate(double a, double b){
23         return a * b;
24     }
25 }
```