File - Main.java

```java
import calculator.Calculator;
import calculator.JCalculator;

public class Main
{
  public static void main(String ... args) {
    if(args.length > 0 && args[0].equals("-t")) {
        new Calculator().run();
    } else new JCalculator();
  }
}

```

File - Node.java

```java
package util;
/**
 * This represents a generic node in a linked structure.
 * It is designed to hold data of a specified type, along
 * with a reference to the next node in the structure.
 *
 * @param <T> The type of the data to be stored within the node.
 */
public class Node<T> {

     T data;
     Node<T> next;

    /**
     * Creates a new node with the provided data and the reference
     * to the next node.
     *
     * @param data The data to be stored within the node.
     * @param next The next node in the linked structure or
     *              {@code null} if this is the last node.
     */
    public Node(T data, Node<T> next){
        this.data = data;
        this.next = next;
    }

}

```

File - Stack.java

```java
package util;

import java.util.EmptyStackException;

/**
 * Generic stack data structure implementation.
 * @param <T> the type of elements in the stack
 */
public class Stack<T> {

    private Node<T> stackTop;
    private int size;

    /**
     * Constructs an empty stack.
     */
    public Stack() {
        stackTop = null;
        size = 0;
    }

    /**
     * Checks if the stack is empty.
     * @return true if the stack is empty, false otherwise.
     */
    public boolean empty() {
        return stackTop == null;
    }

    /**
     * Pushes an element onto the top of the stack.
     * @param data the element to be pushed.
     */
    public void push(T data) {
        stackTop = new Node<>(data, stackTop);
        size++;
    }

    /**
     * Removes and returns the element at the top of the stack.
     * @return the element at the top of the stack
     * @throws EmptyStackException if the stack is empty
     */
    public T pop() throws EmptyStackException {
        if (this.empty()) {
            throw new EmptyStackException();
        }
        T tmp = stackTop.data;
        stackTop = stackTop.next;
        size--;
        return tmp;
    }

    /**
     * Removes all elements from the stack.
     */
    public void clear() {
        stackTop = null;
        size = 0;
        //Garbage collector will do the rest...
    }

    /**
     * @return a string representation of the stack
     */
    public String toString() {
        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

```java
package util;

/**
 * Generic stack iterator, used for the generic stack implementation.
 * @param <T> the type of elements in the stack
 */
public class StackIterator<T> {

    private Node<T> current;

    /**
     * Checks if there is a next element in the iterator.
     * @return true if there is a next element, false otherwise.
     */
    public boolean hasNext() {
        return (current.next != null);
    }

    /**
     * Returns the data of the next element of the iterator and advances it.
     * @return the data of the next element of the iterator.
     */
    public T next() {
        current = current.next;
        return current.data;
    }

    /**
     * Constructs a StackIterator object with the given node.
     * @param current the node
     */
    public StackIterator(Node<T> current) {
        this.current = current;
    }
}
```

File - StackTest.java

```java
package util.test;

import org.junit.Test;
import java.util.EmptyStackException;

import static org.junit.Assert.*;

import util.*;

/**
 * This tests the functions of the class Stack using junit. {@link Stack}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       13/11/2023
 */
public class StackTest {
    /**
     * Test the empty() method of the implemented stack
     */
    @Test
    public void testEmptyStack() {
        Stack<Integer> stack = new Stack<Integer>();
        assertTrue(stack.empty());
    }

    /**
     * Test the push() and pop() methods of the implemented stack
     */
    @Test
    public void testPushAndPop() {
        Stack<Integer> stack = new Stack<Integer>();

        stack.push(1);
        stack.push(2);
        stack.push(3);

        assertFalse(stack.empty());
        assertEquals((Integer)3, stack.pop());
        assertEquals((Integer)2, stack.pop());
        assertEquals((Integer)1, stack.pop());
        assertTrue(stack.empty());
    }

    /**
     * Test if the Stack throw an exception when using pop() on an empty one
     */
    @Test
    public void testPopEmptyStack() {
        Stack<Integer> stack = new Stack<Integer>();
        assertThrows(EmptyStackException.class, stack::pop);
    }

    /**
     * Test if the print of the stack is correct
     */
    @Test
    public void testToString() {
        Stack<Integer> stack = new Stack<Integer>();
        assertEquals("",stack.toString());

        stack.push(1);
        stack.push(2);
        stack.push(3);

        String expected = "3 2 1";
        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          assertArrayEquals(expected, stack.array());
82      }
83
84  }
85
```

File - Clear.java

```java
package calculator;
/**
 * Clear everything that ClearError does but also empties the stack.
 * Extends Operator and implements execute().
 * {@link ClearError}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class Clear extends ClearError {
    public Clear(State calcState) {
        super(calcState);
    }

    /**
     * Clears the current value of the calculator and removes errors if any.
     */
    @Override
    void execute() {

        super.execute();
        calcState.stack.clear();

    }
}

```

File - State.java

```java
package calculator;

import util.Stack;

/**
 * This class holds the current state of a calculator session.
 * This includes the current stack of numbers, the current input or
 * result as a string, and flags indicating if the current input is
 * a result or if there has been an error.
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class State {

    Stack<Double> stack;
    String current;
    boolean isResult;
    boolean error;

    /**
     * Constructs a State object.
     */
    public State() {
        stack    = new Stack<>();
        current  = "0";
        isResult = false;
        error    = false;
    }

    /**
     * @return the current value as a double.
     * @throws NumberFormatException if current isn't a double.
     * @throws NullPointerException if the string is null
     */
    double getCurrentDouble() {
        return Double.parseDouble(current);
    }
}

```

File - UnaryOp.java

```java
package calculator;

/**
 * This class is used as a backbone for all the unary operators.
 *  * It is then required to implement the operate method with the desired operator.
 * {@link Operator}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public abstract class UnaryOp extends Operator {

    /**
     * Constructs a UnaryOp object.
     * @param calcState the calculator state
     */
    public UnaryOp(State calcState) {
        super(calcState);
    }

    /**
     * Executes the unary operation on the current double value.
     * The result is stored in the current String from calcState,
     * and the isResult flag is set to true.
     */
    protected void execute() {
        if(!calcState.current.isEmpty() && !calcState.error) {
            calcState.current = String.valueOf(operate(calcState.getCurrentDouble()));
            calcState.isResult = true;
        }
    }

    protected void setError(){
        calcState.error = true;
    }

    /**
     * Performs the unary operation on the given value.
     *
     * @param value the value to operate on
     * @return the result of the operation
     */
    protected abstract double operate(double value);
}
```

File - BinaryOp.java

```java
package calculator;

/**
 * This class is used as a backbone for all the binary operators.
 * It is then required to implement the operate method with the desired operator.
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       28/11/2023
 */
public abstract class BinaryOp extends Operator {
    public BinaryOp(State calcState) {
        super(calcState);
    }

    /**
     * Executes the binary operation by popping an operand from the stack
     * and using it with the current. The result is then put back in the current.
     * If the stack is empty the error flag is set to true.
     */
    @Override
    protected void execute() {

        if (calcState.stack.empty() || calcState.error) {
            calcState.error = true;
        } else {

            double d = calcState.stack.pop();

            //To cast the double to String
            calcState.current = String.valueOf(operate(d, calcState.getCurrentDouble()));
            calcState.isResult = true;
        }

    }

    /**
     * Performs the binary operation on the given values.
     * @param a the first value.
     * @param b the second value.
     * @return the result of the operation.
     */
    protected abstract double operate(double a, double b);

}

```

File - Operator.java

```java
package calculator;


/**
 * This class is used as a backbone for all the operators and is mainly used to
 * store the current state of the calculator and to give an abstract method to
 * execute the operator.
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public abstract class Operator {
    State calcState;

    /**
     * Executes the operator.
     */
    abstract void execute();

    /**
     * Constructs an Operator object.
     * @param calcState the calculator state.
     */
    Operator(State calcState) {
        this.calcState = calcState;
    }

}

```

File - Opposite.java

```java
package calculator;


/**
 * Transform the value of State.current into it's opposite.
 * Extends Operator and implements execute().
 * {@link Operator}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class Opposite extends Operator {

    public Opposite(State calcState){super(calcState);}

    @Override
    void execute() {
        if(!calcState.error && !calcState.current.equals("0")
                && !calcState.current.equals("NaN")
                && !calcState.current.equals("Infinity")) {
            if(calcState.current.charAt(0) == '-'){
                calcState.current = calcState.current.substring(1);
            }else{
                calcState.current = "-" + calcState.current;
            }
        }
    }
}

```

File - Backspace.java

```java
package calculator;

/**
 * Erase the char at the far right of State.curent.
 * Extends Operator and implements execute().
 * {@link Operator}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class Backspace extends Operator {

    /**
     * Constructs a Backspace object.
     * @param calcState the calculator state
     */
    public Backspace(State calcState) {super(calcState);}

    /**
     * Removes the last character from the current value of the calculator.
     */
    @Override
    void execute() {
        if(!calcState.error && !calcState.current.equals("Infinity")
                && !calcState.current.equals("NaN")){
            if (calcState.current.length() > 1) {
                String cur = calcState.current;
                calcState.current = cur.substring(0, cur.length() - 1);
            } else {
                calcState.current = "0";
            }
        }
    }


}

```

File - Calculator.java

```java
package calculator;

import calculator.binaryop.Addition;
import calculator.binaryop.Division;
import calculator.binaryop.Multiplication;
import calculator.binaryop.Subtraction;
import calculator.unaryop.Inverse;
import calculator.unaryop.Square;
import calculator.unaryop.SquareRoot;

import java.util.HashMap;
import java.util.Scanner;

/**
 * This class allow us to launch the calculator in the terminal.
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       28/11/2023
 */
public class Calculator {

    private final State calcState = new State();
    private final Scanner sin = new Scanner(System.in);
    private final HashMap<String, Operator> operations;

    public Calculator(){
        operations = new HashMap<>();
        operations.put("+",    new Addition(calcState));
        operations.put("-",    new Subtraction(calcState));
        operations.put("*",    new Multiplication(calcState));
        operations.put("/",    new Division(calcState));
        operations.put("sqrt", new SquareRoot(calcState));
        operations.put("inv",  new Inverse(calcState));
        operations.put("sq",   new Square(calcState));
        operations.put("c",    new Clear(calcState));
        operations.put("ce",   new ClearError(calcState));
        operations.put("ms",   new MemoryStore(calcState));
        operations.put("mr",   new MemoryRecall(calcState));
    }

    /**
     * This function runs the calculator in the terminal
     */
    public void run() {
        System.out.println("Java Calculator");

        while (true) {
            String in = sin.nextLine();
            if (in.equals("exit")) break;

            // Verify if the user input is an operation
            if (operations.containsKey(in)) {
                operations.get(in).execute();
            } else {
                // The try{}catch{} verifies if the user input is a double
                try {
                    // Append the entier user input in the State.current
                    // and then set it as a result so that it will
                    // be pushed to the State.stack next time we enter a number.
                    new AddToCurrent(calcState,
                            String.valueOf(Double.parseDouble(in))).execute();
                    calcState.isResult = true;
                } catch (NumberFormatException ignore) {
                    System.out.println("Invalid input");
                    continue; // Ask an input again if it's invalid
                }
```

File - Calculator.java

```
            }

            display();
        }
    }

    /**
     * Display the calculator in the terminal.
     */
    private void display()
    {
        String res;
        if (calcState.error) res = "ERROR :3";
        else res = calcState.current;
        System.out.println(res + " " + calcState.stack.toString());
    }

}

```

File - ClearError.java

```java
package calculator;

/**
 * Put the 2 flag (error, isResult) to false and put tue State.current back to "0".
 * Extends Operator and implements execute().
 * {@link Operator}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class ClearError extends Operator {

    public ClearError(State calcState) {
        super(calcState);
    }

    @Override
    void execute() {

        calcState.current  = "0";
        calcState.error    = false;
        calcState.isResult = false;

    }
}

```

File - JCalculator.java

```java
package calculator;

import calculator.binaryop.*;
import calculator.unaryop.Inverse;
import calculator.unaryop.Square;
import calculator.unaryop.SquareRoot;

import java.awt.Color;
import java.awt.Font;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.JTextField;

//import java.awt.event.*;

public class JCalculator extends JFrame
{
  // Tableau representant une pile vide
  private static final String[] empty = { "< empty stack >" };

  // Zone de texte contenant la valeur introduite ou resultat courant
  private final JTextField jNumber = new JTextField("0");

  // Composant liste representant le contenu de la pile
  private final JList<String> jStack = new JList<>(empty);

  // Contraintes pour le placement des composants graphiques
  private final GridBagConstraints constraints = new GridBagConstraints();

  private final State calcState = new State();

  // Mise à jour de l'interface apres une operation (jList et jStack)
  private void update()
  {
    // Modifier une zone de texte, JTextField.setText(string nom)
    // Modifier un composant liste, JList.setListData(Object[] tableau)
    if(calcState.error) {
      jNumber.setText("ERROR :3");
    } else {
      jNumber.setText(calcState.current);
    }
    if(!calcState.stack.empty()){
      Object[] objets = calcState.stack.array();
      String[] chaines = new String[objets.length];

      for (int i = 0; i < objets.length; i++) {
        chaines[i] = String.valueOf(objets[i]);
      }
        jStack.setListData(chaines);

    } else {
      jStack.setListData(empty);
    }
  }

  // Ajout d'un bouton dans l'interface et de l'operation associee,
  // instance de la classe Operation, possedeant une methode execute()
  private void addOperatorButton(String name, int x, int y, Color color,
                 final Operator operator)
  {
```

```
    JButton b = new JButton(name);
    b.setForeground(color);
    constraints.gridx = x;
    constraints.gridy = y;
    getContentPane().add(b, constraints);
    b.addActionListener((e) -> {
    operator.execute();
    update();
      });
  }

  public JCalculator()
  {
    super("JCalculator");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    getContentPane().setLayout(new GridBagLayout());

    // Contraintes des composants graphiques
    constraints.insets = new Insets(3, 3, 3, 3);
    constraints.fill = GridBagConstraints.HORIZONTAL;

    // Nombre courant
    jNumber.setEditable(false);
    jNumber.setBackground(Color.WHITE);
    jNumber.setHorizontalAlignment(JTextField.RIGHT);
    constraints.gridx = 0;
    constraints.gridy = 0;
    constraints.gridwidth = 5;
    getContentPane().add(jNumber, constraints);
    constraints.gridwidth = 1; // reset width

    // Rappel de la valeur en memoire
    addOperatorButton("MR", 0, 1, Color.RED, new MemoryRecall(calcState));

    // Stockage d'une valeur en memoire
    addOperatorButton("MS", 1, 1, Color.RED, new MemoryStore(calcState));

    // Backspace
    addOperatorButton("<=", 2, 1, Color.RED, new Backspace(calcState));

    // Mise a zero de la valeur courante + suppression des erreurs
    addOperatorButton("CE", 3, 1, Color.RED, new ClearError(calcState));

    // Comme CE + vide la pile
    addOperatorButton("C",  4, 1, Color.RED, new Clear(calcState));

    // Boutons 1-9
    for (int i = 1; i < 10; i++)
      addOperatorButton(String.valueOf(i), (i - 1) % 3, 4 - (i - 1) / 3,
            Color.BLUE, new AddToCurrent(calcState, String.valueOf(i)));
    // Bouton 0
    addOperatorButton("0", 0, 5, Color.BLUE, new AddToCurrent(calcState, "0"));

    // Changement de signe de la valeur courante
    addOperatorButton("+/-", 1, 5, Color.BLUE, new Opposite(calcState));

    // Operateur point (chiffres apres la virgule ensuite)
    addOperatorButton(".", 2, 5, Color.BLUE, new AddToCurrent(calcState, "."));

    // Operateurs arithmetiques a deux operandes: /, *, -, +
    addOperatorButton("/", 3, 2, Color.RED, new Division(calcState));
    addOperatorButton("*", 3, 3, Color.RED, new Multiplication(calcState));
    addOperatorButton("-", 3, 4, Color.RED, new Subtraction(calcState));
    addOperatorButton("+", 3, 5, Color.RED, new Addition(calcState));

    // Operateurs arithmetiques a un operande: 1/x, x^2, Sqrt
    addOperatorButton("1/x", 4, 2, Color.RED, new Inverse(calcState));
```

File - JCalculator.java

```
135     addOperatorButton("x^2", 4, 3, Color.RED, new Square(calcState));
136     addOperatorButton("Sqrt", 4, 4, Color.RED, new SquareRoot(calcState));
137 
138     // Entree: 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

```java
package calculator;

/**
 * Store the State.current in a memory inside this class so that it can be called later.
 * Extends Operator and implements execute().
 * {@link Operator}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class MemoryStore extends Operator {

    static private double mem;

    /**
     * Constructs a MemoryStore object.
     * @param calcState the calculator state
     */
    public MemoryStore(State calcState) {
        super(calcState);
    }

    /**
     * Stores the current value of the calculator in a static variable.
     */
    @Override
    void execute() {
        if(!calcState.current.isEmpty() && !calcState.error) {
            mem = calcState.getCurrentDouble();
        }
    }

    /**
     * @return the value stored in the static variable.
     */
    static double getMem() {
        return mem;
    }


}

```

File - UnaryOpTest.java

```java
package calculator;

import calculator.unaryop.*;
import org.junit.Test;

import static org.junit.Assert.*;

/**
 * This tests the functions, using junit 4.13,
 * of the class Inverse, Square and SquareRoot
 * inheriting the abstract class UnaryOp.
 * {@link UnaryOp}
 * {@link Inverse}
 * {@link Square}
 * {@link SquareRoot}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       04/12/2023
 */
public class UnaryOpTest {
    private final State state = new State();

    /**
     * Call execute() on the unary operation given.
     *
     * @param state The state which is the same the unary op initialized with
     * @param op The unary operation we want to execute on the state
     * @param curr Value that we want to give to the State.current
     *             (execute() is used on it)
     * @return The result of the unary operation in Double
     */
    private double testUOpExecute(State state, UnaryOp op, String curr){
        state.current = curr;
        op.execute();
        return state.getCurrentDouble();
    }

    /**
     * Test if the inverse works normally
     */
    @Test
    public void inverseTest(){
        assertEquals(0.25, testUOpExecute(state, new Inverse(state), "4"), 0.001);
    }

    /**
     * Test if the square operation works
     */
    @Test
    public void squareTest(){
        assertEquals(4, testUOpExecute(state, new Square(state), "2"), 0.001);
    }

    /**
     * Test if the square root operation works
     */
    @Test
    public void squareRootTest(){
        assertEquals(4, testUOpExecute(state, new SquareRoot(state), "16"), 0.001);
    }

    /**
     * Test if that we do not execute() anything if the error flag is true in the state
     */
    @Test
    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

```java
package calculator;
/**
 * Append the toAdd attribute to the State.current.
 * Extends Operator and implements execute().
 * {@link Operator}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class AddToCurrent extends Operator {

    private final String toAdd;
    /**
     * Constructs a AddToCurrent object.
     * @param calcState the calculator state
     * @param toAdd the number to add to the current value of the calculator
     */
    public AddToCurrent(State calcState, String toAdd) {
        super(calcState);
        this.toAdd = toAdd;
    }

    /**
     * Append a number or a dot to the current value of the calculator.
     */
    @Override
     void execute() {
        if(calcState.error) return;

        if(calcState.isResult){
            calcState.isResult = false;
            calcState.stack.push(calcState.getCurrentDouble());
            calcState.current = "";
        }
        //Remove the 0 if adding number, else keep it
        if(calcState.current.equals("0") && !toAdd.equals(".")){
            calcState.current = "";
        }
        //If the current value already contains a dot, don't add another one
        if(!calcState.current.contains(".") || !toAdd.equals(".")) {
            calcState.current += toAdd;
        }
    }
}

```

File - BinaryOpTest.java

```java
package calculator;

import calculator.binaryop.*;
import org.junit.Test;

import static org.junit.Assert.*;

/**
 * This tests the functions, using junit 4.13,
 * of the class Addition, Division, Subtraction and Multiplication
 * inheriting the abstract class BinaryOp.
 * {@link BinaryOp}
 * {@link Addition}
 * {@link Subtraction}
 * {@link Division}
 * {@link Multiplication}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       04/12/2023
 */
public class BinaryOpTest {

    private final State state = new State();

    /**
     * Call execute() on the binary operation given.
     *
     * @param state State of the binary operation.
     * @param op Binary operation given.
     * @param curr Value the State.current will be set (second operand of operation).
     * @param stackTop Value of the top of the State.stack
     *                 (first operand of the operation).
     * @return The result of the binary operation in Double
     */
    private double testBOpExecute(State state, BinaryOp op,
                                  String curr, String stackTop){
        state.current = curr;
        state.stack.push(Double.parseDouble(stackTop));
        op.execute();
        return state.getCurrentDouble();
    }


    /**
     * Test the Addition with valid inputs
     */
    @Test
    public void testAddition(){
        assertEquals(17, testBOpExecute(state, new Addition(state), "9", "8"), 0.001);
    }

    /**
     * Test the Division with valid inputs
     */
    @Test
    public void testDivision(){
        assertEquals(4, testBOpExecute(state, new Division(state), "4", "16"), 0.001);
    }

    /**
     * Test the Multiplication with valid inputs
     */
    @Test
    public void testMultiplication(){
        assertEquals(56,
                testBOpExecute(state, new Multiplication(state), "7", "8"), 0.001);
```

File - BinaryOpTest.java

```
68      }
69
70      /**
71       * Test the Subtraction with valid inputs
72       */
73      @Test
74      public void testSubtraction(){
75          assertEquals(3, testBOpExecute(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, testBOpExecute(state, new Subtraction(state), "1", "1"), 0.001);
85          assertEquals(1, testBOpExecute(state, new Addition(state), "1", "1"), 0.001);
86          assertEquals(1,
87                  testBOpExecute(state, new Multiplication(state), "1", "1"), 0.001);
88          assertEquals(1, testBOpExecute(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

```java
package calculator;

/**
 * Recall the memory stored when MemoryStore is called.
 * Extends Operator and implements execute().
 * {@link Operator}
 * {@link MemoryStore}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class MemoryRecall extends Operator{

    public MemoryRecall(State calcState) {super(calcState);}

    /**
     * Recalls the stored value by the MemoryStore class and puts it
     * in the current value of the calculator.
     */
    @Override
    void execute(){
        if(!calcState.error) {
            calcState.current = String.valueOf(MemoryStore.getMem());
            if (calcState.current.equals("0")) calcState.isResult = false;
        }
    }



}


```

File - OppositeTest.java

```java
package calculator;

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

/**
 * This tests the functions, using junit 4.13, of the class Opposite
 * {@link Opposite}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       04/12/2023
 */
public class OppositeTest {
    State state = new State();

    /**
     * Test that Opposite works
     */
    @Test
    public void testOpposite(){
        state.current = "-1267";
        new Opposite(state).execute();
        assertEquals("1267", state.current);
    }

    /**
     * Test that it does nothing when there's a zero in State.current
     */
    @Test
    public void testZero(){
        new Opposite(state).execute();
        assertEquals("0", state.current);
    }

    /**
     * Test that it does nothing when the error flag is true
     */
    @Test
    public void testError(){
        state.current = "589";
        state.error = true;
        new Opposite(state).execute();
        assertEquals("589", state.current);
    }

    /**
     * Test that it does nothing when the State.current = "NaN"
     */
    @Test
    public void testNaN(){
        state.current = "NaN";
        new Opposite(state).execute();
        assertEquals("NaN", state.current);
    }

    /**
     * Test that it does nothing when the State.current = "Infinity"
     */
    @Test
    public void testInfinity(){
        state.current = "Infinity";
        new Opposite(state).execute();
        assertEquals("Infinity", state.current);
    }
}
```

File - BackspaceTest.java

```java
package calculator;

import org.junit.Test;

import static org.junit.Assert.*;
/**
 * This tests the functions, using junit 4.13, of the class Backspace
 * {@link Backspace}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       04/12/2023
 */
public class BackspaceTest {
    State state = new State();

    /**
     * Test the backspace
     */
    @Test
    public void testBackspace(){
        state.current = "123.45";
        new Backspace(state).execute();
        assertEquals("123.4", state.current);
    }

    /**
     * Test that if we backspace when at State.current.length = 1 it just put 0
     */
    @Test
    public void testLengthAt1(){
        state.current = "1";
        new Backspace(state).execute();
        new Backspace(state).execute();
        new Backspace(state).execute();
        assertEquals("0", state.current);
    }

    /**
     * Test that we cannot backspace when the error flag is true
     */
    @Test
    public void testError(){
        state.current = "31";
        state.error = true;
        new Backspace(state).execute();
        assertEquals("31", state.current);
    }

    /**
     * Test that we cannot backspace when there's "Infinity" in State.current
     * so that we can't have invalid values
     */
    @Test
    public void testInfinity(){
        state.current = "Infinity";
        new Backspace(state).execute();
        assertEquals("Infinity", state.current);
    }

    /**
     * Test that we cannot backspace when there's "NaN" in State.current
     * so that we can't have invalid values
     */
    @Test
    public void testNaN(){
        state.current = "NaN";
```

File - BackspaceTest.java

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

File - AddCurrToStack.java

```java
package calculator;

/**
 * Push the State.current string into the State.stack.
 * Extends Operator and implements execute().
 * {@link Operator}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class AddCurrToStack extends Operator {

    public AddCurrToStack(State calcState){super(calcState);}

    /**
     * Adds the current value of the calculator to the stack.
     */
    @Override
    void execute() {
        if(!calcState.error) {
            calcState.stack.push(calcState.getCurrentDouble());
            calcState.current = "0";
            calcState.isResult = false;
        }
    }
}

```

File - AddToCurrentTest.java

```java
package calculator;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

/**
 * This tests the functions, using junit 4.13, of the class AddToCurrent
 * {@link AddToCurrent}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       04/12/2023
 */
public class AddToCurrentTest {
    State state = new State();

    /**
     * Test if the execute() of AddToCurrent works
     */
    @Test
    public void testAddToCurrent(){
        new AddToCurrent(state, "5").execute();
        assertEquals("5", state.current);
    }

    /**
     * Test if we can make 0.* numbers
     */
    @Test
    public void testDoubleWithZeroAtStart(){
        new AddToCurrent(state, ".").execute();
        new AddToCurrent(state, "5").execute();
        assertEquals("0.5", state.current);
    }

    /**
     * Test if we prohibit the addition of multiple points
     */
    @Test
    public void testMultiplePoints(){
        new AddToCurrent(state, "4").execute();
        new AddToCurrent(state, ".").execute();
        new AddToCurrent(state, "3").execute();
        new AddToCurrent(state, ".").execute();
        assertEquals("4.3", state.current);
    }

    /**
     * Test if we prohibit the addition of multiple zeros
     */
    @Test
    public void testAddMultipleZeros(){
        new AddToCurrent(state, "0").execute();
        new AddToCurrent(state, "0").execute();
        new AddToCurrent(state, "0").execute();
        assertEquals("0", state.current);
    }

    /**
     * Test that we can't change current when error flag of State is true
     */
    @Test
    public void testAddError(){
        state.error = true;
        new AddToCurrent(state, "5").execute();
        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

```java
package calculator;

import org.junit.Test;

import static org.junit.Assert.*;
/**
 * This tests the functions, using junit 4.13, of the class AddCurrToStack
 * {@link AddToCurrent}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       04/12/2023
 */
public class AddCurrToStackTest {

    State state = new State();

    /**
     * Test that AddCurrToStack works
     */
    @Test
    public void testAddCurrToStack(){
        state.current = "5.12";
        new AddCurrToStack(state).execute();
        assertEquals("0", state.current);
        assertEquals(5.12, state.stack.pop(), 0.001);
    }

    /**
     * Test that we can't add something to the stack when the error flag is true
     */
    @Test
    public void testError(){
        state.error = true;
        state.current = "6.78";
        new AddCurrToStack(state).execute();
        assertTrue(state.stack.empty());
    }

}

```

File - ClearErrorAndClearTest.java

```java
package calculator;

import org.junit.Test;

import static org.junit.Assert.*;

/**
 * This tests the functions, using junit 4.13, of the class Clear and ClearError
 * {@link Clear}
 * {@link ClearError}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       04/12/2023
 */
public class ClearErrorAndClearTest {

    State state = new State();

    /**
     * Test that ClearError works and does nothing to the stack
     */
    @Test
    public void testClearError(){
        state.current = "1235";
        state.error = true;
        state.isResult = true;
        state.stack.push(234.);
        new ClearError(state).execute();
        assertFalse(state.error);
        assertFalse(state.isResult);
        assertEquals("0", state.current);
        assertEquals(234, state.stack.pop(), 0.001);
    }

    /**
     * Test that Clear works
     */
    @Test
    public void testClear(){
        state.current = "124";
        state.error = true;
        state.isResult = true;
        state.stack.push(2.67);
        new Clear(state).execute();
        assertFalse(state.error);
        assertFalse(state.isResult);
        assertEquals("0", state.current);
        assertTrue(state.stack.empty());
    }
}

```

File - MemoryStoreAndRecallTest.java

```java
package calculator;

import org.junit.Test;

import static org.junit.Assert.*;

/**
 * This tests the functions, using junit 4.13, of the class MemoryStore and MemoryRecall
 * {@link MemoryStore}
 * {@link MemoryRecall}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       04/12/2023
 */
public class MemoryStoreAndRecallTest {
    State state = new State();

    /**
     * Test that MemoryStore and MemoryRecall work properly
     */
    @Test
    public void testMemoryStoreAndRecall(){
        state.current = "43";
        new MemoryStore(state).execute();
        assertEquals("43", state.current);
        assertEquals(43, MemoryStore.getMem(), 0.001);
        state.current = "0";
        new MemoryRecall(state).execute();
        assertEquals("43.0", state.current);
    }

    /**
     * Test that neither MemoryStore nor MemoryRecall does nothing
     * when the error flag is true
     */
    @Test
    public void testError(){
        state.current = "53";
        state.error = true;
        new MemoryStore(state).execute();
        assertEquals(0, MemoryStore.getMem(), 0.001);
        new MemoryRecall(state).execute();
        assertEquals("53", state.current);
    }



}

```

File - Square.java

```java
package calculator.unaryop;

import calculator.State;
import calculator.UnaryOp;

/**
 * The class implementing the square unary operation.
 * {@link UnaryOp}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class Square extends UnaryOp {
    public Square(State calcState) {
        super(calcState);
    }
    /**
     * @return the square of the given number.
     */
    protected double operate(double a) {

        return a * a;
    }
}
```

File - Inverse.java

```java
package calculator.unaryop;

import calculator.UnaryOp;
import calculator.State;

/**
 * The class implementing the inverse unary operation.
 * {@link UnaryOp}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       04/12/2023
 */
public class Inverse extends UnaryOp {
    public Inverse(State calcState){super(calcState);}
    /**
     * @return the inverse of the given number.
     */
    protected double operate(double a) {
        return 1/a;
    }
}

```

File - SquareRoot.java

```java
package calculator.unaryop;

import calculator.State;
import calculator.UnaryOp;

import static java.lang.Math.sqrt;

/**
 * The class implementing the square root unary operation.
 * {@link UnaryOp}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class SquareRoot extends UnaryOp {
    public SquareRoot(State calcState) {
        super(calcState);
    }

    /**
     * @return the square root of the given number.
     */
    protected double operate(double a) {
        if(a < 0) setError();
        return sqrt(a);
    }
}
```

File - Addition.java

```java
package calculator.binaryop;

import calculator.State;
import calculator.BinaryOp;

/**
 * Addition class that extends the abstract class BinaryOp and implements operate(),
 * it can execute an addition.
 * {@link BinaryOp}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class Addition extends BinaryOp {

    public Addition(State calcState){super(calcState);}

    /**
     * @return the sum of the two given numbers.
     */
     protected double operate(double a, double b){
        return a + b;
    }
}

```

File - Division.java

```java
package calculator.binaryop;

import calculator.State;
import calculator.BinaryOp;

/**
 * Division class that extends the abstract class BinaryOp and implements operate(),
 * it can execute a division.
 * {@link BinaryOp}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class Division extends BinaryOp {

    public Division(State calcState){super(calcState);}

    /**
     * @return the division of the two given numbers.
     */
    protected double operate(double a, double b){
        return a / b;
    }
}
```

File - Subtraction.java

```java
package calculator.binaryop;

import calculator.State;
import calculator.BinaryOp;

/**
 * Subtraction class that extends the abstract class BinaryOp and implements operate(),
 * it can execute a subtraction.
 * {@link BinaryOp}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class Subtraction extends BinaryOp {

    public Subtraction(State calcState){super(calcState);}
    /**
     * @return the difference of the two given numbers.
     */
    protected double operate(double a, double b){
        return a - b;
    }
}
```

File - Multiplication.java

```java
package calculator.binaryop;

import calculator.State;
import calculator.BinaryOp;

/**
 * Multiplication class that extends the abstract class BinaryOp
 * and implements operate(), it can execute a multiplication.
 * {@link BinaryOp}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       22/11/2023
 */
public class Multiplication extends BinaryOp {

    public Multiplication(State calcState){super(calcState);}
    /**
     * @return the multiplication of the two given numbers.
     */
    protected double operate(double a, double b){
        return a * b;
    }
}
```