```java
package main;

/*
 * JCalculator.java
 *
 * Pier Donini, 9 Jan 2004.
 * edited by Minder Valentin and Bron Sacha on Dec 11 2014.
 */

import javax.swing.*;

import operator.*;

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

public class JCalculator extends JFrame {
    // Tableau representant une pile vide
    private 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 jStack = new JList(empty);

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

    /*
     * Mise a 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)
        jNumber.setText(State.getInstance().getValueString());
        Object [] stack = State.getInstance().getStackState();
        if (stack.length == 0) {
            stack = empty;
        }
        jStack.setListData(stack);
    }

    /*
     * 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(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                operator.execute();
                update();
            }
        });
    }

    /*
     * Constructeur
     */
    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 MROperator());

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

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

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

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

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

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

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

        // Operateurs arithmetiques a deux operandes: /, *, -, +
        addOperatorButton("/", 3, 2, Color.RED, new DivOperator());
        addOperatorButton("*", 3, 3, Color.RED, new TimesOperator());
        addOperatorButton("-", 3, 4, Color.RED, new MinusOperator());
        addOperatorButton("+", 3, 5, Color.RED, new PlusOperator());

        // Operateurs arithmetiques a un operande: 1/x, x^2, Sqrt
        addOperatorButton("1/x", 4, 2, Color.RED, new OneOverXOperator());
        addOperatorButton("x^2", 4, 3, Color.RED, new SquareOperator());
        addOperatorButton("Sqrt", 4, 4, Color.RED, new SqrtOperator());

        // Entree: met la valeur courante sur le sommet de la pile
        addOperatorButton("Ent", 4, 5, Color.RED, new EnterOperator());

        // Affichage de la pile
        JLabel jLabel = new JLabel("Stack");
        jLabel.setFont(new Font("Dialog", 0, 12));
        jLabel.setHorizontalAlignment(JLabel.CENTER);
        constraints.gridx = 5;
        constraints.gridy = 0;
        getContentPane().add(jLabel, constraints);

        jStack.setFont(new Font("Dialog", 0, 12));
        jStack.setVisibleRowCount(8);
        JScrollPane scrollPane = new JScrollPane(jStack);
        constraints.gridx = 5;
        constraints.gridy = 1;
        constraints.gridheight = 5;
        getContentPane().add(scrollPane, constraints);
        constraints.gridheight = 1; // reset height

        setResizable(false);
        pack();
    }

    /*
     * main()
     */
    public static void main(String args[]) {
        new JCalculator().setVisible(true);
    }
}

```

```java
package main;

import java.util.Arrays;
import java.util.Scanner;

import operator.*;

/**
 * This class represent the usage of the Calculator from a terminal/console. It
 * allows the user to enter values and operators with the keyboard, and the
 * state of the current value and stack is printed after each operations.
 *
 * @author Sacha Bron
 * @author Valentin Minder
 */
public class Calculator {

    public static void main(String[] args) {

        State state = State.getInstance();
        Scanner scan = new Scanner(System.in);
        System.out.println("Welcome to the REVERSE POLISH TERMINAL CALCULATOR");

        String line = "";
        boolean flag = true;

        while (flag) {
            System.out.print("> ");
            line = scan.nextLine().trim().toLowerCase();
            if (line.equals("exit")) {
                flag = false;
                break;
            } else if (line.equals("+")) {
                new PlusOperator().execute();
            } else if (line.equals("-")) {
                new MinusOperator().execute();
            } else if (line.equals("/")) {
                new DivOperator().execute();
            } else if (line.equals("*")) {
                new TimesOperator().execute();
            } else if (line.equals("sqrt")) {
                new SqrtOperator().execute();
            } else if (line.equals("1/x")) {
                new OneOverXOperator().execute();
            } else if (line.equals("x^2")) {
                new SquareOperator().execute();
            } else if (line.equals("mr")) {
                new MROperator().execute();
            } else if (line.equals("ms")) {
                new MSOperator().execute();
            } else if (line.equals("c")) {
                new COperator().execute();
            } else if (line.equals("ce")) {
                new CEOperator().execute();
            } else if (line.equals("enter")) {
                new EnterOperator().execute();
            } else {
                if (line.length() > 0) {
                    boolean changeSign = false;
                    if (line.charAt(0) == '-') {
                        changeSign = true;
                        line = line.substring(1, line.length());
                    }
                    if (line.charAt(0) == '+') {
                        line = line.substring(1, line.length());
                    }
                    for (int i = 0; i < line.length(); i++) {
                        char a = line.charAt(i);
                        if (a == '.') {
                            new DotOperator().execute();
                        } else if (a != ' ') {
                            try {
                                new DigitOperator(Integer.parseInt(a + ""))
                                        .execute();
                                ;
                            } catch (NumberFormatException e) {
                                System.err
                                        .println("Not a valid number. Try again");
                                break;
                            }
                        }
                    }
                    if (changeSign) {
                        new SignOperator().execute();
                    }
                }
                // in order to make it immutable, so that the next
                // call doesnt modify the value but push it on the stack.
                new MSOperator().execute();
                new MROperator().execute();
            }

            System.out.print(state.getValueString() + " ");
            System.out.println(Arrays.toString(state.getStackState()));
        }
        scan.close();
    }
}

```

```java
package main;

import util.Pile;

/**
 * This class is the Model of the state of the Calculator. It stores the stack,
 * the current value, the memory value, and react to controllers (called by
 * different operators). This is singleton class as there is a single calculator
 * for a program.
 *
 * @author Sacha Bron
 * @author Valentin Minder
 */
public class State {

    /**
     * Private reference to the unique instance of State.
     */
    private static State myInstance;

    /**
     * Private constructor.
     */
    private State() {
        clear();
    }

    /**
     * Public getInstance of the State. If not exists, creates a new one.
     * Otherwise, returns the same state.
     *
     * @return the unique instance of State.
     */
    public static State getInstance() {
        if (myInstance == null) {
            myInstance = new State();
        }
        return myInstance;
    }

    // INTERNAL STATE
    // value currently printed
    private String currentStrValue;
    // value stored in memory by MS
    private String memory;
    // if the current value has an error
    private boolean error;
    // error displayed to the user
    private String errorMessage;
    // if the value is mutable (while typing) or not (after a result)
    private boolean isMutable = true;
    // reference to the stack of computed values
    private Pile pile;

    /**
     * Clear all the machine, including the stack and memory.
     */
    private void clear() {
        clearError();
        pile = new Pile();
        memory = "";
    }

    /**
     * Clear only the error on the current value.
     */
    private void clearError() {
        currentStrValue = "";
        error = false;
        errorMessage = "";
        isMutable = true;
    }

    // NUMERICAL OPERATORS.
    /**
     * To be called before a numerical operator. If it's not mutable, push the
     * value to stack in order to enter a new value and keep the old one in the
     * stack.
     */
    public boolean checkNumericalOperator() {
        if (!isMutable) {
            push();
        }
        return !error;
    }

    /**
     * Add a digit at the end of the current value.
     */
    public void addDigit(int digit) {
        currentStrValue += digit;
    }

    /**
     * Inverse the sign of the current value. WARNING: IN OUR COMPREHENSION,
     * THIS IS A UNARY OPERATOR WHICH IS VALID ON ANY VALID VALUE (MUTABLE OR
     * NOT), CONTRARY TO DIGIT OR DOT, WHICH ARE ONLY ALLOWED ON MUTABLE VALUES.
     */
    public void inverseSign() {
        double val = value();
        if (!error) {
            if (val < 0) {
                currentStrValue = currentStrValue.substring(1,
                        currentStrValue.length());
            } else if (val > 0) {
                currentStrValue = "-" + currentStrValue;
            }
        }
    }

    /**
     * Add a dot at the end of the current value (plus a leading 0 if the value
     * is currently empty)
     */
    public void addDot() {
        // leading 0 is needed in order to have 0.45 and not .45
        if (currentStrValue.length() == 0) {
            currentStrValue += "0";
        }
        // only added if no dot is found so far.
        if (!currentStrValue.contains(".")) {
            currentStrValue += ".";
        }
    }

    // OPERATORS
    /**
     * Checks that it's allowed to compute a two operands operator (the current
     * value must be valid and the stack must have a least one element)
     */
    public boolean beforeTwoOperands() {
        return beforeOneOperand() && hasNext();
    }

    /**
     * Checks that it's allowed to compute a single operand operator (the
     * current value must be valid)
     */
    public boolean beforeOneOperand() {
        value();
        return !error;
    }

    public void operandDiv() {
        if (value() == 0) {
            error = true;
            errorMessage = "div by 0 not allowed!";
        } else {
            setValue(pop() / value());
        }
    }

    public void operandTimes() {
        setValue(pop() * value());
    }

    public void operandPlus() {
        setValue(pop() + value());
    }

    public void operandMinus() {
        setValue(pop() - value());
    }

    public void operandOver() {
        if (value() == 0) {
            error = true;
            errorMessage = "Div. by 0 not allowed!";
        } else {
            setValue(1 / value());
        }
    }

    public void operandSquare() {
        setValue(Math.pow(value(), 2));
    }

    public void operandSqrt() {
        if (value() < 0) {
            error = true;
            errorMessage = "sqrt not allowed for values < 0!";
        } else {
            setValue(Math.sqrt(value()));
        }
    }

    // CONTROLS
    /**
     * Push the value to stack
     */
    public void controlEnter() {
        push();
    }

    /**
     * Only if the current value is mutable (not a computed result), removes the
     * last digit inserted (including dot)
     */
    public void controlBackSpace() {
        if (isMutable) {
            if (currentStrValue.length() > 0) {
                // !! 0.0 verifier TODO
                currentStrValue = currentStrValue.substring(0,
                        currentStrValue.length() - 1);
            }
        }
    }

    /**
     * Stores the current value (only if valid) in the memory, and leave it in
     * the current value.
     */
    public void controlMemoryStore() {
        value();
        if (!error) {
            memory = currentStrValue;
        }
    }

    /**
     * Delete the current value (clearError) and replace it by the memory, which
     * is non mutable;
     */
    public void controlMemoryRecall() {
        clearError();
        currentStrValue = memory;
        isMutable = false;
    }

    /**
     * Clear all the machine.
     */
    public void controlClear() {
        clear();
    }

    /**
     * Clear the error of the current value.
     */
    public void controlClearError() {
        clearError();
    }

    // INSIDE STATE MANAGEMENT
    /**
     * Push the value (only if valid) on the stack.
     */
    private void push() {
        double val = value();
        if (!error) {
            pile.empile(val);
            clearError();
        }
    }

    /**
     * Returns the last inserted value in the stack.
     */
    private double pop() {
        if (hasNext()) {
            return (double) pile.depile();
        }
        return 0;
    }

    /**
     * Checks if the stack has a next value, and stores an error if yes.
     */
    private boolean hasNext() {
        if (pile.getSize() == 0) {
```

```java
            error = true;
            errorMessage = "Empty stack! Operation not allowed!";
            return false;
        }
        return true;
    }

    /**
     * Computes the numerical double value of the current value.
     */
    private double value() {
        try {
            if (currentStrValue.length() == 0) {
                return 0;
            }
            return Double.valueOf(currentStrValue);
        } catch (NumberFormatException e) {
            error = true;
            errorMessage = "Format error:" + currentStrValue;
            return 0;
        }
    }

    /**
     * Set the string current value
     *
     * @param d
     *            the new numerical value.
     */
    private void setValue(double d) {
        currentStrValue = Double.toString(d);
        if (currentStrValue.equalsIgnoreCase("Infinity")
                || currentStrValue.equalsIgnoreCase("-Infinity")) {
            error = true;
            errorMessage = "Limit reached: +/- infinity result.";
        } else
            if (currentStrValue.equalsIgnoreCase("NaN")) {
            error = true;
            errorMessage = "Error NaN: last action produced Not A Number value";
        }
        isMutable = false;
    }

    // INTERACTION WITH OUTSIDE
    /**
     * Getter for the current value as string representation. Returns the error
     * message if appropriate.
     */
    public String getValueString() {
        if (error) {
            return errorMessage;
        }
        if (currentStrValue.length() == 0) {
            return "0";
        }
        return currentStrValue;
    }

    /**
     * Get the stack state as an Object array.
     */
    public Object[] getStackState() {
        return pile.toArray();
    }
}
```

```java
package operator;

/**
 * This is the root class of the hierachy of all operators. All the hierarchy
 * contains checkShouldExecute and exec methods: checkShouldExecute checks if an
 * operator (or a group of operator) should be executed, exec execute the
 * operator (in the deepest defition in the hierarchy). The only non-abstract
 * method (execute) calls the checkShouldExecute() on the operator and then if
 * necessary the exec method().
 * <p>
 * --- Hierarchy --- (lexicographic) <br>
 * Operator <br>
 * --CalculOperator<br>
 * ----OneOperandCalculOperator<br>
 * ------SignOperator<br>
 * ------SqrtOperator<br>
 * ------SquareOperator<br>
 * ------OneOverXOperator<br>
 * ----TwoOperandCalculOperator<br>
 * ------DivOperator<br>
 * ------MinusOperator<br>
 * ------PlusOperator<br>
 * ------TimesOperator<br>
 * --ControlOperator <br>
 * ----BackSpaceOperator<br>
 * ----CEOperator<br>
 * ----COperator<br>
 * ----EnterOperator<br>
 * ----MROperator<br>
 * ----MSOperator<br>
 * --NumberOperator<br>
 * ----DigitOperator<br>
 * ----DotOperator<br>
 * <p>
 * Note: only the leaf are non-abstract, all the other are abstract.
 *
 * @author Sacha Bron
 * @author Valentin Minder
 */

 // multi-page printing of all operators TO SAVE TREES !!!

import main.State;

public abstract class Operator {
    public void execute() {
        if (checkShouldExecute()) {
            exec();
        }
    }

    abstract void exec();

    abstract boolean checkShouldExecute();
}

public abstract class CalculOperator extends Operator {
    abstract void exec();
    abstract boolean checkShouldExecute();
}

public abstract class OneOperandCalculOperator extends CalculOperator {
    abstract void exec();
    boolean checkShouldExecute() {
        // checks that a one-operand-operator could be executed
        return State.getInstance().beforeOneOperand();
    }
}

public class SignOperator extends OneOperandCalculOperator {
    void exec() {
        State.getInstance().inverseSign();
    }
}

public class SqrtOperator extends OneOperandCalculOperator {
    void exec() {
        State.getInstance().operandSqrt();
    }
}

public class SquareOperator extends OneOperandCalculOperator {
    void exec() {
        State.getInstance().operandSquare();
    }
}

public class OneOverXOperator extends OneOperandCalculOperator {
    void exec() {
        State.getInstance().operandOver();
    }
}

public abstract class TwoOperandCalculOperator extends CalculOperator {
    abstract void exec();

    boolean checkShouldExecute() {
        // checks that a two-operands-operator could be executed
        return State.getInstance().beforeTwoOperands();
    }
}

public class DivOperator extends TwoOperandCalculOperator {
    void exec() {
        State.getInstance().operandDiv();
    }
}

```

```java
public class MinusOperator extends TwoOperandCalculOperator {
    void exec() {
        State.getInstance().operandMinus();
    }
}

public class PlusOperator extends TwoOperandCalculOperator {
    void exec() {
        State.getInstance().operandPlus();
    }
}

public class TimesOperator extends TwoOperandCalculOperator {
    void exec() {
        State.getInstance().operandTimes();
    }
}

public abstract class ControlOperator extends Operator {
    abstract void exec();

    boolean checkShouldExecute() {
        // they are always allowed!
        return true;
    }
}

public class BackSpaceOperator extends ControlOperator {
    void exec() {
        State.getInstance().controlBackSpace();
    }
}

public class CEOperator extends ControlOperator {
    void exec() {
        State.getInstance().controlClearError();
    }
}

public class COperator extends ControlOperator {
    void exec() {
        State.getInstance().controlClear();
    }
}

public class EnterOperator extends ControlOperator {
    void exec() {
        State.getInstance().controlEnter();
    }
}

public class MROperator extends ControlOperator {
    void exec() {
        State.getInstance().controlMemoryRecall();
    }
}

public class MSOperator extends ControlOperator {
    void exec() {
        State.getInstance().controlMemoryStore();
    }
}

public abstract class NumberOperator extends Operator {
    abstract void exec();

    boolean checkShouldExecute() {
        // checks that a numerical modifier could be executed
        return State.getInstance().checkNumericalOperator();
    }
}

public class DotOperator extends NumberOperator {

    void exec() {
        State.getInstance().addDot();
    }
}

public class DigitOperator extends NumberOperator {

    private int myValue = 0;

    public DigitOperator(int value) {
        myValue = value;
    }

    void exec() {
        State.getInstance().addDigit(myValue);
    }
}

```