

```
1 package main;
2
3 /*
4  * JCalculator.java
5  *
6  * Pier Donini, 9 Jan 2004.
7  * edited by Minder Valentin and Bron Sacha on Dec 11 2014.
8  */
9
10 import javax.swing.*;
11
12 import operator.*;
13
14 import java.awt.*;
15 import java.awt.event.*;
16
17 public class JCalculator extends JFrame {
18     // Tableau representant une pile vide
19     private final String[] empty = { "< empty stack >" };
20
21     // Zone de texte contenant la valeur introduite ou resultat courant
22     private final JTextField jNumber = new JTextField("0");
23
24     // Composant liste representant le contenu de la pile
25     private final JList jStack = new JList(empty);
26
27     // Contraintes pour le placement des composants graphiques
28     private final GridBagConstraints constraints = new GridBagConstraints();
29
30     /*
31     * Mise a jour de l'interface apres une operation (jList et jStack)
32     */
33     private void update() {
34         // Modifier une zone de texte, JTextField.setText(string nom)
35         // Modifier un composant liste, JList.setListData(Object[] tableau)
36         jNumber.setText(State.getInstance().getValueString());
37         Object [] stack = State.getInstance().getStackState();
38         if (stack.length == 0) {
39             stack = empty;
40         }
41         jStack.setListData(stack);
42     }
43
44     /*
45     * Ajout d'un bouton dans l'interface et de l'operation associee, instance
46     * de la classe Operation, possedeant une methode execute()
47     */
48     private void addOperatorButton(String name, int x, int y, Color color,
49         final Operator operator) {
50         JButton b = new JButton(name);
51         b.setForeground(color);
52         constraints.gridx = x;
53         constraints.gridy = y;
54         getContentPane().add(b, constraints);
55     }
56 }
```

```
55
56     b.addActionListener(new ActionListener() {
57         public void actionPerformed(ActionEvent e) {
58             operator.execute();
59             update();
60         }
61     });
62 }
63
64 /*
65  * Constructeur
66  */
67 public JCalculator() {
68     super("JCalculator");
69     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
70     getContentPane().setLayout(new GridBagLayout());
71
72     // Contraintes des composants graphiques
73     constraints.insets = new Insets(3, 3, 3, 3);
74     constraints.fill = GridBagConstraints.HORIZONTAL;
75
76     // Nombre courant
77     jNumber.setEditable(false);
78     jNumber.setBackground(Color.WHITE);
79     jNumber.setHorizontalAlignment(JTextField.RIGHT);
80     constraints.gridx = 0;
81     constraints.gridy = 0;
82     constraints.gridwidth = 5;
83     getContentPane().add(jNumber, constraints);
84     constraints.gridwidth = 1; // reset width
85
86     // Rappel de la valeur en memoire
87     addOperatorButton("MR", 0, 1, Color.RED, new MROperator());
88
89     // Stockage d'une valeur en memoire
90     addOperatorButton("MS", 1, 1, Color.RED, new MSOperator());
91
92     // Backspace
93     addOperatorButton("<=", 2, 1, Color.RED, new BackSpaceOperator());
94
95     // Mise a zero de la valeur courante + suppression des erreurs
96     addOperatorButton("CE", 3, 1, Color.RED, new CEOperator());
97
98     // Comme CE + vide la pile
99     addOperatorButton("C", 4, 1, Color.RED, new COperator());
100
101     // Boutons 1-9
102     for (int i = 1; i < 10; i++)
103         addOperatorButton(String.valueOf(i), (i - 1) % 3, 4 - (i - 1) / 3,
104             Color.BLUE, new DigitOperator(i));
105     // Bouton 0
106     addOperatorButton("0", 0, 5, Color.BLUE, new DigitOperator(0));
107
108     // Changement de signe de la valeur courante
```

```
109         addOperatorButton("/+-", 1, 5, Color.BLUE, new SignOperator());
110
111         // Operateur point (chiffres apres la virgule ensuite)
112         addOperatorButton(".", 2, 5, Color.BLUE, new DotOperator());
113
114         // Operateurs arithmetiques a deux operandes: /, *, -, +
115         addOperatorButton("/", 3, 2, Color.RED, new DivOperator());
116         addOperatorButton("*", 3, 3, Color.RED, new TimesOperator());
117         addOperatorButton("-", 3, 4, Color.RED, new MinusOperator());
118         addOperatorButton("+", 3, 5, Color.RED, new PlusOperator());
119
120         // Operateurs arithmetiques a un operande: 1/x, x^2, Sqrt
121         addOperatorButton("1/x", 4, 2, Color.RED, new OneOverXOperator());
122         addOperatorButton("x^2", 4, 3, Color.RED, new SquareOperator());
123         addOperatorButton("Sqrt", 4, 4, Color.RED, new SqrtOperator());
124
125         // Entree: met la valeur courante sur le sommet de la pile
126         addOperatorButton("Ent", 4, 5, Color.RED, new EnterOperator());
127
128         // Affichage de la pile
129         JLabel jLabel = new JLabel("Stack");
130         jLabel.setFont(new Font("Dialog", 0, 12));
131         jLabel.setHorizontalAlignment(JLabel.CENTER);
132         constraints.gridx = 5;
133         constraints.gridy = 0;
134         getContentPane().add(jLabel, constraints);
135
136         jStack.setFont(new Font("Dialog", 0, 12));
137         jStack.setVisibleRowCount(8);
138         JScrollPane scrollPane = new JScrollPane(jStack);
139         constraints.gridx = 5;
140         constraints.gridy = 1;
141         constraints.gridheight = 5;
142         getContentPane().add(scrollPane, constraints);
143         constraints.gridheight = 1; // reset height
144
145         setResizable(false);
146         pack();
147     }
148
149     /*
150     * main()
151     */
152     public static void main(String args[]) {
153         new JCalculator().setVisible(true);
154     }
155 }
156
```

```
1 package main;
2
3 import java.util.Arrays;
4 import java.util.Scanner;
5
6 import operator.*;
7
8 /**
9  * This class represent the usage of the Calculator from a terminal/console. It
10  * allows the user to enter values and operators with the keyboard, and the
11  * state of the current value and stack is printed after each operations.
12  *
13  * @author Sacha Bron
14  * @author Valentin Minder
15  */
16 public class Calculator {
17
18     public static void main(String[] args) {
19
20         State state = State.getInstance();
21         Scanner scan = new Scanner(System.in);
22         System.out.println("Welcome to the REVERSE POLISH TERMINAL CALCULATOR");
23
24         String line = "";
25         boolean flag = true;
26
27         while (flag) {
28             System.out.print("> ");
29             line = scan.nextLine().trim().toLowerCase();
30             if (line.equals("exit")) {
31                 flag = false;
32                 break;
33             } else if (line.equals("+")) {
34                 new PlusOperator().execute();
35             } else if (line.equals("-")) {
36                 new MinusOperator().execute();
37             } else if (line.equals("/")) {
38                 new DivOperator().execute();
39             } else if (line.equals("*")) {
40                 new TimesOperator().execute();
41             } else if (line.equals("sqrt")) {
42                 new SqrtOperator().execute();
43             } else if (line.equals("1/x")) {
44                 new OneOverXOperator().execute();
45             } else if (line.equals("x^2")) {
46                 new SquareOperator().execute();
47             } else if (line.equals("mr")) {
48                 new MROperator().execute();
49             } else if (line.equals("ms")) {
50                 new MSOperator().execute();
51             } else if (line.equals("c")) {
52                 new COperator().execute();
53             } else if (line.equals("ce")) {
54                 new CEOperator().execute();
55             }
56         }
57     }
58 }
```

```
55         } else if (line.equals("enter")) {
56             new EnterOperator().execute();
57         } else {
58             if (line.length() > 0) {
59                 boolean changeSign = false;
60                 if (line.charAt(0) == '-') {
61                     changeSign = true;
62                     line = line.substring(1, line.length());
63                 }
64                 if (line.charAt(0) == '+') {
65                     line = line.substring(1, line.length());
66                 }
67                 for (int i = 0; i < line.length(); i++) {
68                     char a = line.charAt(i);
69                     if (a == '.') {
70                         new DotOperator().execute();
71                     } else if (a != ' ') {
72                         try {
73                             new DigitOperator(Integer.parseInt(a + ""))
74                                 .execute();
75                         }
76                         ;
77                     } catch (NumberFormatException e) {
78                         System.err
79                             .println("Not a valid number. Try again");
80                         break;
81                     }
82                 }
83                 if (changeSign) {
84                     new SignOperator().execute();
85                 }
86             }
87             // in order to make it immutable, so that the next
88             // call doesnt modify the value but push it on the stack.
89             new MSOperator().execute();
90             new MROperator().execute();
91         }
92
93         System.out.print(state.getValueString() + " ");
94         System.out.println(Arrays.toString(state.getStackState()));
95     }
96     scan.close();
97 }
98 }
99
```

```
1 package main;
2
3 import util.Pile;
4
5 /**
6  * This class is the Model of the state of the Calculator. It stores the stack,
7  * the current value, the memory value, and react to controllers (called by
8  * different operators). This is singleton class as there is a single calculator
9  * for a program.
10  *
11  * @author Sacha Bron
12  * @author Valentin Minder
13  */
14 public class State {
15
16     /**
17      * Private reference to the unique instance of State.
18      */
19     private static State myInstance;
20
21     /**
22      * Private constructor.
23      */
24     private State() {
25         clear();
26     }
27
28     /**
29      * Public getInstance of the State. If not exists, creates a new one.
30      * Otherwise, returns the same state.
31      *
32      * @return the unique instance of State.
33      */
34     public static State getInstance() {
35         if (myInstance == null) {
36             myInstance = new State();
37         }
38         return myInstance;
39     }
40
41     // INTERNAL STATE
42     // value currently printed
43     private String currentStrValue;
44     // value stored in memory by MS
45     private String memory;
46     // if the current value has an error
47     private boolean error;
48     // error displayed to the user
49     private String errorMessage;
50     // if the value is mutable (while typing) or not (after a result)
51     private boolean isMutable = true;
52     // reference to the stack of computed values
53     private Pile pile;
54 }
```

```
55     /**
56      * Clear all the machine, including the stack and memory.
57      */
58     private void clear() {
59         clearError();
60         pile = new Pile();
61         memory = "";
62     }
63
64     /**
65      * Clear only the error on the current value.
66      */
67     private void clearError() {
68         currentStrValue = "";
69         error = false;
70         errorMessage = "";
71         isMutable = true;
72     }
73
74     // NUMERICAL OPERATORS.
75     /**
76      * To be called before a numerical operator. If it's not mutable, push the
77      * value to stack in order to enter a new value and keep the old one in the
78      * stack.
79      */
80     public boolean checkNumericalOperator() {
81         if (!isMutable) {
82             push();
83         }
84         return !error;
85     }
86
87     /**
88      * Add a digit at the end of the current value.
89      */
90     public void addDigit(int digit) {
91         currentStrValue += digit;
92     }
93
94     /**
95      * Inverse the sign of the current value. WARNING: IN OUR COMPREHENSION,
96      * THIS IS A UNARY OPERATOR WHICH IS VALID ON ANY VALID VALUE (MUTABLE OR
97      * NOT), CONTRARY TO DIGIT OR DOT, WHICH ARE ONLY ALLOWED ON MUTABLE VALUES.
98      */
99     public void inverseSign() {
100         double val = value();
101         if (!error) {
102             if (val < 0) {
103                 currentStrValue = currentStrValue.substring(1,
104                     currentStrValue.length());
105             } else if (val > 0) {
106                 currentStrValue = "-" + currentStrValue;
107             }
108         }
109     }
```

```
109     }
110
111     /**
112     * Add a dot at the end of the current value (plus a leading 0 if the value
113     * is currently empty)
114     */
115     public void addDot() {
116         // leading 0 is needed in order to have 0.45 and not .45
117         if (currentStrValue.length() == 0) {
118             currentStrValue += "0";
119         }
120         // only added if no dot is found so far.
121         if (!currentStrValue.contains(".")) {
122             currentStrValue += ".";
123         }
124     }
125
126     // OPERATORS
127     /**
128     * Checks that it's allowed to compute a two operands operator (the current
129     * value must be valid and the stack must have a least one element)
130     */
131     public boolean beforeTwoOperands() {
132         return beforeOneOperand() && hasNext();
133     }
134
135     /**
136     * Checks that it's allowed to compute a single operand operator (the
137     * current value must be valid)
138     */
139     public boolean beforeOneOperand() {
140         value();
141         return !error;
142     }
143
144     public void operandDiv() {
145         if (value() == 0) {
146             error = true;
147             errorMessage = "div by 0 not allowed!";
148         } else {
149             setValue(pop() / value());
150         }
151     }
152
153     public void operandTimes() {
154         setValue(pop() * value());
155     }
156
157     public void operandPlus() {
158         setValue(pop() + value());
159     }
160
161     public void operandMinus() {
162         setValue(pop() - value());
```



```
163     }
164
165     public void operandOver() {
166         if (value() == 0) {
167             error = true;
168             errorMessage = "Div. by 0 not allowed!";
169         } else {
170             setValue(1 / value());
171         }
172     }
173
174     public void operandSquare() {
175         setValue(Math.pow(value(), 2));
176     }
177
178     public void operandSqrt() {
179         if (value() < 0) {
180             error = true;
181             errorMessage = "sqrt not allowed for values < 0!";
182         } else {
183             setValue(Math.sqrt(value()));
184         }
185     }
186
187     // CONTROLS
188     /**
189      * Push the value to stack
190      */
191     public void controlEnter() {
192         push();
193     }
194
195     /**
196      * Only if the current value is mutable (not a computed result), removes the
197      * last digit inserted (including dot)
198      */
199     public void controlBackSpace() {
200         if (isMutable) {
201             if (currentStrValue.length() > 0) {
202                 // !! 0.0 verifier TODO
203                 currentStrValue = currentStrValue.substring(0,
204                     currentStrValue.length() - 1);
205             }
206         }
207     }
208
209     /**
210      * Stores the current value (only if valid) in the memory, and leave it in
211      * the current value.
212      */
213     public void controlMemoryStore() {
214         value();
215         if (!error) {
216             memory = currentStrValue;
```

```
217     }
218 }
219
220 /**
221  * Delete the current value (clearError) and replace it by the memory, which
222  * is non mutable;
223  */
224 public void controlMemoryRecall() {
225     clearError();
226     currentStrValue = memory;
227     isMutable = false;
228 }
229
230 /**
231  * Clear all the machine.
232  */
233 public void controlClear() {
234     clear();
235 }
236
237 /**
238  * Clear the error of the current value.
239  */
240 public void controlClearError() {
241     clearError();
242 }
243
244 // INSIDE STATE MANAGEMENT
245 /**
246  * Push the value (only if valid) on the stack.
247  */
248 private void push() {
249     double val = value();
250     if (!error) {
251         pile.empile(val);
252         clearError();
253     }
254 }
255
256 /**
257  * Returns the last inserted value in the stack.
258  */
259 private double pop() {
260     if (hasNext()) {
261         return (double) pile.depile();
262     }
263     return 0;
264 }
265
266 /**
267  * Checks if the stack has a next value, and stores an error if yes.
268  */
269 private boolean hasNext() {
270     if (pile.getSize() == 0) {
```

```
271         error = true;
272         errorMessage = "Empty stack! Operation not allowed!";
273         return false;
274     }
275     return true;
276 }
277
278 /**
279  * Computes the numerical double value of the current value.
280  */
281 private double value() {
282     try {
283         if (currentStrValue.length() == 0) {
284             return 0;
285         }
286         return Double.valueOf(currentStrValue);
287     } catch (NumberFormatException e) {
288         error = true;
289         errorMessage = "Format error:" + currentStrValue;
290         return 0;
291     }
292 }
293
294 /**
295  * Set the string current value
296  *
297  * @param d
298  *         the new numerical value.
299  */
300 private void setValue(double d) {
301     currentStrValue = Double.toString(d);
302     if (currentStrValue.equalsIgnoreCase("Infinity")
303         || currentStrValue.equalsIgnoreCase("-Infinity")) {
304         error = true;
305         errorMessage = "Limit reached: +/- infinity result.";
306     } else
307         if (currentStrValue.equalsIgnoreCase("NaN")) {
308             error = true;
309             errorMessage = "Error NaN: last action produced Not A Number value";
310         }
311     isMutable = false;
312 }
313
314 // INTERACTION WITH OUTSIDE
315 /**
316  * Getter for the current value as string representation. Returns the error
317  * message if appropriate.
318  */
319 public String getValueString() {
320     if (error) {
321         return errorMessage;
322     }
323     if (currentStrValue.length() == 0) {
324         return "0";
325     }
326 }
```

```
325     }
326     return currentStrValue;
327 }
328
329 /**
330  * Get the stack state as an Object array.
331  */
332 public Object[] getStackState() {
333     return pile.toArray();
334 }
335 }
```

```
1 package operator;
2
3 /**
4  * This is the root class of the hierachy of all operators. All the hierarchy
5  * contains checkShouldExecute and exec methods: checkShouldExecute checks if an
6  * operator (or a group of operator) should be executed, exec execute the
7  * operator (in the deepest defition in the hierarchy). The only non-abstract
8  * method (execute) calls the checkShouldExecute() on the operator and then if
9  * necessary the exec method().
10 * <p>
11 * --- Hierarchy --- (lexicographic) <br>
12 * Operator <br>
13 * --CalcuOperator<br>
14 * ----OneOperandCalcuOperator<br>
15 * -----SignOperator<br>
16 * -----SqrtOperator<br>
17 * -----SquareOperator<br>
18 * -----OneOverXOperator<br>
19 * ----TwoOperandCalcuOperator<br>
20 * -----DivOperator<br>
21 * -----MinusOperator<br>
22 * -----PlusOperator<br>
23 * -----TimesOperator<br>
24 * --ControlOperator <br>
25 * ----BackSpaceOperator<br>
26 * ----CEOperator<br>
27 * ----COperator<br>
28 * ----EnterOperator<br>
29 * ----MROperator<br>
30 * ----MSOperator<br>
31 * --NumberOperator<br>
32 * ----DigitOperator<br>
33 * ----DotOperator<br>
34 * <p>
35 * Note: only the leaf are non-abstract, all the other are abstract.
36 *
37 * @author Sacha Bron
38 * @author Valentin Minder
39 */
40
41 // multi-page printing of all operators TO SAVE TREES !!!
42
43 import main.State;
44
45 public abstract class Operator {
46     public void execute() {
47         if (checkShouldExecute()) {
48             exec();
49         }
50     }
51
52     abstract void exec();
53
54     abstract boolean checkShouldExecute();
55 }
```

```
55 }
56
57 public abstract class Calc0perator extends Operator {
58     abstract void exec();
59     abstract boolean checkShouldExecute();
60 }
61
62 public abstract class One0operandCalc0perator extends Calc0perator {
63     abstract void exec();
64     boolean checkShouldExecute() {
65         // checks that a one-operand-operator could be executed
66         return State.getInstance().beforeOne0operand();
67     }
68 }
69
70 public class Sign0perator extends One0operandCalc0perator {
71     void exec() {
72         State.getInstance().inverseSign();
73     }
74 }
75
76 public class Sqrt0perator extends One0operandCalc0perator {
77     void exec() {
78         State.getInstance().operandSqrt();
79     }
80 }
81
82 public class Square0perator extends One0operandCalc0perator {
83     void exec() {
84         State.getInstance().operandSquare();
85     }
86 }
87
88 public class One0overX0perator extends One0operandCalc0perator {
89     void exec() {
90         State.getInstance().operandOver();
91     }
92 }
93
94 public abstract class Two0operandCalc0perator extends Calc0perator {
95     abstract void exec();
96
97     boolean checkShouldExecute() {
98         // checks that a two-operands-operator could be executed
99         return State.getInstance().beforeTwo0operands();
100     }
101 }
102
103 public class Div0perator extends Two0operandCalc0perator {
104     void exec() {
105         State.getInstance().operandDiv();
106     }
107 }
108
```

```
109 public class MinusOperator extends TwoOperandCalcu0perator {
110     void exec() {
111         State.getInstance().operandMinus();
112     }
113 }
114
115 public class PlusOperator extends TwoOperandCalcu0perator {
116     void exec() {
117         State.getInstance().operandPlus();
118     }
119 }
120
121 public class TimesOperator extends TwoOperandCalcu0perator {
122     void exec() {
123         State.getInstance().operandTimes();
124     }
125 }
126
127 public abstract class Control0perator extends Operator {
128     abstract void exec();
129
130     boolean checkShouldExecute() {
131         // they are always allowed!
132         return true;
133     }
134 }
135
136 public class BackSpaceOperator extends Control0perator {
137     void exec() {
138         State.getInstance().controlBackSpace();
139     }
140 }
141
142 public class CE0perator extends Control0perator {
143     void exec() {
144         State.getInstance().controlClearError();
145     }
146 }
147
148 public class C0perator extends Control0perator {
149     void exec() {
150         State.getInstance().controlClear();
151     }
152 }
153
154 public class Enter0perator extends Control0perator {
155     void exec() {
156         State.getInstance().controlEnter();
157     }
158 }
159
160 public class MR0perator extends Control0perator {
161     void exec() {
162         State.getInstance().controlMemoryRecall();
```

```
163     }
164 }
165
166 public class MSOperator extends ControlOperator {
167     void exec() {
168         State.getInstance().controlMemoryStore();
169     }
170 }
171
172 public abstract class NumberOperator extends Operator {
173     abstract void exec();
174
175     boolean checkShouldExecute() {
176         // checks that a numerical modifier could be executed
177         return State.getInstance().checkNumericalOperator();
178     }
179 }
180
181 public class DotOperator extends NumberOperator {
182
183     void exec() {
184         State.getInstance().addDot();
185     }
186 }
187
188 public class DigitOperator extends NumberOperator {
189
190     private int myValue = 0;
191
192     public DigitOperator(int value) {
193         myValue = value;
194     }
195
196     void exec() {
197         State.getInstance().addDigit(myValue);
198     }
199 }
200
```