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

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

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

```
1 package main;
2
3
  import java.util.Arrays;
4
  import java.util.Scanner;
5
6
  import operator.*;
7
   /**
8
   * This class represent the usage of the Calculator from a terminal/console. It
9
   * allows the user to enter values and operators with the keyboard, and the
10
   * state of the current value and stack is printed after each operations.
11
12
   * @author Sacha Bron
13
   * @author Valentin Minder
14
15
  public class Calculator {
16
17
       public static void main(String[] args) {
18
19
           State state = State.getInstance();
20
           Scanner scan = new Scanner(System.in);
21
           System.out.println("Welcome to the REVERSE POLISH TERMINAL CALCULATOR");
22
23
           String line = "":
24
           boolean flag = true;
25
26
27
           while (flag) {
               System.out.print("> ");
28
               line = scan.nextLine().trim().toLowerCase();
29
30
               if (line.equals("exit")) {
                   flag = false;
31
                    break;
32
               } else if (line.equals("+")) {
33
                    new PlusOperator().execute();
34
               } else if (line.equals("-")) {
35
                   new MinusOperator().execute();
36
               } else if (line.equals("/")) {
37
                    new DivOperator().execute();
38
               } else if (line.equals("*")) {
39
                    new TimesOperator().execute();
40
               } else if (line.equals("sqrt")) {
41
                    new Sgrt0perator().execute();
42
               } else if (line.equals("1/x")) {
43
                    new OneOverXOperator().execute();
44
               } else if (line.equals("x^2")) {
45
                    new SquareOperator().execute();
46
               } else if (line.equals("mr")) {
47
48
                    new MROperator().execute();
               } else if (line.equals("ms")) {
49
                   new MSOperator().execute();
50
               } else if (line.equals("c")) {
51
                   new COperator().execute();
52
               } else if (line.equals("ce")) {
53
                   new CEOperator().execute();
54
```

```
} else if (line.equals("enter")) {
55
                    new EnterOperator().execute();
56
                } else {
57
                    if (line.length() > 0) {
58
                        boolean changeSign = false;
59
                        if (line.charAt(0) == '-') {
60
                             changeSign = true;
61
                             line = line.substring(1, line.length());
62
                        }
63
                        if (line.charAt(0) == '+') {
64
                             line = line.substring(1, line.length());
65
                        }
66
                        for (int i = 0; i < line.length(); i++) {</pre>
67
                             char a = line.charAt(i);
68
                             if (a == '.') {
69
                                 new DotOperator().execute();
70
                             } else if (a != ' ') {
71
                                 try {
72
73
                                      new DigitOperator(Integer.parseInt(a + ""))
                                              .execute();
74
75
                                 } catch (NumberFormatException e) {
76
77
                                     System.err
                                              .println("Not a valid number. Try again");
78
79
                                     break;
                                 }
80
                             }
81
                        }
82
                        if (changeSign) {
83
                             new SignOperator().execute();
84
                        }
85
                    }
86
                    // in order to make it immutable, so that the next
87
                    // call doesnt modify the value but push it on the stack.
88
89
                    new MSOperator().execute();
                    new MROperator().execute();
90
                }
91
92
                System.out.print(state.getValueString() + " ");
93
                System.out.println(Arrays.toString(state.getStackState()));
94
           }
95
           scan.close();
96
       }
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,
   * the current value, the memory value, and react to controllers (called by
7
   * different operators). This is singleton class as there is a single calculator
8
9
   * for a program.
10
11
   * @author Sacha Bron
   * @author Valentin Minder
12
   */
13
  public class State {
14
15
       /**
16
        * Private reference to the unique instance of State.
17
18
       private static State myInstance;
19
20
       /**
21
22
       * Private constructor.
23
        */
       private State() {
24
25
           clear();
       }
26
27
       /**
28
        * Public getInstance of the State. If not exists, creates a new one.
29
30
        * Otherwise, returns the same state.
31
        * @return the unique instance of State.
32
33
        */
       public static State getInstance() {
34
35
           if (myInstance == null) {
               myInstance = new State();
36
37
38
           return myInstance;
       }
39
40
       // INTERNAL STATE
41
       // value currently printed
42
       private String currentStrValue;
43
       // value stored in memory by MS
44
       private String memory;
45
       // if the current value has an error
46
47
       private boolean error;
48
       // error displayed to the user
       private String errorMessage;
49
       // if the value is mutable (while typing) or not (after a result)
50
       private boolean isMutable = true;
51
       // reference to the stack of computed values
52
       private Pile pile;
53
54
```

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

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

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

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

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

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

```
1 package operator;
2
3
  /**
   * This is the root class of the hierarchy of all operators. All the hierarchy
4
   * contains checkShouldExecute and exec methods: checkShouldExecute checks if an
5
   * operator (or a group of operator) should be executed, exec execute the
6
   * operator (in the deepest defition in the hierarchy). The only non-abstract
7
   * method (execute) calls the checkShouldExecute() on the operator and then if
8
   * necessary the exec method().
9
   * 
10
   * --- Hierarchy --- (lexicographic) <br>
11
   * Operator <br>
12
   * --CalculOperator<br>
13
   * ----OneOperandCalculOperator<br>
14
   * -----SignOperator<br>
15
   * -----Sgrt0perator<br>
16
   * -----SquareOperator<br>
17
   * -----OneOverXOperator<br>
18
   * ----TwoOperandCalculOperator<br>
19
   * -----DivOperator<br>
20
   * -----MinusOperator<br>
21
   * -----PlusOperator<br
22
23
   * ----TimesOperator<br>
   * --ControlOperator <br>
24
   * ----BackSpaceOperator<br>
25
   * ----CEOperator<br>
26
27
   * ----COperator<br>
   * ----EnterOperator<br>
28
   * ----MROperator<br
29
30
   * ----MSOperator<br>
   * --NumberOperator<br>
31
   * ----DigitOperator<br>
32
   * ----DotOperator<br
33
   * 
34
   * Note: only the leaf are non-abstract, all the other are abstract.
35
36
   * @author Sacha Bron
37
   * @author Valentin Minder
38
39
   */
40
   // multi-page printing of all operators TO SAVE TREES !!!
41
42
   import main.State;
43
44
  public abstract class Operator {
45
       public void execute() {
46
           if (checkShouldExecute()) {
47
48
               exec();
           }
49
       }
50
51
       abstract void exec();
52
53
       abstract boolean checkShouldExecute();
54
```

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

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

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