```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;

```

```
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     }
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             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 }
```