Basic Calculator OOP App in Java - Project Report

Mert Samet Kayacıoğlu

23 August 2025

Contents

Calculator

1 Introduction

1.1 Description

This project is a basic calculator application developed in Java as an exercise in object-oriented programming (OOP). The program provides a graphical user interface (GUI) built with the Abstract Window Toolkit (AWT) and allows the user to perform fundamental arithmetic operations such as addition, subtraction, multiplication, and division.

The calculator is implemented using OOP principles, where responsibilities are organized into methods and event listeners that handle user actions like button clicks and keyboard input. While the application is implemented as a single monolithic class (`Calculator`), separation of concerns is demonstrated at the method level. The class manages the interface layout, user interactions, and operation logic, while helper methods encapsulate input validation, number formatting, and error handling. This keeps responsibilities modular inside one class, though not fully separated into multiple classes.

In addition to the standard operations, the calculator supports features like clearing the display, deleting the last character, handling decimal numbers, preventing invalid inputs (e.g., multiple decimal points in one operand), and displaying warnings for errors such as division by zero. The program is intentionally kept simple to highlight Java OOP concepts such as encapsulation, event-driven programming, and modular design, making it a practical learning exercise in applying theory to implementation.

1.2 Objective

The main objective of this project is to practice and demonstrate the principles of object-oriented programming (OOP) in Java through the development of a simple calculator application. The project aims to strengthen understanding of concepts such as encapsulation, modular design, and event-driven programming by applying them in a practical context. Additionally, it provides experience with building graphical user interfaces using AWT, handling user input through both buttons and the keyboard, and managing program behavior through event listeners. By completing this project, the goal is not only to implement basic arithmetic operations but also to develop good coding practices and problem-solving skills in Java application development.

1.3 Background

This project was originally assigned as a homework exercise in the Object-Oriented Programming (OOP) lecture. The initial task was to implement a very basic calculator with limited functionality. However, in order to deepen understanding of Java and OOP concepts, the project was expanded and improved. Additional features decimal number support, and a graphical interface with both more button and keyboard input were implemented. This not only made the calculator more functional and user-friendly but also turned a simple homework assignment into a more advanced practical application of OOP principles.

1.4 Scope

The scope of the project is limited to basic arithmetic operations. It does not include advanced mathematical functions such as trigonometry, exponentiation, or memory storage, as the primary goal is to focus on OOP fundamentals and GUI interaction in Java.

2 System Design

2.1 Architecture and Design Rationale

The calculator is implemented in Java (AWT) using an event-driven, single-window architecture. A single concrete class, `Calculator`, encapsulates both the presentation layer (GUI) and the application logic (parsing, validation, and evaluation). This monolithic approach is deliberate for a compact assignment-scale application: it minimizes boilerplate, keeps control flow explicit, and simplifies deployment while remaining extensible (see §2.11).

A pure AWT stack (`Frame`, `Panel`, `Button`, `TextField`) is used for maximal portability and zero external dependencies. Layout is realized with `GridBagLayout` to reproduce a physical calculator grid while retaining fine control over resizing behavior and component weights.

2.2 Components and Responsibilities

- `Calculator` (concrete class): constructs the UI; registers listeners; maintains transient UI state (display content and the most recent binary operator); validates input; evaluates expressions; formats results; and shows transient feedback messages.
- **Helper methods** (private):
  - `disp()/setDisp()`: read/write the display text.
  - `firstOpIndex(String)`: locate the first binary operator, ignoring a leading sign.
  - `endsWithSign(String)`: detect a trailing unary `'-'` used as a sign (e.g., `5*-`).
  - `hasDotInCurrentOperand(String)`: prevent multiple decimal points within the active operand.
  - `isOperator(char)`: membership test for {`+, -, *, /`}.
  - `formatNumber(double)` / `showResult(double)`: canonicalize numeric output (trimmed fixed-point or scientific notation).
  - `flash(String)`: show a non-blocking, time-limited status message in the display.

2.3 Graphical User Interface

The GUI consists of a non-editable `TextField` (display) and a 4×5 grid of controls arranged via `GridBagLayout`:

- *Digits*: `0--9`.

- *Operators*: `+`, `-`, `*`, `/`.
- *Utilities*: `C` (clear), `Del` (backspace), `.` (decimal point), `=` (evaluate).

The display spans the full width of the grid (columns 0–3). The equals button occupies two vertical cells to mirror common hardware layouts and emphasize its role. Button insets and the panel background (`Color.gray`) provide a clear visual separation.

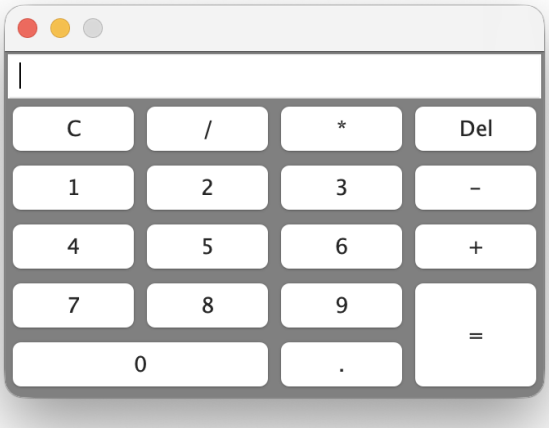

Figure 1: Graphical User Interface of the Calculator application

Focus Management. To ensure immediate keyboard usability, focus is explicitly requested on the display and reinforced on the AWT Event Dispatch Thread (EDT) via `EventQueue.invokeLater(...)`. The frame is centered on screen and sized to a fixed, non-resizable dimension to maintain the grid's visual integrity.

2.4 Event Model and Input Channels

The application implements three listener interfaces:

- `WindowListener`: handles lifecycle events; `windowClosing` terminates the application.
- `ActionListener`: receives button presses; each `Button` is registered with a shared handler.
- `KeyListener`: enables full keyboard control; keystrokes are normalized to synthetic `ActionEvents` when appropriate to reuse the same logic path.

Keyboard Mappings. Digits `0--9`, operators `+ - * /`, decimal point `.`, equals/enter (`=` or `\n`), and clear (`c`/`C`) are supported. Backspace is handled in `keyPressed` by consuming the event and performing a display pop (delete-last-character), keeping keyboard and button behavior aligned.

2.5 State Management and Validation

The UI state consists of:

1. The display buffer (string).
2. The last selected binary operator (`operator` field) or space when none is active.

The display holds, at most, a single binary expression of the form

$$\langle \text{left operand} \rangle \ \langle \text{operator} \rangle \ \langle \text{right operand} \rangle,$$

where each operand may be signed and decimal. The helper `firstOpIndex` deliberately skips the leading sign to correctly detect the first binary operator.

Negative Numbers. A leading `-` is treated as a sign if it occurs at the beginning of the buffer or immediately after another operator (e.g., `5*-2`). The method `endsWithSign` guards against incomplete inputs like `3+ -` or a trailing sign before evaluation.

Decimal Points. `hasDotInCurrentOperand` ensures that each operand contains at most one decimal point. Attempts to add a second decimal in the active operand trigger a transient warning (*"Only one '.' per number!"*).

Operator Entry Rules. If the user enters two binary operators (e.g., `3+`), the last one replaces the previous operator (yielding `3`). An alternative strategy (rejecting the second operator with a warning) was also prototyped, but the replacement model was chosen to keep the user flow uninterrupted.

2.6 Evaluation Semantics

Upon pressing `=`, the buffer is validated to ensure both operands are present and the expression does not end with a dangling sign. The expression is then split at the first binary operator (ignoring a leading sign). Parsing uses `Double.parseDouble`, and a single binary operation is performed:

$$+, -, *, /$$

Division by zero is explicitly detected; the display shows *"Cannot divide by zero"* and the operator state is cleared. There is no operator precedence or chained expression evaluation; the design intentionally restricts input to one binary operation at a time for clarity and robustness.

2.7 Output Formatting Policy

Results are formatted with `formatNumber`:

- For magnitudes in $[10^{-6}, 10^{12})$, numbers are rendered in fixed-point with up to 12 fractional digits, trailing zeros and a trailing decimal point removed.
- For very small or large magnitudes, scientific notation is used (`0.############eX`) with US locale and normalized exponent formatting (`e` instead of `E`, no explicit plus sign).
- `NaN` or infinite results are displayed as *"Overflow"*.

2.8 User Feedback and Responsiveness

Transient messages (e.g., *"Only one '.' per number!"*, *"Complete the expression before '='!"*, *"Number format error!"*) reuse the display area via `flash`. A `java.util.Timer` schedules restoration of the previous content after 2 seconds; the update is marshalled back onto the EDT using `EventQueue.invokeLater`, preserving thread-safety for AWT components.

2.9 Error Handling

The system defends against common user and numeric errors:

- **Input**: multiple decimals per operand, leading `*//`, trailing signs, and malformed numbers trigger readable messages.
- **Arithmetic**: division by zero is intercepted with a clear message.
- **Formatting**: non-finite results are collapsed to *"Overflow"*.

All error paths leave the application in a consistent state, typically by clearing the stored operator and keeping (or restoring) a valid display buffer.

2.10 Program Flow

1. User enters digits/operators via buttons or keyboard.
2. The listener layer normalizes inputs and updates the display buffer while enforcing validation rules (sign handling, decimal constraints, operator replacement).
3. On `=`, the buffer is parsed into $\langle \text{left}, \text{op}, \text{right} \rangle$, evaluated, and the result is formatted and displayed.
4. Users may clear (`C`), delete last character (`Del`/Backspace), or continue by entering a new operator and operand.

2.11 Extensibility and Limitations

The current design intentionally supports a single binary operation. Natural extensions include:

- *Chained expressions and precedence*: introduce a tokenizer and a small expression parser (e.g., shunting-yard) to support multi-operator input.
- *Memory functions*: add `M+`, `M-`, `MR`, `MC` with an internal accumulator.
- *Unary operations*: percent, square root, reciprocal, sign toggle; these can operate on the active operand and reuse the existing validation routines.
- *Internationalization*: adopt locale-aware formatting and input (decimal separators).
- *Separation of concerns*: factor arithmetic and formatting into a dedicated model/service class to facilitate unit testing independent of the GUI.

These enhancements can be integrated without disrupting the current event model by keeping the listener layer thin and delegating to cohesive helper classes.

3 Implementation

3.1 Development Environment

The project was implemented in Java SE (JDK 17) using the Abstract Window Toolkit (AWT). Compilation and execution were verified on multiple operating systems without requiring external libraries, ensuring portability. Development took place in IntelliJ IDEA, though any standard Java environment can be used.

3.2 Class Structure

The application is encapsulated in a single class, `Calculator`. This class is responsible for:

- Constructing and rendering the graphical interface.
- Maintaining the state of the display and active operator.
- Handling user interactions through event listeners.
- Parsing expressions and evaluating results.

Although monolithic in design, this structure was chosen for compactness and assignment scope. Key object-oriented principles are still demonstrated at the method level, including encapsulation, modularity, and abstraction.

3.3 GUI Construction

The graphical interface is built within the constructor of the `Calculator` class. A `GridBagLayout` provides fine-grained control over button placement, while each component is created, configured, and registered with a common event handler.

```
public Calculator() {
    p.setLayout(a);
    b.fill = GridBagConstraints.BOTH;
    t = new TextField(20);
    t.setEditable(false);
    a.setConstraints(t, b);
    p.add(t);

    b1 = new Button("1");
    b1.addActionListener(this);
    p.add(b1);
    ...
}
```

This example shows the initialization of the display field and a digit button; the same pattern is applied to all other components.

3.4 Event Handling

The program implements three listener interfaces:

```
public class Calculator implements WindowListener,
                                   ActionListener,
                                   KeyListener {
    ...
}
```

- **ActionListener:** All buttons invoke the shared `actionPerformed` method, where the source is inspected to determine the corresponding action.
- **KeyListener:** Keyboard input is normalized and routed through the same logic as button presses, ensuring consistent behavior between mouse and keyboard interaction.
- **WindowListener:** Provides graceful termination when the application window is closed.

3.5 Expression Evaluation

The calculator supports single binary operations. When the equals button is pressed, the input string is parsed, validated, and evaluated:

```
if (buttonText.equals("=")) {
    String left  = cur.substring(0, idx);
    String right = cur.substring(idx + 1);
    char op = cur.charAt(idx);

    double l = Double.parseDouble(left);
    double r = Double.parseDouble(right);
    double res;

    switch (op) {
        case '+': res = l + r; break;
        case '-': res = l - r; break;
        case '*': res = l * r; break;
        case '/':
            if (r == 0) { setDisp("Cannot divide by zero"); return; }
            res = l / r; break;
        default: return;
    }
    showResult(res);
}
```

Division by zero is explicitly handled, preventing runtime errors and providing user feedback.

3.6 Formatting and Feedback

Results are formatted by the `formatNumber` method to trim trailing zeros and automatically switch to scientific notation when necessary. User feedback (e.g., invalid input or incomplete expressions) is delivered temporarily via the display:

```
private void flash(String message) {
    final String prev = disp();
    setDisp(message);
    new Timer().schedule(new TimerTask() {
        @Override public void run() {
            EventQueue.invokeLater(() -> setDisp(prev));
        }
    }, 2000);
}
```

This approach allows transient error messages without permanently altering the main display.

3.7 Encapsulation and OOP Practices

While implemented as a single class, the program consistently applies object-oriented principles:

- **Encapsulation:** Display and validation logic are encapsulated in helper methods such as `setDisp`, `hasDotInCurrentOperand`, and `endsWithSign`.
- **Polymorphism:** By implementing multiple listener interfaces (`ActionListener`, `KeyListener`, `WindowListener`), the class uses interface polymorphism to centralize event handling. While not an example of inheritance-based polymorphism, this demonstrates flexible event-driven design.
- **Abstraction:** Utility methods such as number formatting and message flashing abstract away low-level details, simplifying the main event logic.

4 Testing and Results

4.1 Testing Strategy

The calculator was tested using both manual input (keyboard and button clicks) and targeted edge cases. The goal of testing was to ensure correctness of arithmetic operations, proper validation of inputs, and consistent GUI behavior. The following aspects were emphasized:

- **Functional correctness:** addition, subtraction, multiplication, and division produce expected results.
- **Input validation:** prevention of multiple decimal points in one operand, replacement or reinterpretation of consecutive operators (with special support for negative signs), and handling of trailing signs.

- **Error handling:** division by zero, invalid numbers, and incomplete expressions produce clear feedback.
- **User interface:** keyboard and button inputs behave consistently; display updates correctly after every operation.

4.2 Representative Test Cases

Test Case	Input	Expected Result
Addition	$12 + 8 =$	20
Subtraction	$50 - 17 =$	33
Multiplication	$7 * 6 =$	42
Division	$144/12 =$	12
Division by Zero	$9/0 =$	"Cannot divide by zero"
Negative Operand	$5 * -2 =$	-10
Large Result	$9999999999 * 999 =$	9.989999999001e12
Backspace/Delete	$123 \rightarrow$ `Del`	12 remains in display

Table 1: Representative test cases for the calculator application

4.3 Execution Results

The calculator behaved as expected in all tested cases:

- Arithmetic operations yielded correct results.
- Validation rules successfully prevented invalid input.
- Error messages were displayed temporarily and then reverted to the previous valid state.
- Both button presses and keyboard input produced identical behavior, confirming input consistency.

5 Conclusion and Future Work

This project successfully achieved its primary objective: implementing a basic calculator in Java to demonstrate the principles of object-oriented programming (OOP) in a practical context. The application supports core arithmetic operations, integrates both button- and keyboard-based input, and provides robust validation and feedback mechanisms. Through this implementation, key OOP concepts such as encapsulation, modular design, abstraction, and event-driven programming were effectively applied and reinforced.

The project also demonstrated the importance of user interface considerations in software design. By building the GUI with AWT and `GridBagLayout`, the calculator provides a simple but functional interface that balances usability with implementation simplicity. Furthermore, the use of helper methods and validation routines ensured reliable behavior and consistent handling of user inputs and errors.

While the calculator fulfills its goals, its limitations are clear: it evaluates only single binary operations, lacks operator precedence, and omits advanced functions found in scientific calculators. These limitations, however, were intentional to maintain clarity and focus on OOP fundamentals.

Looking forward, several enhancements can be pursued as part of future work:

- Extending support for chained expressions and operator precedence through an expression parser.
- Introducing memory functions (`M+`, `M-`, `MR`, `MC`) to increase usability.
- Adding unary operations such as square root, reciprocal, and percentage.
- Refactoring the design into a Model–View–Controller (MVC) architecture to further illustrate separation of concerns and improve testability.
- Exploring alternative GUI frameworks like Swing or JavaFX for a richer and more modern interface.

In conclusion, the project provided valuable hands-on experience with Java programming, GUI development, and OOP concepts. It not only met the requirements of the assignment but also laid a strong foundation for future enhancements and more advanced applications.

A Source Code

```
import java.awt.*;
import java.awt.event.*;
import java.util.Timer;
import java.util.TimerTask;

public class Calculator implements WindowListener,
                                   ActionListener,
                                   KeyListener {

    GridBagLayout a = new GridBagLayout();
    GridBagConstraints b = new GridBagConstraints();

    Button b1, b2, b3, b4, b5, b6, b7, b8, b9, b0;
    Button b_c, b_sum, b_dif, b_div, b_mul, b_eq, b_dot, b_back;

    Frame f = new Frame();
    Panel p = new Panel();

    TextField t;

    char operator = ' ';

    public Calculator() {
        p.setLayout(a);
        b.fill = GridBagConstraints.BOTH;
        b.insets = new Insets(1, 1, 1, 1);

        // Display
```

```
        b.gridx = 0; b.gridy = 0; b.gridwidth = 4; b.gridheight = 1;
        b.weightx = 0.7;
        b.weighty = 0.7;

        t = new TextField(20);
        t.setEditable(false);
        a.setConstraints(t, b);
        p.add(t);

        // Row 1: C * / Del
        b_c = new Button("C"); b_c.addActionListener(this);
        b.gridx = 0; b.gridy = 1; b.gridwidth = 1; b.gridheight = 1;
        a.setConstraints(b_c, b); p.add(b_c);

        b_div = new Button("/"); b_div.addActionListener(this);
        b.gridx = 1; a.setConstraints(b_div, b); p.add(b_div);

        b_mul = new Button("*"); b_mul.addActionListener(this);
        b.gridx = 2; a.setConstraints(b_mul, b); p.add(b_mul);

        b_back = new Button("Del");
        b_back.addActionListener(this);
        b.gridx = 3; b.gridy = 1; b.gridwidth = 1; b.gridheight = 1;
        a.setConstraints(b_back, b);
        p.add(b_back);

        // Row 2: 1 2 3 +
        b1 = new Button("1"); b1.addActionListener(this);
        b.gridx = 0; b.gridy = 2; a.setConstraints(b1, b); p.add(b1);

        b2 = new Button("2"); b2.addActionListener(this);
        b.gridx = 1; a.setConstraints(b2, b); p.add(b2);

        b3 = new Button("3"); b3.addActionListener(this);
        b.gridx = 2; a.setConstraints(b3, b); p.add(b3);

        b_dif = new Button("-"); b_dif.addActionListener(this);
        b.gridx = 3; b.gridheight = 1; a.setConstraints(b_dif, b); p.
            add(b_dif);

        // Row 3: 4 5 6
        b4 = new Button("4"); b4.addActionListener(this);
        b.gridx = 0; b.gridy = 3; b.gridheight = 1; a.setConstraints(b4
            , b); p.add(b4);

        b5 = new Button("5"); b5.addActionListener(this);
        b.gridx = 1; a.setConstraints(b5, b); p.add(b5);

        b6 = new Button("6"); b6.addActionListener(this);
        b.gridx = 2; a.setConstraints(b6, b); p.add(b6);

        b_sum = new Button("+"); b_sum.addActionListener(this);
        b.gridx = 3; b.gridy = 3; b.gridheight = 1; a.setConstraints(
            b_sum, b); p.add(b_sum);

        // Row 4: 7 8 9 =
        b_eq = new Button("="); b_eq.addActionListener(this);
```

```
        b.gridx = 3; b.gridy = 4; b.gridheight = 2; a.setConstraints(b_eq, b); p.add(b_eq);

        b7 = new Button("7"); b7.addActionListener(this);
        b.gridx = 0; b.gridy = 4; b.gridheight = 1; a.setConstraints(b7, b); p.add(b7);

        b8 = new Button("8"); b8.addActionListener(this);
        b.gridx = 1; a.setConstraints(b8, b); p.add(b8);

        b9 = new Button("9"); b9.addActionListener(this);
        b.gridx = 2; a.setConstraints(b9, b); p.add(b9);

        // Row 5: 0 .
        b0 = new Button("0"); b0.addActionListener(this);
        b.gridx = 0; b.gridy = 5; b.gridwidth = 2; a.setConstraints(b0, b); p.add(b0);

        b_dot = new Button("."); b_dot.addActionListener(this);
        b.gridx = 2; b.gridwidth = 1; a.setConstraints(b_dot, b); p.add(b_dot);

        p.setBackground(Color.gray);
        f.add(p);

        f.addWindowListener(this);
        f.pack();
        f.setSize(330, 240);

        Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
        Dimension size   = f.getSize();
        f.setLocation((screen.width - size.width) / 2, (screen.height - size.height) / 2);

        f.setResizable(false);
        f.setVisible(true);

        t.addKeyListener(this);
        f.addKeyListener(this);
        p.addKeyListener(this);
        t.requestFocus();

        EventQueue.invokeLater(() -> t.requestFocusInWindow());
    }

    public static void main(String[] args) {
        new Calculator();
    }

    // ===== Helpers =====
    private String disp() { return t.getText().trim(); }
    private void setDisp(String s) { t.setText(s); }

    private void flash(String message) {
        final String prev = disp();
        setDisp(message);
        new Timer().schedule(new TimerTask() {
            @Override public void run() {
```

```
                EventQueue.invokeLater(() -> setDisp(prev));
            }
        }, 2000);
    }

    private int firstOpIndex(String s) {
        for (int i = 1; i < s.length(); i++) {
            char c = s.charAt(i);
            if (isOperator(c)) return i;
        }
        return -1;
    }

    private boolean endsWithSign(String s) {
        if (s.isEmpty()) return false;
        int n = s.length();
        if (s.charAt(n - 1) != '-') return false;
        if (n == 1) return true;
        return isOperator(s.charAt(n - 2));
    }

    private boolean hasDotInCurrentOperand(String s) {
        int idx = firstOpIndex(s);
        String operand = (idx == -1) ? s : s.substring(idx + 1);
        return operand.contains(".");
    }

    private boolean isOperator(char c) {
        return c == '+' || c == '-' || c == '*' || c == '/';
    }

    private void showResult(double res) {
        /* String s = String.format(java.util.Locale.US, "%.12f", res);
        s = s.replaceAll("\\.?0+$", "");
        setDisp(s.isEmpty() ? "0" : s); */
        setDisp(formatNumber(res));
    }

    private String formatNumber(double x) {
        if (Double.isNaN(x) || Double.isInfinite(x)) return "Overflow";

        double ax = Math.abs(x);

        if (ax != 0 && (ax < 1e-6 || ax >= 1e12)) {
            java.text.DecimalFormatSymbols sym = java.text.
                DecimalFormatSymbols.getInstance(java.util.Locale.US);
            java.text.DecimalFormat sci = new java.text.DecimalFormat("
                0.############E0", sym);
            String s = sci.format(x);
            s = s.replace("E", "e");
            s = s.replace("e+", "e");
            return s;
        }

        String s = String.format(java.util.Locale.US, "%.12f", x);
        s = s.replaceAll("\\.?0+$", "");
        return s;
    }
```

```
192
193     // ===== Events =====
194     @Override
195     public void actionPerformed(ActionEvent e) {
196         if (!(e.getSource() instanceof Button)) return;
197
198         String buttonText = ((Button) e.getSource()).getLabel();
199         String cur = disp();
200
201         // Delete
202         if (buttonText.equals("Del")) {
203             String curText = disp();
204             if (!curText.isEmpty()) {
205                 setDisp(curText.substring(0, curText.length() - 1));
206             }
207             return;
208         }
209
210         // Clear
211         if (buttonText.equals("C")) {
212             setDisp("");
213             operator = ' ';
214             return;
215         }
216
217         // Digits
218         if ("0123456789".contains(buttonText)) {
219             setDisp(cur + buttonText);
220             return;
221         }
222
223         // Dot
224         if (buttonText.equals(".")) {
225             if (cur.isEmpty() || endsWithSign(cur)) {
226                 setDisp(cur + "0.");
227                 return;
228             }
229             if (hasDotInCurrentOperand(cur)) return;
230             int idx = firstOpIndex(cur);
231             String operand = (idx==-1) ? cur : cur.substring(idx+1);
232             if (operand.contains(".")) {
233                 flash("Only one '.' per number!");
234                 operator=' ';
235                 return;
236             }
237             setDisp(cur + ".");
238             return;
239         }
240
241         // Minus as sign
242         if (buttonText.equals("-")) {
243             if (operator == ' ' && cur.isEmpty()) {
244                 setDisp("-");
245                 return;
246             }
247             if (operator != ' ' && !cur.isEmpty() && cur.charAt(cur.
                length() - 1) == operator) {
248                 setDisp(cur + "-");
```

```
249                 return;
250             }
251         }
252
253         // This code is an alternative approach: if the user enters two
                consecutive operators, it returns an error.
254         /* if (isOperator(buttonText.charAt(0))) {
255             if (operator!=' ') {
256                 flash("Do not enter a second operator!");
257                 operator=' ';
258                 return;
259             }
260             if (cur.isEmpty() && (buttonText.equals("*") || buttonText.
                    equals("/"))) {
261                 flash("Do not enter '" + buttonText + "' before the
                        first number!");
262                 operator=' ';
263                 return;
264             }
265             if (cur.isEmpty()) return;
266
267             if (endsWithSign(cur)) return;
268
269             int idx = firstOpIndex(cur);
270             if (idx != -1 && idx == cur.length() - 1) {
271                 setDisp(cur.substring(0, cur.length() - 1) + buttonText
                        );
272                 operator = buttonText.charAt(0);
273                 return;
274             }
275
276             char last = cur.charAt(cur.length() - 1);
277             if (operator == ' ' && (Character.isDigit(last) || last ==
                    '.')) {
278                 setDisp(cur + buttonText);
279                 operator = buttonText.charAt(0);
280             }
281             return;
282         } */
283
284         // Operators
285         if (isOperator(buttonText.charAt(0))) {
286             if (cur.isEmpty()) {
287                 if (buttonText.equals("*") || buttonText.equals("/")) {
288                     flash("You cannot enter '" + buttonText + "' before
                            the first number!");
289                     return;
290                 }
291                 return;
292             }
293             if (endsWithSign(cur)) return;
294
295             char last = cur.charAt(cur.length()-1);
296             if (isOperator(last)) {
297                 setDisp(cur.substring(0, cur.length()-1) + buttonText);
298                 operator = buttonText.charAt(0);
299                 return;
300             }
```

```
            setDisp(cur + buttonText);
            operator = buttonText.charAt(0);
            return;
        }

        // Equals
        if (buttonText.equals("=")) {
            if (cur.isEmpty() || endsWithSign(cur)) {
                flash("Complete the expression before '='!");
                operator=' ';
                return;
            }
            int idx = firstOpIndex(cur);
            if (idx == -1 || idx == cur.length() - 1) return;

            String left  = cur.substring(0, idx);
            String right = cur.substring(idx + 1);
            char op = cur.charAt(idx);

            try {
                double l = Double.parseDouble(left);
                double r = Double.parseDouble(right);
                double res;

                switch (op) {
                    case '+': res = l + r; break;
                    case '-': res = l - r; break;
                    case '*': res = l * r; break;
                    case '/':
                        if (r == 0) { setDisp("Cannot divide by zero");
                            operator = ' '; return; }
                        res = l / r; break;
                    default: return;
                }

                operator = ' ';
                showResult(res);
            } catch (NumberFormatException ex) {
                flash("Number format error!");
            }
            return;
        }
    }

    @Override
    public void keyTyped(KeyEvent e) {
        char c = e.getKeyChar();

        // Digits
        if (Character.isDigit(c)) {
            setDisp(disp() + c);
        }
        // Operators
        else if (c == '+' || c == '-' || c == '*' || c == '/') {
            actionPerformed(new ActionEvent(new Button(String.valueOf(c
               )), ActionEvent.ACTION_PERFORMED, ""));
        }
        // Decimal point
```

```
357             else if (c == '.') {
358                 actionPerformed(new ActionEvent(new Button("."),
                        ActionEvent.ACTION_PERFORMED, ""));
359             }
360             // Enter or '='
361             else if (c == '\n' || c == '=') {
362                 actionPerformed(new ActionEvent(new Button("="),
                        ActionEvent.ACTION_PERFORMED, ""));
363             }
364             // 'c' or 'C' clears
365             else if (c == 'c' || c == 'C') {
366                 actionPerformed(new ActionEvent(new Button("C"),
                        ActionEvent.ACTION_PERFORMED, ""));
367             }
368         }
369 
370         public void keyPressed(KeyEvent e) {
371             if (e.getKeyCode() == KeyEvent.VK_BACK_SPACE) {
372                 e.consume();
373                 String curText = disp();
374                 if (!curText.isEmpty()) {
375                     setDisp(curText.substring(0, curText.length() - 1));
376                 }
377             }
378         }
379         @Override public void keyReleased(KeyEvent e) {}
380 
381         public void windowOpened(WindowEvent e) {}
382         public void windowClosing(WindowEvent e) { System.exit(0); }
383         public void windowClosed(WindowEvent e) {}
384         public void windowIconified(WindowEvent e) {}
385         public void windowDeiconified(WindowEvent e) {}
386         public void windowActivated(WindowEvent e) {}
387         public void windowDeactivated(WindowEvent e) {}
388 }
```

Listing 1: Calculator.java source code