Aptech®

Unleash your potential

2025

JavaFX

CALCULATOR APPLICATION

TEAM MEMBERS:

ABDULRAHMAN ABDULBAQI ABDULMUMEN ABDULWARETH
AHMED SALAH ABDULJALIL AHMED
AMMAR YAHYA ABDU NOMAN

SUPERVISED BY:

Sina AL-Gahafi

Acknowledgments

All praise is due to **Allah**, the Most Merciful and the Bestower of knowledge, whose guidance and blessings have enabled us to complete this eProject successfully. It is by His will and support that we have been able to reach this milestone.

We extend our sincere thanks to our **supervisors and professors**, whose expert guidance, constructive feedback, and continuous encouragement played a crucial role in shaping the direction of this project. Their mentorship helped us navigate challenges and stay focused on our goals.

We are also deeply grateful to our **team members** for their dedication and teamwork, and to the **Aptech Centre** for providing essential resources, training, and a nurturing environment. This project was made possible through the combined efforts and support of all involved, and we truly value each contributions.

Table of Contents

Project Synopsis:

This project report documents the development of a **Graphical Scientific Calculator** created using **Java** and **JavaFX**. The calculator provides users with an interactive interface to perform a wide variety of mathematical and logical operations efficiently and intuitively. The purpose of this project was to offer a hands-on implementation experience that reinforces key programming concepts, interface design, and modular software development.

Project Objective:

The main objective of this project was to design and develop a user-friendly calculator that not only handles basic arithmetic operations but also includes advanced scientific and number-theoretic functions. By integrating a GUI, the project aimed to enhance user experience while deepening our understanding of Java programming, event-driven interaction, and component-based design using JavaFX.

The application supports a full range of functionalities, including standard operations like addition, subtraction, multiplication, and division, as well as advanced features such as exponentiation, square root, cube, trigonometric calculations (sin, cos, tan, asin, acos, atan), rounding methods (floor, ceiling, round), and comparative functions (min, max). Additionally, it incorporates logic-based operations such as palindrome checks, Armstrong number detection, prime number validation, GCD, LCM, and average calculations.

The project was structured with clean, well-documented code and thoroughly tested for accuracy and responsiveness. Leveraging JavaFX enabled us to create an intuitive graphical user interface with smooth interactivity. The successful completion of this project reflects not only our technical skills but also our ability to collaborate, debug, and deliver a complete real-world application.

Project Analysis:

The following section outlines the detailed problem statement**,** expected output, hardware and software required, and the standards plan followed during the development of the project. It provides a clear understanding of the project's functional requirements and coding expectations.

Problem Statement:

Write a program to create Calculator which should perform various function like Addition, Subtraction, Multiplication, Division, Power, Square, Cube, Square root, round, ceiling, floor, Min Value, Max Value, sin, cos, asin, acos, atan, exponential, Palindrome, Armstrong number, Prime number, Average (first take the input the number of entries expected and then calculate the average), GCD (of two numbers), LCM(of two numbers). The list of option must be stated on the application which should not terminate the program until last option exit is selected.

Excepted Output:

1. Should display all the option this new designed calculated upon.
2. Must calculate mathematical functions and displayed the output based on that.
3. Functions like
    a. Adding, subtracting, Multiplication, Division.
    b. Power, Square, Cube, Square root
    c. Round, ceiling, floor, Min Value, Max Value.
    d. Trigonometric functions such as sin, cos, asin, acos, atan
    e. Exponential function
    f. Palindrome (It must accept at least three digit number as it must ask the number of digit to be entered
    g. and then must be check whether the given number is prime or not.

h. Armstrong number (must check whether the entered is Armstrong or not.
i. Prime number (must calculate whether the number is prime or not
j. Average (first take the input the number of entries expected and then calculate the average),
k. GCD (of the two number entered by the user)
l. LCM (of the two number entered by the user)
m. Exit option (to terminate the program).

Standards Plan:

1. Every code block must have comments.
2. The logic of the program needs to be explained. Proper documentation should be maintained.
3. Complete Project Report along with synopsis, code and documentation should be prepared.

Hardware and Software Required:

Hardware:

- A minimum computer system that will help you access all the tools in the courses is a Pentium 166 or better.
- 128 Megabytes of RAM or better

Software:

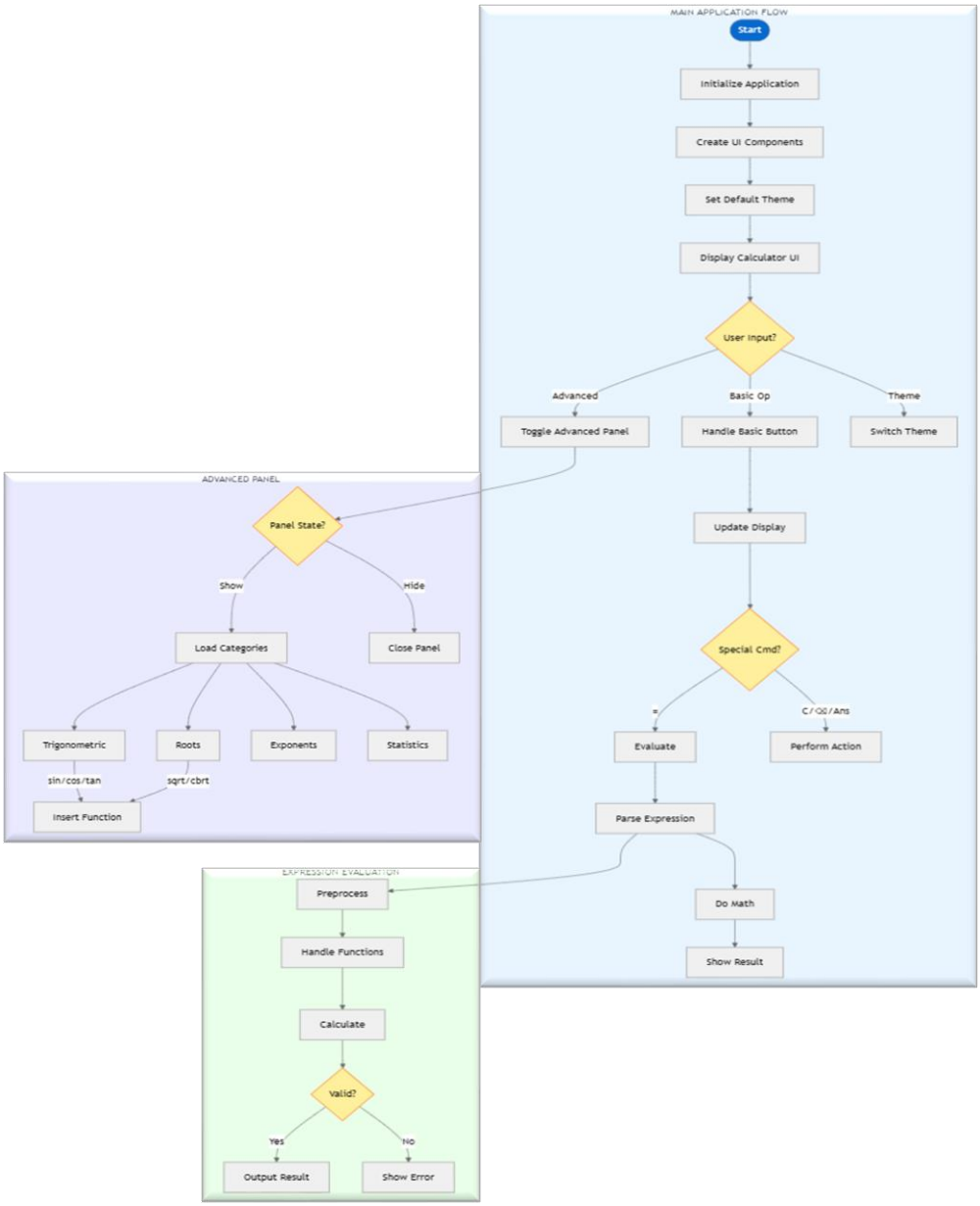
Either or combination of the following Software's are to be used:

- Java J2EE / .NET / C / C++
- Notepad.

Project Design:

Since the application does not involve any database or data storage, the flowchart and Data Flow Diagrams (DFD) have been designed solely around the internal logic and operational flow of the calculator. These diagrams illustrate how user inputs are processed through various functional modules and how results are computed and displayed within the JavaFX interface.

Flowchart:



Data Flow Diagrams (DFD):

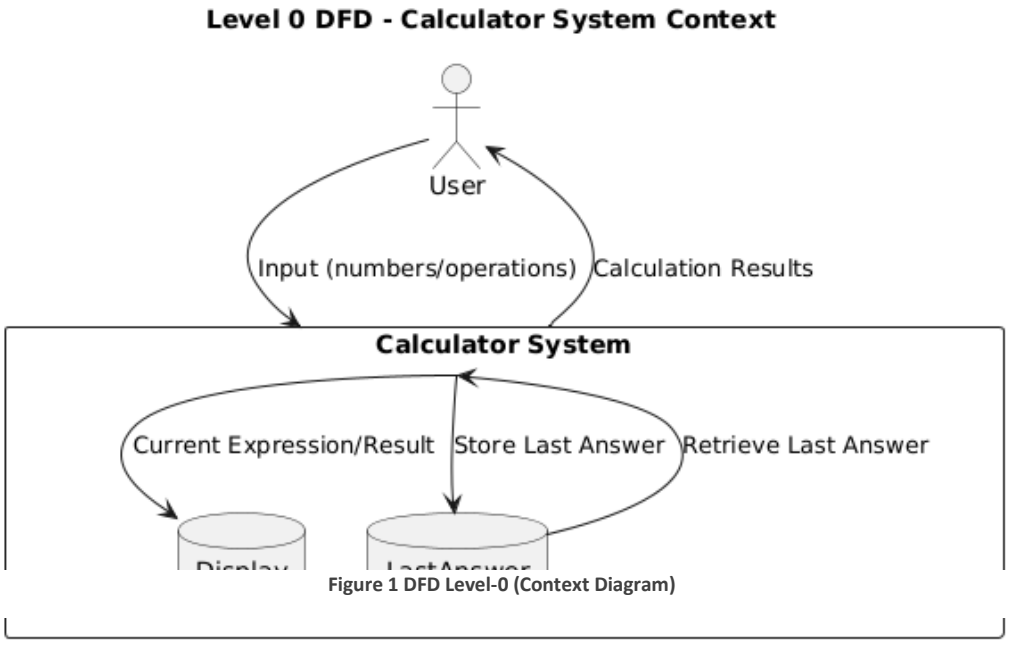


Figure 1 DFD Level-0 (Context Diagram)

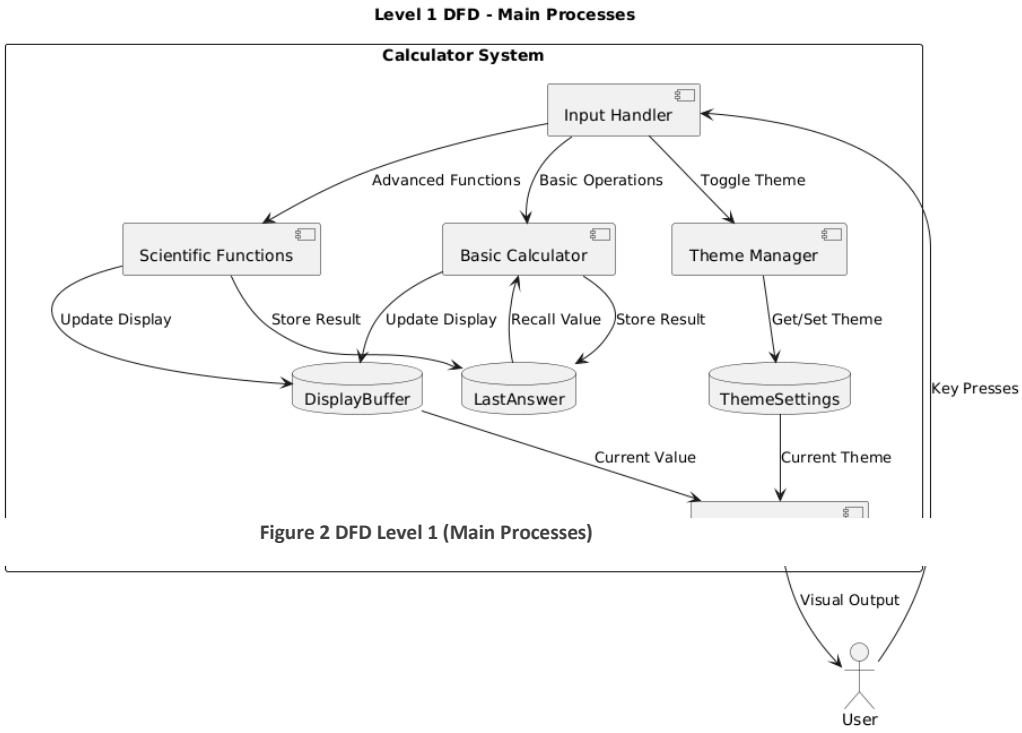


Figure 2 DFD Level 1 (Main Processes)

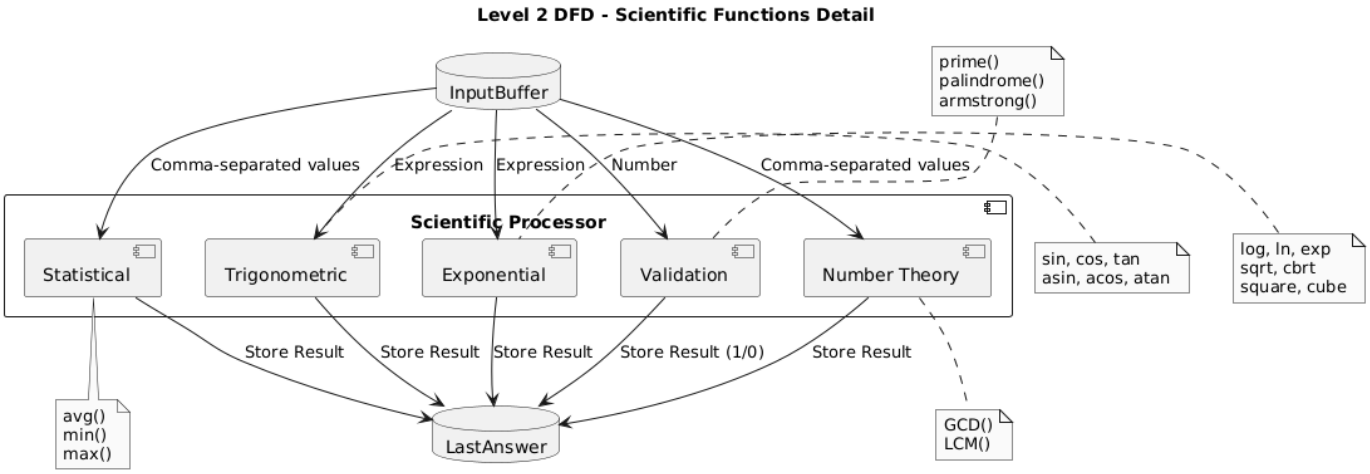
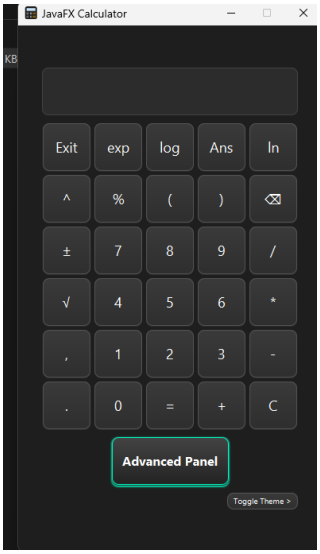


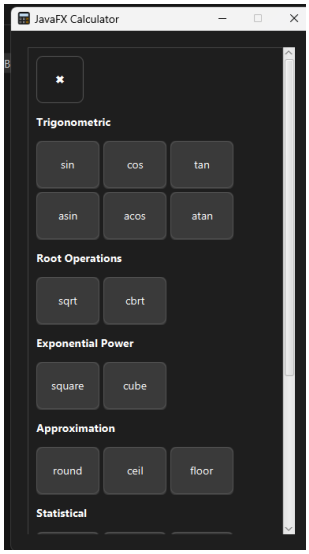
Figure 3 Level 2 (Scientific Functions)

Screenshots:

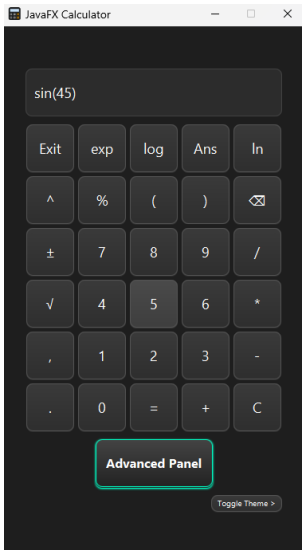
Application Main Interface (Light Theme)



Application Main Interface (Dark Theme)

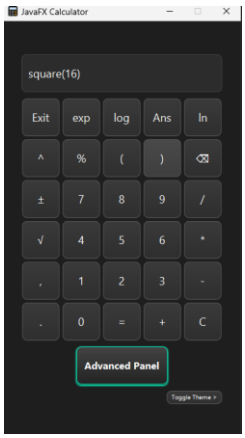


Application Advanced Panel

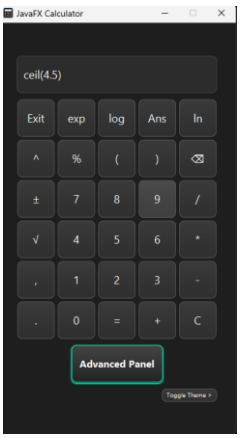


Trigonometric Operations

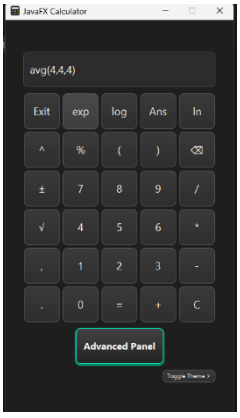
Root Operations



Exponential Power Operatio



Approximation Operations



Statistical Operations

Validation Operations

Number Theory Operations

Source Code with Comments:

Calculator.java

```
// Advanced Calculator Project using JavaFX
// Includes basic and scientific operations with a customizable GUI
(Dark/Light mode)
// Enables a side panel for advanced operations
package calculator;

// Importing all necessary packages and libraries for building the JavaFX
user interface and handling events
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.image.Image;
import javafx.scene.layout.StackPane;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import java.util.*;
import java.util.function.*;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.input.MouseEvent;
import javafx.stage.Stage;
import javafx.application.Platform;
import javafx.scene.control.Alert.AlertType;
import javafx.stage.StageStyle;

public class calculator extends Application {

    // Field for displaying input and results
    private TextField display = new TextField();
    // Side panel for advanced operations
    private VBox advancedPanel = new VBox(15);
    private boolean advancedVisible = false; // Panel visibility state
    private ScrollPane scrollPane; // For scrolling inside the panel
    private double lastAnswer = 0; // Stores the last result
```

```
    private boolean isDarkMode = false; // Theme state

    @Override
    public void start(Stage primaryStage) {

        display.setEditable(false);
        display.setMinHeight(50);
        display.setStyle("-fx-font-size: 18;");

        // Create basic operation buttons
        GridPane basicButtons = createBasicButtons();

        // Button to toggle the side panel
        Button toggleAdvanced = new Button("Advanced Panel");
        toggleAdvanced.getStyleClass().add("toggleAdvanced");
        toggleAdvanced.setOnAction(e -> toggleAdvancedPanel());

        // Setup advanced operations inside the panel
        createAdvancedButtons();

        //Setup the side panel
        advancedPanel.setPadding(new Insets(10));
        advancedPanel.getStyleClass().add("advanced");
        advancedPanel.setTranslateX(300);
        advancedPanel.setPrefWidth(300);
        //advancedPanel = false;

        //Setup scrollPane the side panel
        scrollPane = new ScrollPane(advancedPanel);
        scrollPane.setFitToWidth(true);
        scrollPane.setPrefWidth(300);
        scrollPane.setMouseTransparent(false);
        scrollPane.setTranslateX(300);
        scrollPane.setVisible(false);
        scrollPane.setStyle("-fx-background: transparent; -fx-background-
color: transparent;");

        // Container for theme toggle button
        StackPane theme = new StackPane();

        // Main layout of the calculator
        VBox calculatorLayout = new VBox(10, display, basicButtons,
toggleAdvanced, theme);
        calculatorLayout.setAlignment(Pos.CENTER);
        calculatorLayout.setPadding(new Insets(10));
        StackPane root = new StackPane(calculatorLayout, scrollPane);

        Scene scene = new Scene(root);

        // Hide side panel when clicking outside it
        scene.addEventFilter(MouseEvent.MOUSE_PRESSED, event -> {
            if (advancedVisible &&
!advancedPanel.localToScene(advancedPanel.getBoundsInLocal()).contains(even
t.getSceneX(), event.getSceneY())) {
                hideAdvancedPanel();
            }
        });

        // Button to toggle between dark and light mode
        Button toggleTheme = new Button("Toggle Theme >");
        toggleTheme.setStyle("""
```

```
                    -fx-font-size: 10;-fx-min-width: 80px;
                    -fx-min-height: 20px;""");

        StackPane.setAlignment(toggleTheme, Pos.TOP_RIGHT);
        theme.getChildren().add(toggleTheme);

        // Button to toggle setOnAction...
        toggleTheme.setOnAction(e -> {
            isDarkMode = !isDarkMode;
            scene.getStylesheets().clear();
            if (isDarkMode) {

scene.getStylesheets().add(getClass().getResource("Dark.css").toExternalFor
m());

            } else {

scene.getStylesheets().add(getClass().getResource("light.css").toExternalFo
rm());
            }
        });
        // Load default theme (light)

scene.getStylesheets().add(getClass().getResource("light.css").toExternalFo
rm());

        //Prevent closing the window with the "❌" button
        primaryStage.setOnCloseRequest(evnt -> {
            evnt.consume();
        });

    scene.setOnKeyPressed(event -> {
    String key = event.getText(); // Get the character that was typed

    switch (event.getCode()) {

        case ENTER -> handleInput("=");           // Press Enter to
evaluate
        case BACK_SPACE -> handleInput("⌫");      // Delete last character
        case ESCAPE, DELETE -> handleInput("C");  // Clear the display

        // Handle Shift + 5 to insert %
        case DIGIT5 -> {
            if (event.isShiftDown()) handleInput("%");
            else handleInput("5");
        }

        // Handle Shift + 6 to insert ^
        case DIGIT6 -> {
            if (event.isShiftDown()) handleInput("^");
            else handleInput("6");
        }

        // Handle Shift + 9 to insert (
        case DIGIT9 -> {
            if (event.isShiftDown()) handleInput("(");
            else handleInput("9");
        }

        // Handle Shift + 0 to insert )
        case DIGIT0 -> {
```

```
            if (event.isShiftDown()) handleInput(")");
            else handleInput("0");
        }

        // Digits 1 to 4 and 7 to 8
        case DIGIT1, NUMPAD1 -> handleInput("1");
        case DIGIT2, NUMPAD2 -> handleInput("2");
        case DIGIT3, NUMPAD3 -> handleInput("3");
        case DIGIT4, NUMPAD4 -> handleInput("4");
        case DIGIT7, NUMPAD7 -> handleInput("7");
        case DIGIT8, NUMPAD8 -> handleInput("8");

        // Additional Numpad digit cases not handled above
        case NUMPAD0 -> handleInput("0");
        case NUMPAD5 -> handleInput("5");
        case NUMPAD6 -> handleInput("6");
        case NUMPAD9 -> handleInput("9");

        // Handle math operators from main keyboard or numpad
        case ADD, PLUS -> handleInput("+");
        case SUBTRACT, MINUS -> handleInput("-");
        case MULTIPLY -> handleInput("*");
        case DIVIDE, SLASH -> handleInput("/");

        default -> {
            // Allow typing dot, comma, or math operators
            if (".,*/+-".contains(key)) {
                handleInput(key);
            }

            // Use key 'm' or 'n' to toggle plus/minus
            if (key.equalsIgnoreCase("m") || key.equalsIgnoreCase("n")) {
                handleInput("±");
            }
        }
    }
});

      scene.getRoot().requestFocus();
        //primaryStage.initStyle(StageStyle.UNDECORATED);
        primaryStage.setTitle("JavaFX Calculator");
        primaryStage.setScene(scene);
        primaryStage.setHeight(700);
        primaryStage.getIcons().add(new
Image(String.valueOf(getClass().getResource("icon.png"))));
        primaryStage.setResizable(false);
        primaryStage.show();
    }

    // Toggle side panel visibility
    private void toggleAdvancedPanel() {
        if (advancedVisible) {
            hideAdvancedPanel();
        } else {
            showAdvancedPanel();
        }
    }

    //Show the side panel
    private void showAdvancedPanel() {
        advancedPanel.setTranslateX(0);
```

```
        advancedVisible = true;
        scrollPane.setTranslateX(0);
        scrollPane.setVisible(true);
        display.clear();
    }

    //Hide the side panel
    private void hideAdvancedPanel() {
        scrollPane.setVisible(false);
        advancedVisible = false;
    }

    // Create basic operation buttons (6x5 grid)
    private GridPane createBasicButtons() {
        GridPane grid = new GridPane();
        grid.setHgap(5);
        grid.setVgap(5);
        grid.setAlignment(Pos.CENTER);
        String[][] buttons = {
            {"Exit", "exp", "log", "Ans", "ln"},
            {"^", "%", "(", ")", "□"},
            {"±", "7", "8", "9", "/"},
            {"√", "4", "5", "6", "*"},
            {",", "1", "2", "3", "-"},
            {".", "0", "=", "+", "C"},};

        //Repetition create basic operation buttons (6x5 grid)
        for (int row = 0; row < buttons.length; row++) {
            for (int col = 0; col < buttons[row].length; col++) {
                String symbol = buttons[row][col];
                Button btn = new Button(symbol);
                btn.setMinSize(50, 50);
                btn.setOnAction(e -> handleInput(symbol));
                grid.add(btn, col, row);
            }
        }
        return grid;
    }

    // ================== advanced operation on the side panel
(expression.. ) ====================
    // Create advanced operation buttons on the side panel
    private void createAdvancedButtons() {
        advancedPanel.getChildren().clear();

        //Button hide the side panel
        Button closeBtn = new Button("❌");
        closeBtn.getStyleClass().add("closeBtn");
        closeBtn.setOnAction(e -> hideAdvancedPanel());
        HBox topBar = new HBox(closeBtn);
        topBar.setAlignment(Pos.TOP_LEFT);
        advancedPanel.getChildren().add(topBar);

        // Organize advanced functions into categories
        Map<String, List<String>> operations = new LinkedHashMap<>();
        operations.put("Trigonometric", Arrays.asList("sin", "cos", "tan",
"asin", "acos", "atan"));
        operations.put("Root Operations", Arrays.asList("sqrt", "cbrt"));
        operations.put("Exponential Power", Arrays.asList("square",
"cube"));
```

```
        operations.put("Approximation", Arrays.asList("round", "ceil",
"floor"));
        operations.put("Statistical", Arrays.asList("avg", "min", "max"));
        operations.put("Validation", Arrays.asList("prime", "palindrome",
"armstrong"));
        operations.put("Number Theory", Arrays.asList("GCD", "LCM"));

        // For each entry in the 'operations' map, which holds a category
title and a list of operation names
        for (Map.Entry<String, List<String>> entry : operations.entrySet())
{

            // Get the category title (e.g., "Trigonometric")
            String title = entry.getKey();

            // Get the list of operations in this category (e.g., ["sin",
"cos", "tan"])
            List<String> ops = entry.getValue();
            // Create a label for the category title and apply a CSS style
class "section"
            Label sectionLabel = new Label(title);
            sectionLabel.getStyleClass().add("section");
            // Create a FlowPane to hold the buttons for each operation,
with horizontal and vertical gaps
            FlowPane buttonsPane = new FlowPane();
            buttonsPane.setHgap(5);
            buttonsPane.setVgap(5);

            // For each operation name in the current category list
            for (String op : ops) {
                // Create a button with the operation name as text
                Button opButton = new Button(op);
                opButton.setPrefWidth(80);

                // Define the button's action: when clicked, append the
operation name followed by "(" to the display
                opButton.setOnAction(e -> {
                    display.appendText(op + "(");
                    // Hide the advanced panel after selecting a process
                    hideAdvancedPanel();
                });
                opButton.getStyleClass().add("adv-button");
                buttonsPane.getChildren().add(opButton);
            }
            advancedPanel.getChildren().addAll(sectionLabel, buttonsPane);
        }
    }

    // This method handles user input from calculator buttons
    private void handleInput(String input) {
        if (input.equals("=")) {
            try {
                // Get the expression from the display
                String expression = display.getText();

                // Check if the expression contains boolean logic
operations
                boolean conBoolfunc
                        = expression.contains("palindrome(")
                        || expression.contains("armstrong")
                        || expression.contains("prime");
```

```
                // Evaluate the expression
                String result = String.valueOf(eval(display.getText()));
                lastAnswer = Double.parseDouble(result);  // Store result
for future use
                display.setText(result);

                // If it's a boolean logic function, display true or false
                if (conBoolfunc && (result.equals("1.0") ||
result.equals("1"))) {
                    display.setText("True");
                } else if (conBoolfunc && (result.equals("0.0") ||
result.equals("0"))) {
                    display.setText("false");
                } else {
                    display.setText(result);
                }

            } catch (Exception ex) {
                // Show error on exception
                display.setText("Error:");
            }
        } else if (input.equals("C")) {
            // Clear the display
            display.clear();
        } else if (input.equals("Ans")) {
            // Insert the last calculated result
            display.appendText(Double.toString(lastAnswer));

        } else if (input.equals("±")) {
            try {
                // Negate the current number
                double value =
Double.parseDouble(display.getText().trim());
                display.setText(String.valueOf(-value));
            } catch (Exception ex) {
                display.setText("");
            }
        } else if (input.equals("%")) {
            // Convert to percentage
            try {
                double value =
Double.parseDouble(display.getText().trim());
                display.setText(String.valueOf(value / 100));
            } catch (Exception ex) {
                display.setText("");
            }
        } else if (input.equals("^")) {
            // Add exponentiation symbol
            display.appendText("^");
        } else if (input.equals("√")) {
            // Add exponentiation sqrt
            display.appendText("sqrt(");
            // Insert square root function
        } else if (input.equals("log")) {
            // Add exponentiation log
            display.appendText("log(");
        } else if (input.equals("ln")) {
            // Add exponentiation ln
            display.appendText("ln(");
        } else if (input.equals("exp")) {
```

```
            // Add exponentiation exp
            display.appendText("exp(");
        } else if (input.equals("⌫")) {
            // Remove the last character
            String text = display.getText();
            if (!text.isEmpty()) {
                display.setText(text.substring(0, text.length() - 1));
            }
        } else if (input.equals(",")) {
            // Add comma for multi-value functions
            display.appendText(",");
        } else if (input.equals("Exit")) {
            // Exit the application
            Platform.exit();
        } else {
            // Default: append input to the display
            display.appendText(input);
        }
    }

    // ================== Expression Evaluation Functions
==================
    // Preprocess the expression to convert percentages into valid
mathematical expressions
    private String PreprocessExpression(String expr) {
        return expr.replaceAll("(\\d+(\\.\\d+)?)%", "($1/100)");
    }

    //advanced expressions buttons on the side panel
    // The main evaluation function for mathematical expressions
    // It applies all supported functions and parses the final expression
manually
    private double eval(String expr) {
        expr = PreprocessExpression(expr);

        expr = handleInverseTrigFunctions(expr);

        // Apply unary functions like sin, cos, log, etc.
        expr = applyFunc(expr, "sin", x -> Math.sin(Math.toRadians(x)));
        expr = applyFunc(expr, "cos", x -> Math.cos(Math.toRadians(x)));
        expr = applyFunc(expr, "tan", x -> Math.tan(Math.toRadians(x)));
        expr = applyFunc(expr, "sqrt", Math::sqrt);
        expr = applyFunc(expr, "log", Math::log10);
        expr = applyFunc(expr, "ln", Math::log);
        expr = applyFunc(expr, "exp", Math::exp);
        expr = applyFunc(expr, "sqrt", Math::sqrt);
        expr = applyFunc(expr, "cbrt", Math::cbrt);
        expr = applyFunc(expr, "square", x -> Math.pow(x, 2));
        expr = applyFunc(expr, "cube", x -> Math.pow(x, 3));
        expr = applyFunc(expr, "round", x -> (double) Math.round(x));
        expr = applyFunc(expr, "ceil", Math::ceil);
        expr = applyFunc(expr, "floor", Math::floor);

        // Apply multi-value functions like avg, min, max, etc
        expr = applyMultiFunc(expr, "avg", this::average);
        expr = applyMultiFunc(expr, "min", this::min);
        expr = applyMultiFunc(expr, "max", this::max);
        expr = applyMultiFunc(expr, "GCD", this::GCD);
        expr = applyMultiFunc(expr, "LCM", this::LCM);

        // Apply boolean functions returning true/false
```

```
        expr = applyBooleanFunc(expr, "palindrome", this::isPalindrome);
        expr = applyBooleanFunc(expr, "armstrong", this::isArmstrong);
        expr = applyBooleanFunc(expr, "prime", this::isPrime);

        // Begin manual parsing of mathematical expression
        String finalExpr = expr;

        return new Object() {
            int pos = -1, ch; // pos = current position, ch = current
character
            String expr = finalExpr;

            // Advance to next character
            void nextChar() {
                ch = (++pos < expr.length()) ? expr.charAt(pos) : -1;
            }

            // Consume current character if it matches expected one
            boolean eat(int charToEat) {
                while (ch == ' ') {
                    nextChar();
                }
                if (ch == charToEat) {
                    nextChar();
                    return true;
                }
                return false;
            }

            // Entry point for parsing
            double parse() {
                nextChar();
                double x = parseExpression();
                if (pos < expr.length()) {
                    throw new RuntimeException("Unexpected: " + (char) ch);
                }
                return x;
            }

            // Parse expressions with '+' and '-'
            double parseExpression() {
                double x = parseTerm();
                while (true) {
                    if (eat('+')) {
                        x += parseTerm();
                    } else if (eat('-')) {
                        x -= parseTerm();
                    } else {
                        return x;
                    }
                }
            }

            // Parse terms with '*' and '/'
            double parseTerm() {
                double x = parseFactor();
                while (true) {
                    if (eat('*')) {
                        x *= parseFactor();
                    } else if (eat('/')) {
                        x /= parseFactor();
```

```
                    } else {
                        return x;
                    }
                }
            }

            // Parse numbers, parenthesis, unary signs, and power '^'
            double parseFactor() {
                //Parse numbers
                if (eat('+')) {
                    return parseFactor();
                }
                if (eat('-')) {
                    return -parseFactor();
                }
                double x;

                // Parse parenthesis
                int startPos = pos;
                if (eat('(')) {
                    x = parseExpression();
                    eat(')');

                    // Parse unary signs .
                } else if ((ch >= '0' && ch <= '9') || ch == '.') {
                    while ((ch >= '0' && ch <= '9') || ch == '.') {
                        nextChar();
                    }
                    x = Double.parseDouble(expr.substring(startPos, pos));
                } else {
                    throw new RuntimeException("Unexpected: " + (char) ch);
                }

                // Handle exponentiation
                if (eat('^')) {
                    double exponent = parseFactor();
                    x = Math.pow(x, exponent);
                }
                return x;
            }

        }.parse();
    }


    // ================== Helper functions to apply math functions in
expression ===================
    // Unary functions (take one argument only)
    // Function to apply single-argument functions such as sin, cos, sqrt..
    private String applyFunc(String expr, String name, DoubleUnaryOperator
op) {
        while (expr.contains(name + "(")) {

            // Extract the part inside the function parentheses
            int start = expr.indexOf(name + "(") + name.length() + 1;
            int end = expr.indexOf(")", start);

            // Exit if closing parenthesis is not found
            if (end == -1) {
                break;
            }
```

```
            String innerExpr = expr.substring(start, end);// Expression
inside the parentheses
            double val;
            try {
                val = eval(innerExpr.trim());// Evaluate the inner
expression to get a number
            } catch (Exception ex) {
                return "Error"; // If evaluation fails
            }
            double res;
            try {
                res = op.applyAsDouble(val);// Apply the mathematical
function

                // Round very small results to zero
                if (Math.abs(res) < 1e-10) {
                    res = 0;
                }
                String full = name + "(" + innerExpr + ")";
                expr = expr.replace(full, Double.toString(res));
            } catch (Exception ex) {
                return "Error";
            }
        }
        return expr;
    }

    // Multi-value functions (like avg, min)
    // Function to apply multi-argument functions (comma-separated) such as
avg, min, max
    private String applyMultiFunc(String expr, String name,
Function<String, Double> op) {
        while (expr.contains(name + "(")) {
            int start = expr.indexOf(name + "(") + name.length() + 1;
            int end = expr.indexOf(")", start);
            if (end == -1) {
                break;
            }
            String args = expr.substring(start, end);// Extract the
arguments
            double res = op.apply(args);// Apply the function to the
arguments
            String full = name + "(" + args + ")";
            expr = expr.replace(full, Double.toString(res));
        }
        return expr;
    }

    // Boolean functions (return true/false like prime or palindrome)
    // Function to apply boolean-returning functions like prime,
palindrome, armstrong
    private String applyBooleanFunc(String expr, String name,
Function<Double, Boolean> func) {
        while (expr.contains(name + "(")) {
            int start = expr.indexOf(name + "(") + name.length() + 1;
            int end = expr.indexOf(")", start);
            if (end == -1) {
                break;
            }
            String inner = expr.substring(start, end).trim();
            double value;
```

```
            try {
                value = Double.parseDouble(inner);// Convert input to a
number
            } catch (Exception e) {
                return "Error";
            }
            boolean result;
            try {
                result = func.apply(value);
            } catch (Exception e) {
                return "Error";
            }
            String full = name + "(" + inner + ")";

            expr = expr.replace(full, result ? "1" : "0");
        }
        return expr;
    }



    //
    // ================== Handle Inverse Trigonometric Functions
================================
    //

    //* This method processes all inverse trigonometric functions (asin,
acos, atan)
    //* by detecting them in the expression, evaluating their arguments,
and replacing
    //* them with their calculated degree result.
    private String handleInverseTrigFunctions(String expr) {
        expr = applySafeFunc(expr, "asin", x ->
Math.toDegrees(Math.asin(x)));
        expr = applySafeFunc(expr, "acos", x ->
Math.toDegrees(Math.acos(x)));
        expr = applySafeFunc(expr, "atan", x ->
Math.toDegrees(Math.atan(x)));
        return expr;
    }


    //* A safer version of applyFunc that supports nested expressions
inside a function.
    //* It finds the matching closing parenthesis, evaluates the inner
content,
    //* applies the function, and replaces it in the original expression.
    private String applySafeFunc(String expr, String name,
DoubleUnaryOperator op) {
        while (expr.contains(name + "(")) {
            int startIndex = expr.indexOf(name + "(");
            int openParen = startIndex + name.length() + 1;
            int closeParen = findMatchingParenthesis(expr, openParen - 1);
            if (closeParen == -1) {
                break;
            }
            String innerExpr = expr.substring(openParen, closeParen);
            double val;
            try {
                val = eval(innerExpr.trim());
            } catch (Exception ex) {
```

```
                throw new RuntimeException("Invalid " + name + " input: " +
innerExpr);
            }
            double result = op.applyAsDouble(val);
            String full = expr.substring(startIndex, closeParen + 1);
            expr = expr.replace(full, Double.toString(result));
        }
        return expr;
    }

    //* Utility method to find the matching closing parenthesis in an
expression
    //* given the index of an opening parenthesis.
    private int findMatchingParenthesis(String expr, int openIndex) {
        int count = 0;
        for (int i = openIndex; i < expr.length(); i++) {
            if (expr.charAt(i) == '(') {
                count++;
            } else if (expr.charAt(i) == ')') {
                count--;
                if (count == 0) {
                    return i;
                }
            }
        }
        return -1;
    }


    // ================== Scientific Operation Functions
====================
    // Calculate the average of numbers separated by commas
    private double average(String input) {
        return
Arrays.stream(input.split(",")).mapToDouble(Double::parseDouble).average().
orElse(0);
    }

    // Calculate the minimum number
    private double min(String input) {
        return
Arrays.stream(input.split(",")).mapToDouble(Double::parseDouble).min().orEl
se(0);
    }

    // Calculate the maximum number
    private double max(String input) {
        return
Arrays.stream(input.split(",")).mapToDouble(Double::parseDouble).max().orEl
se(0);
    }

    // Calculate the Greatest Common Divisor (GCD)
    private double GCD(String input) {
        String[] parts = input.split(",");
        int a = Integer.parseInt(parts[0].trim());
        int b = Integer.parseInt(parts[1].trim());
        return computeGCD(a, b);
    }

    // Calculate the Least Common Multiple (LCM)
```

```
    private double LCM(String input) {
        String[] parts = input.split(",");
        int a = Integer.parseInt(parts[0].trim());
        int b = Integer.parseInt(parts[1].trim());
        return computeLCM(a, b);
    }

    // Euclidean algorithm for computing GCD
    private int computeGCD(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return Math.abs(a);
    }

    // Compute LCM using GCD
    private int computeLCM(int a, int b) {
        return Math.abs(a * b) / computeGCD(a, b);
    }

    // Check if a number is a Palindrome (reads the same forward and
backward)
    private boolean isPalindrome(double number) {
        String str = String.valueOf((int) number);
        if (str.length() < 3) {

            // Show warning alert if number has less than 3 digits
            Alert alert = new Alert(AlertType.WARNING);
            alert.setTitle("تـنـبـيـه");
            alert.setHeaderText(null);
            alert.setContentText("A number of 3 or more digits must be
entered for the Palindrome operation");
            alert.showAndWait();
            return false;
        }
        return str.equals(new StringBuilder(str).reverse().toString());
    }

    // Check if number is Armstrong number
    private boolean isArmstrong(double number) {
        int n = (int) number;
        int temp = n, sum = 0, digits = String.valueOf(n).length();
        while (temp != 0) {
            int digit = temp % 10;
            sum += Math.pow(digit, digits);// Raise each digit to the power
of number of digits
            temp /= 10;
        }
        return sum == n;
    }

    // Check if number is Prime
    private boolean isPrime(double number) {
        int n = (int) number;
        if (n < 2) {
            return false;
        }
        for (int i = 2; i <= Math.sqrt(n); i++) {
            if (n % i == 0) {
```

```
                return false;
            }
        }
        return true;
    }

    // Entry point of the JavaFX application
    public static void main(String[] args) {
        launch(args);
    }

}
```

Light.css

```
.root {
-fx-background-color: #f2f2f2;
-fx-padding: 20;
-fx-font-family: "Segoe UI", sans-serif;
-fx-border-color: #333;
-fx-border-width: 1;
}
.text-field {
-fx-font-size: 25px;
/*-fx-min-width: 240px;*/
-fx-min-height: 67px;
-fx-background-color: #ffffff;
-fx-border-color: #cccccc;
-fx-border-radius: 6px;
-fx-background-radius: 6px;
-fx-padding: 10;
-fx-text-fill: #333333;
}
.button {
-fx-font-size: 18px;
-fx-min-width: 60px;
-fx-min-height: 60px;
-fx-background-color: linear-gradient(to bottom, #e0e0e0, #cccccc);
-fx-background-radius: 8px;
-fx-border-radius: 8px;
-fx-border-color: #999999;
-fx-cursor: hand;
}
.button:hover {
-fx-background-color: linear-gradient(to bottom, #d0d0d0, #bbbbbb);
}
.button:pressed {
-fx-background-color: linear-gradient(to bottom, #b0b0b0, #999999);
}



.toggleAdvanced{
-fx-background-color: linear-gradient(to bottom, #e0e0e0, #cccccc);
-fx-text-fill: #333;
```

```
-fx-font-size: 16px;
-fx-font-weight: bold;
-fx-background-radius: 8;
-fx-border-color: #0077cc;
-fx-border-width: 1;
-fx-border-radius: 8;
-fx-effect: dropshadow(gaussian, rgba(0, 119, 204, 0.4), 4, 0.3, 0, 1);
-fx-cursor: hand;
-fx-padding: 8 12;
}
.toggleAdvanced:hover {
    -fx-background-color: #0077cc;
-fx-text-fill: white;
}
.advanced {
-fx-background-color: #f2f2f2;
-fx-border-color: #cccccc;
-fx-border-width: 1;
-fx-effect: dropshadow(gaussian, rgba(0,0,0,0.15), 10, 0.3, 0, 0);
}
.calc-button {
-fx-font-size: 16px;
-fx-background-color: #ffffff;
-fx-text-fill: #000000;
-fx-background-radius: 8;
-fx-border-color: #ccc;
}
.calc-button:hover {
-fx-background-color: #e6e6e6;
}
.adv-button {
-fx-font-size: 14px;
-fx-background-color: #dddddd;
-fx-text-fill: #000000;
-fx-background-radius: 6;
}
.adv-button:hover {
-fx-background-color: #cccccc;
}
.section {
-fx-text-fill: #000000;
/*-fx-border-color: #aaa;*/
-fx-font-weight: bold;
-fx-font-size: 14px;
}
.closeBtn {
    -fx-font-size: 14px;
    -fx-background-color: transparent;
    -fx-text-fill: #000000;
}
```

Dark.css

```
.root {
-fx-background-color: #1e1e1e;
-fx-padding: 20;
-fx-font-family: "Segoe UI", sans-serif;
```

```
}
.text-field {
-fx-font-size: 25px;
-fx-min-width: 240px;
-fx-min-height: 60px;
-fx-background-color: #2c2c2c;
-fx-border-color: #444;
-fx-border-radius: 8px;
-fx-background-radius: 8px;
-fx-padding: 10;
-fx-text-fill: #ffffff;
}
.button {
-fx-font-size: 18px;
-fx-min-width: 60px;
-fx-min-height: 60px;
-fx-background-color: linear-gradient(to bottom, #3a3a3a, #2e2e2e);
-fx-background-radius: 8px;
-fx-border-radius: 8px;
-fx-border-color: #555;
-fx-text-fill: #ffffff;
-fx-cursor: hand;
}
.button:hover {
-fx-background-color: linear-gradient(to bottom, #4a4a4a, #3e3e3e);
}
.button:pressed {
-fx-background-color: #2a2a2a;
}
.titled-pane {
-fx-font-size: 16px;
-fx-background-color: #2e2e2e;
-fx-border-color: #444;
-fx-text-fill: #dddddd;
}
.titled-pane:hover {
    -fx-text-fill: #000;
}

.group-pane .title {
-fx-background-color: linear-gradient(to right, #333, #222);
-fx-text-fill: #00ffc6;
-fx-font-size: 15px;
-fx-font-weight: bold;
-fx-padding: 10 14;
-fx-background-radius: 10;
-fx-border-color: #00ffc6;
-fx-border-width: 1;
-fx-effect: dropshadow(gaussian, #00ffc6, 4, 0.3, 0, 1);
-fx-border-radius: 10;
}
.group-pane .title:hover {
-fx-background-color: #00ffc6;
-fx-text-fill: #000;
}
/* المجموعة أزرار */
.group-button {
-fx-background-color: linear-gradient(to bottom, #444, #222);
-fx-text-fill: white;
-fx-font-size: 14px;
-fx-font-weight: bold;
```

```
-fx-background-radius: 8;
-fx-border-color: #00ffc6;
-fx-border-width: 1;
-fx-border-radius: 8;
-fx-effect: dropshadow(gaussian, #00ffc6, 4, 0.3, 0, 1);
-fx-cursor: hand;
-fx-padding: 8 12;
}
.group-button:hover {
-fx-background-color: #00ffc6;
-fx-text-fill: #000;
}
.toggleAdvanced{
-fx-background-color: linear-gradient(to bottom, #444, #222);
-fx-text-fill: white;
-fx-font-size: 16px;
-fx-font-weight: bold;
-fx-background-radius: 8;
-fx-border-color: #00ffc6;
-fx-border-width: 1;
-fx-border-radius: 8;
-fx-effect: dropshadow(gaussian, #00ffc6, 4, 0.3, 0, 1);
-fx-cursor: hand;
-fx-padding: 8 12;
}
.toggleAdvanced:hover {
-fx-background-color: #00ffc6;
-fx-text-fill: #000;
}
.advanced {
    -fx-background-color: #1e1e1e;
    -fx-border-color: #444;
    -fx-border-width: 1;
    -fx-effect: dropshadow(gaussian, rgba(0,0,0,0.5), 10, 0.3, 0, 0);
}
.calc-button {
-fx-font-size: 16px;
-fx-background-color: #444;
-fx-text-fill: white;
-fx-background-radius: 8;
-fx-border-color: #ddd;

}
.calc-button:hover {
-fx-background-color: #555;
}
.adv-button {
-fx-font-size: 14px;
-fx-background-color: #3a3a3a;
-fx-text-fill: white;
-fx-background-radius: 6;
}
.adv-button:hover {
    -fx-background-color: #ffffff55;

}
.section {
    -fx-text-fill: #ffffff;
/*-fx-border-color: #444;*/
-fx-font-weight: bold;
-fx-font-size: 14px;
```

```
}
.closeBtn {
    -fx-font-size: 14px;
    -fx-background-color: transparent;
    -fx-text-fill: #ffffff;
}
```

User Guide:

No technical setup is required.

To get started, go to the “Compiled Code” folder. Inside, you’ll find two files:

- A .jar file (works on all platforms)
- An .exe file (recommended for Windows users)

Simply double-click the **.**exe file if you're on Windows, or double-click the **.**jar file on any system that supports Java. The calculator will open, and you can start using it right away.

Note: To use the .jar file, you must have Java installed on your computer. We recommend downloading and installing the Liberica Full JDK from https://bell-sw.com/pages/downloads/ — it includes JavaFX and works across all major platforms.

To see how the app works, check the Screenshots section of this guide for examples.

Developer Guide:

his section provides a technical overview of the Calculator Application’s internal structure and functionality. It explains how the code is organized, how each component works, and how different modules interact to deliver the calculator’s features. This guide is intended to help developers understand, maintain, or enhance the application with ease.

Module Descriptions:

```
// Advanced Calculator Project using JavaFX
```

```
// Includes basic and scientific operations with a customizable GUI
(Dark/Light mode)
// Enables a side panel for advanced operations
package calculator;

// Importing all necessary packages and libraries for building the JavaFX
user interface and handling events
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.image.Image;
import javafx.scene.layout.StackPane;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import java.util.*;
import java.util.function.*;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.input.MouseEvent;
import javafx.stage.Stage;
import javafx.application.Platform;
import javafx.scene.control.Alert.AlertType;
import javafx.stage.StageStyle;

public class calculator extends Application {

    // Field for displaying input and results
    private TextField display = new TextField();
    // Side panel for advanced operations
    private VBox advancedPanel = new VBox(15);
    private boolean advancedVisible = false; // Panel visibility state
    private ScrollPane scrollPane; // For scrolling inside the panel
    private double lastAnswer = 0; // Stores the last result
    private boolean isDarkMode = false; // Theme state

    @Override
    public void start(Stage primaryStage) {

        display.setEditable(false);
        display.setMinHeight(50);
        display.setStyle("-fx-font-size: 18;");

        // Create basic operation buttons
        GridPane basicButtons = createBasicButtons();

        // Button to toggle the side panel
        Button toggleAdvanced = new Button("Advanced Panel");
        toggleAdvanced.getStyleClass().add("toggleAdvanced");
        toggleAdvanced.setOnAction(e -> toggleAdvancedPanel());

        // Setup advanced operations inside the panel
        createAdvancedButtons();

        //Setup the side panel
        advancedPanel.setPadding(new Insets(10));
        advancedPanel.getStyleClass().add("advanced");
        advancedPanel.setTranslateX(300);
        advancedPanel.setPrefWidth(300);
        //advancedPanel = false;

        //Setup scrollPane the side panel
```

```
        scrollPane = new ScrollPane(advancedPanel);
        scrollPane.setFitToWidth(true);
        scrollPane.setPrefWidth(300);
        scrollPane.setMouseTransparent(false);
        scrollPane.setTranslateX(300);
        scrollPane.setVisible(false);
        scrollPane.setStyle("-fx-background: transparent; -fx-background-
color: transparent;");

        // Container for theme toggle button
        StackPane theme = new StackPane();

        // Main layout of the calculator
        VBox calculatorLayout = new VBox(10, display, basicButtons,
toggleAdvanced, theme);
        calculatorLayout.setAlignment(Pos.CENTER);
        calculatorLayout.setPadding(new Insets(10));
        StackPane root = new StackPane(calculatorLayout, scrollPane);

        Scene scene = new Scene(root);

        // Hide side panel when clicking outside it
        scene.addEventFilter(MouseEvent.MOUSE_PRESSED, event -> {
            if (advancedVisible &&
!advancedPanel.localToScene(advancedPanel.getBoundsInLocal()).contains(even
t.getSceneX(), event.getSceneY())) {
                hideAdvancedPanel();
            }
        });

        // Button to toggle between dark and light mode
        Button toggleTheme = new Button("Toggle Theme >");
        toggleTheme.setStyle("""
                    -fx-font-size: 10;-fx-min-width: 80px;
                    -fx-min-height: 20px;""");

        StackPane.setAlignment(toggleTheme, Pos.TOP_RIGHT);
        theme.getChildren().add(toggleTheme);
```

Module Description: Calculator Application:

Imports

The module imports various JavaFX packages and classes necessary for building the user interface, handling events, and managing application behavior. Key imports include:

- **Application**: The base class for JavaFX applications.
- **Scene**, **Stage**: Classes for managing the application window and its content.

- **Button**, **TextField**, **VBox**, **GridPane**, **ScrollPane**: UI components for user interaction and layout management.
- **Insets**, **Pos**: Classes for managing layout properties and positioning.
- **MouseEvent**: For handling mouse interactions.
- **Platform**: For managing the application lifecycle.
- **Alert**: For displaying alerts to the user.

Fields

- **TextField display**: A non-editable text field that displays user input and results.
- **VBox advancedPanel**: A vertical box layout that serves as the container for advanced operation buttons.
- **boolean advancedVisible**: A flag indicating whether the advanced panel is currently visible.
- **ScrollPane scrollPane**: A scrollable pane that contains the advanced panel, allowing users to scroll through advanced operations.
- **double lastAnswer**: A variable to store the last calculated result for easy access.
- **boolean isDarkMode**: A flag to track the current theme state (dark or light mode).

start Method:

The start method is the entry point of the JavaFX application, where the primary stage and scene are set up. Key functionalities include:

1. **Display Setup**:
    a. The display is configured to be non-editable, with a minimum height and font size.
2. **Basic Operation Buttons**:

   a. A grid of buttons for basic calculator operations is created using the createBasicButtons() method (not shown in the snippet).
3. **Advanced Panel Toggle**:
   a. A button labeled "Advanced Panel" is created to toggle the visibility of the advanced operations panel. The button's action is linked to the toggleAdvancedPanel() method (not shown in the snippet).
4. **Advanced Operations Setup**:
   a. The advanced operations are initialized using the createAdvancedButtons() method (not shown in the snippet).
5. **Advanced Panel Configuration**:
   a. The advancedPanel is styled and positioned, with padding and width settings applied.
6. **Scroll Pane Configuration**:
   a. A ScrollPane is created to allow scrolling within the advanced panel, with specific styling to ensure it blends with the overall design.
7. **Theme Toggle Button**:
   a. A button for toggling between dark and light themes is created and positioned in the top right corner of the application.
8. **Main Layout**:
   a. The main layout of the calculator is constructed using a VBox that contains the display, basic buttons, the advanced panel toggle button, and the theme toggle button. This layout is then added to a StackPane to accommodate the scrollable advanced panel.
9. **Mouse Event Filter**:
   a. An event filter is added to the scene to hide the advanced panel when the user clicks outside of it, enhancing user experience.

```
  // Button to toggle setOnAction...
        toggleTheme.setOnAction(e -> {
            isDarkMode = !isDarkMode;
            scene.getStylesheets().clear();
            if (isDarkMode) {

scene.getStylesheets().add(getClass().getResource("Dark.css").toExternalFor
m());

            } else {

scene.getStylesheets().add(getClass().getResource("light.css").toExternalFo
rm());
            }
        });
        // Load default theme (light)

scene.getStylesheets().add(getClass().getResource("light.css").toExternalFo
rm());

        //Prevent closing the window with the "❌" button
        primaryStage.setOnCloseRequest(evnt -> {
            evnt.consume();
        });

    scene.setOnKeyPressed(event -> {
    String key = event.getText(); // Get the character that was typed

    switch (event.getCode()) {

        case ENTER -> handleInput("=");           // Press Enter to
evaluate
        case BACK_SPACE -> handleInput("⌫");      // Delete last character
        case ESCAPE, DELETE -> handleInput("C");  // Clear the display

        // Handle Shift + 5 to insert %
        case DIGIT5 -> {
            if (event.isShiftDown()) handleInput("%");
            else handleInput("5");
        }

        // Handle Shift + 6 to insert ^
        case DIGIT6 -> {
            if (event.isShiftDown()) handleInput("^");
            else handleInput("6");
        }

        // Handle Shift + 9 to insert (
        case DIGIT9 -> {
            if (event.isShiftDown()) handleInput("(");
            else handleInput("9");
        }

        // Handle Shift + 0 to insert )
        case DIGIT0 -> {
```

```
            if (event.isShiftDown()) handleInput(")");
            else handleInput("0");
        }

        // Digits 1 to 4 and 7 to 8
        case DIGIT1, NUMPAD1 -> handleInput("1");
        case DIGIT2, NUMPAD2 -> handleInput("2");
        case DIGIT3, NUMPAD3 -> handleInput("3");
        case DIGIT4, NUMPAD4 -> handleInput("4");
        case DIGIT7, NUMPAD7 -> handleInput("7");
        case DIGIT8, NUMPAD8 -> handleInput("8");

        // Additional Numpad digit cases not handled above
        case NUMPAD0 -> handleInput("0");
        case NUMPAD5 -> handleInput("5");
        case NUMPAD6 -> handleInput("6");
        case NUMPAD9 -> handleInput("9");

        // Handle math operators from main keyboard or numpad
        case ADD, PLUS -> handleInput("+");
        case SUBTRACT, MINUS -> handleInput("-");
        case MULTIPLY -> handleInput("*");
        case DIVIDE, SLASH -> handleInput("/");

        default -> {
            // Allow typing dot, comma, or math operators
            if (".,*/+-".contains(key)) {
                handleInput(key);
            }

            // Use key 'm' or 'n' to toggle plus/minus
            if (key.equalsIgnoreCase("m") || key.equalsIgnoreCase("n")) {
                handleInput("±");
            }
        }
    }
});

      scene.getRoot().requestFocus();


        //primaryStage.initStyle(StageStyle.UNDECORATED);
        primaryStage.setTitle("JavaFX Calculator");
        primaryStage.setScene(scene);
        primaryStage.setHeight(700);
        primaryStage.getIcons().add(new
Image(String.valueOf(getClass().getResource("icon.png"))));
        primaryStage.setResizable(false);
        primaryStage.show();
    }

    // Toggle side panel visibility
    private void toggleAdvancedPanel() {
        if (advancedVisible) {
            hideAdvancedPanel();
        } else {
            showAdvancedPanel();
        }
    }

    //Show the side panel
```

```
    private void showAdvancedPanel() {
        advancedPanel.setTranslateX(0);
        advancedVisible = true;
        scrollPane.setTranslateX(0);
        scrollPane.setVisible(true);
        display.clear();
    }

    //Hide the side panel
    private void hideAdvancedPanel() {
        scrollPane.setVisible(false);
        advancedVisible = false;
    }

    // Create basic operation buttons (6x5 grid)
    private GridPane createBasicButtons() {
        GridPane grid = new GridPane();
        grid.setHgap(5);
        grid.setVgap(5);
        grid.setAlignment(Pos.CENTER);
        String[][] buttons = {
            {"Exit", "exp", "log", "Ans", "ln"},
            {"^", "%", "(", ")", "□"},
            {"±", "7", "8", "9", "/"},
            {"√", "4", "5", "6", "*"},
            {",", "1", "2", "3", "-"},
            {".", "0", "=", "+", "C"},};

        //Repetition create basic operation buttons (6x5 grid)
        for (int row = 0; row < buttons.length; row++) {
            for (int col = 0; col < buttons[row].length; col++) {
                String symbol = buttons[row][col];
                Button btn = new Button(symbol);
                btn.setMinSize(50, 50);
                btn.setOnAction(e -> handleInput(symbol));
                grid.add(btn, col, row);
            }
        }
        return grid;
    }

    // ================== advanced operation on the side panel
(expression.. ) ====================
    // Create advanced operation buttons on the side panel
    private void createAdvancedButtons() {
        advancedPanel.getChildren().clear();

        //Button hide the side panel
        Button closeBtn = new Button("❌");
        closeBtn.getStyleClass().add("closeBtn");
        closeBtn.setOnAction(e -> hideAdvancedPanel());
        HBox topBar = new HBox(closeBtn);
        topBar.setAlignment(Pos.TOP_LEFT);
        advancedPanel.getChildren().add(topBar);

        // Organize advanced functions into categories
        Map<String, List<String>> operations = new LinkedHashMap<>();
        operations.put("Trigonometric", Arrays.asList("sin", "cos", "tan",
"asin", "acos", "atan"));
        operations.put("Root Operations", Arrays.asList("sqrt", "cbrt"));
```

```
        operations.put("Exponential Power", Arrays.asList("square",
"cube"));
        operations.put("Approximation", Arrays.asList("round", "ceil",
"floor"));
        operations.put("Statistical", Arrays.asList("avg", "min", "max"));
        operations.put("Validation", Arrays.asList("prime", "palindrome",
"armstrong"));
        operations.put("Number Theory", Arrays.asList("GCD", "LCM"));
```

Module Descriptions: Theme Management and Advanced Operations:

Theme Toggle Button Action:

- The **toggleTheme** button is configured to switch between dark and light themes when clicked.
- The **isDarkMode** boolean flag is toggled to track the current theme state.
- The current stylesheet is cleared and replaced with either **Dark.css** or **light.css** based on the theme state.

Default Theme Loading:

- The application initially loads the light theme by default when the scene is created.

Preventing Window Closure:

- The application prevents the window from closing when the user attempts to click the close button (the "X" button). This can be useful for applications that require confirmation before exiting.

Handling Key Press Events:

- The application processes key press events to manage user input in a calculator-like interface. It allows users to enter numbers, mathematical

operators, and special characters through both the main keyboard and the numpad.

- Specific keys are mapped to actions, such as evaluating input with the "Enter" key, deleting the last character with "Backspace," and clearing the display with "Escape" or "Delete."
- The application also supports special inputs, such as inserting percentage ("%") and exponentiation ("^") symbols when the Shift key is held down.
- Additionally, it allows toggling between positive and negative values using the "m" or "n" keys, inserting the plus/minus symbol ("±").
- After setting up the key press handler, the application ensures that the input field is focused, allowing immediate user interaction.

Stage Configuration:

- The primary stage is configured with a title, scene, height, icon, and a non-resizable property. This ensures a consistent user experience.

Advanced Operations Panel:

1. **Toggling Advanced Panel Visibility:**

- The **toggleAdvancedPanel()** method checks the visibility state of the advanced panel and either shows or hides it accordingly.

Showing the Advanced Panel:

- The **showAdvancedPanel** method makes the advanced panel visible by resetting its translation and updating the visibility state. It also clears the display for a fresh start.

Hiding the Advanced Panel:

- The **hideAdvancedPanel** method hides the advanced panel and updates the visibility state.

Creating Basic Operation Buttons:

- The **createBasicButtons** method constructs a grid of buttons for basic calculator operations. Each button is assigned an action to handle user input.

Creating Advanced Operation Buttons:

- The **createAdvancedButtons** method populates the advanced panel with buttons for various advanced mathematical operations. It organizes these operations into categories such as trigonometric functions, root operations, and statistical functions

Organizing Advanced Functions:

- Advanced operations are organized into a **Map** where each category (e.g., "Trigonometric", "Root Operations") is associated with a list of corresponding functions. This structure allows for easy expansion and management of advanced features.

```
      // For each entry in the 'operations' map, which holds a category
title and a list of operation names
        for (Map.Entry<String, List<String>> entry : operations.entrySet())
{

            // Get the category title (e.g., "Trigonometric")
            String title = entry.getKey();

            // Get the list of operations in this category (e.g., ["sin",
"cos", "tan"])
```

```
            List<String> ops = entry.getValue();
            // Create a label for the category title and apply a CSS style
class "section"
            Label sectionLabel = new Label(title);
            sectionLabel.getStyleClass().add("section");
            // Create a FlowPane to hold the buttons for each operation,
with horizontal and vertical gaps
            FlowPane buttonsPane = new FlowPane();
            buttonsPane.setHgap(5);
            buttonsPane.setVgap(5);

            // For each operation name in the current category list
            for (String op : ops) {
                // Create a button with the operation name as text
                Button opButton = new Button(op);
                opButton.setPrefWidth(80);

                // Define the button's action: when clicked, append the
operation name followed by "(" to the display
                opButton.setOnAction(e -> {
                    display.appendText(op + "(");
                    // Hide the advanced panel after selecting a process
                    hideAdvancedPanel();
                });
                opButton.getStyleClass().add("adv-button");
                buttonsPane.getChildren().add(opButton);
            }
            advancedPanel.getChildren().addAll(sectionLabel, buttonsPane);
        }
    }

    // This method handles user input from calculator buttons
    private void handleInput(String input) {
        if (input.equals("=")) {
            try {
                // Get the expression from the display
                String expression = display.getText();

                // Check if the expression contains boolean logic
operations
                boolean conBoolfunc
                        = expression.contains("palindrome(")
                        || expression.contains("armstrong")
                        || expression.contains("prime");

                // Evaluate the expression
                String result = String.valueOf(eval(display.getText()));
                lastAnswer = Double.parseDouble(result);  // Store result
for future use
                display.setText(result);

                // If it's a boolean logic function, display true or false
                if (conBoolfunc && (result.equals("1.0") ||
result.equals("1"))) {
                    display.setText("True");
                } else if (conBoolfunc && (result.equals("0.0") ||
result.equals("0"))) {
                    display.setText("false");
                } else {
                    display.setText(result);
                }
```

```
            } catch (Exception ex) {
                // Show error on exception
                display.setText("Error:");
            }
        } else if (input.equals("C")) {
            // Clear the display
            display.clear();
        } else if (input.equals("Ans")) {
            // Insert the last calculated result
            display.appendText(Double.toString(lastAnswer));

        } else if (input.equals("±")) {
            try {
                // Negate the current number
                double value =
Double.parseDouble(display.getText().trim());
                display.setText(String.valueOf(-value));
            } catch (Exception ex) {
                display.setText("");
            }
        } else if (input.equals("%")) {
            // Convert to percentage
            try {
                double value =
Double.parseDouble(display.getText().trim());
                display.setText(String.valueOf(value / 100));
            } catch (Exception ex) {
                display.setText("");
            }
        } else if (input.equals("^")) {
            // Add exponentiation symbol
            display.appendText("^");
        } else if (input.equals("√")) {
            // Add exponentiation sqrt
            display.appendText("sqrt(");
            // Insert square root function
        } else if (input.equals("log")) {
            // Add exponentiation log
            display.appendText("log(");
        } else if (input.equals("ln")) {
            // Add exponentiation ln
            display.appendText("ln(");
        } else if (input.equals("exp")) {
            // Add exponentiation exp
            display.appendText("exp(");
        } else if (input.equals("⌫")) {
            // Remove the last character
            String text = display.getText();
            if (!text.isEmpty()) {
                display.setText(text.substring(0, text.length() - 1));
            }
        } else if (input.equals(",")) {
            // Add comma for multi-value functions
            display.appendText(",");
        } else if (input.equals("Exit")) {
            // Exit the application
            Platform.exit();
        } else {
            // Default: append input to the display
            display.appendText(input);
```

```
        }
    }
```

Module Description: Advanced Operations and Input Handling:

Advanced Operations Display

1. **Dynamic Creation of Advanced Operation Buttons**:
    - The module iterates through a map that categorizes advanced operations into different sections (e.g., "Trigonometric", "Root Operations"). For each category, it creates a label to represent the category title and a **FlowPane** to hold the corresponding operation buttons.
    - Each operation button is created with the operation name as its text. When clicked, the button appends the operation name followed by an opening parenthesis to the display, allowing users to easily input complex expressions. After selecting an operation, the advanced panel is hidden to streamline the user interface.
2. **Styling and Layout**:
    - The buttons are styled with a specific CSS class to ensure a consistent look and feel. The **FlowPane** is configured with horizontal and vertical gaps to provide adequate spacing between buttons, enhancing usability.

User Input Handling

1. **Input Processing**:
    - The module includes a method to handle user input from the calculator buttons. It processes various types of input, including numerical values, operations, and special commands.

2. **Evaluation of Expressions**:
    - When the user presses the "=" button, the expression displayed is evaluated. The module checks for specific boolean logic functions (e.g., "palindrome", "armstrong", "prime") and evaluates the expression accordingly. The result is displayed, and if the expression corresponds to a boolean function, it shows "True" or "False" based on the evaluation.
3. **Clearing the Display**:
    - The "C" button allows users to clear the display, resetting it for new input.
4. **Using Last Answer**:
    - The "Ans" button enables users to insert the last calculated result into the display, facilitating quick calculations.
5. **Negation and Percentage Conversion**:
    - The module provides functionality to negate the current number and convert it to a percentage. It handles potential exceptions that may arise from invalid input.
6. **Special Operations**:
    - The module supports various mathematical operations, such as exponentiation, square root, logarithm, and natural logarithm. Each operation appends the appropriate function to the display, allowing users to construct complex expressions.
7. **Backspace Functionality**:
    - The "Delete" button allows users to remove the last character from the display, providing a way to correct mistakes.
8. **Exiting the Application**:
    - The "Exit" button enables users to close the application gracefully.

9. **Default Behavior**:

    - For any other input, the module appends the input directly to the display, allowing users to build their expressions freely.

```
    // ================== Expression Evaluation Functions
====================
```

```
    // Preprocess the expression to convert percentages into valid
mathematical expressions
    private String PreprocessExpression(String expr) {
        return expr.replaceAll("(\\d+(\\.\\d+)?)%", "($1/100)");
    }

    //advanced expressions buttons on the side panel
    // The main evaluation function for mathematical expressions
    // It applies all supported functions and parses the final expression
manually
    private double eval(String expr) {
        expr = PreprocessExpression(expr);

        expr = handleInverseTrigFunctions(expr);

        // Apply unary functions like sin, cos, log, etc.
        expr = applyFunc(expr, "sin", x -> Math.sin(Math.toRadians(x)));
        expr = applyFunc(expr, "cos", x -> Math.cos(Math.toRadians(x)));
        expr = applyFunc(expr, "tan", x -> Math.tan(Math.toRadians(x)));
        expr = applyFunc(expr, "sqrt", Math::sqrt);
        expr = applyFunc(expr, "log", Math::log10);
        expr = applyFunc(expr, "ln", Math::log);
        expr = applyFunc(expr, "exp", Math::exp);
        expr = applyFunc(expr, "sqrt", Math::sqrt);
        expr = applyFunc(expr, "cbrt", Math::cbrt);
        expr = applyFunc(expr, "square", x -> Math.pow(x, 2));
        expr = applyFunc(expr, "cube", x -> Math.pow(x, 3));
        expr = applyFunc(expr, "round", x -> (double) Math.round(x));
        expr = applyFunc(expr, "ceil", Math::ceil);
        expr = applyFunc(expr, "floor", Math::floor);

        // Apply multi-value functions like avg, min, max, etc
        expr = applyMultiFunc(expr, "avg", this::average);
        expr = applyMultiFunc(expr, "min", this::min);
        expr = applyMultiFunc(expr, "max", this::max);
        expr = applyMultiFunc(expr, "GCD", this::GCD);
        expr = applyMultiFunc(expr, "LCM", this::LCM);

        // Apply boolean functions returning true/false
        expr = applyBooleanFunc(expr, "palindrome", this::isPalindrome);
        expr = applyBooleanFunc(expr, "armstrong", this::isArmstrong);
        expr = applyBooleanFunc(expr, "prime", this::isPrime);

        // Begin manual parsing of mathematical expression
        String finalExpr = expr;

        return new Object() {
            int pos = -1, ch; // pos = current position, ch = current
character
            String expr = finalExpr;

            // Advance to next character
            void nextChar() {
                ch = (++pos < expr.length()) ? expr.charAt(pos) : -1;
            }

            // Consume current character if it matches expected one
            boolean eat(int charToEat) {
                while (ch == ' ') {
                    nextChar();
                }
```

```
            if (ch == charToEat) {
                nextChar();
                return true;
            }
            return false;
        }

        // Entry point for parsing
        double parse() {
            nextChar();
            double x = parseExpression();
            if (pos < expr.length()) {
                throw new RuntimeException("Unexpected: " + (char) ch);
            }
            return x;
        }
```

Module Description: Expression Evaluation Functions:

Expression Preprocessing

1. **Preprocessing Percentages:**
    - The **PreprocessExpression** method converts percentage values in the expression into valid mathematical expressions. It replaces occurrences of numbers followed by a percentage sign with their equivalent fraction (e.g., converting "50%" to "(50/100)"). This ensures that percentage calculations are handled correctly during evaluation.

Main Evaluation Function

2. **Expression Evaluation:**
    - The **eval** method serves as the core function for evaluating mathematical expressions. It first preprocesses the expression to handle percentages and then applies various mathematical functions to the expression.
3. **Handling Inverse Trigonometric Functions:**
    - The method includes a call to **handleInverseTrigFunctions**, which likely processes any inverse trigonometric functions present in the

expression (though the implementation details are not provided in the snippet).

4. **Applying Unary Functions:**
    - The method applies a series of unary mathematical functions such as sine, cosine, tangent, square root, logarithm (base 10), natural logarithm, exponential, cube root, squaring, cubing, rounding, ceiling, and floor functions. Each function is applied using the **applyFunc** method, which takes the expression and a lambda function that defines the mathematical operation.
5. **Applying Multi-Value Functions:**
    - The module also supports multi-value functions like average, minimum, maximum, greatest common divisor (GCD), and least common multiple (LCM). These functions are applied using the **applyMultiFunc** method, which processes the expression accordingly.
6. **Applying Boolean Functions:**
    - The evaluation process includes checks for boolean functions such as palindrome, Armstrong number, and prime number. These functions return true or false based on the evaluation of the expression, and they are applied using the **applyBooleanFunc** method.

Manual Parsing of Mathematical Expressions

7. **Manual Parsing Logic:**
    - The module implements a manual parsing mechanism to interpret the mathematical expression. It defines an anonymous inner class that maintains the current position and character being processed in the expression.

- The **nextChar** method advances to the next character in the expression, while the **eat** method consumes the current character if it matches the expected character.

8. **Entry Point for Parsing:**

- The **parse** method serves as the entry point for parsing the expression. It initializes the parsing process and checks for any unexpected characters in the expression after parsing is complete.

```
        // Parse expressions with '+' and '-'
            double parseExpression() {
```

```
            double x = parseTerm();
            while (true) {
                if (eat('+')) {
                    x += parseTerm();
                } else if (eat('-')) {
                    x -= parseTerm();
                } else {
                    return x;
                }
            }
        }

        // Parse terms with '*' and '/'
        double parseTerm() {
            double x = parseFactor();
            while (true) {
                if (eat('*')) {
                    x *= parseFactor();
                } else if (eat('/')) {
                    x /= parseFactor();
                } else {
                    return x;
                }
            }
        }

        // Parse numbers, parenthesis, unary signs, and power '^'
        double parseFactor() {
            //Parse numbers
            if (eat('+')) {
                return parseFactor();
            }
            if (eat('-')) {
                return -parseFactor();
            }
            double x;

            // Parse parenthesis
            int startPos = pos;
            if (eat('(')) {
                x = parseExpression();
                eat(')');

                // Parse unary signs .
            } else if ((ch >= '0' && ch <= '9') || ch == '.') {
                while ((ch >= '0' && ch <= '9') || ch == '.') {
                    nextChar();
                }
                x = Double.parseDouble(expr.substring(startPos, pos));
            } else {
                throw new RuntimeException("Unexpected: " + (char) ch);
            }

            // Handle exponentiation
            if (eat('^')) {
                double exponent = parseFactor();
                x = Math.pow(x, exponent);
            }
            return x;
        }
```

```
        }.parse();
    }


    // ================== Helper functions to apply math functions in
expression ====================
    // Unary functions (take one argument only)
    // Function to apply single-argument functions such as sin, cos, sqrt..
    private String applyFunc(String expr, String name, DoubleUnaryOperator
op) {
        while (expr.contains(name + "(")) {

            // Extract the part inside the function parentheses
            int start = expr.indexOf(name + "(") + name.length() + 1;
            int end = expr.indexOf(")", start);

            // Exit if closing parenthesis is not found
            if (end == -1) {
                break;
            }
            String innerExpr = expr.substring(start, end);// Expression
inside the parentheses
            double val;
            try {
                val = eval(innerExpr.trim());// Evaluate the inner
expression to get a number
            } catch (Exception ex) {
                return "Error"; // If evaluation fails
            }
            double res;
            try {
                res = op.applyAsDouble(val);// Apply the mathematical
function

                // Round very small results to zero
                if (Math.abs(res) < 1e-10) {
                    res = 0;
                }
                String full = name + "(" + innerExpr + ")";
                expr = expr.replace(full, Double.toString(res));
            } catch (Exception ex) {
                return "Error";
            }
        }
        return expr;
    }
```

Module Description: Expression Parsing and Function Application:

Expression Parsing

1. **Parsing Expressions**:

    - The **parseExpression** method is designed to handle addition and subtraction operations. It starts by parsing a term and then continuously checks for '+' or

'-' operators to combine terms accordingly. This method ensures that expressions are evaluated in the correct order of operations.

2. **Parsing Terms**:
    - The **parseTerm** method focuses on multiplication and division operations. Similar to **parseExpression**, it begins by parsing a factor and then checks for '*' or '/' operators to combine factors. This method ensures that multiplication and division are prioritized over addition and subtraction.
3. **Parsing Factors**:
    - The **parseFactor** method is responsible for parsing individual numbers, parentheses, unary signs, and exponentiation. It handles:
        - Unary signs: It checks for '+' or '-' to determine the sign of the number.
        - Parentheses: If an opening parenthesis is encountered, it recursively calls **parseExpression** to evaluate the expression within the parentheses.
        - Numbers: It parses numeric values, including decimals, and converts them to **double**.
        - Exponentiation: If the '^' operator is found, it calculates the power of the base using the **Math.pow** function.
4. **Error Handling**:
    - Throughout the parsing methods, exceptions are thrown for unexpected characters or malformed expressions, ensuring that the user is informed of any errors in their input.

Function Application

5. **Applying Unary Functions**:
    - The **applyFunc** method is responsible for applying unary mathematical functions (e.g., sine, cosine, square root) to expressions. It searches for occurrences of a specified function in the expression and evaluates the inner expression contained within the parentheses.
    - The method extracts the inner expression, evaluates it, and applies the specified mathematical function using a **DoubleUnaryOperator**. If the evaluation is successful, the result replaces the original function call in the expression.

6. **Error Management**:
    - If any errors occur during the evaluation of the inner expression or the application of the function, the method returns "Error" to indicate that the evaluation failed. This ensures that the user receives feedback on invalid operations.
7. **Rounding Small Results**:
    - The method includes logic to round very small results (close to zero) to exactly zero, which helps in maintaining numerical stability and accuracy in calculations.

```
// Multi-value functions (like avg, min)
```

```
    // Function to apply multi-argument functions (comma-separated) such as
avg, min, max
    private String applyMultiFunc(String expr, String name,
Function<String, Double> op) {
        while (expr.contains(name + "(")) {
            int start = expr.indexOf(name + "(") + name.length() + 1;
            int end = expr.indexOf(")", start);
            if (end == -1) {
                break;
            }
            String args = expr.substring(start, end);// Extract the
arguments
            double res = op.apply(args);// Apply the function to the
arguments
            String full = name + "(" + args + ")";
            expr = expr.replace(full, Double.toString(res));
        }
        return expr;
    }

    // Boolean functions (return true/false like prime or palindrome)
    // Function to apply boolean-returning functions like prime,
palindrome, armstrong
    private String applyBooleanFunc(String expr, String name,
Function<Double, Boolean> func) {
        while (expr.contains(name + "(")) {
            int start = expr.indexOf(name + "(") + name.length() + 1;
            int end = expr.indexOf(")", start);
            if (end == -1) {
                break;
            }
            String inner = expr.substring(start, end).trim();
            double value;
            try {
                value = Double.parseDouble(inner);// Convert input to a
number
            } catch (Exception e) {
                return "Error";
            }
            boolean result;
            try {
                result = func.apply(value);
            } catch (Exception e) {
                return "Error";
            }
            String full = name + "(" + inner + ")";

            expr = expr.replace(full, result ? "1" : "0");
        }
        return expr;
    }



    //
    // ================== Handle Inverse Trigonometric Functions
================================
    //

    //* This method processes all inverse trigonometric functions (asin,
acos, atan)
```

```
    //* by detecting them in the expression, evaluating their arguments,
and replacing
    //* them with their calculated degree result.
    private String handleInverseTrigFunctions(String expr) {
        expr = applySafeFunc(expr, "asin", x ->
Math.toDegrees(Math.asin(x)));
        expr = applySafeFunc(expr, "acos", x ->
Math.toDegrees(Math.acos(x)));
        expr = applySafeFunc(expr, "atan", x ->
Math.toDegrees(Math.atan(x)));
        return expr;
    }


    //* A safer version of applyFunc that supports nested expressions
inside a function.
    //* It finds the matching closing parenthesis, evaluates the inner
content,
    //* applies the function, and replaces it in the original expression.
    private String applySafeFunc(String expr, String name,
DoubleUnaryOperator op) {
        while (expr.contains(name + "(")) {
            int startIndex = expr.indexOf(name + "(");
            int openParen = startIndex + name.length() + 1;
            int closeParen = findMatchingParenthesis(expr, openParen - 1);
            if (closeParen == -1) {
                break;
            }
            String innerExpr = expr.substring(openParen, closeParen);
            double val;
            try {
                val = eval(innerExpr.trim());
            } catch (Exception ex) {
                throw new RuntimeException("Invalid " + name + " input: " +
innerExpr);
            }
            double result = op.applyAsDouble(val);
            String full = expr.substring(startIndex, closeParen + 1);
            expr = expr.replace(full, Double.toString(result));
        }
        return expr;
    }

    //* Utility method to find the matching closing parenthesis in an
expression
    //* given the index of an opening parenthesis.
    private int findMatchingParenthesis(String expr, int openIndex) {
        int count = 0;
        for (int i = openIndex; i < expr.length(); i++) {
            if (expr.charAt(i) == '(') {
                count++;
            } else if (expr.charAt(i) == ')') {
                count--;
                if (count == 0) {
                    return i;
                }
            }
        }
        return -1;
    }
```

Module Description: Multi-Value and Boolean Functions, Inverse Trigonometric Functions:

Multi-Value Functions

1. **Applying Multi-Argument Functions:**

    - The **applyMultiFunc** method is designed to handle functions that take multiple arguments, such as average (avg), minimum (min), and maximum (max). It searches for occurrences of the specified function in the expression and extracts the arguments contained within the parentheses.
    - The method applies the specified function to the extracted arguments using a provided **Function<String, Double>.** The result replaces the original function call in the expression, allowing for seamless integration of multi-value calculations.

Boolean Functions

2. **Applying Boolean-Returning Functions:**

    - The **applyBooleanFunc** method processes boolean functions that return true or false, such as checking for prime numbers, palindromes, or Armstrong numbers. Similar to multi-value functions, it searches for the specified function in the expression and extracts the inner argument.
    - The method attempts to convert the argument to a double and applies the boolean function using a provided **Function<Double, Boolean>.** The result (true or false) is then replaced in the expression with "1" for true and "0" for false, enabling logical evaluations within mathematical expressions.

Inverse Trigonometric Functions

3. **Handling Inverse Trigonometric Functions:**

    - The **handleInverseTrigFunctions** method processes inverse trigonometric functions such as arcsine (asin), arccosine (acos), and arctangent (atan). It detects these functions in the expression, evaluates their arguments, and replaces them with their calculated degree results.
    - The method utilizes the **applySafeFunc** method to ensure that nested expressions within the function calls are handled correctly.

4. **Safe Function Application:**

- The **applySafeFunc** method is a safer version of the applyFunc method, designed to support nested expressions. It finds the matching closing parenthesis for a function call, evaluates the inner content, applies the specified function, and replaces the original function call in the expression with the result.
- This method enhances the robustness of function evaluations by ensuring that complex expressions are processed accurately.

5. **Finding Matching Parentheses:**

- The **findMatchingParenthesis** utility method is used to locate the matching closing parenthesis for a given opening parenthesis in the expression. It counts the number of opening and closing parentheses to ensure that nested structures are correctly identified. If a matching parenthesis is found, its index is returned; otherwise, -1 is returned, indicating an error in the expression**.**

```
    // ================== Scientific Operation Functions
==================
    // Calculate the average of numbers separated by commas
    private double average(String input) {
        return
Arrays.stream(input.split(",")).mapToDouble(Double::parseDouble).average().
orElse(0);
    }

    // Calculate the minimum number
    private double min(String input) {
        return
Arrays.stream(input.split(",")).mapToDouble(Double::parseDouble).min().orEl
se(0);
    }

    // Calculate the maximum number
    private double max(String input) {
        return
Arrays.stream(input.split(",")).mapToDouble(Double::parseDouble).max().orEl
se(0);
    }

    // Calculate the Greatest Common Divisor (GCD)
    private double GCD(String input) {
        String[] parts = input.split(",");
        int a = Integer.parseInt(parts[0].trim());
        int b = Integer.parseInt(parts[1].trim());
        return computeGCD(a, b);
    }

    // Calculate the Least Common Multiple (LCM)
    private double LCM(String input) {
        String[] parts = input.split(",");
        int a = Integer.parseInt(parts[0].trim());
        int b = Integer.parseInt(parts[1].trim());
        return computeLCM(a, b);
    }

    // Euclidean algorithm for computing GCD
    private int computeGCD(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return Math.abs(a);
    }

    // Compute LCM using GCD
    private int computeLCM(int a, int b) {
        return Math.abs(a * b) / computeGCD(a, b);
    }

    // Check if a number is a Palindrome (reads the same forward and
backward)
    private boolean isPalindrome(double number) {
        String str = String.valueOf((int) number);
        if (str.length() < 3) {

            // Show warning alert if number has less than 3 digits
```

```
            Alert alert = new Alert(AlertType.WARNING);
            alert.setTitle("Warning");
            alert.setHeaderText(null);
            alert.setContentText("A number of 3 or more digits must be
entered for the Palindrome operation");
            alert.showAndWait();
            return false;
        }
        return str.equals(new StringBuilder(str).reverse().toString());
    }

    // Check if number is Armstrong number
    private boolean isArmstrong(double number) {
        int n = (int) number;
        int temp = n, sum = 0, digits = String.valueOf(n).length();
        while (temp != 0) {
            int digit = temp % 10;
            sum += Math.pow(digit, digits);// Raise each digit to the power
of number of digits
            temp /= 10;
        }
        return sum == n;
    }

    // Check if number is Prime
    private boolean isPrime(double number) {
        int n = (int) number;
        if (n < 2) {
            return false;
        }
        for (int i = 2; i <= Math.sqrt(n); i++) {
            if (n % i == 0) {
                return false;
            }
        }
        return true;
    }

    // Entry point of the JavaFX application
    public static void main(String[] args) {
        launch(args);
    }

}
```

Module Description: Scientific Operation Functions:

Statistical and Mathematical Functions

1. Calculating the Average:

- The **average** method computes the average of a list of numbers provided as a comma-separated string. It splits the input string, converts each part to a double, and calculates the average using **Java Streams.** If no numbers are provided, it returns 0.

2. **Finding the Minimum:**
    - The **min** method determines the minimum value from a list of numbers given in a comma-separated format. Similar to the average function, it utilizes **Java Streams** to parse the input and find the minimum value.
3. **Finding the Maximum:**
    - The **max** method identifies the maximum value from a list of numbers provided as a comma-separated string. It follows the same approach as the min method to parse the input and compute the maximum.
4. **Calculating the Greatest Common Divisor (GCD):**
    - The **GCD** method calculates the GCD of two integers provided in a comma-separated format. It splits the input, parses the integers, and calls the **computeGCD** method to perform the calculation.
5. **Calculating the Least Common Multiple (LCM):**
    - The **LCM** method computes the LCM of two integers, similar to the GCD method. It splits the input, parses the integers, and calls the **computeLCM** method to obtain the result.
6. **Euclidean Algorithm for GCD:**
    - The **computeGCD** method implements the **Euclidean algorithm** to find the GCD of two integers. It repeatedly applies the modulus operation until one of the numbers becomes zero, returning the absolute value of the other number as the GCD.
7. **Computing LCM Using GCD:**

- The **computeLCM** method calculates the LCM of two integers using the relationship between GCD and LCM: .

Numerical Property Checks

8. **Checking for Palindromes:**
    - The **isPalindrome** method checks if a given number reads the same forwards and backwards. It converts the number to a string and compares it with its reverse. If the number has fewer than three digits, it displays a warning alert to the user.
9. **Checking for Armstrong Numbers:**
    - The **isArmstrong** method determines if a number is an Armstrong number. It calculates the sum of its digits raised to the power of the number of digits and checks if this sum equals the original number.
10. **Checking for Prime Numbers:**
    - The **isPrime** method checks if a number is prime. It verifies that the number is greater than 1 and checks for divisibility by all integers up to the square root of the number. If any divisor is found, the number is not prime.

Application Entry Point

1. **Main Method:**
    - The **main** method serves as the entry point for the JavaFX application. It calls the **launch** method to start the application.