Simple Calculator using JavaFX

Tanishq Padwal 664972389

Calculator GUI

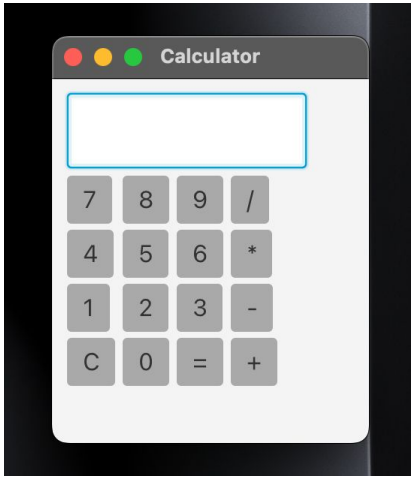
GUI Structure and Initialization

Application Window Setup

- Title Configuration: The application window is titled "Calculator" to indicate its function. This is achieved through the code primaryStage.setTitle("Calculator").
- Scene Size: A scene with dimensions of 250x250 pixels provides a compact and efficient interface for the calculator. This is specified by new Scene(grid, 250, 250).

Display Field Configuration

- TextField as Display Area: A TextField named display is utilized to show both the input expressions and the calculation results to the user.
- Non-Editable: To ensure that the display area can only be modified through the calculator buttons and not through keyboard input, display.setEditable(false) is used.
- Preferred Height: The display area is given a preferred height of 50 pixels (display.setPrefHeight(50)), ensuring that it is prominently visible and can comfortably show the input and results.



Efficient Layout with GridPane

GridPane Overview

- What Is GridPane? A layout pane in JavaFX for arranging UI elements in a grid format.
- Purpose: Ensures a clean, structured arrangement of buttons and display fields.

GridPane Features in Our Calculator

- Spacing: Utilizes setHgap(5) and setVgap(5) for consistent gaps between elements.
- Padding: Employs setPadding(new Insets(10)) for edge spacing, enhancing layout aesthetics.
- Component Placement: Positions elements precisely with grid.add(Node, columnIndex, rowIndex), optimizing the user interface.

Benefits for Calculator GUI

- Flexible Design: Allows easy modifications and additions to the UI.
- User-Friendly Interface: Offers a logically organized layout, improving usability and interaction.

```
GridPane grid = new GridPane();
grid.setPadding(new Insets( v: 10));
grid.setHgap(5);
grid.setVgap(5);

Button[] buttons = new Button[16];
String[] labels = {"7", "8", "9", "/", "4", "5", "6", "*", "1", "2", "3", "-", "C", "0", "=", "+"};
for (int i = 0; i < 16; i++) {
    Button button = new Button(labels[i]);
    button.setOnAction(event -> handleButtonAction(button.getText()));
    button.setStyle("-fx-background-color: #D3D3D3; -fx-font-size: 16px;");
    buttons[i] = button;
}

grid.add(display, i: 0, i1: 0, i2: 4, i3: 1);

int buttonIndex = 0;
for (int row = 1; row < 5; row++) {
    for (int col = 0; col < 4; col++) {
        buttons[buttonIndex].setStyle("-fx-background-color: #A9A9A9; -fx-font-size: 16px;");
        grid.add(buttons[buttonIndex], col, row);
        buttonIndex++;
    }
}
```

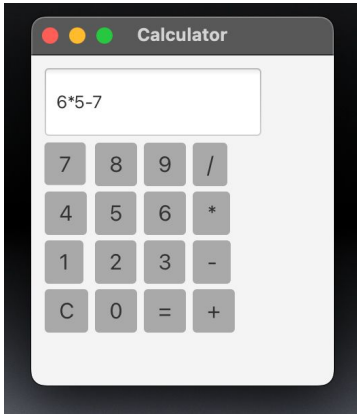
Event Handling in Calculator GUI

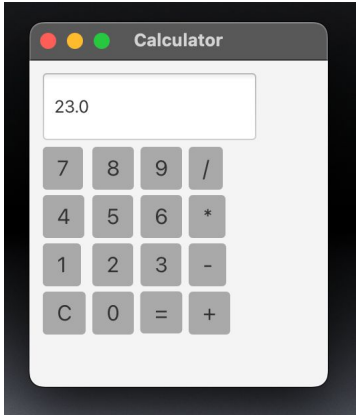
Responding to User Input

- Event Handlers: Each button is equipped with an event handler using button.setOnAction(event -> handleButtonAction(button.getText())). This mechanism ensures the calculator reacts dynamically to clicks, processing each button press accordingly.

Function Execution

- Action Processing: The handleButtonAction(String text) method deciphers the button's text (number, operator, or function) and updates the display or calculates results.
- Numbers and Operators: Appends digits or operators (+, -, *, /) to the current expression in the display.
- Equals (=): Computes the expression's result and displays it.
- Clear (C): Clears the display for a new calculation.





Styling and Appearance

Customizing Button and Scene Styling

- CSS Styling for Buttons: Each button in our calculator is styled for visual appeal and user interaction feedback. Using -fx-background-color: #A9A9A9; -fx-font-size: 16px;, we set a uniform background color and font size, making the buttons distinct and easily readable.
- Scene Background Color: To ensure a cohesive and pleasant user interface, the scene's background is set to a light gray (Color.LIGHTGRAY) with scene.setFill(Color.LIGHTGRAY);. This choice of background color minimizes strain on the eyes and enhances element visibility.

Impact on User Experience

- Consistency and Accessibility: Uniform styling across buttons ensures a consistent look and feel, aiding in user navigation and accessibility.
- Aesthetic Appeal: The combination of carefully chosen colors and styles enhances the overall aesthetic appeal of the calculator, making it more engaging and enjoyable to use.

```
Button[] buttons = new Button[16];
String[] labels = {"7", "8", "9", "/", "4", "5", "6", "*", "1", "2", "3", "-", "C", "0", "=", "+"};
for (int i = 0; i < 16; i++) {
    Button button = new Button(labels[i]);
    button.setOnAction(event -> handleButtonAction(button.getText()));
    button.setStyle("-fx-background-color: #D3D3D3; -fx-font-size: 16px;");
    buttons[i] = button;
}

grid.add(display, i: 0, i1: 0, i2: 4, i3: 1);

int buttonIndex = 0;
for (int row = 1; row < 5; row++) {
    for (int col = 0; col < 4; col++) {
        buttons[buttonIndex].setStyle("-fx-background-color: #A9A9A9; -fx-font-size: 16px;");
        grid.add(buttons[buttonIndex], col, row);
        buttonIndex++;
    }
}

Scene scene = new Scene(grid, v: 250, v1: 250);
scene.setFill(Color.LIGHTGRAY);
primaryStage.setScene(scene);
primaryStage.show();
```

Stack Operations

Handling Numerical Values and the operandStack:

Numerical Detection & Collection:

- Detecting Numbers: Identifies when a digit appears, signaling the start of a numerical value (either an integer or a floating-point number).
- Building Numbers: Uses a StringBuilder to gather all connected digits (and any decimal point) to form the complete number as a string.

Conversion & Stacking:

- Converting to Double: Converts the assembled string representation of the number into a double data type.
- Stacking Operands: Pushes the converted double value onto the operandStack, where it's stored for later calculation use.

```
} else if (Character.isDigit(ch)) {

    StringBuilder num = new StringBuilder();
    while (i < expression.length() && (Character.isDigit(expression.charAt(i)) || expression.charAt(i) == '.')) {
        num.append(expression.charAt(i++));
    }
    operandStack.push(Double.parseDouble(num.toString()));
    i--;
```

Handling Operators and the operatorStack:

Operator Handling & Precedence:

- Detecting Operators: Identifies when an arithmetic operator (+, -, *, /) is encountered in the expression.
- Deciding Action Based on Precedence: Uses a precedence rule to decide whether to apply an operation immediately or to store the operator for later.

Applying Operations:

- Evaluating Stack for Precedence: Checks if any operator in the operatorStack has equal or higher precedence than the current one.

Operation Execution: If conditions are met, it performs the following steps:

- Removes the last two numbers from the operandStack.
- Removes the top operator from the operatorStack.
- Applies the arithmetic operation.
- Stores the result back in the operandStack.

```
} else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
    // If the character is an operator, perform operations based on operator precedence
    while (!operatorStack.isEmpty() && precedence(operatorStack.peek()) >= precedence(ch)) {
        applyOperation();
    }
    operatorStack.push(ch);
```

Stacking the Operator:

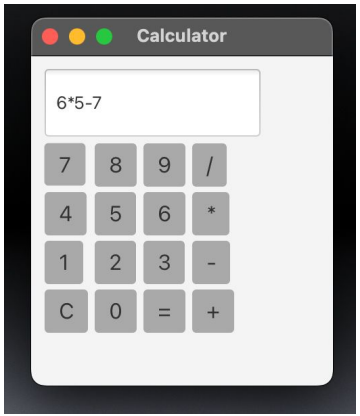
- Storing Operators for Later: If the current operator does not trigger immediate action, or after an operation is applied, it is added to the operatorStack for future operations, ensuring that operator precedence rules are observed throughout the expression evaluation.

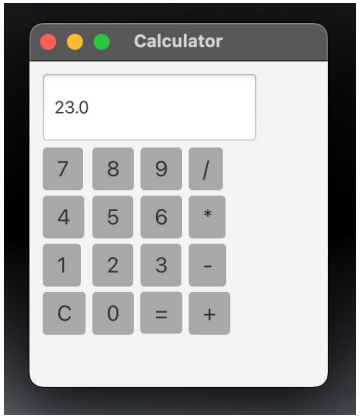
How the Stacks Work Together:

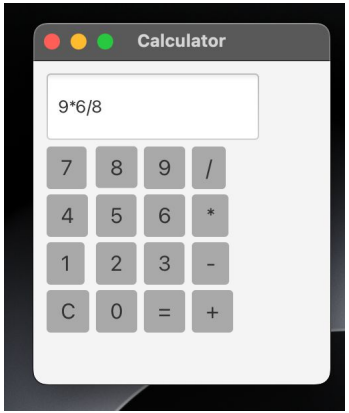
operandStack holds the values. As the expression is parsed, numbers are converted to double and stored here.

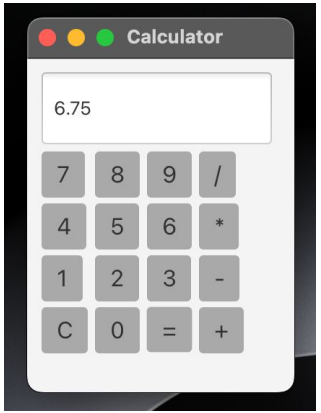
operatorStack holds the operators. It's used to manage the order in which operations are applied, based on the precedence of each operator.

Output


Calculator
6*5-7
7 8 9 /
4 5 6 *
1 2 3 -
C 0 = +


Calculator
23.0
7 8 9 /
4 5 6 *
1 2 3 -
C 0 = +


Calculator
9*6/8
7 8 9 /
4 5 6 *
1 2 3 -
C 0 = +


Calculator
6.75
7 8 9 /
4 5 6 *
1 2 3 -
C 0 = +