University of Sheffield

COM2008 : Systems Design and Security

Tutorial 01 - Calculator Application with Java Swing

Department of Computer Science

Contents

Chapter 1

Introduction

1.1 Learning Objectives

1. Understand the purpose of Swing in Java for building graphical user interfaces (GUIs).
2. Familiarize yourself with common layout managers in Swing, such as BorderLayout, FlowLayout, GridLayout, GridBagLayout, BoxLayout, CardLayout, GroupLayout, and SpringLayout.
3. Learn the characteristics and use cases of different Swing buttons, including JButton, JToggleButton, JCheckBox, JRadioButton, JMenuItem, JToolBar, JSlider, JSpinner, and JButtonGroup.
4. Explore how to add action listeners to Swing components to capture and respond to user interactions, such as button clicks and menu selections.
5. Learn how to create a Swing-based calculator application in Java.
6. Understand the steps involved in setting up a Swing GUI, including creating the main class, initializing components, and defining the layout.
7. Explore how to declare and initialize Swing components like buttons and text fields.
8. Gain insights into the use of layout managers to organize components effectively within a GUI.

Chapter 2

A Swing Refresher

A JFrame is an instance of the javax.swing.JFrame class. It serves as the top-level container for a Swing GUI application, providing the main window frame in which you can organize and display your GUI components. You can set various properties of the JFrame, such as its title, size, layout manager, default close operation, and more.

Every JFrame has a ContentPane container, which is where you typically add the GUI components of your application. The content pane is an essential part of the JFrame and is responsible for managing the layout of its child components. You can access the content pane of a JFrame using the getContentPane() method. When you add components directly to the JFrame, they are actually added to the content pane.

A JPanel is another Swing container class, used to group and organize related components within a JFrame. You can add one or more JPanels to a JFrame to divide the GUI into logical sections or to provide a structured layout for your components. To add a JPanel to a JFrame, you typically create an instance of JPanel, customize it with the desired components (buttons, labels, etc.), and then add the JPanel to the JFrame by invoking the add() method on the content pane.

2.1 Swing Layouts

Java Swing is a framework for building graphical user interfaces (GUIs) in Java applications. Layout managers in Java Swing are responsible for arranging and positioning components (such as buttons, labels, and text fields) within containers (such as frames and panels) to create a visually appealing and functional user interface. Here's a brief overview of some commonly used Java Swing layout managers:

1. BorderLayout:
    - Divides the container into five regions: North, South, East, West, and Center.
    - Components are added to these regions, and they expand to fill the available space.

2. FlowLayout:
    - Arranges components in a row from left to right until the row is full, then moves to the next row.
    - Useful for creating simple, flow-based layouts.
3. GridLayout:
    - Divides the container into a grid of rows and columns.
    - Components are added to specific grid cells.
    - All cells have the same size.
4. GridBagLayout:
    - Provides fine-grained control over component placement.
    - Components are added with constraints that specify their position and behavior in the grid.
5. BoxLayout:
    - Arranges components in a single row or column.
    - Useful for creating linear layouts with components of varying sizes.
6. CardLayout:
    - Manages a stack of components, displaying only one at a time.
    - Useful for creating wizard-style interfaces or tabbed panes.
7. GroupLayout:
    - Introduced in Java SE 6, GroupLayout is a versatile and powerful layout manager.
    - It supports both horizontal and vertical grouping of components.
    - Offers a more intuitive way to define complex layouts.
8. SpringLayout:
    - Provides a flexible and constraint-based layout system.
    - Components are attached to springs, which define their positioning and behavior.

Layout managers help maintain platform independence and ensure that your GUIs adapt to different screen sizes and resolutions. By choosing the appropriate layout manager for your Java Swing application, you can create responsive and visually appealing user interfaces.

2.2 Swing Buttons

Java Swing provides several types of buttons that can be used to create interactive user interfaces. Buttons are essential components for triggering actions or events in a Java Swing application. Here's a brief overview of some commonly used Java Swing buttons:

1. JButton:
    - The most basic type of button in Swing.
    - Represents a simple push button.
    - Typically used to trigger actions when clicked.
2. JToggleButton:
    - Extends JButton.
    - Toggles between two states: selected and deselected.
    - Useful for implementing options that can be turned on and off.
3. JCheckBox:
    - Represents a checkbox with a label.
    - Used for enabling or disabling options or choices.
    - Can be checked (selected) or unchecked (deselected).
4. JRadioButton:
    - Represents a radio button.
    - Used in groups where only one option can be selected at a time.
    - Typically, you create multiple radio buttons and group them together using a ButtonGroup.
5. JMenuItem:
    - Represents a menu item in a menu bar or popup menu.
    - Used for triggering actions in menu-driven applications.
    - Can have icons, labels, and keyboard shortcuts.
6. JToolBar:
    - Represents a toolbar containing various types of components, including buttons.
    - Commonly used for providing quick access to frequently used actions.
7. JSlider:
    - Represents a slider control that allows users to select a value within a specified range.

- Useful for adjusting settings or values.

8. JSpinner:
   - Provides a way to select a value from a set of predefined values or a range.
   - Typically used in conjunction with a SpinnerModel to define the allowable values.
9. JButtonGroup:
   - Not a button itself, but a way to group related radio buttons.
   - Ensures that only one radio button within the group can be selected at a time.

Buttons in Java Swing can be customized with various properties, such as text, icons, tooltips, and event listeners to respond to user interactions. They play a crucial role in creating interactive user interfaces and enabling users to perform actions in a graphical application.

2.3 Swing Action Listeners

In Java Swing, action listeners are a fundamental part of event handling. They are used to capture and respond to user actions, such as clicking a button or selecting a menu item, within a graphical user interface (GUI) application. Here's a brief overview of Java Swing action listeners:

2.3.1 What is an Action Listener

An action listener is an interface provided by Swing for monitoring and responding to user actions, specifically those related to components like buttons, menu items, and other interactive elements.

2.3.2 How It Works

To use an action listener, you typically implement the ActionListener interface in a class and override its actionPerformed(ActionEvent e) method. This method is automatically called when the associated component, like a button, generates an action event (e.g., a button click).

2.3.3 Adding an Action Listener

You can associate an action listener with a Swing component by using the addActionListener() method of the component. For example, to make a button respond to clicks, you would call button.addActionListener(myActionListener).

2.3.4 Handling Events

Inside the actionPerformed(ActionEvent e) method of your action listener class, you write the code that responds to the event triggered by the user. This is where you define what should happen when the associated component's action event occurs.

2.3.5 Action Command

Components like buttons can have an associated action command (a string) that helps identify which component triggered the event. You can use e.getActionCommand() within the actionPerformed() method to differentiate between multiple components using the same action listener.

2.3.6 Common Use Cases

Action listeners are commonly used to perform actions like opening a new window, processing user input, saving data, or navigating between different parts of a GUI application. They enable you to make your Swing application interactive and responsive to user actions.

Chapter 3

Creating the Calculator App

3.1 Creating the GUI

3.1.1 Step 1 - Creating the CalculatorWindow Class

Start by creating a new project using your preferred IDE. Create two files -

1. Main.java - This class is used to start the application and launch a new instance of the CalculatorWindow class.
2. CalculatorWindow.java - This class contains the code for the calculator.

3.1.2 Step 2 - Running the Application

Once the CaculatorWindow class has been created, you can create a new instance of the CalculatorWindow class from the Main class and make the window visible using the following code.

```
1   import javax.swing.*;
2
3   public class Main {
4       public static void main(String[] args) {
5           final String text = "Calculator";
6           SwingUtilities.invokeLater(new Runnable() {
7               @Override
8               public void run() {
9                   final CalculatorWindow window = new CalculatorWindow(text);
10                  window.setVisible(true);
11              }
12          });
13      }
14  }
```

Figure 3.1: Code Snippet of the Main Class

This Java code snippet is the main entry point for a Swing-based calculator application. It begins by defining the text to be displayed in the calculator window, which is set as "Calculator." Inside the main method, it uses SwingUtilities.invokeLater() to ensure that Swing components are initialized and updated on the event dispatch thread (EDT), which is the thread responsible for managing Swing GUI interactions. Within the run() method of an anonymous Runnable object, it creates an instance of the CalculatorWindow class (presumably representing the calculator's GUI window) and sets its visibility to true, making the window visible to the user. This approach ensures proper GUI thread synchronization and is a common practice when working with Swing applications to prevent potential threading issues.

3.1.3 Step 3 - Creating the Components

First the following packages should be imported. The `javax.swing.*;` statement includes the entire javax.swing package, which is a crucial part of Java's Swing Windowwork for creating graphical user interfaces (GUIs). It provides classes and components for creating GUI elements like windows, buttons, labels, and text fields. Importing this package allows you to use these classes and components in your code without specifying the full package name, making it easier to develop interactive and visually appealing applications.

The `import java.awt.event.ActionEvent;` and `import java.awt.event.ActionListener;` statements bring in specific classes from the java.awt.event package. ActionEvent is a class representing actions triggered by user interactions with GUI components, such as button clicks or menu selections. ActionListener is an interface that defines how to handle these action events. By importing these classes, you can implement event-driven behavior in your applications, specifying what should occur when users interact with your GUI elements, making your software interactive and responsive.

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5  import java.util.Stack;
6
```

Figure 3.2: Code snippet of the Imported Packages

The following snippet declares four private instance variables within the CalculatorWindow class. outputTextField is a text field used for displaying user input and calculation results, operandStack is a stack for storing numeric operands entered by the user, currentOperator represents the current arithmetic operator chosen for calculations, and newInput is a boolean flag indicating whether the user is currently entering a new numeric input or appending digits to an existing one. These variables collectively play a crucial role in managing the internal state of the calculator, facilitating user interactions, and enabling the execution of arithmetic operations with proper context and data storage.

```
7   public class CalculatorWindow extends JFrame {
8
9       private JTextField outputTextField;
10      private Stack<Double> operandStack;
11      private Operator currentOperator;
12      private boolean newInput;
```

Figure 3.3: Code snippet of the instance variables

The following code snippet defines an enumeration called Operator in Java, which serves as a custom data type with a finite set of distinct values: ADD, SUBTRACT, MULTIPLY, and DIVIDE. These values represent the four fundamental arithmetic operations. Enums like this are particularly useful for enhancing code readability and maintainability by providing meaningful, self-explanatory labels for commonly used constants. In this case, the Operator enum simplifies the representation of mathematical operators in programs, ensuring clarity and reducing the risk of errors when dealing with operations such as addition, subtraction, multiplication, and division.

```
14      enum Operator {
15          ADD, SUBTRACT, MULTIPLY, DIVIDE
16      }
```

Figure 3.4: Code snippet of the Operator Enum

The following code snippet is from the constructor of a Java class called CalculatorWindow, which represents a graphical calculator window. It begins by calling the superclass constructor with a specified text parameter, likely used for setting the window's title. Inside the constructor, several key components are initialized. The operandStack is created as an empty stack to hold numeric values for calculations, currentOperator is initially set to null to signify no active operator, and newInput is set to true to indicate the start of a new user input. The code then proceeds to create and configure a JTextField named outputTextField, which is used for displaying user input and calculation results. It sets various properties of this text field, such as making it non-editable, specifying its dimensions, font style, text color, and margin. Finally, a JPanel named panel is created and configured with a grid layout of 5 rows and 4 columns, presumably to hold buttons for digits and operators in the calculator's user interface.

```
18      public CalculatorWindow(String text) {
19          super(text);
20          operandStack = new Stack<>();
21          currentOperator = null;
22          newInput = true;
23
24          outputTextField = new JTextField();
25          outputTextField.setEditable(b:false);
26          outputTextField.setMaximumSize(new Dimension(width:225, height:40));
27          outputTextField.setFont(new Font(name:"Monospaced", Font.BOLD, size:20));
28          outputTextField.setDisabledTextColor(new Color(r:0, g:0, b:0));
29          outputTextField.setMargin(new Insets(top:0, left:5, bottom:0, right:0));
30
31          JPanel panel = new JPanel();
32          panel.setLayout(new GridLayout(rows:5, cols:4));
33
```

Figure 3.5: Initial code snippet of the constructor

The following code snippet is part of the constructor of a Java class representing a graphical calculator window. It sets up the calculator's user interface by creating and adding various buttons to a JPanel called panel. Initially, a for loop is used to generate digit buttons labeled from 0 to 9, and these buttons are added to the panel using the addButton method. Following that, operator buttons for addition, subtraction, multiplication, and division are created and added to the panel. Finally, "equals" and "clear" buttons are added. After configuring the buttons, the code arranges the calculator's components within the window's BorderLayout, positioning the outputTextField at the top (North) and placing the button panel in the center. It also specifies that the application should exit when the calculator window is closed (setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE) and packs the window to ensure proper sizing based on its content. This snippet is crucial for setting up the calculator's graphical user interface, making it ready for user interaction and arithmetic calculations.

```
34          // Create and add buttons for digits 0-9
35          for (int i = 0; i <= 9; i++) {
36              addButton(panel, String.valueOf(i));
37          }
38
39          // Create and add operator buttons
40          addButton(panel, label:"+");
41          addButton(panel, label:"-");
42          addButton(panel, label:"*");
43          addButton(panel, label:"/");
44
45          // Create equals and clear buttons
46          addButton(panel, label:"=");
47          addButton(panel, label:"C");
48
49          this.add(outputTextField, BorderLayout.NORTH);
50          this.add(panel, BorderLayout.CENTER);
51          this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
52          this.pack();
53      }
```

Figure 3.6: Remaining code snippet of the constructor

The `addButton` function is a private helper method within a Java class representing a graphical calculator window. Its purpose is to streamline the process of creating, configuring, and adding buttons to the calculator's user interface. When called, it takes two parameters: a JPanel named panel to which the button will be added, and a String named label that determines the text displayed on the button. Inside the function, a new JButton named button is created with the specified label. It then configures this button by setting its font style to "Monospaced," making the text bold, and setting the font size to 22 for a consistent appearance. Furthermore, an ActionListener is attached to the button to handle button click events. This listener calls the handleButtonClick method with the label as an argument, allowing for specific actions to be taken based on the button's label. Finally, the configured button is added to the provided panel, effectively incorporating it into the calculator's graphical user interface. This function simplifies the process of button creation and event handling, making it more efficient and organized within the calculator's code.

```java
private void addButton(JPanel panel, String label) {
    JButton button = new JButton(label);
    button.setFont(new Font(name:"Monospaced", Font.BOLD, size:22));
    button.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            handleButtonClick(label);
        }
    });
    panel.add(button);
}
```

Figure 3.7: Code snippet of the addButton function

The `handleButtonClick` function plays a crucial role in managing user interactions and processing button clicks within a Java class representing a graphical calculator. When called with a String parameter label representing the label of a clicked button, this function handles various scenarios based on the button label.

- If the label matches a digit (0-9), it checks if a new input is starting (newInput is true). If so, it sets the outputTextField to the digit, updates the newInput flag to false, and pushes the digit's numeric value onto the operandStack. If it's not a new input, it appends the digit to the existing input while considering the current number's digits and positioning it correctly in the stack.
- If the label is "C," it clears the outputTextField, empties the operandStack, resets the currentOperator to null, and sets newInput to true to indicate the start of a new input.
- If the label is "=", it checks if the user has entered a valid expression (not a new input and an operator is selected). If so, it performs the calculation using the last two operands and the current operator, updates the outputTextField with the result, pushes the result onto the operandStack, resets the currentOperator to null, and sets newInput to true.
- For all other cases (non-digit, non-"C," non-"=" buttons), it identifies the operator from the label, and if it's not a new input and a previous operator was selected, it evaluates the previous expression (equivalent to pressing "=") and then sets the currentOperator to the newly selected operator and sets newInput to true for the next input.

This function manages the core logic of the calculator's interaction, allowing it to handle digits, operators, clearing, and calculation, ensuring a smooth and accurate user experience.

```java
private void handleButtonClick(String label) {
    if (label.matches(regex:"[0-9]")) {
        if (newInput) {
            outputTextField.setText(label);
            newInput = false;
            operandStack.push(Double.parseDouble(label));
        } else {
            newInput = false;
            outputTextField.setText(outputTextField.getText() + label);
            double currentNumber = operandStack.pop();
            // Handle appending digits to the current number
            operandStack.push((currentNumber * 10) + Double.parseDouble(label));
        }
    } else if (label.equals(anObject:"C")) {
        // Handle clearing the calculator
        outputTextField.setText(t:"");
        operandStack.clear();
        currentOperator = null;
        newInput = true;
    } else if (label.equals(anObject:"=")) {
        if (!newInput && currentOperator != null) {
            double operand2 = operandStack.pop();
            double operand1 = operandStack.pop();
            // Perform the calculation and display the result
            double result = performOperation(operand1, operand2, currentOperator);
            outputTextField.setText(String.valueOf(result));
            operandStack.push(result);
            currentOperator = null;
            newInput = true;
        }
    } else {
        // Handle operator selection
        Operator operator = getOperatorFromLabel(label);
        // If not a new input and a previous operator exists, evaluate the previous
        // expression
        if (!newInput && currentOperator != null) {
            // Evaluate the previous expression
            handleButtonClick(label:"=");
        }
        currentOperator = operator;
        newInput = true;
    }
}
```

Figure 3.8: Code snippet of the handleButtonClick function

The `getOperatorFromLabel` function is a private method within a Java class that's responsible for mapping a string label, typically representing an arithmetic operator (+, -, , /), to an enum value of the Operator enum type. It uses a switch statement to examine the label,

and depending on its value, it returns the corresponding Operator enum value. If the label is ”+” it returns Operator.ADD, if it’s ”-” it returns Operator.SUBTRACT, if it’s ”” it returns Operator.MULTIPLY, and if it’s ”/” it returns Operator.DIVIDE. If the label doesn’t match any of these cases, it throws an IllegalArgumentException with a message indicating that the operator is invalid. This function ensures that the provided operator labels are correctly mapped to the appropriate Operator enum values, enhancing code readability and safety when working with arithmetic operations in the calculator.

```
120     private Operator getOperatorFromLabel(String label) {
121         switch (label) {
122             case "+":
123                 return Operator.ADD;
124             case "-":
125                 return Operator.SUBTRACT;
126             case "*":
127                 return Operator.MULTIPLY;
128             case "/":
129                 return Operator.DIVIDE;
130             default:
131                 throw new IllegalArgumentException("Invalid operator: " + label);
132         }
133     }
```

Figure 3.9: Code snippet of the getOperatorFromLabel function

The `performOperation` function is a private method within a Java class that handles the actual execution of arithmetic operations based on two operands (operand1 and operand2) and the specified arithmetic operator represented by the Operator enum type. Inside the function, a switch statement is used to determine which operation to perform based on the operator. If it’s Operator.ADD, it returns the sum of operand1 and operand2. If it’s Operator.SUBTRACT, it returns the result of subtracting operand2 from operand1. For Operator.MULTIPLY, it returns the product of the two operands. For Operator.DIVIDE, it checks if operand2 is not zero before returning the result of dividing operand1 by operand2. If operand2 is zero, it displays an error message using JOptionPane and returns Double.NaN to represent a ”Not-a-Number” result. If the operator does not match any of these cases, it throws an IllegalArgumentException with an error message indicating that the operator is invalid. This function ensures that arithmetic operations are performed accurately while handling special cases like division by zero and invalid operators.

```java
private double performOperation(double operand1, double operand2, Operator operator) {
    switch (operator) {
        case ADD:
            return operand1 + operand2;
        case SUBTRACT:
            return operand1 - operand2;
        case MULTIPLY:
            return operand1 * operand2;
        case DIVIDE:
            if (operand2 != 0) {
                return operand1 / operand2;
            } else {
                JOptionPane.showMessageDialog(this, message:"Division by zero", title:"Error", JOptionPane.ERROR_MESSAGE);
                return Double.NaN;
            }
        default:
            throw new IllegalArgumentException("Invalid operator: " + operator);
    }
}
```

Figure 3.10: Code snippet of the performOperation function

Chapter 4

Conclusion

In conclusion, this tutorial has provided a comprehensive overview of Swing in Java and guided you through the creation of a Swing-based calculator application. Here are some key takeaways from this tutorial:

1. **Learning Objectives**: This tutorial aimed to help you understand Swing's role in Java for building graphical user interfaces (GUIs) and familiarize you with layout managers, various types of Swing buttons, and action listeners.

2. **Swing Refresher**: You learned about Swing layouts, including BorderLayout, FlowLayout, GridLayout, GridBagLayout, BoxLayout, CardLayout, GroupLayout, and SpringLayout. You also explored different types of Swing buttons and the importance of action listeners in handling user interactions.

3. **Creating the Calculator App**: In the practical part of the tutorial, you created a functional calculator application using Swing. The steps included setting up the GUI, creating and configuring components like buttons and text fields, and implementing logic to handle user inputs, perform calculations, and display results.

This tutorial has equipped you with essential skills for designing and developing interactive GUI applications in Java using Swing. You've learned how to create user-friendly interfaces and respond to user actions effectively. These skills can be applied to a wide range of Java applications that require graphical interfaces.