A
Seminar Report
On
Java Swings
submitted

in the partial fulfillment of requirements

for the award of the Degree of

Bachelor of Technology
in
Computer Science & Engineering

Supervisor:
Seema Arya
Assistant Professor
(Computer Science & Engineering Department)

Submitted By:
Name: Jitendra Verma
21E1MICSM10P027

Department of Computer Science & Engineering

Modi Institute of Technology, Kota

Rajasthan Technical University

September, 2024

MODI INSTITUTE OF TECHNOLOGY, KOTA

Session 2024-2025

Certificate

This is to certify that the seminar entitled **Java Swings** is submitted by Jitendra Verma bearing Enrollment No. 21E1MICSM10P027, in the partial fulfillment of the requirements for the award of degree of Bachelor of Technology in "Computer Science & Engineering", in the academic year 2024-25.

Supervisor
Seema Arya
Assistant Professor

Seema Arya
(Computer Science & Engineering Department)

ACKNOWLEDGEMENT

No volume of words is enough to express my deep gratitude towards my guide **Mrs. Seema Arya**, Assistant Professor, Computer Science & Engineering Department, Modi Institute of Technology, Kota, who has been very concerned and has aided me for all the materials essential for the work and preparation of this report. She helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am also thankful to **Mrs. Seema Arya** (Head & Assistant Professor, CSE Dept. MIT, Kota) for his kind help, continued interest and inspiration during this work.

I also want to express my gratitude to **Dr. Vikas Soni** (Principal, Modi Institute of Technology, Kota) for his kind support and suggestions.

Jitendra Verma

INDEX

CHAPTER - 1

INTRODUCTION TO JAVA SWING

Java Swing is a powerful and versatile toolkit for building graphical user interfaces (GUIs) in Java applications. As part of the Java Foundation Classes (JFC), Swing provides a rich set of components that allow developers to create interactive and visually appealing applications. Unlike its predecessor, the Abstract Window Toolkit (AWT), Swing is lightweight and platform-independent, making it a preferred choice for many desktop applications.

Overview of Java Swing

Swing is built on the Model-View-Controller (MVC) architecture, which separates the application logic (model) from the user interface (view) and the input handling (controller). This separation promotes better organization and maintainability of code. Swing components are rendered in Java rather than relying on native system components, enabling a consistent look and feel across different operating systems.

Key features of Swing include:

- **Rich Set of Components**: Swing provides a wide range of components such as buttons, text fields, tables, and trees, allowing for the creation of complex user interfaces.
- **Customizable Look and Feel**: Developers can customize the appearance of Swing applications by selecting different Look and Feel settings, enabling a consistent or themed design.
- **Lightweight Components**: Swing components are lightweight, which means they do not rely on native GUI components, resulting in better performance and a more uniform experience.
- **Event Handling**: Swing uses an event-driven programming model, allowing applications to respond dynamically to user actions such as mouse clicks and keyboard input.

CHAPTER - 2

IMPORTANCE OF GUI IN JAVA APPLICATIONS

Graphical User Interfaces are crucial for enhancing user experience and interaction in software applications. A well-designed GUI can make applications more accessible and easier to use, allowing users to navigate features intuitively. The importance of GUI in Java applications can be summarized as follows:

- **User Engagement**: GUIs provide visual elements that engage users and encourage interaction, making software more appealing and user-friendly.
- **Accessibility**: A good GUI helps accommodate users with varying levels of technical expertise, making applications usable for a broader audience.
- **Productivity**: Intuitive interfaces streamline tasks, reducing the time required for users to accomplish their goals within the application.
- **Feedback and Interaction**: GUIs can provide immediate feedback to user actions, enhancing the interactive experience and ensuring users are aware of the application's state.

CHAPTER - 3

BRIEF HISTORY AND EVOLUTION

Java Swing was introduced in the late 1990s as part of Java Foundation Classes. Here's a brief timeline of its evolution:

1. **Early Java (1995)**: Java was first released with AWT as its primary GUI toolkit. AWT components are heavyweight, meaning they rely on the native GUI components of the operating system, leading to inconsistencies across platforms.
2. **Introduction of Swing (1998)**: Swing was introduced with Java 1.2 as a part of the Java Foundation Classes. It offered a more flexible and feature-rich alternative to AWT. Swing components are lightweight, allowing for better performance and more customization options.
3. **Integration and Enhancements**: Over the years, Swing has been continuously updated and integrated with Java's overall ecosystem. Features such as the Pluggable Look and Feel (allowing applications to change their appearance dynamically) and the introduction of new components have kept Swing relevant.
4. **Swing Today**: While newer frameworks like JavaFX have emerged, Swing remains a popular choice for many developers, especially for legacy applications and when a straightforward desktop application is needed. Its robustness and extensive documentation ensure that it is still widely used in the industry.

CHAPTER - 4

GETTING STARTED WITH JAVA SWING

To develop applications using Java Swing, you need to set up your development environment properly. This section will guide you through the process, introduce the Java Development Kit (JDK) and popular Integrated Development Environments (IDEs), and show you how to create a simple Swing application.

Setting Up the Java Development Environment

1. **Download and Install the JDK**:
   - Visit the Oracle website or OpenJDK to download the latest version of the Java Development Kit (JDK).
   - Follow the installation instructions for your operating system (Windows, macOS, or Linux).
2. **Set Up Environment Variables (if necessary)**:
   - For Windows, you might need to set the JAVA_HOME environment variable and update the PATH variable to include the JDK's bin directory.
   - On macOS or Linux, ensure that the JDK's bin directory is in your PATH by adding a line like export PATH=$PATH:/path/to/jdk/bin in your shell configuration file (e.g., .bashrc, .zshrc).
3. **Choose an IDE**:
   - IDEs make it easier to write, debug, and manage your code. Here are a few popular choices for Java development:

Introduction to JDK and IDEs

- **Java Development Kit (JDK)**:
  - The JDK is a software development kit that provides the necessary tools for developing Java applications, including the Java Runtime Environment (JRE), the Java compiler (javac), and other tools for managing Java applications.
- **Popular IDEs**:
  - **Eclipse**:
    - ✦ A widely used open-source IDE with a strong ecosystem of plugins.
    - ✦ Great for Java development with features like code completion, debugging, and a visual GUI builder.
    - ✦ Download Eclipse.
  - **IntelliJ IDEA**:
    - ✦ A powerful IDE with intelligent code assistance and a user-friendly interface.
    - ✦ Offers a free Community Edition and a paid Ultimate Edition with additional features.
    - ✦ Download IntelliJ IDEA.
  - **NetBeans**:
    - ✦ Another open-source IDE that provides good support for Java development.
    - ✦ Includes features like a GUI builder and support for various Java technologies.
    - ✦ Download NetBeans.

Creating a Simple Swing Application

Once you have the JDK and your IDE set up, you can create a simple Swing application. Below is a step-by-step guide to create a basic "Hello, World!" application using Swing.

1. **Open your IDE**:
   - Start your chosen IDE and create a new Java project.
2. **Create a new Java Class**:
   - Create a new Java class file named HelloWorldSwing.java.
3. **Write the Swing Code**:
   - In the HelloWorldSwing.java file, write the following code:

java

```
import javax.swing.*;

public class HelloWorldSwing {
  public static void main(String[] args) {
    // Create a JFrame instance
    JFrame frame = new JFrame("Hello, World!");

    // Create a JLabel
    JLabel label = new JLabel("Hello, World!", SwingConstants.CENTER);

    // Add the label to the frame
    frame.add(label);

    // Set the default close operation
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Set the frame size
    frame.setSize(300, 200);

    // Make the frame visible
    frame.setVisible(true);
  }
}
```

4. **Run the Application**:
   - Compile and run the HelloWorldSwing class. You should see a window pop up displaying "Hello, World!".
5. **Explore Further**:
   - Experiment with modifying the text, size, or adding more components like buttons or text fields.

CHAPTER - 5

SWING ARCHITECTURE

Java Swing is designed around a robust architecture that enhances the development of graphical user interfaces (GUIs). Understanding the key components of this architecture—specifically the Model-View-Controller (MVC) pattern, components and containers, and the event dispatching thread—is essential for creating efficient and maintainable Swing applications.

Understanding the MVC (Model-View-Controller) Pattern

The MVC pattern is a design paradigm that separates an application into three interconnected components:

1. **Model**:
   - Represents the data and the business logic of the application.
   - Notifies the view about changes in the data, allowing the view to update accordingly.
   - In Swing, the model can include classes like DefaultListModel or TableModel that manage data for components like lists and tables.
2. **View**:
   - The user interface component that displays the data to the user.
   - In Swing, the view consists of various components (like JButton, JLabel, etc.) that present the data visually.
   - It listens for changes in the model and updates itself when necessary.
3. **Controller**:
   - Acts as an intermediary between the model and the view, handling user input and updating the model.
   - In Swing applications, event listeners often serve as controllers, responding to user actions (like button clicks) and updating the model accordingly.

This separation of concerns enhances code organization and maintainability, making it easier to manage complex applications.

Components and Containers

In Swing, components are the building blocks of the user interface, while containers are special components that hold other components.

1. **Components**:
   - Swing provides a rich set of GUI components that can be used to build applications. Examples include:
     - ✦ **JButton**: A clickable button.
     - ✦ **JLabel**: Displays text or images.
     - ✦ **JTextField**: For single-line text input. ✦ **JTextArea**: For multi-line text input.
     - ✦ **JComboBox**: A dropdown list for selection. 2. **Containers**:
   - Containers are used to group components together and control their layout. Common container types include:
     - ✦ **JFrame**: The main window of the application.
     - ✦ **JPanel**: A flexible container for organizing components.

- **JScrollPane**: Provides a scrollable view for components that may exceed the visible area.
- **JTabbedPane**: Allows for tabbed navigation between multiple components.

Swing uses a hierarchy of containers, allowing for nested layouts. This hierarchy allows for flexible user interfaces that can adapt to various screen sizes and resolutions.

Event Dispatching Thread

Swing is single-threaded, which means that all GUI-related tasks must be executed on a single thread called the Event Dispatch Thread (EDT). Understanding the EDT is crucial for ensuring that your Swing applications remain responsive and do not freeze during long-running tasks.

1. **Threading Model**:
   - The EDT is responsible for managing the event handling in Swing. This includes responding to user inputs (like clicks and key presses) and updating the UI.
   - All updates to the Swing components should occur on the EDT to prevent inconsistencies and ensure thread safety.
2. **Using `SwingUtilities.invokeLater()`**:
   - To safely update the GUI from outside the EDT (such as from background threads), you should use SwingUtilities.invokeLater(). This method schedules the specified Runnable to be executed on the EDT:

java

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        // Update the UI components here
        label.setText("Updated Text");
    }
});
```

3. **Long-Running Tasks**:
   - For tasks that may take a long time (like network calls or intensive computations), use a SwingWorker. This allows you to run the task in a background thread while safely updating the GUI when the task is complete.

CHAPTER - 6

BASIC SWING COMPONENTS

Java Swing provides a rich set of components that form the building blocks of graphical user interfaces (GUIs). Understanding these components is essential for developing interactive and user-friendly applications. Below, we will explore some of the most commonly used Swing components, along with examples of how to create and use them.

JFrame: Creating a Window

`JFrame` is the main window that serves as the primary container for your GUI application. It represents the window in which components are added.

Example:

java

```
import javax.swing.*;

public class SimpleFrame {
  public static void main(String[] args) {
    JFrame frame = new JFrame("My First Window");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(400, 300);
    frame.setVisible(true);
  }
}
```

- **Key Methods**:
  - setDefaultCloseOperation(int): Defines the operation that will happen when the user closes the window.
  - setSize(int width, int height): Sets the dimensions of the window.
  - setVisible(boolean): Displays the window.

JPanel: A Container for Other Components

`JPanel` is a versatile container used to organize and group components within a window. It can also be used for custom layouts and drawing.

Example:

java

```
JPanel panel = new JPanel();
panel.setBackground(Color.LIGHT_GRAY);
frame.add(panel);
```

- Panels can hold other components, making them useful for organizing UI layouts.

JButton: Creating Buttons

`JButton` allows users to perform actions by clicking. It can display text or images.

Example:

java

```
JButton button = new JButton("Click Me!");
button.addActionListener(e -> System.out.println("Button Clicked!"));
panel.add(button);
```

- **Key Methods**:
  - addActionListener(ActionListener): Adds an action listener to respond to clicks.

JLabel: Displaying Text and Images

`JLabel` is used to display static text or images. It can be used for labeling other components.

Example:

java

```
JLabel label = new JLabel("Hello, World!", SwingConstants.CENTER);
panel.add(label);
```

- **Key Properties**:
  - SwingConstants.CENTER: Aligns the text or image within the label.

JTextField: Input Fields for Text

`JTextField` allows users to enter a single line of text input.

Example:

java

```
JTextField textField = new JTextField(20);
panel.add(textField);
```

- **Key Properties**:
  - The integer parameter defines the number of columns (width) of the text field.

JTextArea: Multi-Line Text Input

`JTextArea` is used for inputting and displaying multiple lines of text.

Example:

java

```
JTextArea textArea = new JTextArea(5, 20);
JScrollPane scrollPane = new JScrollPane(textArea);
panel.add(scrollPane);
```

- **Key Properties**:
  - Wraps the text area in a JScrollPane to allow for scrolling.

JCheckBox: Boolean Options

`JCheckBox` provides a way for users to select or deselect options, representing boolean values.

Example:

java

```
JCheckBox checkBox = new JCheckBox("Accept Terms");
panel.add(checkBox);
```

- **Key Methods**:
  - isSelected(): Returns whether the checkbox is selected.

JRadioButton: Selecting One Option from a Group

`JRadioButton` allows users to select one option from a group of choices.

Example:

java

```
JRadioButton radioButton1 = new JRadioButton("Option 1");
JRadioButton radioButton2 = new JRadioButton("Option 2");
ButtonGroup group = new ButtonGroup();
group.add(radioButton1);
group.add(radioButton2);
panel.add(radioButton1);
panel.add(radioButton2);
```

- **Key Methods**:
  - Use ButtonGroup to group radio buttons together, ensuring only one can be selected at a time.

JComboBox: Dropdown Selections

`JComboBox` presents a dropdown list for users to choose from.

Example:

java

```
String[] options = {"Option 1", "Option 2", "Option 3"};
JComboBox<String> comboBox = new JComboBox<>(options);
panel.add(comboBox);
```

- **Key Methods**:
  - getSelectedItem(): Retrieves the currently selected item from the combo box.

JList: Displaying Lists of Items

`JList` displays a list of items for selection. It can be used to show a collection of data.

Example:

java

```
DefaultListModel<String> listModel = new DefaultListModel<>();
listModel.addElement("Item 1");
listModel.addElement("Item 2");
JList<String> list = new JList<>(listModel);
JScrollPane listScrollPane = new JScrollPane(list);
panel.add(listScrollPane);
```

- **Key Methods**:
    - getSelectedValue(): Retrieves the currently selected value from the list.

CHAPTER - 7

LAYOUT MANAGERS

In Swing, layout managers are crucial for organizing and positioning components within containers. They control the size and arrangement of components, ensuring that your GUI remains responsive and visually appealing across different screen sizes and resolutions.

Importance of Layout Managers in Swing

- **Consistent Layouts**: Layout managers help maintain a consistent appearance for your GUI by automatically adjusting the placement and size of components as the window is resized.
- **Ease of Use**: They simplify the process of managing component positions without requiring manual calculations for positioning and sizing.
- **Responsive Design**: Layout managers ensure that applications adapt well to various screen sizes and resolutions, making GUIs more user-friendly.
- **Maintainability**: Using layout managers makes it easier to change the layout of an application without altering the underlying code significantly.

Common Layout Managers

Swing provides several built-in layout managers, each with its unique behavior and use cases:

1. FlowLayout

`FlowLayout` arranges components in a left-to-right flow, wrapping to the next line when there is no more horizontal space.

Example:

java

```
JPanel panel = new JPanel(new FlowLayout());
panel.add(new JButton("Button 1"));
panel.add(new JButton("Button 2"));
panel.add(new JButton("Button 3"));
```

- **Key Features**:
  - Components are added sequentially.
  - You can set alignment (left, center, right).
  - Ideal for toolbars or simple button panels.

2. BorderLayout

`BorderLayout` divides the container into five regions: North, South, East, West, and Center. Each region can contain one component.

Example:

java

```
JPanel panel = new JPanel(new BorderLayout());
```

```
panel.add(new JButton("North"),    BorderLayout.NORTH);
panel.add(new JButton("South"),    BorderLayout.SOUTH);
panel.add(new JButton("Center"), BorderLayout.CENTER);
```

- **Key Features**:
  - Only one component can occupy each region.
  - The Center region expands to fill available space.
  - Useful for creating structured layouts.

3. GridLayout

`GridLayout` arranges components in a grid format with equal-sized cells. You specify the number of rows and columns.

Example:

java

```
JPanel panel = new JPanel(new GridLayout(2, 2)); // 2 rows, 2 columns
panel.add(new JButton("Button 1"));
panel.add(new JButton("Button 2"));
panel.add(new JButton("Button 3"));
panel.add(new JButton("Button 4"));
```

- **Key Features**:
  - All cells are the same size.
  - Components fill the grid in row-major order (left to right, top to bottom).
  - Great for calculator layouts or grid-based interfaces.

4. BoxLayout

`BoxLayout` arranges components either vertically or horizontally, allowing for flexible component alignment and spacing.

Example:

java

```
JPanel panel = new JPanel();
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS)); // Vertical
panel.add(new JButton("Button 1"));
panel.add(Box.createVerticalStrut(10)); // Adds space between components
panel.add(new JButton("Button 2"));
```

- **Key Features**:
  - Allows for custom spacing and alignment.
  - Components can be aligned to the left, right, or center.
  - Useful for creating forms or stacked layouts.

Custom Layouts

While the built-in layout managers cover many use cases, sometimes you may need a specific arrangement that they cannot provide. In such cases, you can create a custom layout manager

by implementing the `LayoutManager` interface or extending one of the existing layout managers.

Steps to Create a Custom Layout:

1. **Implement LayoutManager**:
   - Define methods such as addLayoutComponent(), layoutContainer(), and preferredLayoutSize().
2. **Override Layout Logic**:
   - Implement your own positioning and sizing logic in the layoutContainer() method.

Example:

java

```
import java.awt.*;

public class CustomLayout implements LayoutManager {
   @Override
   public void addLayoutComponent(String name, Component comp) {}

   @Override
   public void removeLayoutComponent(Component comp) {}

   @Override
   public Dimension preferredLayoutSize(Container parent) {
      return new Dimension(400, 200); // Example size
   }

   @Override
   public void layoutContainer(Container parent) {
      // Custom layout logic goes here
   }
}
```

CHAPTER - 8

EVENT HANDLING IN SWING

Event handling is a critical aspect of building interactive applications in Java Swing. It allows your application to respond to user actions such as clicks, key presses, and mouse movements. This section covers the fundamentals of events and listeners, common event listeners, and how to create custom listeners.

Understanding Events and Listeners

1. **Events**:
   - Events are occurrences that happen in the application, usually as a result of user actions (e.g., clicking a button, typing a key).
   - Each event is represented by a specific event object that contains information about the event.
2. **Listeners**:
   - Listeners are interfaces that define methods that are called when a particular event occurs.
   - You register a listener with a component, and when the event occurs, the listener's methods are invoked.
   - In Swing, event handling is typically achieved using the Observer design pattern, where the component acts as the subject that notifies registered listeners.

Common Event Listeners

Here are some of the most commonly used event listeners in Swing:

1. ActionListener

`ActionListener` is used to handle action events, such as button clicks or menu item selections.

Example:

java

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ActionListenerExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("ActionListener Example");
    JButton button = new JButton("Click Me!");

    button.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        System.out.println("Button was clicked!");
      }
    });
```

```
        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

2. MouseListener

`MouseListener` is used to handle mouse events, such as mouse clicks, entry, exit, press, and release.

Example:

java

```
import javax.swing.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class MouseListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("MouseListener Example");
        JLabel label = new JLabel("Hover over or click me!");

        label.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                System.out.println("Label was clicked!");
            }

            @Override
            public void mouseEntered(MouseEvent e) {
                label.setText("Mouse is over me!");
            }

            @Override
            public void mouseExited(MouseEvent e) {
                label.setText("Hover over or click me!");
            }
        });

        frame.add(label);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

3. KeyListener

`KeyListener` is used to handle keyboard events, such as key presses, releases, and typed keys.

Example:

java

```
import javax.swing.*;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;

public class KeyListenerExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("KeyListener Example");
    JTextField textField = new JTextField();

    textField.addKeyListener(new KeyAdapter() {
      @Override
      public void keyPressed(KeyEvent e) {
        System.out.println("Key pressed: " + e.getKeyChar());
      }
    });

    frame.add(textField);
    frame.setSize(300, 200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

Creating Custom Listeners

In some cases, you might want to create a custom listener that listens for specific events that are not covered by the built-in listeners. To do this, you can define your own interface and use it to handle events.

Steps to Create a Custom Listener:

1. **Define the Listener Interface**:
   - Create an interface with methods that represent the events you want to handle.
2. **Implement the Listener**:
   - Implement this interface in the class that will respond to the events.
3. **Register the Listener**:
   - Create a method to register the listener and notify it when the event occurs.

Example:

java

```
// Step 1: Define the custom listener interface
interface CustomListener {
  void onCustomEvent(String message);
}

// Step 2: Create a component that fires events
class CustomComponent {
  private CustomListener listener;

  public void setCustomListener(CustomListener listener) {
    this.listener = listener;
  }

  public void triggerEvent() {
    if (listener != null) {
```

```
            listener.onCustomEvent("Custom event occurred!");
        }
    }
}

// Step 3: Use the custom listener
public class CustomListenerExample {
    public static void main(String[] args) {
        CustomComponent customComponent = new CustomComponent();

        customComponent.setCustomListener(message -> System.out.println(message));

        // Trigger the event
        customComponent.triggerEvent();
    }
}
```

CHAPTER - 9

ADVANCED SWING COMPONENTS

Java Swing provides a variety of advanced components that enhance the functionality and usability of GUI applications. These components allow for better organization, representation of complex data, and integration of graphics and background tasks. Below, we'll explore some of the advanced Swing components and utilities.

JMenuBar: Creating Menus

`JMenuBar` is used to create a menu bar at the top of a window. It can contain multiple menus, each of which can have items and submenus.

Example:

java

```
import javax.swing.*;

public class MenuBarExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Menu Bar Example");
    JMenuBar menuBar = new JMenuBar();

    JMenu fileMenu = new JMenu("File");
    JMenuItem exitItem = new JMenuItem("Exit");
    exitItem.addActionListener(e -> System.exit(0));
    fileMenu.add(exitItem);

    menuBar.add(fileMenu);
    frame.setJMenuBar(menuBar);

    frame.setSize(400, 300);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

JToolBar: Adding Toolbars

`JToolBar` provides a way to add a toolbar to your application, which can contain buttons and other components for quick access to functions.

Example:

java

```
import javax.swing.*;

public class ToolBarExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("ToolBar Example");
    JToolBar toolBar = new JToolBar();
```

```
        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");
        toolBar.add(button1);
        toolBar.add(button2);

        frame.add(toolBar, "North");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

JTabbedPane: Tabs for Organizing Content

`JTabbedPane` allows you to create tabs that can be used to organize different panels or content areas.

Example:

java

```
import javax.swing.*;

public class TabbedPaneExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Tabbed Pane Example");
        JTabbedPane tabbedPane = new JTabbedPane();

        tabbedPane.addTab("Tab 1", new JLabel("Content for Tab 1"));
        tabbedPane.addTab("Tab 2", new JLabel("Content for Tab 2"));

        frame.add(tabbedPane);
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

JSplitPane: Split Views

`JSplitPane` allows you to create a split view in your application, where two components can be resized by dragging a divider.

Example:

java

```
import javax.swing.*;

public class SplitPaneExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Split Pane Example");
        JSplitPane splitPane = new JSplitPane();

        splitPane.setLeftComponent(new JLabel("Left Component"));
        splitPane.setRightComponent(new JLabel("Right Component"));

        frame.add(splitPane);
```

```
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

JScrollPane: Scrollable Views

`JScrollPane` provides a scrollable view for other components, allowing users to scroll through content that exceeds the visible area.

Example:

java

```
import javax.swing.*;

public class ScrollPaneExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Scroll Pane Example");
        JTextArea textArea = new JTextArea(10, 30);
        JScrollPane scrollPane = new JScrollPane(textArea);

        frame.add(scrollPane);
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

JTree: Hierarchical Data Representation

`JTree` is used to display hierarchical data in a tree structure, allowing users to expand and collapse nodes.

Example:

java

```
import javax.swing.*;

public class TreeExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Tree Example");
        DefaultMutableTreeNode root = new DefaultMutableTreeNode("Root");
        DefaultMutableTreeNode child1 = new DefaultMutableTreeNode("Child 1");
        DefaultMutableTreeNode child2 = new DefaultMutableTreeNode("Child 2");

        root.add(child1);
        root.add(child2);

        JTree tree = new JTree(root);
        frame.add(new JScrollPane(tree));

        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
```

```
}
```

JTable: Displaying Tabular Data

`JTable` allows you to display and edit tabular data in rows and columns.

Example:

java

```
import javax.swing.*;
import javax.swing.table.DefaultTableModel;

public class TableExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Table Example");
    String[] columnNames = {"Name", "Age", "Gender"};
    Object[][] data = {
      {"Alice", 25, "Female"},
      {"Bob", 30, "Male"}
    };

    JTable table = new JTable(new DefaultTableModel(data, columnNames));
    frame.add(new JScrollPane(table));

    frame.setSize(300, 200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

Swing and Graphics

Swing provides capabilities for custom graphics through the `paintComponent` method, allowing developers to draw shapes, images, and more.

Custom Painting with the paintComponent Method

You can override the `paintComponent(Graphics g)` method of a component to perform custom drawing.

Example:

java

```
import javax.swing.*;
import java.awt.*;

public class CustomPaintingExample extends JPanel {
  @Override
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.RED);
    g.fillRect(20, 20, 100, 100); // Draw a red rectangle
  }
```

```
    public static void main(String[] args) {
        JFrame frame = new JFrame("Custom Painting Example");
        CustomPaintingExample panel = new CustomPaintingExample();
        frame.add(panel);
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Drawing Shapes and Images

You can use the `Graphics` object to draw shapes and images within the `paintComponent` method.

Example:

java

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.BLUE);
    g.drawOval(50, 50, 100, 100); // Draw a blue circle
    Image img = Toolkit.getDefaultToolkit().getImage("path/to/image.png");
    g.drawImage(img, 10, 10, this); // Draw an image
}
```

Using Graphics2D for Advanced Graphics

`Graphics2D` provides more sophisticated control over graphics rendering, including transformations, anti-aliasing, and advanced shapes.

Example:

java

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.Ellipse2D;

public class Graphics2DExample extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2d = (Graphics2D) g;
        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
        g2d.setColor(Color.GREEN);
        g2d.fill(new Ellipse2D.Double(50, 50, 100, 100)); // Draw a green circle
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame("Graphics2D Example");
        Graphics2DExample panel = new Graphics2DExample();
        frame.add(panel);
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    frame.setVisible(true);
  }
}
```

Swing Utilities

Swing provides several utilities to enhance functionality, including `SwingWorker` for background tasks, `Swing Timer` for scheduling tasks, and `JOptionPane` for dialogs and notifications.

SwingWorker for Background Tasks

`SwingWorker` allows you to perform long-running tasks in the background without freezing the GUI. It can also safely update the GUI once the task is complete.

Example:

java

```
import javax.swing.*;
import java.util.List;

public class SwingWorkerExample {
  public static void main(String[] args) {
    JButton button = new JButton("Start Task");
    button.addActionListener(e -> {
      SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>() {
        @Override
        protected Void doInBackground() {
          // Simulate a long-running task
          try {
            Thread.sleep(2000);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
          return null;
        }

        @Override
        protected void done() {
          System.out.println("Task completed!");
        }
      };
      worker.execute();
    });

    JFrame frame = new JFrame("SwingWorker Example");
    frame.add(button);
    frame.setSize(300, 200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

Swing Timer for Scheduling Tasks

`Swing Timer` is used for scheduling repeated tasks in the Swing event dispatch thread.

Example:

java

```
import javax.swing.*;

public class TimerExample {
  public static void main(String[] args) {
    Timer timer = new Timer(1000, e -> System.out.println("Timer ticked!"));
    timer.start();

    JFrame frame = new JFrame("Timer Example");
    frame.setSize(300, 200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

JOptionPane for Dialogs and Notifications

`JOptionPane` provides a simple way to create pop-up dialogs for user input, notifications, and confirmations.

Example:

java

```
import javax.swing.*;

public class JOptionPaneExample {
  public static void main(String[] args) {
    JOptionPane.showMessageDialog(null, "This is a message dialog!");
    String input = JOptionPane.showInputDialog("Enter your name:");
    System.out.println("Hello, " + input + "!");
  }
}
```

CHAPTER - 10

LOOK AND FEEL

The "look and feel" of a Swing application refers to its visual appearance and user interaction characteristics. Java Swing allows developers to customize the look and feel to create aesthetically pleasing and user-friendly applications.

Customizing the Look and Feel of a Swing Application

Swing provides a mechanism to change the look and feel (L&F) of an application. You can choose from the default options provided by Swing or customize your own.

Example: Setting a different look and feel

java

```
import javax.swing.*;

public class LookAndFeelExample {
  public static void main(String[] args) {
    try {
      // Set the look and feel to Nimbus
      UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");
    } catch (Exception e) {
      e.printStackTrace();
    }

    JFrame frame = new JFrame("Look and Feel Example");
    JButton button = new JButton("Click Me");
    frame.add(button);
    frame.setSize(300, 200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

Setting System Look and Feel vs. Cross-Platform Look and Feel

1. **System Look and Feel**:
   - Adapts the appearance of the Swing application to match the native look and feel of the operating system.
   - This enhances user experience by making the application feel familiar.

   Example:

   java

   ```
   try {
     UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
   } catch (Exception e) {
     e.printStackTrace();
   }
   ```

2. **Cross-Platform Look and Feel**:

- Ensures a consistent appearance across all platforms.
- This is useful for applications that need to maintain a uniform look regardless of the operating system.

Example:

java

```
try {
   UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
} catch (Exception e) {
   e.printStackTrace();
}
```

Using Third-Party Look and Feel Libraries

In addition to the built-in options, several third-party libraries provide enhanced and visually appealing look and feel options. Popular libraries include:

- **Substance**: A sophisticated look and feel with themes and animations.
- **FlatLaf**: A modern Flat look and feel for Swing applications.

Example (Using FlatLaf):

java

```
import com.formdev.flatlaf.FlatLightLaf;
import javax.swing.*;

public class FlatLafExample {
  public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(new FlatLightLaf());
    } catch (Exception e) {
      e.printStackTrace();
    }

    JFrame frame = new JFrame("FlatLaf Example");
    JButton button = new JButton("Click Me");
    frame.add(button);
    frame.setSize(300, 200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

Best Practices in Swing Development

To create efficient and user-friendly Swing applications, consider the following best practices:

Managing Resources

- **Resource Management**: Keep track of memory and other resources. Use try-with-resources where applicable to ensure resources are closed properly.

- **Icons and Images**: Load images and icons once and reuse them to reduce memory consumption.

Threading Considerations

- **Event Dispatch Thread (EDT)**: Always update the GUI components on the EDT to ensure thread safety. Use SwingUtilities.invokeLater() for this purpose.

Example:

java

```
SwingUtilities.invokeLater(() -> {
    // Update GUI components here
});
```

Performance Optimizations

- **Reduce Component Creation**: Avoid creating new components unnecessarily. Reuse existing components where possible.
- **Use Proper Layouts**: Choose appropriate layout managers to avoid excessive resizing and rendering operations.
- **Optimize Painting**: Override paintComponent() efficiently, and use double buffering to reduce flickering.

Accessibility Considerations

- **Keyboard Navigation**: Ensure that all components are accessible via the keyboard.
- **Screen Readers**: Provide meaningful descriptions and use appropriate accessibility properties (like setAccessibleDescription).

Deploying Swing Applications

Deploying Swing applications involves packaging, creating runnable JAR files, and managing distribution and versioning.

Packaging Swing Applications

- **JAR Files**: Package your application into a JAR file using tools like jar command or build tools like Maven or Gradle.
- **Manifest File**: Include a manifest file in your JAR to specify the main class and dependencies.

Example: Create a JAR file

bash

```
jar cfm MyApp.jar Manifest.txt -C build/classes .
```

Creating Runnable JAR Files

To create a runnable JAR file, specify the main class in the manifest file. Users can then run your application with:

bash

```
java -jar MyApp.jar
```

Distributing and Versioning Applications

- **Distribution**: Host your JAR files on a server, or use platforms like GitHub for distribution.
- **Versioning**: Implement a versioning system within your application to help users identify updates. You can check for updates on startup.

Example: Simple version check

java

```
public static final String VERSION = "1.0.0";

public static void main(String[] args) {
    System.out.println("MyApp version " + VERSION);
}
```

REFRENCE

https://docs.oracle.com/javase/8/docs