COD3R.AADI

Java Foundation Classes (JFC)

SWING

GUI WITH JAVA SWING

@cod3r.aadi

www.github.com/adityaaerpule

About the Author

Aditya Aerpule *is a dedicated software developer with expertise in Java, web development, and database technologies. Currently pursuing a Master of Computer Applications (MCA) from RGPV, Bhopal, Aditya is passionate about building innovative and efficient software solutions. With a strong foundation in Java, MySQL, and creating dynamic user interfaces using Java Swing, he enjoys exploring various aspects of software development and continuously expanding his knowledge in the field.*

Beyond academics, Aditya is also an accomplished individual in yoga and Mallakhamb, with appearances on various TV shows showcasing his versatility. His unique blend of technical skills and personal discipline brings a fresh perspective to his work.

*In his book, "**GUI with JAVA SWING**," Aditya aims to make Java Swing accessible to beginners by providing practical examples and clear explanations of key concepts. His goal is to empower aspiring developers to build intuitive and responsive desktop applications.*

Aditya remains committed to contributing to the tech community through learning, teaching, and developing innovative solutions.

CONTENT

1. **Introduction to Swing** **01**
   - Overview of Swing
   - Difference between AWT and Swing
   - Swing architecture
2. **Swing Containers** **02**
   - JFrame
   - JPanel
   - JDialog
   - JApplet
3. **Swing Components** **04**
   - JLabel
   - JButton
   - JTextField
   - JTextArea
   - JCheckBox
   - JRadioButton
   - JComboBox
   - JList
   - JTable
   - JTree
   - JTabbedPane
   - JScrollPane
4. **Layouts in Swing** **08**
   - FlowLayout
   - BorderLayout
   - GridLayout
   - BoxLayout
   - GroupLayout
   - CardLayout
5. **Event Handling** **10**
   - ActionListener
   - KeyListener
   - MouseListener
   - WindowListener
   - FocusListener
6. **Swing Models** **13**
   - DefaultTableModel
   - DefaultListModel
   - ListSelectionModel
   - TreeModel
7. **Swing Menus** **16**
   - JMenuBar
   - JMenu
   - JMenuItem
   - JPopupMenu
8. **Dialog Boxes** **18**
   - JOptionPane
   - JColorChooser
   - JFileChooser
9. **Swing Timers** **21**
   - javax.swing.Timer
10. **Custom Rendering** **22**
    - Cell rendering for lists, tables, and trees
11. **Pluggable Look and Feel (PLAF)** **23**
    - Changing the Look and Feel
    - UIManager class
12. **Concurrency in Swing** **24**
    - SwingWorker
    - Event Dispatch Thread (EDT)
13. **Advanced Components** **26**
    - JProgressBar
    - JSlider
    - JSpinner
    - JSplitPane
14. **Swing Utilities** **28**
    - SwingUtilities class
    - invokeLater()

1. Introduction to Swing

Overview of Swing:

- Swing is a part of Java's JFC (Java Foundation Classes) used to create graphical user interfaces (GUIs).
- It provides a rich set of lightweight components like buttons, labels, text fields, tables, etc.
- Swing is platform-independent and built on top of the Abstract Window Toolkit (AWT) but offers more powerful and flexible components.

Difference between AWT and Swing:

- AWT components are heavyweight, which means they depend on the underlying platform's GUI (e.g., Windows, macOS).
- Swing components are lightweight, meaning they are written entirely in Java and are drawn by the Java code itself, making them more flexible and consistent across platforms.
- Swing offers more features and better customization compared to AWT.

Swing Architecture:

- Model-View-Controller (MVC): Swing components follow the MVC architecture, separating the internal representation (Model), the way it is presented to the user (View), and how the user interacts with it (Controller).
    - Model: Manages the data or the state.
    - View: Displays the data.
    - Controller: Manages the user interaction.

2. Swing Containers

Swing containers are special components that can hold other Swing components like buttons, text fields, labels, etc. Containers help in organizing the layout and structure of the user interface (UI). Here are the main containers:

1. JFrame:

- **Purpose**: A top-level container that represents a window with a title bar and borders.
- **Usage**: Used to create the main window of a GUI application.
- **Key Features**:
    - Supports adding components like buttons, labels, etc., inside its content pane.
    - Provides methods like setTitle(), setSize(), setVisible(), and setDefaultCloseOperation().

  Example:

```
JFrame frame = new JFrame("My Frame");
frame.setSize(400, 300);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
```

2. JPanel:

- **Purpose**: A generic lightweight container used to group other components together.
- **Usage**: Often used as an intermediary container to organize components, manage layout, or for custom painting.
- **Key Features**:
    - Does not have a title bar or border (unlike JFrame).
    - You can add a JPanel to another container like JFrame.

  Example:

```
JPanel panel = new JPanel();
panel.add(new JButton("Click Me"));
frame.add(panel);
```

3. JDialog:

- **Purpose**: A top-level container used for creating a dialog box (a pop-up window used for interactions).
- **Usage**: Commonly used for warnings, confirmations, or additional information outside the main window.
- **Key Features**:
    - Can be modal (blocking input to other windows) or non-modal.
    - Does not have a menu bar but can contain buttons, text fields, etc.

Example:

```
JDialog dialog = new JDialog(frame, "My Dialog", true); // true = modal
dialog.setSize(200, 150);
dialog.setVisible(true);
```

4. JApplet:

- **Purpose**: A top-level container used to create web-based applications.
- **Usage**: JApplet extends the Applet class and is used to embed Java applications in a web browser.
- **Key Features**:
    - Deprecated in modern Java versions due to security issues with applets.
    - Contains similar features as JFrame but within a browser context.

Example:

```
public class MyApplet extends JApplet {
   public void init() {
      add(new JLabel("Hello, Applet!"));
   }
}
```

Key Concepts:
Content Pane: In JFrame, JDialog, and JApplet, components are added to the **content pane**, not directly to the container itself. You can access the content pane with getContentPane().**Top-Level Containers**: JFrame, JDialog, and JApplet are considered top-level containers, meaning they can act as independent windows. In contrast, JPanel is a simple container that must be placed within a top-level container.

3. Swing Components

Swing provides a wide range of components that can be added to containers like JFrame and JPanel to build graphical user interfaces. Here are the most commonly used Swing components:

1. JLabel:

- **Purpose**: A display area for a short text string or an image.
- **Usage**: Used to display static information or simple instructions.
- **Key Features**:
    - Can contain text, an image, or both.
    - Text alignment, font, and color can be customized.

Example:

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

2. JButton:

- **Purpose**: A clickable button that triggers an action when pressed.
- **Usage**: Used for initiating events or actions (e.g., submitting a form).
- **Key Features**:
  - Supports text and icons.
  - Can register ActionListener for handling button clicks.

Example:

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

3. JTextField:

- **Purpose**: A single-line text field for user input.
- **Usage**: Used to collect short text inputs like usernames or search queries.
- **Key Features**:
  - Can set default text and manage input length.
  - Supports ActionListener for event handling.

Example:

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

4. JTextArea:

- **Purpose**: A multi-line area to display or allow the editing of text.
- **Usage**: Used for larger input areas, like comments or descriptions.
- **Key Features**:
  - Can set number of rows and columns.
  - Supports word wrapping and scrollbars.

Example:

```
JTextArea textArea = new JTextArea(5, 20); // 5 rows, 20 columns
panel.add(new JScrollPane(textArea)); // Optional scrollbar
```

5. JCheckBox:

- **Purpose**: A box that can be checked or unchecked, representing a boolean choice.
- **Usage**: Used for toggling settings or options.
- **Key Features**:
  - Can add ActionListener to capture when the state changes.

Example:

```
JCheckBox checkBox = new JCheckBox("I agree");
panel.add(checkBox);
```

6. JRadioButton:

- **Purpose**: A radio button that allows a user to select one option from a group of choices.
- **Usage**: Used when only one selection from a group is allowed (e.g., gender, payment options).
- **Key Features**:
    - Must be grouped using ButtonGroup to enforce single selection.

Example:

```
JRadioButton option1 = new JRadioButton("Option 1");
JRadioButton option2 = new JRadioButton("Option 2");
ButtonGroup group = new ButtonGroup();
group.add(option1);
group.add(option2);
panel.add(option1);
panel.add(option2);
```

7. JComboBox:

- **Purpose**: A dropdown list that allows users to select one item from a list.
- **Usage**: Used for selecting an item from multiple predefined options (e.g., country, state).
- **Key Features**:
    - Can add items programmatically or in the constructor.
    - Supports event listeners for selection changes.

Example:

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

8. JList:

- **Purpose**: A list component that displays a list of items, allowing single or multiple selections.
- **Usage**: Used for displaying and selecting from multiple items.
- **Key Features**:
    - Supports single or multiple item selection.
    - Requires ListModel for data management.

Example:

```
String[] data = { "Item 1", "Item 2", "Item 3" };
JList<String> list = new JList<>(data);
panel.add(new JScrollPane(list));
```

9. JTable:

- **Purpose**: A component that displays data in a two-dimensional table format.
- **Usage**: Used for displaying and editing tabular data (e.g., spreadsheets, data grids).
- **Key Features**:
    - Supports customizable table models and event handling.
    - Allows editing and sorting of table data.

Example:

```
String[] columnNames = { "Name", "Age", "Gender" };
```

```
Object[][] data = {
   { "John", 25, "Male" },
   { "Sara", 22, "Female" }
};
JTable table = new JTable(data, columnNames);
panel.add(new JScrollPane(table));
```

10. JTree:

- **Purpose**: A component that displays a hierarchical tree of data.
- **Usage**: Used for representing hierarchical structures like file systems or category trees.
- **Key Features**:
    - Nodes can be expanded and collapsed.
    - Uses TreeModel to manage data.

Example:

```
DefaultMutableTreeNode root = new DefaultMutableTreeNode("Root");
DefaultMutableTreeNode child1 = new DefaultMutableTreeNode("Child 1");
root.add(child1);
JTree tree = new JTree(root);
panel.add(new JScrollPane(tree));
```

11. JTabbedPane:

- **Purpose**: A component that allows switching between different tabs of content.
- **Usage**: Used for organizing multiple panes of content, each with its own tab.
- **Key Features**:
    - Each tab can hold different components.
    - Supports icons and tooltips on tabs.

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"));
panel.add(tabbedPane);
```

12. JScrollPane:

- **Purpose**: A container that provides scrollbars when content exceeds the visible area.
- **Usage**: Used to wrap components like JTextArea, JList, JTable, or JTree that might have large amounts of content.
- **Key Features**:
    - Automatically provides vertical and/or horizontal scrolling.

Example:

```
JTextArea textArea = new JTextArea(10, 20);
JScrollPane scrollPane = new JScrollPane(textArea);
panel.add(scrollPane);
```

4. Layouts in Swing

Layout managers are responsible for arranging the components within a container in Java Swing. They determine how components are sized and positioned, allowing developers to control the appearance of the user interface. Below are the main layout managers in Swing:

1. FlowLayout:

- **Purpose**: Arranges components in a row, one after another, and wraps them to the next line if there isn't enough horizontal space.
- **Usage**: Suitable for simple forms or dialogs where components are laid out in a sequence.
- **Key Features**:
  - Components are placed in the order they are added.
  - Default alignment is **center**, but can be changed to left or right.

Example:

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

2. BorderLayout:

- **Purpose**: Divides the container into five regions: **North**, **South**, **East**, **West**, and **Center**. Each region can hold only one component.
- **Usage**: Commonly used for creating main windows where different sections (e.g., header, footer, sidebar) are needed.
- **Key Features**:
  - The **Center** region expands to take up the remaining space.
  - If no components are added to certain regions, they are left empty.

Example:

```
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);
```

3. GridLayout:

- **Purpose**: Arranges components in a grid of rows and columns, with each cell of equal size.
- **Usage**: Suitable for forms, calculators, or any situation where components need to be uniformly spaced.
- **Key Features**:
  - Components are placed in the grid in the order they are added.
  - Can specify the number of rows and columns.

Example:

```
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"));
```

4. BoxLayout:

- **Purpose**: Arranges components either vertically or horizontally in a single row or column.
- **Usage**: Ideal when you need to align components in one direction.
- **Key Features**:
    - Supports both horizontal (BoxLayout.X_AXIS) and vertical (BoxLayout.Y_AXIS) alignment.
    - Components are placed in the order they are added.

Example:

```
JPanel panel = new JPanel();
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS)); // Vertical alignment
panel.add(new JButton("Button 1"));
panel.add(new JButton("Button 2"));
```

5. GroupLayout:

- **Purpose**: Allows precise control over the position of components by defining both horizontal and vertical groupings.
- **Usage**: Typically used with GUI builders (like NetBeans) to create complex layouts.
- **Key Features**:
    - Offers more flexibility by grouping components and controlling their size and alignment.
    - Supports automatic alignment and resizing.

Example:

```
JPanel panel = new JPanel();
GroupLayout layout = new GroupLayout(panel);
panel.setLayout(layout);
layout.setAutoCreateGaps(true);

JButton button1 = new JButton("Button 1");
JButton button2 = new JButton("Button 2");

layout.setHorizontalGroup(
   layout.createSequentialGroup()
      .addComponent(button1)
      .addComponent(button2)
);
layout.setVerticalGroup(
   layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
      .addComponent(button1)
      .addComponent(button2)
);
```

6. CardLayout:

- **Purpose**: Allows multiple components to be stacked on top of each other, with only one component visible at a time.
- **Usage**: Commonly used for implementing a wizard or tab-like navigation where users can switch between views.
- **Key Features**:
    - Components are organized like a deck of cards.
    - Methods like next(), previous(), and show() can switch between the components.

Example:

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

```
panel.add(new JLabel("Card 1"), "Card 1");
panel.add(new JLabel("Card 2"), "Card 2");

CardLayout cl = (CardLayout) panel.getLayout();
cl.show(panel, "Card 2"); // Show "Card 2"
```

Key Concepts:

- **Nested Layouts**: Multiple layout managers can be nested. For example, a JPanel using FlowLayout can be added to a JPanel using BorderLayout, allowing more complex layouts.
- **Custom Layouts**: You can also implement custom layouts by extending LayoutManager or LayoutManager2 for specific positioning needs.

5. Event Handling in Swing

In Java Swing, **event handling** refers to responding to user interactions like button clicks, key presses, or mouse movements. The event-handling mechanism follows the **Delegation Event Model**, where events are dispatched to specific event handlers based on the source of the event and the type of event. The key components of this system include:

1. **Event Source**: The component (e.g., a JButton) that generates an event.
2. **Event Object**: An object representing the event (e.g., ActionEvent for button clicks).
3. **Event Listener**: The interface that receives and processes the event (e.g., ActionListener).

Key Event Listeners in Swing

Here are the most common event listeners and how they work:

1. ActionListener

- **Purpose**: Handles action events like button clicks or pressing the Enter key in a text field.
- **Usage**: Primarily used with buttons (JButton), menu items (JMenuItem), and text fields (JTextField).
- **Interface Method**: void actionPerformed(ActionEvent e)
- **Example**:

```
JButton button = new JButton("Click Me");
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      System.out.println("Button clicked!");
   }
});
```

2. KeyListener

- **Purpose**: Handles keyboard events, such as pressing or releasing keys.
- **Usage**: Used when you want to capture user keyboard input (e.g., games, form validation).
- **Interface Methods**:
    - void keyPressed(KeyEvent e) – Triggered when a key is pressed.
    - void keyReleased(KeyEvent e) – Triggered when a key is released.
    - void keyTyped(KeyEvent e) – Triggered when a key is typed (pressed and released).
- **Example**:

```
JTextField textField = new JTextField(20);
textField.addKeyListener(new KeyAdapter() {
   public void keyPressed(KeyEvent e) {
      System.out.println("Key Pressed: " + e.getKeyChar());
```

```
    }
});
```

3. MouseListener

- **Purpose**: Handles mouse events like clicks, entering or exiting a component, and pressing or releasing mouse buttons.
- **Usage**: Used for adding interactive mouse features (e.g., clicking on labels, detecting right-clicks, drawing, etc.).
- **Interface Methods**:
    - void mouseClicked(MouseEvent e)
    - void mouseEntered(MouseEvent e)
    - void mouseExited(MouseEvent e)
    - void mousePressed(MouseEvent e)
    - void mouseReleased(MouseEvent e)
- **Example**:

```
JLabel label = new JLabel("Click Me");
label.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        System.out.println("Mouse clicked at: " + e.getX() + ", " + e.getY());
    }
});
```

4. MouseMotionListener

- **Purpose**: Handles mouse movement events like moving or dragging the mouse.
- **Usage**: Useful for interactive UIs like drawing apps or custom cursors.
- **Interface Methods**:
    - void mouseDragged(MouseEvent e)
    - void mouseMoved(MouseEvent e)
- **Example**:

```
JLabel label = new JLabel("Drag Me");
label.addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent e) {
        System.out.println("Mouse dragged at: " + e.getX() + ", " + e.getY());
    }
});
```

5. WindowListener

- **Purpose**: Handles window events like opening, closing, minimizing, or activating a window.
- **Usage**: Often used in JFrame for controlling what happens when a window is closed.
- **Interface Methods**:
    - void windowOpened(WindowEvent e)
    - void windowClosing(WindowEvent e)
    - void windowClosed(WindowEvent e)
    - void windowIconified(WindowEvent e)
    - void windowDeiconified(WindowEvent e)
    - void windowActivated(WindowEvent e)
    - void windowDeactivated(WindowEvent e)
- **Example**:

```
JFrame frame = new JFrame("Window Listener Example");
frame.addWindowListener(new WindowAdapter() {
```

```
    public void windowClosing(WindowEvent e) {
        System.out.println("Window is closing");
        System.exit(0); // Close the application
    }
});
```

6. FocusListener

- **Purpose**: Handles events when a component gains or loses focus.
- **Usage**: Typically used for input fields where the behavior changes based on focus (e.g., changing field background color when selected).
- **Interface Methods**:
    - void focusGained(FocusEvent e)
    - void focusLost(FocusEvent e)
- **Example**:

```
JTextField textField = new JTextField(20);
textField.addFocusListener(new FocusAdapter() {
    public void focusGained(FocusEvent e) {
        System.out.println("Focus gained on text field.");
    }
});
```

Event Handling Process

1. **Registering Event Listeners**: You attach an event listener to a component using the addXListener() method, where X represents the type of listener (e.g., addActionListener()).
2. **Handling Events**: Once an event occurs (e.g., a button click), an event object is created (e.g., ActionEvent) and passed to the listener's method (e.g., actionPerformed()), where the event is processed.

Key Concepts

- **Adapter Classes**: For event listeners that have multiple methods (e.g., MouseListener, WindowListener), adapter classes like MouseAdapter and WindowAdapter are provided. These adapters implement the listener interfaces with empty method bodies, so you can override only the methods you need.
- **Anonymous Inner Classes**: You can implement event listeners using anonymous inner classes to keep your code more concise. This is common when you only need to handle one event type in a specific component.

```
JButton button = new JButton("Click Me");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});
```

- **Lambda Expressions**: If you're using Java 8 or later, you can use lambda expressions for event handling, making the code even more concise for functional interfaces like ActionListener.

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

Summary of Key Event Listeners:

Listener	Purpose	Common Components
ActionListener	Button clicks and action events	JButton, JMenuItem, JTextField

Listener	Purpose	Common Components
KeyListener	Keyboard key presses/releases	JTextField, JTextArea
MouseListener	Mouse clicks and component entry/exit	All components
MouseMotionListener	Mouse movement and dragging	All components
WindowListener	Window open, close, minimize events	JFrame, JDialog
FocusListener	Component focus gain/loss	JTextField, JButton

Event handling is a core concept in Swing, as it connects user actions to the functionality of the application. Understanding how to register and process these events is essential for building interactive Swing-based UIs.

6. Swing Models

In Java Swing, models are the data representations used by various components like lists, tables, and trees. The **Model-View-Controller (MVC)** architecture is employed, where the **model** represents the data, the **view** is the UI component that displays the data, and the **controller** handles the interaction between the user and the data. Swing components like JList, JTable, and JTree rely on models to separate the data from the view, allowing for a more flexible and reusable design.

Swing provides several default models for common components, but you can also create custom models by implementing specific interfaces.

1. ListModel (JList)

- **Purpose**: Defines the data model for a JList, representing a list of items.
- **Usage**: Used to manage the data behind a JList, separating the list's content from its presentation.
- **Implementations**:
  - **DefaultListModel**: The standard implementation of ListModel, which supports dynamic addition and removal of elements.
  - **Custom ListModel**: Implementing your own ListModel interface to customize how the list is populated.
- **Key Methods in ListModel**:
  - int getSize(): Returns the number of items in the list.
  - Object getElementAt(int index): Returns the item at the specified index.
  - void addElement(Object obj): Adds an element to the list (specific to DefaultListModel).
- **Example** (using DefaultListModel):

```
DefaultListModel<String> listModel = new DefaultListModel<>();
listModel.addElement("Item 1");
listModel.addElement("Item 2");

JList<String> list = new JList<>(listModel);
panel.add(new JScrollPane(list));
```

2. TableModel (JTable)

- **Purpose**: Defines the data model for a JTable, representing a two-dimensional grid of data.
- **Usage**: Used to manage data in tables, separating the structure of the data (rows and columns) from the way it is displayed.
- **Implementations**:
  - **DefaultTableModel**: A basic implementation of TableModel that supports adding, removing, and updating data in the table.
  - **Custom TableModel**: Implementing your own TableModel interface for custom behavior, useful for more complex tables.
- **Key Methods in TableModel**:

    - int getRowCount(): Returns the number of rows in the table.
    - int getColumnCount(): Returns the number of columns in the table.
    - Object getValueAt(int row, int column): Returns the value at the specified cell.
    - void setValueAt(Object value, int row, int column): Sets the value at a particular cell (specific to DefaultTableModel).
- **Example** (using DefaultTableModel):

```
String[] columnNames = { "Name", "Age" };
Object[][] data = {
   { "John", 25 },
   { "Jane", 30 }
};

DefaultTableModel tableModel = new DefaultTableModel(data, columnNames);
JTable table = new JTable(tableModel);
panel.add(new JScrollPane(table));
```

3. TreeModel (JTree)

- **Purpose**: Defines the data model for a JTree, representing a hierarchical structure of nodes.
- **Usage**: Used to manage the hierarchical data that a JTree displays, separating the nodes and the tree's structure from its display.
- **Implementations**:
    - **DefaultTreeModel**: The default implementation for managing tree nodes.
    - **Custom TreeModel**: Implementing your own TreeModel interface for complex, custom trees.
- **Key Methods in TreeModel**:
    - Object getRoot(): Returns the root node of the tree.
    - Object getChild(Object parent, int index): Returns the child of a parent node at the specified index.
    - int getChildCount(Object parent): Returns the number of children a parent node has.
    - boolean isLeaf(Object node): Returns whether the node is a leaf node (i.e., has no children).
- **Example** (using DefaultTreeModel):

```
DefaultMutableTreeNode root = new DefaultMutableTreeNode("Root");
DefaultMutableTreeNode child1 = new DefaultMutableTreeNode("Child 1");
root.add(child1);

DefaultTreeModel treeModel = new DefaultTreeModel(root);
JTree tree = new JTree(treeModel);
panel.add(new JScrollPane(tree));
```

4. ComboBoxModel (JComboBox)

- **Purpose**: Defines the data model for a JComboBox, representing a list of items, with one item selected at a time.
- **Usage**: Used to manage the list of items in a dropdown, separating the data from the visual presentation.
- **Implementations**:
    - **DefaultComboBoxModel**: The standard implementation for a combo box that allows dynamic updates to the list.
    - **Custom ComboBoxModel**: Implementing your own ComboBoxModel interface to control the selection and the list of items.
- **Key Methods in ComboBoxModel**:
    - void setSelectedItem(Object item): Sets the currently selected item.
    - Object getSelectedItem(): Returns the currently selected item.
    - int getSize(): Returns the number of items in the list.
    - Object getElementAt(int index): Returns the item at the specified index.
- **Example** (using DefaultComboBoxModel):

```
DefaultComboBoxModel<String> comboBoxModel = new DefaultComboBoxModel<>();
comboBoxModel.addElement("Item 1");
comboBoxModel.addElement("Item 2");

JComboBox<String> comboBox = new JComboBox<>(comboBoxModel);
panel.add(comboBox);
```

5. SpinnerModel (JSpinner)

- **Purpose**: Defines the data model for a JSpinner, representing a sequence of values.
- **Usage**: Used to manage the value in a spinner component, separating the value manipulation from the view.
- **Implementations**:
  - **SpinnerNumberModel**: A model for numerical values.
  - **SpinnerDateModel**: A model for dates.
  - **SpinnerListModel**: A model for a list of values.
- **Key Methods in SpinnerModel**:
  - Object getValue(): Returns the current value of the spinner.
  - void setValue(Object value): Sets the current value of the spinner.
  - Object getNextValue(): Returns the next value in the sequence.
  - Object getPreviousValue(): Returns the previous value in the sequence.
- **Example** (using SpinnerNumberModel):

```
SpinnerModel spinnerModel = new SpinnerNumberModel(1, 1, 100, 1); // value, min, max, step
JSpinner spinner = new JSpinner(spinnerModel);
panel.add(spinner);
```

Custom Models

While Swing provides default models for most components, there are cases where you need more control over how data is represented and manipulated. In such cases, you can create custom models by implementing the appropriate interface (ListModel, TableModel, TreeModel, etc.).

For example, you can create a custom TableModel that retrieves data from a database or a custom ListModel that dynamically fetches items from a web service. Custom models allow you to decouple your data from the view, ensuring that the UI remains responsive and flexible even as the data changes.

Summary of Common Swing Models:

Model	Component	Default Implementation	Custom Implementation
ListModel	JList	DefaultListModel	Implement ListModel
TableModel	JTable	DefaultTableModel	Implement TableModel
TreeModel	JTree	DefaultTreeModel	Implement TreeModel
ComboBoxModel	JComboBox	DefaultComboBoxModel	Implement ComboBoxModel
SpinnerModel	JSpinner	SpinnerNumberModel, etc.	Implement SpinnerModel

7. Swing Menus

Menus in Java Swing provide a way to organize commands and options in a structured manner, enhancing the user experience by offering accessible and easy-to-navigate interfaces. Swing menus consist of several components, including menus, menu items, and toolbars.

Types of Menus in Swing

1. **JMenuBar**: The container for menus, typically displayed at the top of a window.
2. **JMenu**: Represents a single menu within the menu bar, which can contain multiple menu items or submenus.
3. **JMenuItem**: A selectable item in a menu that triggers an action when clicked.
4. **JCheckBoxMenuItem**: A menu item that can be selected or deselected, representing a binary choice.
5. **JRadioButtonMenuItem**: A menu item that is part of a group of items, where only one item can be selected at a time.
6. **JToolBar**: A horizontal or vertical bar that contains buttons or menu items for quick access to frequently used actions.

Creating Menus

To create a menu in Swing, follow these steps:

1. **Create a JMenuBar**: Initialize the menu bar and add it to the frame.
2. **Create JMenu**: Initialize the menus and add them to the menu bar.
3. **Create JMenuItem**: Initialize the menu items and add them to the respective menus.
4. **Add ActionListeners**: Attach action listeners to handle user interactions.

Example: Creating a Simple Menu

Here's a complete example of creating a menu bar with menus and menu items in a JFrame:

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

public class MenuExample {

public static void main(String[] args) {
JFrame frame = new JFrame("Menu Example");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(400, 300);

// Create the menu bar
JMenuBar menuBar = new JMenuBar();

// Create File menu
JMenu fileMenu = new JMenu("File");
JMenuItem newItem = new JMenuItem("New");
JMenuItem openItem = new JMenuItem("Open");
JMenuItem exitItem = new JMenuItem("Exit");

// Add action listener for Exit
exitItem.addActionListener(e -> System.exit(0));

fileMenu.add(newItem);
fileMenu.add(openItem);
fileMenu.addSeparator(); // Adds a separator line
```

```
        fileMenu.add(exitItem);

        // Create Edit menu
        JMenu editMenu = new JMenu("Edit");
        JMenuItem cutItem = new JMenuItem("Cut");
        JMenuItem copyItem = new JMenuItem("Copy");
        JMenuItem pasteItem = new JMenuItem("Paste");

        editMenu.add(cutItem);
        editMenu.add(copyItem);
        editMenu.add(pasteItem);

        // Add menus to the menu bar
        menuBar.add(fileMenu);
        menuBar.add(editMenu);

        // Set the menu bar on the frame
        frame.setJMenuBar(menuBar);

        // Display the frame
        frame.setVisible(true);
        }
        }
```

Explanation of the Example

1. **Creating the JFrame**: A JFrame is created as the main window.
2. **JMenuBar**: A JMenuBar is created to hold the menus.
3. **JMenu**: Two menus, "File" and "Edit", are created and populated with menu items.
4. **JMenuItem**: Each menu item is created and added to the corresponding menu.
5. **Action Listener**: The exit menu item has an action listener that terminates the application when clicked.
6. **Adding Menus to MenuBar**: The menus are added to the menu bar, which is then set on the frame.

CheckBox and RadioButton Menu Items

In addition to standard menu items, Swing provides checkboxes and radio buttons for specific functionalities:

1. JCheckBoxMenuItem

- Represents a menu item that can be checked or unchecked.
- Useful for toggling settings or preferences.

Example:

```
JCheckBoxMenuItem showStatusItem = new JCheckBoxMenuItem("Show Status Bar");
showStatusItem.addActionListener(e -> {
    // Toggle status bar visibility
});
editMenu.add(showStatusItem);
```

2. JRadioButtonMenuItem

- Used to represent a set of options where only one can be selected at a time.
- Grouped together to ensure mutual exclusivity.

Example:

```
JRadioButtonMenuItem option1 = new JRadioButtonMenuItem("Option 1");
JRadioButtonMenuItem option2 = new JRadioButtonMenuItem("Option 2");
ButtonGroup group = new ButtonGroup();
group.add(option1);
group.add(option2);
editMenu.add(option1);
editMenu.add(option2);
```

Toolbars

Toolbars provide a quick-access interface for commonly used actions, similar to menus but typically more visible. Toolbars can contain buttons, menu items, and other components.

Example of a Toolbar:

```
JToolBar toolBar = new JToolBar("Main Toolbar");
JButton newButton = new JButton("New");
toolBar.add(newButton);
frame.getContentPane().add(toolBar, BorderLayout.NORTH);
```

Summary of Menu Components

Component	Description
JMenuBar	Container for menus, usually at the top of the window.
JMenu	Represents a single menu, can contain menu items or submenus.
JMenuItem	A selectable item in a menu that triggers an action.
JCheckBoxMenuItem	A menu item that can be checked or unchecked.
JRadioButtonMenuItem	A menu item that belongs to a group, allowing one selection.
JToolBar	A bar containing buttons or menu items for quick access.

8. Dialog Boxes in Swing

Dialog boxes in Java Swing are specialized windows used to display information or prompt users for input. They provide a way to interact with the user without requiring the user to navigate away from the main application window. Dialogs can be modal or non-modal, depending on whether they block input to other windows while they are open.

Types of Dialogs

1. **JDialog**: The base class for creating dialog boxes in Swing.
2. **JOptionPane**: A utility class for creating standard dialog boxes for messages, confirmations, and input.

1. Using JDialog

JDialog is a customizable dialog box that can be used for various purposes, such as displaying custom content or gathering input from the user.

Key Features of JDialog:

- Can be modal or non-modal.
- Can contain any Swing component (buttons, text fields, etc.).
- Can be customized in terms of size, layout, and content.

Example: Creating a Custom JDialog

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

public class CustomDialogExample {

    public static void main(String[] args) {
        JFrame frame = new JFrame("Custom Dialog Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        JButton button = new JButton("Open Dialog");
        button.addActionListener(e -> openDialog(frame));

        frame.getContentPane().add(button, BorderLayout.CENTER);
        frame.setVisible(true);
    }

    private static void openDialog(JFrame parent) {
        JDialog dialog = new JDialog(parent, "Custom Dialog", true);
        dialog.setLayout(new FlowLayout());
        dialog.setSize(300, 150);

        JLabel label = new JLabel("This is a custom dialog.");
        JButton okButton = new JButton("OK");
        okButton.addActionListener(e -> dialog.dispose());

        dialog.add(label);
        dialog.add(okButton);
        dialog.setVisible(true);
    }
}
```

Explanation of the Example

1. **Creating the JFrame**: A main window is created with a button to open the dialog.
2. **Opening the JDialog**: The openDialog method creates a JDialog that is modal (blocks interaction with the parent frame until closed).
3. **Adding Components**: A label and an OK button are added to the dialog.
4. **Displaying the Dialog**: The dialog is displayed using setVisible(true).

2. Using JOptionPane

JOptionPane is a convenient class for creating standard dialog boxes without the need for extensive customization. It can be used for showing messages, confirming actions, and prompting for input.

Common Methods of JOptionPane:

- **showMessageDialog**: Displays a message dialog.
- **showConfirmDialog**: Displays a dialog with options for confirmation (Yes/No/Cancel).

- **showInputDialog**: Prompts the user for input.

Example: Using JOptionPane

```
import javax.swing.*;

public class JOptionPaneExample {

    public static void main(String[] args) {
        // Show a message dialog
        JOptionPane.showMessageDialog(null, "Hello, this is a message!");

        // Show a confirmation dialog
        int response = JOptionPane.showConfirmDialog(null, "Do you want to continue?", "Confirm",
JOptionPane.YES_NO_OPTION);
        if (response == JOptionPane.YES_OPTION) {
            System.out.println("User chose to continue.");
        } else {
            System.out.println("User chose not to continue.");
        }

        // Show an input dialog
        String name = JOptionPane.showInputDialog("Enter your name:");
        System.out.println("Hello, " + name + "!");
    }
}
```

Explanation of the Example

1. **Message Dialog**: The showMessageDialog method displays a simple message.
2. **Confirmation Dialog**: The showConfirmDialog method prompts the user with a Yes/No option. The response is captured and used to determine the action.
3. **Input Dialog**: The showInputDialog method prompts the user for a string input and captures it.

Modal vs Non-Modal Dialogs

- **Modal Dialog**: Prevents interaction with other windows of the application until it is closed. It is useful for critical operations where user input is necessary before proceeding.
- **Non-Modal Dialog**: Allows the user to interact with other windows while the dialog is open. It is suitable for auxiliary tasks that do not require immediate attention.

Example of a Non-Modal Dialog

```
private static void openNonModalDialog(JFrame parent) {
    JDialog dialog = new JDialog(parent, "Non-Modal Dialog", false);
    dialog.setLayout(new FlowLayout());
    dialog.setSize(300, 150);

    JLabel label = new JLabel("This is a non-modal dialog.");
    JButton okButton = new JButton("OK");
    okButton.addActionListener(e -> dialog.dispose());

    dialog.add(label);
    dialog.add(okButton);
    dialog.setVisible(true);
}
```

Customizing Dialogs

You can customize dialogs by adding components such as text fields, checkboxes, or panels to meet your application's specific needs. Here's an example of a dialog with input fields:

```
private static void openInputDialog(JFrame parent) {
    JDialog dialog = new JDialog(parent, "Input Dialog", true);
    dialog.setLayout(new GridLayout(3, 2));
    dialog.setSize(300, 150);

    dialog.add(new JLabel("Username:"));
    JTextField userField = new JTextField();
    dialog.add(userField);

    dialog.add(new JLabel("Password:"));
    JPasswordField passField = new JPasswordField();
    dialog.add(passField);

    JButton submitButton = new JButton("Submit");
    submitButton.addActionListener(e -> {
        String username = userField.getText();
        String password = new String(passField.getPassword());
        System.out.println("Username: " + username + ", Password: " + password);
        dialog.dispose();
    });

    dialog.add(submitButton);
    dialog.setVisible(true);
}
```

9. Swing Timers

The javax.swing.Timer class is used to schedule a task to be executed after a specified delay or at regular intervals. It is particularly useful for creating animations, updating UI components, and managing events that require timing.

Key Features of Swing Timer:

- **Event Dispatch Thread (EDT)**: Timers run on the EDT, ensuring that updates to the UI are thread-safe.
- **Non-blocking**: Timers do not block the execution of the program; they simply schedule an event for later execution.

Creating a Timer

To create a timer, you need to specify the delay (in milliseconds) and an ActionListener that defines the action to be performed when the timer fires.

Example: Basic Timer Example

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

public class TimerExample {

    public static void main(String[] args) {
        JFrame frame = new JFrame("Timer Example");
```

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

        JLabel label = new JLabel("Timer: 0");
        frame.getContentPane().add(label);

        Timer timer = new Timer(1000, new ActionListener() {
            private int count = 0;

            @Override
            public void actionPerformed(ActionEvent e) {
                count++;
                label.setText("Timer: " + count);
            }
        });

        timer.start(); // Start the timer

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

Explanation of the Example

1. **Creating the JFrame**: A simple window with a label is created.
2. **Timer Initialization**: A Timer is created with a delay of 1000 milliseconds (1 second) and an ActionListener.
3. **Action Performed**: Every time the timer fires, the count is incremented, and the label is updated.
4. **Starting the Timer**: The start() method is called to begin the timer.

10. Custom Rendering

Custom rendering in Swing allows you to control how components display their data. This is particularly important for lists, tables, and trees, where you may need to customize the appearance of items based on their content or state.

Cell Rendering for Lists, Tables, and Trees

- **List Rendering**: Customize how items in a JList are displayed.
- **Table Rendering**: Control how cells in a JTable are rendered.
- **Tree Rendering**: Define the appearance of nodes in a JTree.

Example: Custom List Cell Renderer

To create a custom renderer for a JList, implement the ListCellRenderer interface.

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

public class CustomListRenderer extends JFrame {

    public CustomListRenderer() {
        String[] data = {"Apple", "Banana", "Cherry"};
        JList<String> list = new JList<>(data);

        list.setCellRenderer(new ListCellRenderer<String>() {
            @Override
```

```
        public Component getListCellRendererComponent(JList<? extends String> list, String value, int index,
boolean isSelected, boolean cellHasFocus) {
            JLabel label = new JLabel(value);
            if (isSelected) {
                label.setBackground(Color.BLUE);
                label.setForeground(Color.WHITE);
            } else {
                label.setBackground(Color.WHITE);
                label.setForeground(Color.BLACK);
            }
            label.setOpaque(true);
            return label;
        }
    });

    add(new JScrollPane(list));
    setTitle("Custom List Renderer");
    setSize(300, 200);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(CustomListRenderer::new);
  }
}
```

Explanation of the Example

1. **Custom Renderer**: Implement ListCellRenderer to define how each item is displayed.
2. **getListCellRendererComponent**: Customize the label based on whether the item is selected.
3. **Background and Foreground**: Change colors based on selection state.
4. **Setting Renderer**: The custom renderer is set on the JList.

11. Pluggable Look and Feel (PLAF)

Swing supports a pluggable look and feel (PLAF), allowing developers to change the appearance of the entire application without altering the underlying code. This feature enables a consistent UI that can mimic native operating system styles or adopt custom themes.

Changing the Look and Feel

You can change the look and feel at runtime or before the application starts. Swing provides several built-in look and feels, including:

- **Metal**: The default Swing look and feel.
- **Nimbus**: A modern look and feel introduced in Java 6.
- **System**: Uses the look and feel of the operating system.

Example: Setting the Look and Feel

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

Explanation of the Example

1. **UIManager**: Use the UIManager class to set the desired look and feel.
2. **Setting Nimbus**: The Nimbus look and feel is applied before creating any UI components.
3. **Creating the JFrame**: A simple frame with a button is created, displaying the new look.

12. UIManager Class

The UIManager class is a central class in Swing that manages the look and feel of the application. It provides methods to get and set various UI properties and styles.

Key Methods of UIManager

- **setLookAndFeel(String name)**: Sets the look and feel for the application.
- **getLookAndFeel()**: Returns the current look and feel.
- **get(String key)**: Retrieves the value of a specific UI property.
- **put(String key, Object value)**: Sets a specific UI property value.

Example: Customizing UI Properties

You can customize individual UI properties using the UIManager.

```
import javax.swing.*;

public class CustomUIManagerExample {

    public static void main(String[] args) {
        // Set custom properties
        UIManager.put("Button.background", Color.BLUE);
        UIManager.put("Button.foreground", Color.WHITE);

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

Explanation of the Example

1. **Custom Properties**: The background and foreground colors of buttons are set using UIManager.put().
2. **Creating Components**: The customized button will reflect the new properties.

12. Concurrency in Swing

Concurrency in Swing is essential for maintaining a responsive user interface (UI). Swing is not thread-safe, meaning that all updates to the UI must occur on the Event Dispatch Thread (EDT). If long-running tasks are performed on the EDT, the UI will freeze, leading to a poor user experience.

Key Concepts

1. **Event Dispatch Thread (EDT)**: The thread responsible for handling all UI events, including user actions and painting components. Only one thread can modify the UI at a time.
2. **Background Tasks**: To perform time-consuming tasks without blocking the EDT, use background threads or Swing utilities.

Managing Concurrency in Swing

1. SwingWorker

SwingWorker is a class that facilitates the execution of background tasks while providing hooks to update the UI upon completion. It allows you to run tasks in a separate thread and safely update the UI on the EDT.

Example of Using SwingWorker

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

public class SwingWorkerExample {

    public static void main(String[] args) {
        JFrame frame = new JFrame("SwingWorker Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
        JButton button = new JButton("Start Task");
        JLabel label = new JLabel("Result: ");

        button.addActionListener(e -> {
            button.setEnabled(false); // Disable the button
            label.setText("Processing...");

            // Create a SwingWorker
            SwingWorker<String, Void> worker = new SwingWorker<>() {
                @Override
                protected String doInBackground() throws Exception {
                    // Simulate a long-running task
                    Thread.sleep(3000);
                    return "Task Completed!";
                }

                @Override
                protected void done() {
                    try {
                        // Get the result and update the label on the EDT
```

```
                String result = get();
                label.setText(result);
            } catch (Exception ex) {
                label.setText("Error: " + ex.getMessage());
            } finally {
                button.setEnabled(true); // Re-enable the button
            }
        }
    };

    worker.execute(); // Start the worker
});

frame.setLayout(new FlowLayout());
frame.add(button);
frame.add(label);
frame.setVisible(true);
}
}
```

Explanation of the Example

1. **Creating a JFrame**: A simple window with a button and label is created.
2. **Button ActionListener**: When the button is clicked, it disables itself and updates the label.
3. **SwingWorker**:
    - doInBackground(): Contains the long-running task (simulated with Thread.sleep).
    - done(): Called on the EDT after doInBackground() completes. It updates the label with the result.
4. **Executing the Worker**: The worker is started with execute().

13. Advanced Components in Swing

Swing provides several advanced components that allow developers to create more complex and interactive UIs. Here are some of the notable advanced components:

1. JTable

JTable is used for displaying tabular data in rows and columns. It supports sorting, filtering, and editing.

Basic Example:

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

public class JTableExample {

    public static void main(String[] args) {
        JFrame frame = new JFrame("JTable Example");
        String[] columnNames = {"Name", "Age", "Gender"};
        Object[][] data = {
            {"Alice", 30, "Female"},
            {"Bob", 25, "Male"},
            {"Charlie", 35, "Male"}
        };

        DefaultTableModel model = new DefaultTableModel(data, columnNames);
        JTable table = new JTable(model);
```

```
        JScrollPane scrollPane = new JScrollPane(table);
        frame.add(scrollPane);

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

Explanation of the Example

1. **Creating a JTable**: A table is created using DefaultTableModel for easy data manipulation.
2. **Scroll Pane**: The table is placed in a JScrollPane for better navigation.
3. **Displaying the Frame**: The table is displayed in a JFrame.

2. JTree

JTree is used to display hierarchical data in a tree structure. It supports nodes, leaf nodes, and can be customized for various data types.

Basic Example:

```
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;

public class JTreeExample {

    public static void main(String[] args) {
        JFrame frame = new JFrame("JTree 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.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Explanation of the Example

1. **Creating a JTree**: A JTree is created with a root node and two child nodes.
2. **Scroll Pane**: The tree is added to a JScrollPane for scrolling.
3. **Displaying the Frame**: The tree is displayed in a JFrame.

3. JTabbedPane

JTabbedPane allows users to switch between different panels or views using tabs.

Basic Example:

```
import javax.swing.*;

public class JTabbedPaneExample {

   public static void main(String[] args) {
      JFrame frame = new JFrame("JTabbedPane 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.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(300, 200);
      frame.setVisible(true);
   }
}
```

Explanation of the Example

1. **Creating a JTabbedPane**: Tabs are created with labels containing content.
2. **Displaying the Frame**: The tabbed pane is displayed in a JFrame.

14. Swing Utilities

The SwingUtilities class provides utility methods for working with Swing components and ensuring thread safety in Swing applications. Since Swing is not thread-safe, you must ensure that any updates to Swing components are performed on the Event Dispatch Thread (EDT). SwingUtilities helps with this by providing methods to manage UI updates in a safe manner.

Key Methods in SwingUtilities

1. **invokeLater(Runnable runnable)**: This method ensures that the provided Runnable is executed on the EDT as soon as possible, without blocking the current thread.
2. **invokeAndWait(Runnable runnable)**: This method also schedules the Runnable on the EDT, but the calling thread waits for the task to complete before continuing. This method is rarely used as it may lead to deadlocks if misused.

SwingUtilities.invokeLater()

The invokeLater() method is the most commonly used method in Swing applications for ensuring that a block of code is executed on the EDT. Since the EDT handles all UI updates, failure to use invokeLater() can result in race conditions, inconsistent UI updates, or freezing of the application.

Example of invokeLater()

```
import javax.swing.*;

public class SwingUtilitiesExample {

   public static void main(String[] args) {
      // Create a Runnable to perform a UI task
      Runnable createUI = new Runnable() {
         public void run() {
            JFrame frame = new JFrame("SwingUtilities Example");
```

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

    // Schedule the task for execution on the EDT using invokeLater()
    SwingUtilities.invokeLater(createUI);
  }
}
```

Explanation of the Example

1. **Creating a Runnable**: A Runnable is created to define the task, which in this case is the creation of a simple Swing UI with a button.
2. **invokeLater()**: The Runnable is passed to SwingUtilities.invokeLater(), ensuring that the UI creation happens on the EDT.

Why Use invokeLater()?

- **Ensures Thread Safety**: Since all UI updates must happen on the EDT, invokeLater() ensures that the code block passed to it will be executed on the correct thread.
- **Prevents Freezing**: If long-running tasks block the EDT, the UI will become unresponsive. Using invokeLater() prevents tasks from locking the main thread.

Detailed Workflow of invokeLater()

1. **Non-UI Thread**: When you execute invokeLater() from a background thread (non-EDT), it places the task in a queue to be processed by the EDT.
2. **Execution on EDT**: The EDT eventually picks up the task and executes it, ensuring that all updates to the UI are thread-safe.

This approach is used in situations where the main application is running background threads (such as reading a file or fetching data from the network), and you want to update the UI with the result when the task is completed.

SwingUtilities.invokeAndWait()

The invokeAndWait() method is similar to invokeLater(), but it blocks the calling thread until the task is completed on the EDT. This is used in rare cases when the calling thread must wait for the result of a UI task before proceeding.

Example of invokeAndWait():

```
import javax.swing.*;

public class SwingUtilitiesInvokeAndWaitExample {

  public static void main(String[] args) {
    try {
      // Call invokeAndWait to ensure the UI is built on the EDT
      SwingUtilities.invokeAndWait(new Runnable() {
        public void run() {
          JFrame frame = new JFrame("invokeAndWait Example");
          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          frame.setSize(300, 200);
```

```
                JButton button = new JButton("Click Me");
                frame.add(button);
                frame.setVisible(true);
            }
        });
    } catch (Exception e) {
        e.printStackTrace();
    }

    System.out.println("UI created and task completed!");
  }
}
```

Explanation of the Example

1. **invokeAndWait()**: The method ensures that the Runnable task (creating the UI) is executed on the EDT.
2. **Blocking Behavior**: Unlike invokeLater(), the calling thread is blocked until the UI task completes, ensuring the next line of code (printing the message) doesn't execute until the UI is fully created.

When to Use invokeAndWait()?

- **When UI Must Be Updated Immediately**: You may use invokeAndWait() when the main thread needs to wait for a UI update before proceeding.
- **Avoid in Long-Running Tasks**: It should not be used for time-consuming tasks on the EDT as it can cause deadlocks or UI freezes.

Best Practices for Using invokeLater() and invokeAndWait()

1. **Use invokeLater() for Non-Critical Updates**: Most UI updates should be done with invokeLater(). It does not block the main thread, ensuring smooth UI performance.
2. **Limit invokeAndWait() Usage**: Only use invokeAndWait() when absolutely necessary, as it blocks the calling thread, which can potentially lead to deadlocks.
3. **Perform Heavy Tasks Off the EDT**: Long-running tasks, such as network operations or file I/O, should always be executed in background threads (like with SwingWorker), and their results should be passed to invokeLater() for UI updates.
4. **Check EDT Status**: Before modifying the UI directly, always ensure you are on the EDT by using SwingUtilities.isEventDispatchThread(). If not, switch to invokeLater() to execute the task safely.