**Java Swing Assignment-2**

**1.Custom Painting and Graphics:**

**i.Creating a Custom Component for Custom Painting:** To create a custom component in Swing that performs custom painting, you need to subclass a Swing component such as JPanel and override its paintComponent(Graphics g) method. This method is called whenever the component needs to be repainted, such as when it is first displayed, resized, or otherwise rendered.

**ii.Role of the paintComponent() Method:** The paintComponent() method is central to custom painting in Swing. It is responsible for all the drawing on the component. This method provides a Graphics object, which acts as a drawing context. You can use this object to draw shapes, text, and images on the component. It’s important to call super.paintComponent(g) within this method to ensure that the component's background and other default elements are drawn before your custom painting.

**iii.Optimizing Painting in Swing Applications:** To handle performance issues in painting, you can:

* + Use double buffering (enabled by default in Swing).
  + Minimize the area that needs to be repainted by calling repaint(Rectangle r) with a smaller region.
  + Avoid complex painting operations and reduce the number of objects created during painting.
  + Cache computed values and images if they do not change frequently.

**2.Concurrency and Swing:**

**i.Issues with Concurrency in Swing:** Swing is not thread-safe, meaning that updates to the GUI should only be done on the Event Dispatch Thread (EDT). Potential issues include race conditions, inconsistent GUI states, and deadlocks if Swing components are updated from multiple threads.

**ii.Using the SwingWorker Class:** SwingWorker is designed to perform background tasks while keeping the GUI responsive. It allows you to execute lengthy operations in a background thread and then update the GUI on the EDT. This separation ensures that the GUI remains responsive during long-running tasks.

**iii.Event Dispatch Thread (EDT):** The EDT is responsible for handling all the GUI-related events and updates. It is important because all Swing component updates must occur on this thread to maintain thread safety and avoid inconsistencies in the GUI. Use SwingUtilities.invokeLater() or SwingUtilities.invokeAndWait() to ensure that updates are executed on the EDT.

**3.Event Handling and Listeners:**

**i.Types of Event Listeners:** Swing provides various event listeners such as ActionListener, MouseListener, KeyListener, and FocusListener to handle different types of events. Custom events can be created by defining custom event classes and listener interfaces, then firing the events from within components.

**Example of MouseListener and MouseMotionListener:**

JPanel panel = new JPanel();

panel.addMouseListener(new MouseAdapter() {

@Override

public void mousePressed(MouseEvent e) {

// Handle mouse press

}

});

panel.addMouseMotionListener(new MouseMotionAdapter() {

@Override

public void mouseDragged(MouseEvent e) {

// Handle mouse drag

}

});

**4.MVC Architecture in Swing:**

**i.Applying MVC in Swing Components:** The MVC design pattern separates the application logic into three components: Model (data), View (UI), and Controller (business logic). Swing components like JTable follow this pattern, where the table model represents the data, the JTable itself is the view, and the controller manages the interactions.

**ii.Separating Model, View, and Controller:**

* + **Model:** Represents the data and business logic.
  + **View:** Renders the UI components.
  + **Controller:** Handles user input and updates the model.

**Example:**

// Model

public class MyModel {

private int value;

public int getValue() { return value; }

public void setValue(int value) { this.value = value; }

}

// View

public class MyView extends JFrame {

private JButton button = new JButton("Click Me");

public MyView() {

add(button);

}

public JButton getButton() { return button; }

}

// Controller

public class MyController {

private MyModel model;

private MyView view;

public MyController(MyModel model, MyView view) {

this.model = model;

this.view = view;

view.getButton().addActionListener(e -> updateModel());

}

public void updateModel() {

model.setValue(model.getValue() + 1);

}

}

**5.Advanced Components and Layouts:**

**i.Creating a Custom Layout Manager:** To create a custom layout manager, implement the LayoutManager or LayoutManager2 interface and define methods such as addLayoutComponent(), removeLayoutComponent(), preferredLayoutSize(), minimumLayoutSize(), and layoutContainer().

**ii.Comparison of Layout Managers:**

* + **BorderLayout:** Divides the container into five regions (North, South, East, West, Center). Suitable for simple layouts with distinct regions.
  + **GridBagLayout:** Provides a flexible grid-based layout with complex constraints. Ideal for complex forms with varying component sizes.
  + **GroupLayout:** Designed for use with GUI builders, it groups components and aligns them horizontally and vertically. Useful for complex, visually consistent layouts.

**iii.Implementing a Custom JTable Model:** Subclass AbstractTableModel and override methods like getRowCount(), getColumnCount(), and getValueAt(). This allows for displaying complex data structures in a table.

**6.Look and Feel Customization:**

**i.Changing the Look and Feel:** Use UIManager.setLookAndFeel() to change the look and feel of a Swing application. You can choose from built-in options like Metal, Nimbus, and system look and feels, or create custom look and feels by subclassing LookAndFeel and overriding its methods.

**ii.UIManager and UIDefaults:** UIManager manages the look and feel settings of Swing components. UIDefaults is a hashtable that stores default settings and can be customized to change the appearance of components globally.

**7.Dialogs and Menus:**

**i.Creating Custom Dialogs:** Use JDialog to create custom dialogs. Modal dialogs block user interaction with other windows until dismissed. Customize dialogs by adding components and setting properties.

**ii.Context-Sensitive Popup Menus:** Create JPopupMenu and add JMenuItems. Use a MouseListener to show the popup menu at the appropriate location based on user interaction.

**8.Animation and Timers:**

**i.Implementing Animations:** Use javax.swing.Timer to create animations. This class triggers action events at specified intervals, allowing you to update and repaint components to create the illusion of movement.

**Example of Animation:**

Timer timer = new Timer(100, e -> {

// Update position of shape

// Repaint component

});

timer.start();

**9.Handling High DPI Displays:**

**i.Challenges and Solutions:** High DPI displays can cause UI components to appear too small. Address this by:

* + Using vector graphics or scalable images.
  + Setting appropriate font sizes and scaling factors.
  + Updating the look and feel to support high DPI settings.

**10.Integration with Other Technologies:**

**i.Embedding JavaFX Components:** Use JFXPanel to embed JavaFX components in a Swing application. This allows for interoperability between the two technologies, combining Swing's mature features with JavaFX's modern UI capabilities.

**ii.Benefits and Drawbacks of JavaFX vs. Swing:**

* + **Benefits:** JavaFX offers modern UI controls, better styling with CSS, and improved graphics capabilities.
  + **Drawbacks:** Swing is more mature, widely used, and has better integration with legacy applications. JavaFX can be more complex for simple applications.

**11. Basic GUI Application:**

**i.Creating a Simple Swing Application:**

To create a basic Java Swing application, you need to:

- Create a `JFrame` to serve as the main window.

- Add a `JLabel` to display the "Hello, World!" message.

- Add a `JButton` and register an `ActionListener` with it.

- In the `ActionListener`, update the `JLabel` text to change the message when the button is clicked.

**12. Event Handling:**

**i.Swing Application with Text Field and Button:**

To create this application:

- Create a `JFrame` and add a `JTextField` for user input.

- Add a `JButton` and register an `ActionListener`.

- Add a `JLabel` below the button to display the text.

- In the `ActionListener`, retrieve the text from the `JTextField` and set it to the `JLabel`.

**13. Layout Management:**

**i.Designing a Form with User Input Fields:**

To design a form:

- Create a `JFrame` and choose an appropriate layout manager such as `GridBagLayout` or `GroupLayout`.

- Add `JTextField`s for name, email, and password input.

- Add a `JButton` labeled "Submit".

- Arrange the components using the layout manager to ensure a neat and user-friendly interface.

**14. Custom Component:**

**i.Creating a Custom Swing Component:**

To implement a custom component:

- Subclass `JComponent` or `JPanel`.

- Override the `paintComponent(Graphics g)` method to draw a colored rectangle.

- Add buttons or a `JColorChooser` to allow users to change the color.

- In the button or color chooser event handler, update the component's state and call `repaint()`.

**15. JTable Example:**

**Displaying Data in a JTable:**

To create this application:

- Create a `JFrame` and add a `JTable` to display student data (name, age, grade).

- Implement a custom `TableModel` to manage the data.

- Add buttons for adding, editing, and deleting rows.

- Use appropriate dialogs or input fields to manage row data and update the `TableModel`.

**16. File Chooser:**

**i.Opening a File Chooser Dialog:**

To develop this application:

- Create a `JFrame` and add a `JButton` to open the file chooser.

- Register an `ActionListener` with the button to display a `JFileChooser` dialog.

- When a file is selected, retrieve its path and size.

- Display the file information in a `JLabel`.

**17. Drawing Application:**

**i.Building a Basic Drawing Application:**

To build this application:

- Create a `JFrame` and add a custom `JPanel` for drawing.

- Override the `paintComponent(Graphics g)` method in the custom panel to draw shapes.

- Add controls for selecting shapes, colors, and clearing the canvas.

- Use mouse listeners to handle drawing actions based on user input.

**18. Dialog Boxes:**

**i.Creating a Custom Dialog Box:**

To create a custom dialog:

- Create a `JFrame` and add a menu with an "About" option.

- Implement a `JDialog` to display application information.

- Configure the `JDialog` to be modal to block input to other windows until it is closed.

- Add components to the dialog to display relevant information.

**19. Timer-Based Animation:**

**i.Animating an Object with Timer:**

To write this program:

- Create a `JFrame` and a custom `JPanel`.

- Use a `javax.swing.Timer` to update the position of the object (e.g., a ball) at regular intervals.

- Override the `paintComponent(Graphics g)` method in the custom panel to draw the object.

- In the timer's action listener, update the object's position and call `repaint()`.

**20. Tab-based Interface:**

**i.Implementing a Tabbed Pane Interface:**

To implement this interface:

- Create a `JFrame` and add a `JTabbedPane`.

- Add tabs to the `JTabbedPane`, each containing different content like a form, table, or drawing canvas.

- Configure each tab with appropriate components and layouts to display the desired content.

**21. Look and Feel:**

**Changing Look and Feel at Runtime:**

To create this application:

- Create a `JFrame` and add a dropdown menu for look and feel options.

- Use `UIManager` to change the look and feel based on the selected option.

- Include built-in look and feels like system, Nimbus, and any custom look and feels.

- Update the UI to apply the new look and feel.

**22. Split Pane:**

**Using JSplitPane to Divide Window:**

To develop this application:

- Create a `JFrame` and add a `JSplitPane`.

- Add components to each side of the `JSplitPane`.

- Allow users to resize the split using the divider.

- Dynamically update content in both sections based on user interactions.

**23. Tree and List Components:**

**Displaying File System Structure:**

To design this application:

- Create a `JFrame` and add a `JTree` to display the file system structure.

- Add a `JList` to display files in the selected directory.

- Implement functionality to open files from the list by adding appropriate event listeners.

**24. Drag-and-Drop Functionality:**

**Implementing Drag-and-Drop.**

To create this application:

- Create a `JFrame` and add components like a list and a panel.

- Implement drag-and-drop functionality by adding `TransferHandler` to the components.

- Handle the data transfer process to move items from the list to the panel or other components.

**25. Custom Painting:**

**Overriding paintComponent() for Custom Scene:**

To write this application:

- Create a `JFrame` and a custom `JPanel`.

- Override the `paintComponent(Graphics g)` method in the custom panel to draw a scene like a house or landscape.

- Add controls to modify scene elements, such as changing colors or sizes.

- Update the custom panel based on user inputs and call `repaint()` to reflect changes.