Q1. Swing Components and Containers

a) Differentiate between heavyweight and lightweight components in Java Swing

Heavyweight Components: - Rely on native peer components from the underlying OS - Each component creates its own opaque native window - More resource-intensive - Examples: Frame, Dialog, Window

Lightweight Components: - Written entirely in Java - Don't rely on native peers - Share the same opaque window - More efficient and flexible - Examples: JButton, JLabel, JPanel

b) Role of the JPanel container in Swing applications

JPanel is a general-purpose lightweight container that: 1. Provides an area to organize and group other components 2. Helps with layout management by acting as a sub-container 3. Can be used to divide complex GUIs into logical sections 4. Supports double buffering for smooth painting 5. Can have its own layout manager different from the parent container

c) Four commonly used Swing components and their purposes

1. **JButton**: Creates clickable buttons that trigger actions when pressed
2. **JLabel**: Displays uneditable text or images (often used for descriptions)
3. **JTextField**: Allows single-line text input from the user
4. **JComboBox**: Provides a drop-down list of selectable items

d) How JFrame class is utilized to create a main window

JFrame is used as follows: 1. Create an instance: `JFrame frame = new JFrame("Title");` 2. Set size: `frame.setSize(width, height);` 3. Set layout manager (optional): `frame.setLayout(new FlowLayout());` 4. Add components: `frame.add(component);` 5. Set default close operation: `frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);` 6. Make visible: `frame.setVisible(true);`

JFrame provides the main application window with title bar, borders, and other window decorations.

e) Significance of setDefaultCloseOperation() method in JFrame

The `setDefaultCloseOperation()` method determines what happens when the user clicks the close button (X) on the JFrame. Common options: - `JFrame.EXIT_ON_CLOSE`: Terminates the application - `JFrame.HIDE_ON_CLOSE`:

Hides the frame (default) - `JFrame.DISPOSE_ON_CLOSE`: Releases resources but may not exit - `JFrame.DO_NOTHING_ON_CLOSE`: Requires manual handling

This method is crucial for controlling application behavior on window closure.

Q2. Layout Managers (5 Marks)

a) Compare FlowLayout and BorderLayout in terms of component arrangement.

- **FlowLayout**:
  - Arranges components in a **left-to-right, top-to-bottom** flow.
  - Components are placed in the order they are added.
  - If the container is resized, components may wrap to the next line.
  - Default layout for `JPanel`.
- **BorderLayout**:
  - Divides the container into **five regions**: `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER`.
  - Each region can hold **only one component**.
  - Expands components to fill their respective regions.
  - Default layout for `JFrame`'s content pane.

b) How does GridLayout manage components within a container?

- **GridLayout** divides the container into a **grid of equally sized cells** (rows × columns).
- Components are added **left-to-right, top-to-bottom** in the grid.
- All components **resize to fill their cell**, ensuring uniform size.
- Example: `new GridLayout(2, 3)` creates a 2-row × 3-column grid.

c) Provide a code snippet to set a BoxLayout for a JPanel along the Y-axis.

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

d) What is the default layout manager for a JFrame's content pane?

- The default layout manager for `JFrame`'s content pane is **`BorderLayout`**.

e) Explain how CardLayout can be used to switch between different panels in a GUI.

- **CardLayout** allows multiple panels (cards) to share the same display space.
- Only **one panel is visible at a time**.

- Switching between panels is done using methods like:
  - `next(Container)` → Shows the next card.
  - `previous(Container)` → Shows the previous card.
  - `show(Container, String)` → Displays a specific card by name.
- Example:

```
JPanel cards = new JPanel(new CardLayout());
cards.add(new JPanel(), "Panel1");
cards.add(new JPanel(), "Panel2");

// Switch to "Panel2"
CardLayout cl = (CardLayout) cards.getLayout();
cl.show(cards, "Panel2");
```

This allows for **tab-like navigation** in Swing applications.

Q3. Swing Event Handling (5 Marks)

a) Define the delegation event model in Java Swing. The **Delegation Event Model** is a mechanism in Java Swing where: - **Event Sources** (e.g., buttons, text fields) generate events (e.g., clicks, key presses). - **Event Listeners** (interfaces like `ActionListener`, `MouseListener`) are registered to handle these events. - When an event occurs, the **source delegates the event** to the registered listener for processing. - Promotes **loose coupling** between event sources and event handlers.

b) List the steps involved in handling an event in Swing.

1. **Identify the event source** (e.g., a `JButton`).
2. **Implement the appropriate listener interface** (e.g., `ActionListener`).
3. **Register the listener** with the event source using `addXxxListener()` (e.g., `button.addActionListener()`).
4. **Override the required methods** (e.g., `actionPerformed()` for `ActionListener`).
5. **Write event-handling logic** inside the overridden method.

c) What is the purpose of the ActionListener interface? Provide an example scenario.

- **Purpose**: `ActionListener` is used to handle **action events**, such as button clicks or menu selections.
- **Example Scenario**:

```
JButton button = new JButton("Click Me!");
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
```

```
        System.out.println("Button clicked!");
    }
});
```

Here, clicking the button triggers `actionPerformed()`, printing a message.

d) Differentiate between MouseListener and MouseMotionListener interfaces.

MouseListener	MouseMotionListener
Handles **basic mouse events** (clicks, enters, exits).	Handles **mouse movement & dragging** events.
Methods: `mouseClicked()`, `mousePressed()`, `mouseReleased()`, `mouseEntered()`, `mouseExited()`.	Methods: `mouseMoved()`, `mouseDragged()`.
Example: Detecting a button click.	Example: Tracking mouse drag for drawing.

e) Explain how anonymous inner classes can be used for event handling in Swing.

- **Anonymous Inner Classes** allow defining a listener **inline** without creating a separate class.
- **Example**:

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

- **Advantages**:
  - **Concise**: No need for a separate class.
  - **Direct access** to outer class variables (if `final` or effectively final).
- **Disadvantage**: Can lead to **less readable code** if overused.

Q4. Event Classes and Listener Interfaces (5 Marks)

a) Match the following event classes with their corresponding listener interfaces:

Event Class	Listener Interface
ActionEvent	`ActionListener`
ItemEvent	`ItemListener`
KeyEvent	`KeyListener`
MouseEvent	`MouseListener` / `MouseMotionListener`
WindowEvent	`WindowListener` / `WindowStateListener`

b) What method must be implemented when using the ActionListener interface?

- The **`actionPerformed(ActionEvent e)`** method must be implemented.
- Example:

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

c) Describe a scenario where a FocusListener would be appropriately used.

- **FocusListener** is used when tracking **focus changes** on components (e.g., `JTextField`, `JButton`).
- **Example Scenario**:
  - Validating user input **when a text field loses focus** (e.g., checking if an email is properly formatted).
  - Changing the UI (e.g., highlighting a `JTextField` when it gains focus).
- **Methods**:
  - `focusGained(FocusEvent e)` → Called when component gets focus.
  - `focusLost(FocusEvent e)` → Called when component loses focus.

d) How does the ItemListener interface respond to item selection changes?

- **ItemListener** is triggered when **selectable components** (e.g., `JCheckBox`, `JComboBox`, `JRadioButton`) change state.
- **Method to implement**:

```
void itemStateChanged(ItemEvent e)
```

- **Example**:

```
JCheckBox checkBox = new JCheckBox("Enable Feature");
checkBox.addItemListener(new ItemListener() {
```

```
    @Override
    public void itemStateChanged(ItemEvent e) {
        if (e.getStateChange() == ItemEvent.SELECTED) {
            System.out.println("Checkbox selected!");
        }
    }
});
```

e) Provide a brief explanation of the WindowAdapter class and its advantages.

- **WindowAdapter** is an **abstract adapter class** that implements `WindowListener` with **empty default methods**.
- **Advantages**:
  - Avoids forcing developers to implement **all methods** of `WindowListener`.
  - Only required methods (e.g., `windowClosing()`) need to be overridden.
- **Example**:

```
frame.addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e) {
        System.exit(0); // Close application on window close
    }
});
```