Unit-06

Introducing Swing and Java Database Connectivity (JDBC)

6.1 Design philosophy of Swing

Swing, a part of Java's Standard Library (javax.swing), is a GUI widget toolkit for creating graphical user interfaces. It follows a specific design philosophy that sets it apart from other GUI toolkits. The design philosophy of Swing can be summarized as follows:

- Platform Independence
- Pluggable Look and Feel (PLAF)
- Lightweight Components
- MVC Architecture
- Single-Threaded Model
- Customizability and Extensibility
- Rich Set of Components
- Event-Driven Programming
- Accessibility
- Internationalization

6.2 Components and Containers

Components

Components in Java represent the building blocks of graphical user interfaces (GUIs). They are objects that interact with the user and display information. Examples of components include buttons, labels, text fields, checkboxes, radio buttons, and sliders. Each component has its own set of properties and behaviors that define how it looks and responds to user input.

Containers

Containers in Java GUI programming are components that can contain other components. They provide structure and organization to the GUI by holding and arranging components within them. Examples of containers include frames (windows), panels, dialogs, and applets. Containers manage the layout of their child components, determining how components are positioned and sized relative to each other within the container.

Components are individual elements that interact with users or display information (e.g., buttons, text fields) whereas Containers are components themselves that can hold other components, providing structure and layout management for GUIs (e.g., frames, panels).

```
import javax.swing.*;


public class ComponentsAndContainers {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Example Frame");
        JButton button = new JButton("Click Me");
        JLabel label = new JLabel("Hello, World!");
        frame.add(button); // Adding a button to the frame (container)
        frame.add(label);  // Adding a label to the frame (container)
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }  }
```

6.3 Layout Managers

Layout managers in Java GUI programming are responsible for determining the size and position of components within containers (such as frames, panels, and dialogs). They provide automatic layout capabilities, allowing developers to design flexible and resizable graphical user interfaces that adjust to different screen sizes and user preferences.

Common Layout Managers in Java:

1. **FlowLayout**: Arranges components in a row, wrapping to the next row if necessary.
2. **BorderLayout**: Divides the container into five regions: North, South, East, West, and Center.
3. **GridLayout**: Arranges components in a grid of rows and columns.

4. **BoxLayout**: Arranges components either vertically or horizontally in a single line.
5. **GridBagLayout**: Offers a flexible and powerful grid-based layout, allowing precise control over component positioning and sizing.
6. **CardLayout**: Manages multiple components (cards) stacked on top of each other, showing only one at a time.

Example of Using Layout Managers:

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


public class LayoutManager {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Layout Manager Example");
        JPanel panel = new JPanel(new BorderLayout());
        JButton northButton = new JButton("North");
        panel.add(northButton, BorderLayout.NORTH);
        JButton southButton = new JButton("South");
        panel.add(southButton, BorderLayout.SOUTH);
        JButton eastButton = new JButton("East");
        panel.add(eastButton, BorderLayout.EAST);
        JButton westButton = new JButton("West");
        panel.add(westButton, BorderLayout.WEST);
        JButton centerButton = new JButton("Center");
        panel.add(centerButton, BorderLayout.CENTER);
        frame.add(panel);
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }   }
```

6.4 Swing Event Handling

Swing event handling in Java refers to the mechanism through which Swing components respond to user interactions, such as button clicks, mouse movements, key presses, and window events. It allows developers to define how their GUI components react to user actions by associating event listeners with these components.

Common Swing Event Listeners:

- **ActionListener**: Handles action events, such as button clicks.
- **MouseListener**: Handles mouse events, such as clicks, movements, and entering/exiting components.
- **KeyListener**: Handles keyboard events, such as key presses and releases.
- **FocusListener**: Handles focus events, such as component gaining or losing focus.
- **WindowListener**: Handles window events, such as opening, closing, and iconifying windows.

6.5 Basic Swing Components: JButton, JTextField, JCheckBox, JList

1. JButton

JButton represents a clickable button that triggers an action when pressed. It is used for performing actions such as submitting a form, initiating a process, or navigating to another screen.

2. JTextField

JTextField Provides a single-line text input field for users to enter text. It is used in forms or dialogs where users need to input textual data, such as entering usernames, passwords, or search queries.

3. JCheckBox

JCheckBox represents a checkable box that allows users to select or deselect an option. It is used when users need to make binary choices or select multiple options from a list, such as enabling/disabling features or selecting preference**s.**

4. JList

JList displays a list of items from which users can select one or more items. It is used for presenting lists of items, such as contacts, items in a shopping cart, or options in a dropdown menu.

6.6 Use Anonymous Inner Classes to Handle Events

Anonymous inner classes in Java are a type of inner class that allows you to declare and instantiate a class at the same time, without explicitly naming the class. They are particularly useful for implementing interfaces or extending classes on-the-fly where a separate named class would be overkill or unnecessary.

Handling Events with Anonymous Inner Classes

We'll create a `JFrame` with a `JButton` and a `JTextField`. When the button is clicked, it will retrieve text from the text field and display it in the console using an anonymous inner class for event handling.

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class AnonymousInnerClassExample {

    public static void main(String[] args) {

    JFrame frame = new JFrame("Event Handling with Anonymous Inner Class");

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    frame.setSize(300, 150);

    // Create components

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

    JTextField textField = new JTextField(20);

    // Add components to content pane
```

```
Container contentPane = frame.getContentPane();
contentPane.setLayout(new FlowLayout());
contentPane.add(new JLabel("Enter text: "));
contentPane.add(textField);
contentPane.add(button);
// Add ActionListener using anonymous inner class
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String text = textField.getText();
        System.out.println("Text entered: " + text);
    } });
// Display the JFrame
frame.setVisible(true);   }  }
```

6.7 The Design of JDBC

The Java Database Connectivity (JDBC) API is a standard Java API that allows Java applications to interact with relational databases. Its design revolves around providing a consistent, vendor-neutral interface for database access, ensuring portability and ease of use across different database systems. The various design principles and components that define JDBC are:

- **Interface-Based**: Provides a set of interfaces (`Connection`, `Statement`, `ResultSet`, etc.) for database interaction, ensuring portability across different databases.
- **DriverManager**: Manages database drivers dynamically at runtime, enabling Java applications to connect to specific databases based on the URL provided.

- **Connection Management**: Offers methods (`getConnection()`, `createStatement()`, `prepareStatement()`) for establishing connections, managing transactions, and accessing database metadata.
- **Statement Execution**: Supports executing SQL queries (`Statement`), parameterized queries (`PreparedStatement`), and stored procedures (`CallableStatement`).
- **Result Handling**: Facilitates navigation through query results (`ResultSet`), accessing column values, and retrieving metadata about result sets and database structure.
- **Error Handling**: Uses `SQLException` to manage errors during database operations, ensuring robust error recovery and application stability.

6.8 Executing SQL Statements

Executing SQL statements in JDBC involves several steps, including establishing a connection to the database, creating and executing SQL statements, processing the results (if any), and properly closing resources to release database and JDBC resources

```
import java.sql.*;
public class ExecuteSQL{
    public static void main(String[] args) {
        String jdbcUrl = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "username";
        String password = "password";
        try (Connection connection = DriverManager.getConnection(jdbcUrl, username, password);
             Statement statement = connection.createStatement()) {
            // Execute SELECT query
            ResultSet resultSet = statement.executeQuery("SELECT * FROM users");
            // Process ResultSet
            while (resultSet.next()) {
```

```
            int id = resultSet.getInt("id");
            String username = resultSet.getString("username");
            String email = resultSet.getString("email");
            System.out.println("ID: " + id + ", Username: " + username + ", Email: " +
email);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
  }
}
```

6.9 Query Execution

Query execution in JDBC involves preparing and executing SQL statements to interact with a database.

```
import java.sql.*;
public class QueryExecution {
  public static void main(String[] args) {
    String jdbcUrl = "jdbc:mysql://localhost:3306/mydatabase";
    String username = "username";
    String password = "password";
    try (Connection connection = DriverManager.getConnection(jdbcUrl, username,
password);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery("SELECT * FROM users")) {
        // Process ResultSet
        while (resultSet.next()) {
```

```
            int id = resultSet.getInt("id");
            String username = resultSet.getString("username");
            String email = resultSet.getString("email");
            System.out.println("ID: " + id + ", Username: " + username + ", Email: " +
email);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
  }
}
```