**Lab 1**

**Theory**

**Swing**

Swing is a part of Java's standard library designed to create graphical user interfaces (GUIs). Unlike the older Abstract Window Toolkit (AWT), Swing components are lightweight, meaning they are written entirely in Java and do not rely on native operating system components. Swing provides a rich set of GUI components like buttons, text fields, tables, and more, allowing for the creation of complex and visually appealing interfaces.

**JFrame**

A JFrame is the basic building block for any Swing application. It represents a window with a title bar, border, and buttons for closing, minimizing, and maximizing. A JFrame is a top-level container that can hold other GUI components. To create a JFrame, instantiate the JFrame class, set its size, define its default close operation, and make it visible.

**Components in Jframe**

Components like JLabel and JTextArea are used to display text and information within a Swing application. A JLabel is a display area for a short text string or an image, or both. A JTextArea is a multi-line area that displays plain text. These components are added to containers like JFrame to display information to the user.

Images can be displayed in Swing using the ImageIcon class. An ImageIcon is an implementation of the Icon interface that paints icons from images. You can load an image from a file or a URL and display it in components like JLabel

Swing provides the Graphics and Graphics2D classes for drawing 2D shapes, such as lines, rectangles, and ellipses. By overriding the paintComponent method of a JPanel or similar component, you can use these classes to render custom graphics. Graphics2D offers more advanced control over geometry, coordinate transformations, color management, and text layout.

**Layout Management**

Layout managers control the size and position of components within a container. Swing provides several layout managers, including:

* BorderLayout: Divides the container into five regions: North, South, East, West, and Center.
* FlowLayout: Arranges components in a left-to-right flow, similar to words on a page.
* GridLayout: Arranges components in a grid with equal-sized cells.
* BoxLayout: Arranges components either vertically or horizontally.

**Event Handling**

Event handling in Swing involves detecting and responding to user actions like button clicks, mouse movements, and key presses. Swing uses the delegation event model, which defines standard and consistent mechanisms to generate and process events.

**Event Handling Basics**

Event handling is based on three key components: event objects, event listeners, and event sources. An event object (e.g., ActionEvent) contains information about an event. An event listener (e.g., ActionListener) defines methods to handle specific events. An event source (e.g., JButton) generates events.

**Event Classes**

Event classes represent various types of events that can occur in a Swing application, such as ActionEvent for action events, MouseEvent for mouse events, and KeyEvent for keyboard events. These classes provide information about the event, such as its type, the component that generated it, and additional data.

**Lab 2**

**Theory**

**Event Listeners and Adapter Classes**

Event listeners are interfaces that define methods to handle events. For example, ActionListener has a single method, actionPerformed, which is called when an action event occurs. Adapter classes provide empty implementations of event listener interfaces, allowing developers to override only the methods they need.

**Swing and the MVC Design Pattern**

Swing components follow the Model-View-Controller (MVC) design pattern, which separates the representation of information from the user's interaction with it. The model represents the data, the view displays the data, and the controller handles user input. This separation allows for more modular, flexible, and maintainable code.

**Basic Swing Components**

Swing provides a wide range of basic components for building GUIs:

* JButton: A push button that can trigger an action event when clicked.
* JLabel: A display area for a short text string or an image, or both.
* JTextField: A single-line text field for user input.
* JTextArea: A multi-line area to display plain text.
* JCheckBox: A check box that can be either checked or unchecked.
* JRadioButton: A radio button that can be selected or deselected.
* JComboBox: A drop-down list of items.
* JList: A list of items that can be selected.

**Lab 3**

**Theory**

**JDBC**

JDBC is a Java-based API that allows Java applications to interact with relational databases. It provides methods to connect to a database, execute SQL queries, and retrieve results. JDBC abstracts the underlying database details, allowing developers to use a consistent API regardless of the database management system (DBMS) used.

JDBC is typically used for:

* **Connecting to a database:** Establishing a connection to the DBMS.
* **Executing SQL queries:** Running SQL statements to query or update the database.
* **Retrieving and processing results:** Fetching data from the database and handling it within the application.
* **Handling transactions:** Managing transactions to ensure data integrity.

To configure JDBC, you need:

* **JDBC Driver:** The appropriate driver for the database you are connecting to.
* **Database URL:** A string that specifies the database location and connection parameters.
* **Database Credentials:** Username and password to authenticate with the database.

**JDBC Driver Types**

JDBC drivers are categorized into four types based on their architecture and how they interact with the database:

* **Type 1: JDBC-ODBC Bridge Driver**
  + This driver translates JDBC calls into ODBC calls and relies on the ODBC driver to communicate with the database.
  + **Pros:** Easy to use and suitable for quick prototyping.
  + **Cons:** Not suitable for production environments due to performance overhead and platform dependence.
* **Type 2: Native-API Driver (partially Java driver)**
  + This driver converts JDBC calls into database-specific calls using native client-side libraries.
  + **Pros:** Better performance than Type 1 drivers.
  + **Cons:** Requires native libraries on the client machine, leading to platform dependence.
* **Type 3: Network Protocol Driver (pure Java driver)**
  + This driver translates JDBC calls into a database-independent network protocol, which is then translated to database-specific calls by a middleware server.
  + **Pros:** Platform-independent, and can be used for accessing multiple databases.
  + **Cons:** Requires a middleware server.
* **Type 4: Thin Driver (pure Java driver)**
  + This driver converts JDBC calls directly into database-specific protocol calls.
  + **Pros:** Platform-independent, no need for native libraries or middleware, and provides the best performance.
  + **Cons:** Database-specific, requiring a separate driver for each database.