UNIT-5

Note: For your exams, you can skip coding examples of Event Handling.

Event Handling

1. Event Delegation Model

The Event Delegation Model in Java is a design pattern used to handle events generated by GUI components. This model works by delegating the responsibility of handling events to dedicated objects called event listeners. This allows for a clean separation of event generation and event handling logic, making the code more modular and easier to maintain.

Key Components of the Event Delegation Model

1. **Event Source**: The GUI component that generates the event (e.g., a button, a text field).
2. **Event Object**: An object that encapsulates information about the event (e.g., ActionEvent, MouseEvent).
3. **Event Listener**: An object that implements one or more event listener interfaces to handle events (e.g., ActionListener, MouseListener).
4. **Event Listener Interface**: An interface that defines the methods that must be implemented by an event listener (e.g., actionPerformed for ActionListener).

How It Works

1. **Event Source Registration**: The event source (e.g., a button) registers one or more event listeners.
2. **Event Generation**: When the user interacts with the event source (e.g., clicking a button), an event is generated.
3. **Event Dispatching**: The event is dispatched to the registered event listeners.
4. **Event Handling**: The event listener handles the event by executing the appropriate method defined in the event listener interface.

Example

Here's a simple example to illustrate the Event Delegation Model in Java:

Step-by-Step Implementation

1. **Create the GUI Application**: Use Swing to create a basic GUI application with a button.
2. **Register an Event Listener**: Register an ActionListener to the button to handle click events.
3. **Implement the Event Handling Logic**: Define what happens when the button is clicked.

Code Example

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

public class EventDelegationModelExample {
   public static void main(String[] args) {
      // Create a JFrame (the main window)
      JFrame frame = new JFrame("Event Delegation Model Example");

      // Create a JButton (a button component)
      JButton button = new JButton("Click Me!");

      // Register an ActionListener with the button
      button.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent e) {
            // Handle the button click event
            System.out.println("Button clicked!");
         }
      });

      // Set the layout manager
      frame.setLayout(new FlowLayout());

      // Add the button to the frame
      frame.add(button);

      // Set the default close operation and size of the frame
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(300, 200);

      // Make the frame visible
      frame.setVisible(true);
```

```
    }
}
```

Explanation

1. **Event Source**: The JButton is the event source in this example.
2. **Event Object**: An ActionEvent object is created when the button is clicked.
3. **Event Listener**: An anonymous inner class implements the ActionListener interface and handles the event.
4. **Event Listener Interface**: The ActionListener interface defines the actionPerformed method, which is called when the button is clicked.

Benefits of the Event Delegation Model

- **Decoupling**: Separates event generation from event handling, making the code more modular.
- **Reusability**: Event listeners can be reused across different components and applications.
- **Maintainability**: Easier to maintain and update event handling logic without affecting the event source.

In summary, the Event Delegation Model in Java provides a robust and flexible way to handle events in GUI applications, promoting clean code architecture and ease of maintenance.

2. Classes

In Java event handling, several key classes are used to manage events. These classes represent various types of events and provide methods to access event details. The primary classes involved in Java event handling include:

1. **AWTEvent**: The root class for all AWT events.
2. **ActionEvent**: Represents events triggered by actions, such as button clicks.
3. **MouseEvent**: Represents mouse-related events, such as clicks and movements.
4. **KeyEvent**: Represents keyboard-related events, such as key presses.
5. **WindowEvent**: Represents events related to window operations, such as opening, closing, and activating a window.

6. **ItemEvent**: Represents events generated by item selection, such as checkbox and list selection.
7. **FocusEvent**: Represents focus-related events, such as gaining or losing focus.

1. AWTEvent

AWTEvent is the superclass for all AWT event classes. It encapsulates event information and is used to dispatch events to listeners.

Example:

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

public class AWTEventExample extends Frame {
    public AWTEventExample() {
        setSize(300, 200);
        setVisible(true);

        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent we) {
                dispose();
            }
        });
    }

    public static void main(String[] args) {
        new AWTEventExample();
    }
}
```

2. ActionEvent

ActionEvent is used for events generated by buttons, menu items, or other components that can trigger an action.

Example:

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

public class ActionEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ActionEvent Example");
        JButton button = new JButton("Click Me!");

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

        frame.setLayout(new FlowLayout());
        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

3. MouseEvent

MouseEvent handles mouse-related events, including mouse clicks, movements, and drags.

Example:

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

public class MouseEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("MouseEvent Example");
        JLabel label = new JLabel("Click inside the frame!");

        frame.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                System.out.println("Mouse clicked at coordinates: " + e.getX() + ", " + e.getY());
            }
        });

        frame.setLayout(new FlowLayout());
        frame.add(label);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

4. KeyEvent

KeyEvent is used for keyboard-related events, such as key presses and releases.

Example:

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

public class KeyEventExample extends JFrame {
    public KeyEventExample() {
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                System.out.println("Key pressed: " + KeyEvent.getKeyText(e.getKeyCode()));
            }
        });
    }

    public static void main(String[] args) {
        new KeyEventExample();
    }
}
```

5. WindowEvent

WindowEvent deals with events related to window operations, such as opening, closing, and activating a window.

Example:

```
import java.awt.*;
import java.awt.event.*;

public class WindowEventExample extends Frame {
    public WindowEventExample() {
        setSize(300, 200);
        setVisible(true);

        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent we) {
                System.out.println("Window is closing");
                dispose();
            }
        });
    }

    public static void main(String[] args) {
        new WindowEventExample();
    }
}
```

6. ItemEvent

ItemEvent represents events generated by item selection, such as checkbox and list selection.

Example:

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

public class ItemEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ItemEvent Example");
        JCheckBox checkBox = new JCheckBox("Select Me!");

        checkBox.addItemListener(new ItemListener() {
            @Override
            public void itemStateChanged(ItemEvent e) {
                if (e.getStateChange() == ItemEvent.SELECTED) {
                    System.out.println("Checkbox selected");
                } else {
                    System.out.println("Checkbox deselected");
                }
            }
        });

        frame.setLayout(new FlowLayout());
        frame.add(checkBox);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

7. FocusEvent

FocusEvent handles focus-related events, such as gaining or losing focus.

Example:

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

public class FocusEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("FocusEvent Example");
        JTextField textField = new JTextField("Click here to focus");

        textField.addFocusListener(new FocusAdapter() {
            @Override
            public void focusGained(FocusEvent e) {
                System.out.println("Text field gained focus");
            }

            @Override
            public void focusLost(FocusEvent e) {
                System.out.println("Text field lost focus");
            }
        });

        frame.setLayout(new FlowLayout());
        frame.add(textField);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

These classes play a crucial role in Java's event handling mechanism, providing a structured way to manage and respond to various user interactions within GUI applications.

3. Event Listener Interfaces

In Java event handling, event listener interfaces are a core concept. They define the methods that must be implemented to handle specific types of events. These interfaces ensure that event handling code is consistent and adheres to a particular contract. Here are some of the most commonly used event listener interfaces in Java:

1. ActionListener

The ActionListener interface is used for handling action events, typically from buttons, menu items, or other components that can trigger actions.

Example:

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

public class ActionListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ActionListener Example");
        JButton button = new JButton("Click Me!");

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

        frame.setLayout(new FlowLayout());
        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

2. MouseListener

The MouseListener interface handles mouse events such as clicks, presses, releases, enters, and exits.

Example:

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

public class MouseListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("MouseListener Example");
        JLabel label = new JLabel("Click inside the frame!");

        // Implementing the MouseListener interface
        frame.addMouseListener(new MouseListener() {
            @Override
            public void mouseClicked(MouseEvent e) {
                System.out.println("Mouse clicked at coordinates: " + e.getX() + ", " + e.getY());
            }

            @Override
            public void mousePressed(MouseEvent e) {
                // Not implemented
            }

            @Override
            public void mouseReleased(MouseEvent e) {
                // Not implemented
            }

            @Override
            public void mouseEntered(MouseEvent e) {
                // Not implemented
            }

            @Override
            public void mouseExited(MouseEvent e) {
                // Not implemented
            }
        });

        frame.setLayout(new FlowLayout());
        frame.add(label);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

3. KeyListener

The KeyListener interface handles keyboard events such as key presses, releases, and typing.

Example:

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

public class KeyListenerExample extends JFrame {
    public KeyListenerExample() {
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        // Implementing the KeyListener interface
        addKeyListener(new KeyListener() {
            @Override
            public void keyTyped(KeyEvent e) {
                System.out.println("Key typed: " + e.getKeyChar());
            }

            @Override
            public void keyPressed(KeyEvent e) {
                System.out.println("Key pressed: " + KeyEvent.getKeyText(e.getKeyCode()));
            }

            @Override
            public void keyReleased(KeyEvent e) {
                System.out.println("Key released: " + KeyEvent.getKeyText(e.getKeyCode()));
            }
        });
    }
    public static void main(String[] args) {
        new KeyListenerExample();
    }
}
```

4. WindowListener

The WindowListener interface handles window events such as opening, closing, activated, deactivated, etc.

Example:

```
import java.awt.*;
import java.awt.event.*;

public class WindowListenerExample extends Frame {
    public WindowListenerExample() {
        setSize(300, 200);
        setVisible(true);

        // Implementing the WindowListener interface
        addWindowListener(new WindowListener() {
            @Override
            public void windowOpened(WindowEvent e) {
                System.out.println("Window opened");
            }

            @Override
            public void windowClosing(WindowEvent e) {
                System.out.println("Window closing");
                dispose();
            }

            @Override
            public void windowClosed(WindowEvent e) {
                System.out.println("Window closed");
            }

            @Override
            public void windowIconified(WindowEvent e) {
                System.out.println("Window iconified");
            }

            @Override
            public void windowDeiconified(WindowEvent e) {
                System.out.println("Window deiconified");
            }

            @Override
            public void windowActivated(WindowEvent e) {
                System.out.println("Window activated");
            }

            @Override
            public void windowDeactivated(WindowEvent e) {
                System.out.println("Window deactivated");
            }
        });
    }

    public static void main(String[] args) {
        new WindowListenerExample();
    }
}
```

5. ItemListener

The ItemListener interface handles events generated by item selection, such as checkboxes and lists.

Example:

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

public class ItemListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ItemListener Example");
        JCheckBox checkBox = new JCheckBox("Select Me!");
        // Implementing the ItemListener interface
        checkBox.addItemListener(new ItemListener() {
            @Override
            public void itemStateChanged(ItemEvent e) {
                if (e.getStateChange() == ItemEvent.SELECTED) {
                    System.out.println("Checkbox selected");
                } else {
                    System.out.println("Checkbox deselected");
                }
            }
        });

        frame.setLayout(new FlowLayout());
        frame.add(checkBox);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

6. FocusListener

The FocusListener interface handles focus events, such as when a component gains or loses focus.

Example:

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

public class FocusListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("FocusListener Example");
        JTextField textField = new JTextField("Click here to focus");

        // Implementing the FocusListener interface
        textField.addFocusListener(new FocusListener() {
            @Override
            public void focusGained(FocusEvent e) {
                System.out.println("Text field gained focus");
            }

            @Override
            public void focusLost(FocusEvent e) {
                System.out.println("Text field lost focus");
            }
        });

        frame.setLayout(new FlowLayout());
        frame.add(textField);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

7. ComponentListener

The ComponentListener interface handles events related to component size, position, or visibility changes.

Example:

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

public class ComponentListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ComponentListener Example");
        JButton button = new JButton("Resize or Move the Frame");

        // Implementing the ComponentListener interface
        frame.addComponentListener(new ComponentListener() {
            @Override
            public void componentResized(ComponentEvent e) {
                System.out.println("Component resized");
            }
            @Override
            public void componentMoved(ComponentEvent e) {
                System.out.println("Component moved");
            }

            @Override
            public void componentShown(ComponentEvent e) {
                System.out.println("Component shown");
            }

            @Override
            public void componentHidden(ComponentEvent e) {
                System.out.println("Component hidden");
            }
        });

        frame.setLayout(new FlowLayout());
        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

These examples illustrate how to use various event listener interfaces to handle different types of events in Java. By implementing these interfaces, you can create responsive and interactive GUI applications.

4. Adapter Classes

In Java event handling, adapter classes provide default implementations for the methods in event listener interfaces. These classes are especially useful when you do not need to implement all the methods of an event listener interface. By extending an adapter class, you can override only the methods you need, thereby simplifying the code.

Common Adapter Classes

Some of the commonly used adapter classes in Java include:

1. **MouseAdapter**
2. **KeyAdapter**
3. **FocusAdapter**
4. **WindowAdapter**
5. **ComponentAdapter**

1. MouseAdapter

The MouseAdapter class provides empty implementations for all methods in the MouseListener and MouseMotionListener interfaces. You can extend this class to handle only the mouse events you are interested in.

Example:

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

public class MouseAdapterExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("MouseAdapter Example");
        JLabel label = new JLabel("Click or move the mouse inside the frame!");

        frame.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                System.out.println("Mouse clicked at coordinates: " + e.getX() + ", " + e.getY());
            }
        });

        frame.addMouseMotionListener(new MouseAdapter() {
            @Override
            public void mouseMoved(MouseEvent e) {
                System.out.println("Mouse moved at coordinates: " + e.getX() + ", " + e.getY());
            }
        });

        frame.setLayout(new FlowLayout());
        frame.add(label);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

2. KeyAdapter

The KeyAdapter class provides empty implementations for all methods in the KeyListener interface. You can extend this class to handle only the key events you are interested in.

Example:

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

public class KeyAdapterExample extends JFrame {
    public KeyAdapterExample() {
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                System.out.println("Key pressed: " + KeyEvent.getKeyText(e.getKeyCode()));
            }
        });
    }

    public static void main(String[] args) {
        new KeyAdapterExample();
    }
}
```

3. FocusAdapter

The FocusAdapter class provides empty implementations for all methods in the FocusListener interface. You can extend this class to handle only the focus events you are interested in.

Example:

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

public class FocusAdapterExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("FocusAdapter Example");
        JTextField textField = new JTextField("Click here to focus");

        textField.addFocusListener(new FocusAdapter() {
            @Override
            public void focusGained(FocusEvent e) {
                System.out.println("Text field gained focus");
            }

            @Override
            public void focusLost(FocusEvent e) {
                System.out.println("Text field lost focus");
            }
        });

        frame.setLayout(new FlowLayout());
        frame.add(textField);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

4. WindowAdapter

The WindowAdapter class provides empty implementations for all methods in the WindowListener interface. You can extend this class to handle only the window events you are interested in.

Example:

```java
import java.awt.*;
import java.awt.event.*;

public class WindowAdapterExample extends Frame {
    public WindowAdapterExample() {
        setSize(300, 200);
        setVisible(true);

        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                System.out.println("Window is closing");
                dispose();
            }
        });
    }

    public static void main(String[] args) {
        new WindowAdapterExample();
    }
}
```

5. ComponentAdapter

The ComponentAdapter class provides empty implementations for all methods in the ComponentListener interface. You can extend this class to handle only the component events you are interested in.

Example:

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

public class ComponentAdapterExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ComponentAdapter Example");
        JButton button = new JButton("Resize or Move the Frame");
        frame.addComponentListener(new ComponentAdapter() {
            @Override
            public void componentResized(ComponentEvent e) {
                System.out.println("Component resized");
            }

            @Override
            public void componentMoved(ComponentEvent e) {
                System.out.println("Component moved");
            }
        });

        frame.setLayout(new FlowLayout());
        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Benefits of Using Adapter Classes

1. **Simplicity**: Adapter classes simplify the process of event handling by allowing you to override only the methods you need, instead of implementing all methods of an interface.
2. **Readability**: Code becomes more readable and manageable since only relevant methods are overridden.
3. **Maintainability**: It is easier to maintain and update the event handling code as the focus is on specific methods.

By using adapter classes, developers can create more concise and focused event handling code, improving both development efficiency and code clarity.

Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is an API that enables Java applications to interact with a wide range of databases. JDBC provides methods for querying and updating data in a database, making it a vital tool for Java developers working with relational databases.

Key Components of JDBC

1. **JDBC Drivers**: These are the implementations provided by database vendors. They act as a bridge between the Java application and the database.
2. **Connection**: This interface represents a session with a specific database.
3. **Statement**: This interface is used to execute SQL queries.
4. **ResultSet**: This interface provides methods for navigating and retrieving results from a query.
5. **PreparedStatement**: This interface is used to execute precompiled SQL queries with parameter substitution.
6. **CallableStatement**: This interface is used to execute stored procedures in the database.

Steps to Connect to a Database using JDBC

1. **Load the JDBC Driver**: This step loads the driver into memory.
2. **Establish a Connection**: This step creates a connection to the database.
3. **Create a Statement**: This step creates a statement object to send SQL queries to the database.
4. **Execute the Query**: This step executes the SQL query and retrieves the results.
5. **Process the Results**: This step processes the results returned by the query.
6. **Close the Connection**: This step closes the database connection to free up resources.

Example Code

Here's a simple example to illustrate JDBC connectivity and basic operations like querying a database.

Step-by-Step Implementation

1. **Load the JDBC Driver**
2. **Establish a Connection**

3. **Create a Statement**
4. **Execute a Query**
5. **Process the Results**
6. **Close the Connection**

Code Example

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JDBCExample {
    public static void main(String[] args) {
        String jdbcUrl = "jdbc:mysql://localhost:3306/testdb";
        String username = "root";
        String password = "password";
        Connection connection = null;
        Statement statement = null;
        ResultSet resultSet = null;

        try {
            // 1. Load the JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");
            System.out.println("Driver loaded!");

            // 2. Establish a connection
            connection = DriverManager.getConnection(jdbcUrl, username, password);
            System.out.println("Connected to the database!");

            // 3. Create a statement
            statement = connection.createStatement();

            // 4. Execute a query
            String sql = "SELECT * FROM users";
            resultSet = statement.executeQuery(sql);

            // 5. Process the results
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                String email = resultSet.getString("email");
                System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 6. Close the connection
            try {
                if (resultSet != null) resultSet.close();
                if (statement != null) statement.close();
                if (connection != null) connection.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Explanation

1. **Load the JDBC Driver**: The Class.forName("com.mysql.cj.jdbc.Driver") line loads the MySQL JDBC driver.
2. **Establish a Connection**: The DriverManager.getConnection method establishes a connection to the database using the specified URL, username, and password.
3. **Create a Statement**: The connection.createStatement() method creates a Statement object for sending SQL statements to the database.
4. **Execute a Query**: The statement.executeQuery(sql) method executes the SQL query and returns a ResultSet object containing the results.
5. **Process the Results**: The while loop iterates through the ResultSet to retrieve and process the data.
6. **Close the Connection**: The finally block ensures that the ResultSet, Statement, and Connection objects are closed, releasing database resources.

Conclusion

JDBC provides a powerful and flexible way to connect Java applications to various databases. By understanding and utilizing JDBC's core components and methods, you can effectively manage database operations in your Java applications.