can you tell me step by step process to code this problem and also

Make sure your answers include:

- Aim
- Algorithm / Procedure
- Program / Implementation
- Output / Result
- Viva notes

The PDF has a full list of 15 Advanced Java problems but your question does not specify **which one** you need the step-by-step solution for.[1]

Need clarification

Here are the problems available in your file:[1]

- JSP + JSTL + EL product list with custom price tag
- File upload servlet (multipart/form-data) storing files on server
- Feedback form using Servlets + DB
- Filter that logs IP, time, URL before servlet
- Servlet using HttpSession to track visit count
- Multi-tier JSP/Servlet/JDBC app: register, login, profile
- RMI employee management (serialization)
- RMI calculator
- Socket-based client–server chat
- JDBC login system
- JDBC prepared statement: bulk insert + GUI table
- JDBC full CRUD on Student table
- Theme-switching GUI Look & Feel app
- Swing calculator
- MVC-based Swing form with validation

What you should do

Reply with **the exact question number and title** (for example: “Q5: servlet that tracks user visits using HttpSession”).[1]

Once you choose one, the answer will include, for that specific problem:

- Aim
- Algorithm / Procedure (clear steps)
- Program / Implementation (Java code)
- Output / Result (what you show in screenshot / observation table)
- Viva notes (short Q&A theory points)

⁂

question 1

Aim:
To create a JSP application that uses JSTL and EL to display a list of products fetched from a database using `<c:forEach>` and a custom tag for formatting prices.[2]

Algorithm / Procedure

1. Create a database table `products` with fields like `id`, `name`, `price` and insert sample rows.[2]
2. Write a DAO class in Java to connect to the database (JDBC) and return a `List<Product>`.[2]
3. In a servlet, call the DAO, get the product list, and set it as a request attribute (for example `request.setAttribute("products", list)`), then forward to `products.jsp`.[2]
4. Configure JSTL in the JSP by adding the core taglib directive and enable EL (default in JSP 2.x).[2]
5. Create a custom tag (simple tag handler class + TLD file) that takes a `price` attribute and outputs a formatted string like “₹1,234.50”.[2]
6. In `products.jsp`, iterate over the product list using `<c:forEach>` and display fields with EL, using the custom tag to format the price column.[2]
7. Deploy the web app on a servlet container (Tomcat), access the servlet URL, and verify the product list output.[2]

Program / Implementation

Below is a minimal example structure (you can adapt package names, DB config, and deployment descriptor as per your lab environment).

1) Product model

```
public class Product {
    private int id;
    private String name;
    private double price;

    public Product() {}
    public Product(int id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }
}
```

2) DAO class

```
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class ProductDAO {
    private static final String URL = "jdbc:mysql://localhost:3306/advanced_java";
    private static final String USER = "root";
    private static final String PASS = "root";

    public List<Product> getAllProducts() {
        List<Product> list = new ArrayList<>();

        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection con = DriverManager.getConnection(URL, USER, PASS);
            String sql = "SELECT id, name, price FROM products";
            PreparedStatement ps = con.prepareStatement(sql);
            ResultSet rs = ps.executeQuery();

            while (rs.next()) {
                Product p = new Product();
                p.setId(rs.getInt("id"));
                p.setName(rs.getString("name"));
                p.setPrice(rs.getDouble("price"));
                list.add(p);
            }

            rs.close();
            ps.close();
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

        return list;
    }
}
```

3) Servlet to fetch products

```
import java.io.IOException;
import java.util.List;
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import jakarta.servlet.annotation.WebServlet;

@WebServlet("/products")
public class ProductServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        ProductDAO dao = new ProductDAO();
        List<Product> products = dao.getAllProducts();

        request.setAttribute("products", products);
        RequestDispatcher rd = request.getRequestDispatcher("products.jsp");
        rd.forward(request, response);
    }
}
```

4) Custom tag to format price

Tag handler

```
import java.io.IOException;
import java.text.NumberFormat;
import java.util.Locale;
import jakarta.servlet.jsp.tagext.SimpleTagSupport;
import jakarta.servlet.jsp.JspWriter;

public class PriceFormatterTag extends SimpleTagSupport {
    private double value;

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

    public void doTag() throws IOException {
        JspWriter out = getJspContext().getOut();
        NumberFormat nf = NumberFormat.getCurrencyInstance(new Locale("en", "IN"));
        String formatted = nf.format(value);
        out.print(formatted);
    }
}
```

TLD file (e.g., `price.tld` in `WEB-INF`)

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib xmlns="https://jakarta.ee/xml/ns/jakartaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
                            https://jakarta.ee/xml/ns/jakartaee/web-jsptaglibrary_3_0.xsd
        version="3.0">
    <tlib-version>1.0</tlib-version>
    <short-name>price</short-name>
    <uri>http://example.com/tags/price</uri>

    <tag>
        <name>formatPrice</name>
        <tag-class>PriceFormatterTag</tag-class>
        <body-content>empty</body-content>
        <attribute>
            <name>value</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
            <type>double</type>
        </attribute>
    </tag>
</taglib>
```

5) JSP page (`products.jsp`)

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://example.com/tags/price" prefix="p" %>

<html>
<head>
    <title>Product List</title>
</head>
<body>
<h2>Product List</h2>

<table border="1" cellpadding="5">
    <tr>
        <th>ID</th>
        <th>Name</th>
        <th>Price</th>
    </tr>
    <c:forEach var="p" items="${products}">
        <tr>
            <td>${p.id}</td>
            <td>${p.name}</td>
            <td><p:formatPrice value="${p.price}" /></td>
        </tr>
    </c:forEach>
</table>

</body>
</html>
```

Ensure JSTL libraries are in `WEB-INF/lib` and that `web.xml` or annotations correctly map the servlet and TLD (if your container needs explicit mapping).

Output / Result

- When the user opens `/products`, the servlet loads all rows from the `products` table and forwards them to `products.jsp`.[2]
- The JSP displays an HTML table listing each product's ID, name, and a nicely formatted price (currency format) using the custom tag for every row processed by `<c:forEach>`.[2]

Viva notes

- JSTL (JavaServer Pages Standard Tag Library) provides ready-made tags for common tasks like iteration, conditionals, formatting, and SQL, making JSP pages cleaner than using scriptlets.[2]
- Expression Language (EL) allows simple access to Java objects stored in scopes (request, session, application) using syntax like `${products}` instead of Java code.[2]
- `<c:forEach>` is a core JSTL tag used to iterate over collections or arrays in JSP views.[2]
- Custom tags encapsulate reusable view logic in tag handlers, improving separation of concerns and making JSPs easier to maintain.[2]
- MVC pattern: the servlet acts as controller, DAO as model accessing the database, and JSP with JSTL/EL is the view that renders data.[2]

⁂

question 2

Aim:
To implement a file upload servlet that accepts files sent with `multipart/form-data` from a HTML form and stores the uploaded files on the server.[3]

Algorithm / Procedure

1. Create a HTML/JSP page with a form that uses `method="post"` and `enctype="multipart/form-data"` and contains an `<input type="file">` element.[3]
2. Configure a servlet with `@MultipartConfig` (or in `web.xml`) so it can handle multipart requests.[3]
3. In the servlet's `doPost`, obtain the uploaded file as a `Part` object using `request.getPart("fileFieldName")`.[3]

4. Read the original file name from the `Part` header and choose a server folder path (for example, a directory under the web app or some absolute path on disk).[3]
5. Use `Part.write(...)` or an input/output stream to save the file bytes from the request to the chosen directory on the server.[3]
6. After saving, send a response back to the client indicating success and optionally show the uploaded file name and path.[3]
7. Run the application on Tomcat, open the upload form, select a file, submit, and verify that the file appears in the target upload directory.[3]

Program / Implementation

1) Upload form (upload.jsp)

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
<head>
    <title>File Upload Example</title>
</head>
<body>
<h2>Upload a File</h2>
<form action="upload" method="post" enctype="multipart/form-data">
    Select file:
    <input type="file" name="file" required />
    <br/><br/>
    <input type="submit" value="Upload" />
</form>
</body>
</html>
```

2) File upload servlet (using Jakarta Servlet 5+ style)

```
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.MultipartConfig;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.Part;

@WebServlet("/upload")
@MultipartConfig
public class FileUploadServlet extends HttpServlet {

    private static final String UPLOAD_DIR = "uploads";
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        // Get the upload folder on the server
        String applicationPath = request.getServletContext().getRealPath("");
        String uploadPath = applicationPath + File.separator + UPLOAD_DIR;

        File uploadDir = new File(uploadPath);
        if (!uploadDir.exists()) {
            uploadDir.mkdir();
        }

        // Get the file part
        Part filePart = request.getPart("file");
        String fileName = getFileName(filePart);

        // Target file on disk
        File file = new File(uploadDir, fileName);

        // Save uploaded file bytes
        try (InputStream input = filePart.getInputStream()) {
            Files.copy(input, file.toPath(), StandardCopyOption.REPLACE_EXISTING);
        }

        response.setContentType("text/html");
        response.getWriter().println("<h3>File uploaded successfully!</h3>");
        response.getWriter().println("Saved as: " + file.getAbsolutePath());
    }

    // Extract file name from header
    private String getFileName(Part part) {
        String contentDisp = part.getHeader("content-disposition");
        String[] tokens = contentDisp.split(";");
        for (String token : tokens) {
            if (token.trim().startsWith("filename")) {
                return token.substring(token.indexOf('=') + 1).trim().replace("\"", "");
            }
        }
        return "unknown";
    }
}
```

If you use older `javax.servlet` instead of `jakarta.servlet`, just change the import package names accordingly.

Output / Result

- The browser shows an upload form where the user can choose a file and click “Upload”.[3]
- After submission, the servlet stores the file under the `uploads` directory inside the web application folder and responds with a confirmation message showing the saved file path.[3]

Viva notes

- `multipart/form-data` is required in the form's `enctype` attribute to send files in the HTTP request body.[3]
- The `@MultipartConfig` annotation (or equivalent configuration) enables the servlet to parse `Part` objects from the request representing uploaded files.[3]
- A `Part` gives metadata like file name and size, plus an `InputStream` for the file content, which can then be written to the server's file system.[3]
- File upload in servlets is typically handled via POST requests because they can carry large request bodies.[3]
- Security considerations include limiting file size, validating file type, and controlling upload paths to avoid overwriting important system files.[3]

⁂

question 9

Aim:
To build a simple client–server chat application in Java where a server accepts a connection from a client and both can exchange text messages using TCP sockets.[4]

Algorithm / Procedure

1. Create a server program that: opens a `ServerSocket` on a fixed port, waits with `accept()`, and obtains a `Socket` for the connected client.[4]
2. From this `Socket`, create input and output streams to read incoming messages and send outgoing responses.[4]
3. Run a loop on the server side to continuously read messages from the client and display them, and allow the server user to type replies that are sent back.[4]
4. Create a client program that: creates a `Socket` to connect to the server's IP and port, then obtains input and output streams for communication.[4]
5. Use two threads (or interleaved loops) on both client and server: one to read from the socket and print to console, another to read from keyboard and send to the other side.[4]
6. Define a termination condition (for example, when either side types `"exit"`) and then close streams, socket, and server socket gracefully.[4]
7. First start the server program, then run the client, and verify that typing messages at one end appears at the other end.[4]

Program / Implementation

Below is a simple console-based one-client chat (single client connected to server at a time).

1) ChatServer.java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class ChatServer {

    public static void main(String[] args) {
        int port = 5000;

        try (ServerSocket serverSocket = new ServerSocket(port)) {
            System.out.println("Server started. Waiting for client on port " + port + "..

            Socket socket = serverSocket.accept();
            System.out.println("Client connected: " + socket.getInetAddress());

            BufferedReader in = new BufferedReader(
                    new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

            BufferedReader console = new BufferedReader(
                    new InputStreamReader(System.in));

            // Thread to read messages from client
            Thread reader = new Thread(() -> {
                String msgFromClient;
                try {
                    while ((msgFromClient = in.readLine()) != null) {
                        System.out.println("Client: " + msgFromClient);
                        if ("exit".equalsIgnoreCase(msgFromClient)) {
                            System.out.println("Client ended the chat.");
                            break;
                        }
                    }
                } catch (IOException e) {
                    System.out.println("Connection closed.");
                }
            });
            reader.start();

            // Main thread sends messages to client
            String msgToClient;
            while ((msgToClient = console.readLine()) != null) {
                out.println(msgToClient);
                if ("exit".equalsIgnoreCase(msgToClient)) {
                    System.out.println("Chat ended.");
                    break;
                }
            }

            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2) ChatClient.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class ChatClient {

    public static void main(String[] args) {
        String host = "localhost"; // or server IP
        int port = 5000;

        try (Socket socket = new Socket(host, port)) {
            System.out.println("Connected to server.");

            BufferedReader in = new BufferedReader(
                    new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

            BufferedReader console = new BufferedReader(
                    new InputStreamReader(System.in));

            // Thread to read messages from server
            Thread reader = new Thread(() -> {
                String msgFromServer;
                try {
                    while ((msgFromServer = in.readLine()) != null) {
                        System.out.println("Server: " + msgFromServer);
                        if ("exit".equalsIgnoreCase(msgFromServer)) {
                            System.out.println("Server ended the chat.");
                            break;
                        }
                    }
                } catch (IOException e) {
                    System.out.println("Connection closed.");
                }
            });
            reader.start();

            // Main thread sends messages to server
            String msgToServer;
            while ((msgToServer = console.readLine()) != null) {
                out.println(msgToServer);
                if ("exit".equalsIgnoreCase(msgToServer)) {
                    System.out.println("Chat ended.");
                    break;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

How to run

1. Compile both classes and run `ChatServer` first on one terminal/PC.
2. Run `ChatClient` on the same or another machine (with `host` set to server IP).
3. Type messages in either terminal and see them appear on the other side; type `exit` to close.

Output / Result

- The server console shows “Server started. Waiting for client...” and then “Client connected” when the client connects.[4]
- Messages typed on the client side appear prefixed by “Client:” on the server console, and messages typed on the server appear prefixed by “Server:” on the client console, until either side types `exit` to end the chat.[4]

Viva notes

- Java sockets use the TCP protocol for reliable, connection-oriented communication between client and server.[4]
- `ServerSocket` listens on a port and `accept()` blocks until a client connects, returning a `Socket` used for communication.[4]
- The `Socket` provides input and output streams to read and write data over the network.[4]
- Multithreading is useful in chat applications so one thread can continuously listen for incoming messages while another handles user input and sending messages.[4]
- Typical chat applications define a protocol (for example, lines of text terminated by newline) and a termination keyword like `exit` to close the connection gracefully.[4]

⁂

question 11

Aim:
To write a Java program that uses JDBC prepared statements to insert multiple (bulk) records into a database table and then display the stored records in a Swing GUI table.[5]

Algorithm / Procedure

1. Create a database (for example `advanced_java`) and a table (for example `employees` with columns `id`, `name`, `salary`).[5]
2. Design a Swing form with input fields (ID, Name, Salary), an "Add" button to insert the record, and a `JTable` to display all records.[5]
3. Establish a JDBC connection to the database in a helper class using `DriverManager.getConnection(...)`.[5]
4. In the "Add" button handler, read values from text fields and prepare a SQL `INSERT` query using `PreparedStatement` with placeholders (`?`).[5]
5. Set parameter values on the `PreparedStatement`, execute it to insert each record, and optionally allow repeated inserts (bulk) by clicking "Add" multiple times or using `addBatch()` and `executeBatch()`.[5]
6. After insertion, execute a `SELECT` query using `PreparedStatement` to fetch all rows, and load the result set into a `DefaultTableModel` bound to the `JTable`.[5]
7. Refresh the table model each time records are inserted, so the GUI always shows the latest data.[5]

Program / Implementation

Below is a simple single-file example for clarity; in real labs you may separate DAO and GUI classes.

1) DB connection helper (optional)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBUtil {
    private static final String URL = "jdbc:mysql://localhost:3306/advanced_java";
    private static final String USER = "root";
    private static final String PASS = "root";

    public static Connection getConnection() throws SQLException, ClassNotFoundException
        Class.forName("com.mysql.cj.jdbc.Driver");
        return DriverManager.getConnection(URL, USER, PASS);
    }
}
```

2) GUI with bulk insert and table (JDBCPreparedGUI.java)

```
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.sql.*;
```

```java
public class JDBCPreparedGUI extends JFrame {

    private JTextField txtId, txtName, txtSalary;
    private JTable table;
    private DefaultTableModel model;

    public JDBCPreparedGUI() {
        setTitle("JDBC PreparedStatement - Bulk Insert & Display");
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        // Input panel
        JPanel inputPanel = new JPanel(new GridLayout(4, 2, 5, 5));
        inputPanel.add(new JLabel("ID:"));
        txtId = new JTextField();
        inputPanel.add(txtId);

        inputPanel.add(new JLabel("Name:"));
        txtName = new JTextField();
        inputPanel.add(txtName);

        inputPanel.add(new JLabel("Salary:"));
        txtSalary = new JTextField();
        inputPanel.add(txtSalary);

        JButton btnAdd = new JButton("Add Record");
        btnAdd.addActionListener(this::addRecord);
        inputPanel.add(btnAdd);

        JButton btnLoad = new JButton("Load Records");
        btnLoad.addActionListener(e -> loadTableData());
        inputPanel.add(btnLoad);

        // Table
        model = new DefaultTableModel(new String[]{"ID", "Name", "Salary"}, 0);
        table = new JTable(model);
        JScrollPane scrollPane = new JScrollPane(table);

        add(inputPanel, BorderLayout.NORTH);
        add(scrollPane, BorderLayout.CENTER);

        loadTableData();
    }

    private void addRecord(ActionEvent e) {
        String idStr = txtId.getText().trim();
        String name = txtName.getText().trim();
        String salaryStr = txtSalary.getText().trim();

        if (idStr.isEmpty() || name.isEmpty() || salaryStr.isEmpty()) {
            JOptionPane.showMessageDialog(this, "All fields are required");
            return;
        }
        int id = Integer.parseInt(idStr);
        double salary = Double.parseDouble(salaryStr);

        String sql = "INSERT INTO employees (id, name, salary) VALUES (?, ?, ?)";

        try (Connection con = DBUtil.getConnection();
             PreparedStatement ps = con.prepareStatement(sql)) {

            ps.setInt(1, id);
            ps.setString(2, name);
            ps.setDouble(3, salary);

            ps.executeUpdate(); // single insert

            JOptionPane.showMessageDialog(this, "Record inserted successfully");
            clearFields();
            loadTableData(); // refresh table

        } catch (Exception ex) {
            ex.printStackTrace();
            JOptionPane.showMessageDialog(this, "Error: " + ex.getMessage());
        }
    }

    private void loadTableData() {
        model.setRowCount(0); // clear table
        String sql = "SELECT id, name, salary FROM employees";
        try (Connection con = DBUtil.getConnection();
             PreparedStatement ps = con.prepareStatement(sql);
             ResultSet rs = ps.executeQuery()) {

            while (rs.next()) {
                Object[] row = {
                        rs.getInt("id"),
                        rs.getString("name"),
                        rs.getDouble("salary")
                };
                model.addRow(row);
            }

        } catch (Exception ex) {
            ex.printStackTrace();
            JOptionPane.showMessageDialog(this, "Error loading data: " + ex.getMessage())
        }
    }

    private void clearFields() {
        txtId.setText("");
        txtName.setText("");
        txtSalary.setText("");
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new JDBCPreparedGUI().setVisible(true));
    }
}
```

3) Example SQL for table (MySQL)

```
CREATE DATABASE advanced_java;
USE advanced_java;

CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    salary DOUBLE
);
```

To treat this as "bulk insert", you can enter several records one by one, or modify `addRecord` to use `addBatch()` if your teacher specifically wants a batch example.

Output / Result

- The GUI window shows text fields for ID, Name, Salary, buttons "Add Record" and "Load Records", and a table at the bottom. [5]
- After entering multiple records and clicking "Add Record" each time, the table displays all rows currently stored in the `employees` table, confirming insertion through JDBC prepared statements. [5]

Viva notes

- A `PreparedStatement` precompiles SQL with placeholders (`?`), which improves performance for repeated inserts and helps prevent SQL injection. [5]
- `setInt`, `setString`, `setDouble` and similar methods bind Java values to SQL parameters before execution. [5]
- `executeUpdate()` is used for `INSERT`, `UPDATE`, and `DELETE` queries, whereas `executeQuery()` returns a `ResultSet` for `SELECT` queries. [5]
- In a `JTable`, data is managed by a table model (here `DefaultTableModel`), and rows can be filled from a `ResultSet` to show DB data visually. [5]
- Using JDBC with Swing demonstrates integration of database operations with a desktop GUI, which is typical in small management systems. [5]

⁂

question 14

Aim:
To create a Swing-based calculator application that performs basic arithmetic operations using event-driven programming. [6]

Algorithm / Procedure

1. Design a `JFrame` with a `JTextField` display at the top and buttons for digits (0–9), operations (+, −, ×, ÷), clear, and equals, arranged using layouts like `BorderLayout` and `GridLayout`.[6]
2. Attach `ActionListener`s to each button so that when a button is clicked, the actionPerformed method updates the display or performs the selected operation.[6]
3. When a digit button is pressed, append the digit to the current text in the display; when an operator is pressed, store the first operand and operator, then clear or prepare the display for the second operand.[6]
4. When the equals button is pressed, parse the second operand, perform the operation based on the stored operator, and show the result in the display.[6]
5. Implement the clear button to reset operands, operator, and display; run the application and test all operations.[6]

Program / Implementation

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

public class SwingCalculator extends JFrame implements ActionListener {

    private JTextField display;
    private String firstOperand = "";
    private String operator = "";
    private boolean startNewNumber = true;

    public SwingCalculator() {
        setTitle("Swing Calculator");
        setSize(300, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        display = new JTextField();
        display.setEditable(false);
        display.setHorizontalAlignment(JTextField.RIGHT);
        display.setFont(new Font("Arial", Font.BOLD, 24));
        add(display, BorderLayout.NORTH);

        String[] buttons = {
                "7", "8", "9", "/",
                "4", "5", "6", "*",
                "1", "2", "3", "-",
                "0", "C", "=", "+"
        };

        JPanel panel = new JPanel(new GridLayout(4, 4, 5, 5));
        for (String text : buttons) {
            JButton btn = new JButton(text);
            btn.setFont(new Font("Arial", Font.BOLD, 18));
            btn.addActionListener(this);
            panel.add(btn);
        }
        add(panel, BorderLayout.CENTER);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();

        if (cmd.matches("[0-9]")) { // digit
            if (startNewNumber) {
                display.setText(cmd);
                startNewNumber = false;
            } else {
                display.setText(display.getText() + cmd);
            }
        } else if (cmd.equals("C")) { // clear
            display.setText("");
            firstOperand = "";
            operator = "";
            startNewNumber = true;
        } else if (cmd.equals("=")) {
            if (!firstOperand.isEmpty() && !operator.isEmpty() && !display.getText().isEm
                double a = Double.parseDouble(firstOperand);
                double b = Double.parseDouble(display.getText());
                double result = 0;

                switch (operator) {
                    case "+": result = a + b; break;
                    case "-": result = a - b; break;
                    case "*": result = a * b; break;
                    case "/":
                        if (b == 0) {
                            display.setText("Error");
                            startNewNumber = true;
                            return;
                        }
                        result = a / b;
                        break;
                }
                display.setText(String.valueOf(result));
                firstOperand = "";
                operator = "";
                startNewNumber = true;
            }
        } else { // operator + - * /
            if (!display.getText().isEmpty()) {
                firstOperand = display.getText();
                operator = cmd;
                startNewNumber = true;
            }
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new SwingCalculator().setVisible(true));
    }
}
```

Output / Result

- A window titled "Swing Calculator" appears with a numeric display and 16 buttons arranged in a 4×4 grid for digits, operators, clear, and equals.[6]
- Clicking number and operator buttons allows entering expressions like 7 + 3, and pressing `=` shows the result in the display; `C` clears the current calculation.[6]

Viva notes

- Event-driven programming in Swing uses listeners such as `ActionListener` that react to user actions like button clicks.[6]
- `JFrame` is the main window container, `JButton` components generate events, and `JTextField` is used here as the calculator display.[6]
- Layout managers like `BorderLayout` and `GridLayout` control how components are arranged in the GUI without manual pixel positioning.[6]
- The calculator stores operands and operator as variables, performs arithmetic in response to the `=` event, and updates the view, illustrating separation of GUI and logic.[6]
- Proper input validation (for example, division by zero handling) and resetting state are important to avoid runtime errors and incorrect results.[6]

⁂

answer all the remaining ones

There are 11 remaining questions (3, 4, 5, 6, 7, 8, 10, 12, 13, 15), and each needs Aim, Algorithm, Program, Output and Viva, which is too long for a single answer within limits.[7]

How to continue efficiently

To make sure you get complete lab-record style answers without cutting important parts, it is better to do them in small groups. Please choose:[7]

- Next: 3, 4, 5
- Or: 6, 7, 8
- Or: 10, 12, 13
- Or: 15 alone (MVC Swing form is slightly longer)

Reply with which group you want first (for example: "Do 3, 4, 5 now"), and that group will be answered with: Aim, Algorithm/Procedure, Program, Output, Viva notes for each question.

⁂

3,4,5

Aim for Q3:
To build a feedback form web application using Servlets that collects user feedback from an HTML form and stores it in a database.[8]

Algorithm / Procedure (Q3)

1. Create a database and table (for example `feedback(id, name, email, comments)` with `id` auto-increment).[8]
2. Design an HTML/JSP feedback form with fields for name, email, and comments, posting to a servlet via POST.[8]
3. In the servlet's `doPost`, read the form parameters using `request.getParameter(...)`.[8]
4. Establish a JDBC connection, create a `PreparedStatement` for `INSERT`, set parameters, and execute to store the feedback.[8]
5. Show a success message or thank-you page, then close the DB connection.[8]

Program / Implementation (Q3)

Minimal form (feedback.jsp):

```
<form action="feedback" method="post">
    Name: <input type="text" name="name" required><br>
    Email: <input type="email" name="email" required><br>
    Feedback:<br>
    <textarea name="comments" rows="4" cols="30" required></textarea><br>
    <input type="submit" value="Submit">
</form>
```

Servlet (FeedbackServlet.java, imports `jakarta.servlet.*`, `jakarta.servlet.http.*`, `java.sql.*`):

```
@WebServlet("/feedback")
public class FeedbackServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String name = request.getParameter("name");
        String email = request.getParameter("email");
        String comments = request.getParameter("comments");

        String url = "jdbc:mysql://localhost:3306/advanced_java";
        String user = "root";
        String pass = "root";

        String sql = "INSERT INTO feedback(name, email, comments) VALUES (?, ?, ?)";

        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection con = DriverManager.getConnection(url, user, pass);
            PreparedStatement ps = con.prepareStatement(sql);
            ps.setString(1, name);
            ps.setString(2, email);
            ps.setString(3, comments);
            ps.executeUpdate();
            ps.close();
            con.close();

            response.setContentType("text/html");
            response.getWriter().println("<h3>Thank you for your feedback!</h3>");
        } catch (Exception e) {
            e.printStackTrace();
            response.getWriter().println("Error: " + e.getMessage());
        }
    }
}
```

Output / Result (Q3)

- User fills the feedback form and submits; a thank-you message is displayed in the browser. [8]
- The feedback record appears in the `feedback` table in the database with name, email, and comments stored. [8]

Viva notes (Q3)

- Servlets process HTTP requests, read parameters, and generate dynamic responses. [8]
- POST is preferred for form submissions that change server data or send large input. [8]
- JDBC `PreparedStatement` lets the application safely insert user input into the database. [8]
- Web applications typically follow MVC: HTML/JSP as view, Servlet as controller, and DB as model. [8]

Aim for Q4:
To write a web application where a filter logs request details such as IP address, timestamp, and requested URL before the target servlet runs. [8]

Algorithm / Procedure (Q4)

1. Create a servlet (for example `HelloServlet`) that just prints a simple response. [8]
2. Implement a `Filter` class that intercepts requests, obtains `request.getRemoteAddr()`, request URI, and current time. [8]
3. Log these details to console or a file, then call `chain.doFilter(request, response)` to continue processing. [8]
4. Configure the filter mapping to the servlet or a URL pattern using `@WebFilter` or `web.xml`. [8]
5. Deploy the app and access the servlet URL to see log entries for each request. [8]

Program / Implementation (Q4)

Simple servlet:

```
@WebServlet("/hello")
public class HelloServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        response.getWriter().println("<h3>Hello from servlet</h3>");
    }
}
```

Logging filter:

```
import jakarta.servlet.*;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.http.HttpServletRequest;
import java.io.IOException;
import java.time.LocalDateTime;

@WebFilter("/*")
public class LoggingFilter implements Filter {

    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {

        HttpServletRequest req = (HttpServletRequest) request;
        String ip = request.getRemoteAddr();
        String url = req.getRequestURI();
        LocalDateTime now = LocalDateTime.now();

        System.out.println("Request from IP: " + ip +
                ", URL: " + url +
                ", Time: " + now);

        chain.doFilter(request, response);
    }
}
```

Output / Result (Q4)

- When `/hello` is accessed, the browser displays the “Hello from servlet” message.[8]
- The server console prints log lines showing client IP, requested URL, and timestamp for each request.[8]

Viva notes (Q4)

- Filters in Java EE/Jakarta EE are used to preprocess and postprocess requests and responses (logging, authentication, compression, etc.).[8]
- `doFilter` is executed before the request reaches the servlet and after it returns, depending on where code is placed around `chain.doFilter`.[8]
- `@WebFilter` or `web.xml` defines which URL patterns are intercepted by a filter.[8]
- Filters promote separation of cross-cutting concerns from business logic.[8]

Aim for Q5:
To develop a servlet that tracks user visits using `HttpSession` and displays how many times the user has visited the page.[8]

Algorithm / Procedure (Q5)

1. Create a servlet mapped to a URL like `/counter`.[8]
2. In `doGet`, obtain the `HttpSession` using `request.getSession()`.[8]
3. Retrieve a session attribute (for example `visitCount`); if null, set it to 1; otherwise increment it and store back.[8]
4. Display the current visit count in the response HTML.[8]
5. Deploy and access the servlet multiple times in the same browser to see the count increase; closing the browser or expiring the session resets the count.[8]

Program / Implementation (Q5)

```java
import jakarta.servlet.*;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.*;
import java.io.IOException;

@WebServlet("/counter")
public class VisitCounterServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        HttpSession session = request.getSession();
        Integer count = (Integer) session.getAttribute("visitCount");
        if (count == null) {
            count = 1;
        } else {
            count++;
        }
        session.setAttribute("visitCount", count);

        response.setContentType("text/html");
        response.getWriter().println("<h3>You have visited this page " +
                count + " times in this session.</h3>");
    }
}
```

Output / Result (Q5)

- On first visit, the page shows "You have visited this page 1 times in this session."[8]
- Refreshing the page increases the number; if the session expires or a new browser session starts, the count restarts from 1.[8]

Viva notes (Q5)

- `HttpSession` stores user-specific data across multiple requests, enabling session management on top of stateless HTTP.[8]
- The server uses a session ID (often via cookie `JSESSIONID`) to identify which session belongs to which client.[8]
- Session attributes are key–value pairs accessible using `setAttribute` and `getAttribute`.[8]
- Session timeout can be configured in `web.xml` or programmatically; after timeout, a new session is created.[8]

⁂

question5

Aim:
To develop a servlet that uses `HttpSession` to count and display how many times a user has visited the page in the current session.[9]

Algorithm / Procedure

1. Create a servlet mapped to a URL like `/counter`.[9]
2. In `doGet`, get the `HttpSession` using `request.getSession()`; this creates a new session if one does not exist.[9]
3. Read a session attribute `visitCount`; if it is `null`, initialize it to 1, otherwise increment the existing value.[9]
4. Store the updated count back in the session and generate an HTML response showing the visit count.[9]
5. Redeploy and access the servlet multiple times in the same browser window to see the counter increase; closing the browser or session expiry resets the count.[9]

Program / Implementation

```
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;

@WebServlet("/counter")
public class VisitCounterServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        // Get or create session
        HttpSession session = request.getSession();

        // Read visit count from session
        Integer count = (Integer) session.getAttribute("visitCount");
        if (count == null) {
            count = 1;          // first visit
        } else {
            count = count + 1;  // subsequent visits
        }

        // Store updated count
        session.setAttribute("visitCount", count);

        // Output to browser
        response.setContentType("text/html");
        response.getWriter().println(
                "<h3>You have visited this page " + count +
                " times in this session.</h3>"
        );
    }
}
```

Output / Result

- First time opening `/counter` in a browser shows: “You have visited this page 1 times in this session.”[9]
- Refreshing or revisiting increments the count (2, 3, 4, ...) until the session expires or the browser starts a new session, after which it starts again from 1.[9]

Viva notes

- `HttpSession` provides a way to store user-specific data across multiple HTTP requests, compensating for HTTP being stateless.[9]
- The server identifies each session using a session ID (often stored in a cookie like `JSESSIONID`).[9]
- Session attributes are stored as key–value pairs and accessed with `setAttribute` and `getAttribute`.[9]
- Session timeout can be configured in `web.xml` or via `session.setMaxInactiveInterval`, after which the session and its attributes are discarded.[9]

⁂

Question 6 – JSP, Servlet, JDBC (Register, Login, Profile)

Aim:
To create a multi-tier web application using JSP, Servlet, and JDBC that allows users to register, log in, and view their profile.[10]

Algorithm / Procedure (Q6)

1. Create a database and `users` table with fields like `id`, `name`, `email`, `password`.[10]
2. Design `register.jsp` to collect user details and submit them to a `RegisterServlet` using POST.[10]
3. In `RegisterServlet`, read parameters, insert a new user record into the DB using JDBC, then redirect to the login page.[10]
4. Design `login.jsp` to collect email and password and submit them to a `LoginServlet`.[10]
5. In `LoginServlet`, validate the credentials using a `SELECT` query; if valid, store user info in `HttpSession` and forward to `profile.jsp`.[10]
6. In `profile.jsp`, read user details from the session and display them.[10]

Program / Implementation (Q6) – core pieces

SQL:

```
CREATE DATABASE advanced_java;
USE advanced_java;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(50) UNIQUE,
    password VARCHAR(50)
);
```

`register.jsp`:

```
<form action="register" method="post">
    Name: <input type="text" name="name" required><br>
    Email: <input type="email" name="email" required><br>
    Password: <input type="password" name="password" required><br>
    <input type="submit" value="Register">
</form>
```

`RegisterServlet`:

```
@WebServlet("/register")
public class RegisterServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String name = request.getParameter("name");
        String email = request.getParameter("email");
        String pass = request.getParameter("password");

        String url = "jdbc:mysql://localhost:3306/advanced_java";
        String user = "root";
        String pwd = "root";

        String sql = "INSERT INTO users(name, email, password) VALUES (?, ?, ?)";

        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection con = DriverManager.getConnection(url, user, pwd);
            PreparedStatement ps = con.prepareStatement(sql);
            ps.setString(1, name);
            ps.setString(2, email);
            ps.setString(3, pass);
            ps.executeUpdate();
            ps.close();
            con.close();
            response.sendRedirect("login.jsp");
        } catch (Exception e) {
            e.printStackTrace();
            response.getWriter().println("Error: " + e.getMessage());
        }
    }
}
```

`login.jsp`:

```
<form action="login" method="post">
    Email: <input type="email" name="email" required><br>
    Password: <input type="password" name="password" required><br>
    <input type="submit" value="Login">
</form>
```

`LoginServlet`:

```
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String email = request.getParameter("email");
        String pass = request.getParameter("password");

        String url = "jdbc:mysql://localhost:3306/advanced_java";
        String user = "root";
        String pwd = "root";

        String sql = "SELECT name, email FROM users WHERE email=? AND password=?";

        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection con = DriverManager.getConnection(url, user, pwd);
            PreparedStatement ps = con.prepareStatement(sql);
            ps.setString(1, email);
            ps.setString(2, pass);
            ResultSet rs = ps.executeQuery();

            if (rs.next()) {
                HttpSession session = request.getSession();
                session.setAttribute("name", rs.getString("name"));
                session.setAttribute("email", rs.getString("email"));
                response.sendRedirect("profile.jsp");
            } else {
                response.getWriter().println("Invalid credentials");
            }

            rs.close();
            ps.close();
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
            response.getWriter().println("Error: " + e.getMessage());
        }
    }
}
```

`profile.jsp`:

```
<%
    String name = (String) session.getAttribute("name");
    String email = (String) session.getAttribute("email");
%>
<h3>User Profile</h3>
Name: <%= name %><br>
Email: <%= email %><br>
```

Output / Result (Q6)

- New users can register, after which they are redirected to the login page.[10]
- After successful login, the profile page displays the logged-in user’s name and email stored in the database.[10]

Viva notes (Q6)

- JSP is used for the view, Servlets for controller logic, and JDBC for database access, forming a multi-tier architecture.[10]
- `HttpSession` maintains the logged-in user's state across multiple requests.[10]
- Separating registration, login, and profile pages follows the MVC pattern and improves maintainability.[10]

Question 7 – RMI Employee Management

Aim:
To develop an RMI system for employee management that remotely stores and retrieves employee details using serialization.[10]

Algorithm / Procedure (Q7)

1. Define a remote interface (for example `EmployeeService`) that extends `Remote` and declares methods like `addEmployee(Employee e)` and `List<Employee> getEmployees()` throwing `RemoteException`.[10]
2. Create a serializable `Employee` class with fields like `id`, `name`, `salary` and implement `Serializable`.[10]
3. Implement the remote interface in a class (for example `EmployeeServiceImpl`) that maintains an in-memory list or simple file/DB storage.[10]
4. In the RMI server program, create an instance of the implementation, export it, and bind it to the RMI registry with a name.[10]
5. In the client program, lookup the remote object from the registry, call `addEmployee` to store employee details and `getEmployees` to retrieve and display them.[10]

Program / Implementation (Q7) – core pieces

Remote interface:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

public interface EmployeeService extends Remote {
    void addEmployee(Employee e) throws RemoteException;
    java.util.List<Employee> getEmployees() throws RemoteException;
}
```

`Employee` class:

```
import java.io.Serializable;

public class Employee implements Serializable {
    private int id;
    private String name;
    private double salary;

    public Employee() {}
    public Employee(int id, String name, double salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }
    public int getId() { return id; }
    public String getName() { return name; }
    public double getSalary() { return salary; }
}
```

Implementation:

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.List;

public class EmployeeServiceImpl extends UnicastRemoteObject implements EmployeeService 

    private List<Employee> employees;

    public EmployeeServiceImpl() throws RemoteException {
        employees = new ArrayList<>();
    }

    @Override
    public void addEmployee(Employee e) throws RemoteException {
        employees.add(e);
        System.out.println("Employee added: " + e.getName());
    }

    @Override
    public List<Employee> getEmployees() throws RemoteException {
        return employees;
    }
}
```

Server:

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class EmployeeServer {
    public static void main(String[] args) {
        try {
            EmployeeService service = new EmployeeServiceImpl();
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.rebind("EmployeeService", service);
            System.out.println("Employee RMI Server is ready.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Client:

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.List;

public class EmployeeClient {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);
            EmployeeService service = (EmployeeService) registry.lookup("EmployeeService'

            service.addEmployee(new Employee(1, "Alice", 50000));
            service.addEmployee(new Employee(2, "Bob", 60000));

            List<Employee> list = service.getEmployees();
            for (Employee e : list) {
                System.out.println(e.getId() + " " + e.getName() + " " + e.getSalary());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Output / Result (Q7)

- The server console prints that the RMI server is ready, and each time `addEmployee` is called it logs a message.[10]
- The client, after adding employees, prints the list of employees with id, name, and salary values received from the remote object.[10]

Viva notes (Q7)

- RMI (Remote Method Invocation) allows a Java program to invoke methods of an object located on another JVM as if it were local.[10]
- Remote interfaces must extend `Remote` and their methods must declare `RemoteException`.[10]
- Classes whose instances are sent over the network (like `Employee`) must implement `Serializable` (or be remote stubs).[10]
- The RMI registry acts like a naming service where server objects are bound and clients look them up by name.[10]

⁂

question 8

Aim:
To create an RMI-based calculator service that performs basic arithmetic operations (addition, subtraction, multiplication, division) remotely between a client and server.[11]

Algorithm / Procedure

1. Define a remote interface `Calculator` that extends `Remote` and declares methods like `add`, `sub`, `mul`, `div` throwing `RemoteException`.[11]
2. Implement this interface in a class `CalculatorImpl` that extends `UnicastRemoteObject` and provides actual arithmetic logic.[11]
3. In the RMI server class, create an instance of `CalculatorImpl`, create or get the RMI registry, and bind the object with a name like `"CalcService"`.[11]
4. In the client program, look up `"CalcService"` in the registry, obtain the remote reference, and call its methods with user-supplied numbers, printing results.[11]
5. Start the server first, then run the client to test remote calculations.[11]

Program / Implementation

1) Remote interface (Calculator.java)

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Calculator extends Remote {
    double add(double a, double b) throws RemoteException;
    double sub(double a, double b) throws RemoteException;
    double mul(double a, double b) throws RemoteException;
    double div(double a, double b) throws RemoteException;
}
```

2) Implementation (CalculatorImpl.java)

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class CalculatorImpl extends UnicastRemoteObject implements Calculator {

    public CalculatorImpl() throws RemoteException {
        super();
    }

    public double add(double a, double b) throws RemoteException {
        return a + b;
    }

    public double sub(double a, double b) throws RemoteException {
        return a - b;
    }

    public double mul(double a, double b) throws RemoteException {
        return a * b;
    }

    public double div(double a, double b) throws RemoteException {
        if (b == 0) {
            throw new RemoteException("Division by zero");
        }
        return a / b;
    }
}
```

3) Server (CalcServer.java)

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class CalcServer {
    public static void main(String[] args) {
        try {
            Calculator calc = new CalculatorImpl();
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.rebind("CalcService", calc);
            System.out.println("Calculator RMI Server is ready.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

4) Client (CalcClient.java)

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Scanner;

public class CalcClient {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);
            Calculator calc = (Calculator) registry.lookup("CalcService");

            Scanner sc = new Scanner(System.in);
            System.out.print("Enter first number: ");
            double a = sc.nextDouble();
            System.out.print("Enter second number: ");
            double b = sc.nextDouble();

            System.out.println("a + b = " + calc.add(a, b));
            System.out.println("a - b = " + calc.sub(a, b));
            System.out.println("a * b = " + calc.mul(a, b));
            try {
                System.out.println("a / b = " + calc.div(a, b));
            } catch (Exception ex) {
                System.out.println("Error: " + ex.getMessage());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Output / Result

- The server console prints “Calculator RMI Server is ready.” once the remote object is bound.[11]
- The client, after entering two numbers, prints the results of add, sub, mul, and div, with division by zero reported as an error.[11]

Viva notes

- RMI allows a client JVM to invoke methods on an object located in a server JVM over the network as if it were local.[11]
- Remote interfaces extend `Remote`, and all remote methods must declare `RemoteException`.[11]
- The implementation class usually extends `UnicastRemoteObject` so that stubs and networking details are handled automatically.[11]
- The RMI registry is a name service where remote objects are registered (`rebind`) and clients find them (`lookup`) using a logical name like `"CalcService"`.[11]

⁂

Question 12 – JDBC CRUD on Student

Aim:
To implement a Java application that performs full CRUD (Create, Read, Update, Delete) operations on a “Student” table using JDBC connectivity.[12]

Algorithm / Procedure (Q12)

1. Create database `advanced_java` and table `student(id, name, course, marks)` with `id` as primary key.[12]
2. Write a Java class with menu options for Insert, View, Update, Delete using console input or simple methods.[12]
3. For each option, open a JDBC connection, prepare SQL with `PreparedStatement`, set parameters, and execute (`INSERT`, `SELECT`, `UPDATE`, `DELETE`).[12]
4. For read operations, use `ResultSet` to fetch and display all student records.[12]

5. Close all JDBC resources after each operation.[12]

Program / Implementation (Q12)

Example SQL:

```
CREATE DATABASE advanced_java;
USE advanced_java;

CREATE TABLE student (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    course VARCHAR(50),
    marks INT
);
```

Java (console menu):

```
import java.sql.*;
import java.util.Scanner;

public class StudentCRUD {

    private static final String URL = "jdbc:mysql://localhost:3306/advanced_java";
    private static final String USER = "root";
    private static final String PASS = "root";

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int choice;
        do {
            System.out.println("1. Insert  2. View  3. Update  4. Delete  5. Exit");
            choice = sc.nextInt();
            switch (choice) {
                case 1: insertStudent(sc); break;
                case 2: viewStudents(); break;
                case 3: updateStudent(sc); break;
                case 4: deleteStudent(sc); break;
            }
        } while (choice != 5);
        sc.close();
    }

    private static Connection getConnection() throws Exception {
        Class.forName("com.mysql.cj.jdbc.Driver");
        return DriverManager.getConnection(URL, USER, PASS);
    }

    private static void insertStudent(Scanner sc) {
        try (Connection con = getConnection()) {
            System.out.print("ID: ");
            int id = sc.nextInt();
            sc.nextLine();
            System.out.print("Name: ");
            String name = sc.nextLine();
            System.out.print("Course: ");
            String course = sc.nextLine();
            System.out.print("Marks: ");
            int marks = sc.nextInt();

            String sql = "INSERT INTO student(id, name, course, marks) VALUES (?, ?, ?, ?
            PreparedStatement ps = con.prepareStatement(sql);
            ps.setInt(1, id);
            ps.setString(2, name);
            ps.setString(3, course);
            ps.setInt(4, marks);
            ps.executeUpdate();
            System.out.println("Student inserted.");
            ps.close();
        } catch (Exception e) { e.printStackTrace(); }
    }

    private static void viewStudents() {
        try (Connection con = getConnection()) {
            String sql = "SELECT * FROM student";
            PreparedStatement ps = con.prepareStatement(sql);
            ResultSet rs = ps.executeQuery();
            while (rs.next()) {
                System.out.println(rs.getInt("id") + " "
                        + rs.getString("name") + " "
                        + rs.getString("course") + " "
                        + rs.getInt("marks"));
            }
            rs.close();
            ps.close();
        } catch (Exception e) { e.printStackTrace(); }
    }

    private static void updateStudent(Scanner sc) {
        try (Connection con = getConnection()) {
            System.out.print("Enter ID to update: ");
            int id = sc.nextInt();
            sc.nextLine();
            System.out.print("New Name: ");
            String name = sc.nextLine();
            System.out.print("New Course: ");
            String course = sc.nextLine();
            System.out.print("New Marks: ");
            int marks = sc.nextInt();

            String sql = "UPDATE student SET name=?, course=?, marks=? WHERE id=?";
            PreparedStatement ps = con.prepareStatement(sql);
            ps.setString(1, name);
            ps.setString(2, course);
            ps.setInt(3, marks);
            ps.setInt(4, id);
            ps.executeUpdate();
            System.out.println("Student updated.");
            ps.close();
        } catch (Exception e) { e.printStackTrace(); }
    }

    private static void deleteStudent(Scanner sc) {
        try (Connection con = getConnection()) {
            System.out.print("Enter ID to delete: ");
            int id = sc.nextInt();

            String sql = "DELETE FROM student WHERE id=?";
            PreparedStatement ps = con.prepareStatement(sql);
            ps.setInt(1, id);
            ps.executeUpdate();
            System.out.println("Student deleted.");
            ps.close();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

Output / Result (Q12)

- Console menu lets you insert many students, list them all, modify details for a given ID, and delete a student; database table `student` reflects each change.[12]

Viva notes (Q12)

- CRUD stands for Create, Read, Update, Delete—the four basic operations on persistent data.[12]
- JDBC `PreparedStatement` is used for all SQL operations to handle parameters safely.[12]
- `ResultSet` is used with `SELECT` to iterate over records returned from the database.[12]
- Keys (like primary key `id`) uniquely identify rows, which is important for updates and deletes.[12]

Question 13 – Theme-switching GUI (Look & Feel)

Aim:
To build a Swing GUI application that allows the user to switch between different Look & Feel themes at runtime.[12]

Algorithm / Procedure (Q13)

1. Create a `JFrame` with some sample components (labels, buttons, text fields) so theme changes are visible.[12]
2. Add controls (for example, a `JComboBox` or menu) listing available Look & Feels like Metal, Nimbus, System.[12]
3. On selection change, call `UIManager.setLookAndFeel(...)` with the chosen L&F class name.[12]
4. Call `SwingUtilities.updateComponentTreeUI(frame)` and `frame.pack()` or `frame.repaint()` to apply the new theme.[12]

5. Run the app and switch themes to observe UI changes.[12]

Program / Implementation (Q13)

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

public class ThemeSwitcher extends JFrame {

    private JComboBox<String> combo;

    public ThemeSwitcher() {
        setTitle("Theme Switching GUI");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        // Sample components
        JLabel label = new JLabel("Sample Text:");
        JTextField textField = new JTextField(15);
        JButton button = new JButton("Click Me");

        combo = new JComboBox<>(new String[]{"Metal", "Nimbus", "System"});
        combo.addActionListener(this::changeTheme);

        JPanel top = new JPanel();
        top.add(new JLabel("Select Theme:"));
        top.add(combo);

        JPanel center = new JPanel();
        center.add(label);
        center.add(textField);
        center.add(button);

        add(top, BorderLayout.NORTH);
        add(center, BorderLayout.CENTER);
    }

    private void changeTheme(ActionEvent e) {
        String choice = (String) combo.getSelectedItem();
        try {
            if ("Metal".equals(choice)) {
                UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
            } else if ("Nimbus".equals(choice)) {
                UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");
            } else if ("System".equals(choice)) {
                UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
            }
            SwingUtilities.updateComponentTreeUI(this);
            this.pack();
        } catch (Exception ex) {
            ex.printStackTrace();
            JOptionPane.showMessageDialog(this, "Error applying theme: " + ex.getMessage(
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new ThemeSwitcher().setVisible(true));
    }
}
```

Output / Result (Q13)

- The window shows text field, button, and a combo box for theme selection.[12]
- When the user selects Metal, Nimbus, or System, the entire window's appearance changes immediately to the chosen Look & Feel.[12]

Viva notes (Q13)

- Look & Feel in Swing controls how components are drawn (colors, borders, widgets), separate from program logic.[12]
- `UIManager.setLookAndFeel` selects the theme; `updateComponentTreeUI` reapplies it to all components.[12]
- Common built-in L&Fs include Metal, Nimbus, System, and cross-platform defaults.[12]
- Theme switching demonstrates Swing's pluggable Look & Feel architecture, where visual styles can be swapped without changing core code.[12]

⁂

question 14,15

Answer for Question 14 was already given earlier; repeating it briefly and then adding Question 15.[13]

Question 14 – Swing Calculator (event-driven)

Aim:
To create a Swing-based calculator application that performs basic arithmetic operations using event-driven programming.[13]

Algorithm / Procedure (Q14)

1. Create a `JFrame` with a non-editable `JTextField` as the display and a grid of `JButton`s for digits, operators, clear, and equals.[13]
2. Add an `ActionListener` to each button; digit buttons append to the display, operator buttons store the first operand and operator, and `=` performs the calculation using the second operand.[13]
3. Handle division by zero and clear operation to reset state.[13]

Program / Implementation (Q14)

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

public class SwingCalculator extends JFrame implements ActionListener {

    private JTextField display;
    private String firstOperand = "";
    private String operator = "";
    private boolean startNewNumber = true;

    public SwingCalculator() {
        setTitle("Swing Calculator");
        setSize(300, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        display = new JTextField();
        display.setEditable(false);
        display.setHorizontalAlignment(JTextField.RIGHT);
        display.setFont(new Font("Arial", Font.BOLD, 24));
        add(display, BorderLayout.NORTH);

        String[] buttons = {
                "7", "8", "9", "/",
                "4", "5", "6", "*",
                "1", "2", "3", "-",
                "0", "C", "=", "+"
        };

        JPanel panel = new JPanel(new GridLayout(4, 4, 5, 5));
        for (String text : buttons) {
            JButton btn = new JButton(text);
            btn.setFont(new Font("Arial", Font.BOLD, 18));
            btn.addActionListener(this);
            panel.add(btn);
        }
        add(panel, BorderLayout.CENTER);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();

        if (cmd.matches("[0-9]")) {
            if (startNewNumber) {
                display.setText(cmd);
                startNewNumber = false;
            } else {
                display.setText(display.getText() + cmd);
            }
        } else if (cmd.equals("C")) {
            display.setText("");
            firstOperand = "";
            operator = "";
            startNewNumber = true;
        } else if (cmd.equals("=")) {
            if (!firstOperand.isEmpty() && !operator.isEmpty() && !display.getText().isEm
                double a = Double.parseDouble(firstOperand);
                double b = Double.parseDouble(display.getText());
                double result = 0;

                switch (operator) {
                    case "+": result = a + b; break;
                    case "-": result = a - b; break;
                    case "*": result = a * b; break;
                    case "/":
                        if (b == 0) {
                            display.setText("Error");
                            startNewNumber = true;
                            return;
                        }
                        result = a / b;
                        break;
                }
                display.setText(String.valueOf(result));
                firstOperand = "";
                operator = "";
                startNewNumber = true;
            }
        } else { // operator
            if (!display.getText().isEmpty()) {
                firstOperand = display.getText();
                operator = cmd;
                startNewNumber = true;
            }
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new SwingCalculator().setVisible(true));
    }
}
```

Output / Result (Q14)

- A calculator window with numeric and operator buttons; pressing buttons like `7 + 3 =` shows `10` in the display, `C` clears the display.[13]

Viva notes (Q14)

- Event-driven programming uses listeners like `ActionListener` to react to user actions instead of sequential code.[13]
- Swing components (`JFrame`, `JButton`, `JTextField`) and layout managers (`BorderLayout`, `GridLayout`) build the GUI.[13]

- The calculator logic maintains operands and operator as state, updated by button click events.[13]

Question 15 – MVC Swing Form with Validation

Aim:
To develop an MVC-based Swing form with text fields, buttons, checkboxes, and radio buttons that performs form validation before accepting input.[13]

Algorithm / Procedure (Q15)

1. Design the Model class (for example `User`) with fields like name, email, gender, and interests, plus getters/setters.[13]
2. Design the View as a `JFrame` containing `JTextField`s (name, email), `JRadioButton`s for gender, `JCheckBox`es for hobbies, and buttons (`Submit`, `Clear`).[13]
3. Design the Controller as listeners that read values from the view, validate them (non-empty, email format, at least one gender selected), and if valid, populate the Model and show a success message; otherwise, show error messages.[13]
4. Run the application; try invalid input to see validation messages and valid input to see accepted data.[13]

Program / Implementation (Q15) – simple combined version

Model (`User.java`):

```
public class User {
    private String name;
    private String email;
    private String gender;
    private String hobbies;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    public String getGender() { return gender; }
    public void setGender(String gender) { this.gender = gender; }

    public String getHobbies() { return hobbies; }
    public void setHobbies(String hobbies) { this.hobbies = hobbies; }
}
```

View + Controller (`UserFormMVC.java`):

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

```java
import java.awt.event.ActionEvent;

public class UserFormMVC extends JFrame {

    private JTextField txtName, txtEmail;
    private JRadioButton rbMale, rbFemale;
    private JCheckBox cbSports, cbMusic, cbReading;
    private JButton btnSubmit, btnClear;
    private User model;

    public UserFormMVC() {
        model = new User();

        setTitle("MVC Swing Form with Validation");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        txtName = new JTextField(15);
        txtEmail = new JTextField(15);

        rbMale = new JRadioButton("Male");
        rbFemale = new JRadioButton("Female");
        ButtonGroup group = new ButtonGroup();
        group.add(rbMale);
        group.add(rbFemale);

        cbSports = new JCheckBox("Sports");
        cbMusic = new JCheckBox("Music");
        cbReading = new JCheckBox("Reading");

        btnSubmit = new JButton("Submit");
        btnClear = new JButton("Clear");

        btnSubmit.addActionListener(this::handleSubmit);
        btnClear.addActionListener(e -> clearForm());

        JPanel panel = new JPanel(new GridLayout(6, 2, 5, 5));
        panel.add(new JLabel("Name:"));
        panel.add(txtName);
        panel.add(new JLabel("Email:"));
        panel.add(txtEmail);
        panel.add(new JLabel("Gender:"));

        JPanel genderPanel = new JPanel();
        genderPanel.add(rbMale);
        genderPanel.add(rbFemale);
        panel.add(genderPanel);

        panel.add(new JLabel("Hobbies:"));
        JPanel hobbyPanel = new JPanel();
        hobbyPanel.add(cbSports);
        hobbyPanel.add(cbMusic);
        hobbyPanel.add(cbReading);
        panel.add(hobbyPanel);
        panel.add(btnSubmit);
        panel.add(btnClear);

        add(panel);
    }

    private void handleSubmit(ActionEvent e) {
        String name = txtName.getText().trim();
        String email = txtEmail.getText().trim();
        String gender = rbMale.isSelected() ? "Male" : rbFemale.isSelected() ? "Female" :
        StringBuilder hobbies = new StringBuilder();
        if (cbSports.isSelected()) hobbies.append("Sports ");
        if (cbMusic.isSelected()) hobbies.append("Music ");
        if (cbReading.isSelected()) hobbies.append("Reading ");

        // Validation
        if (name.isEmpty()) {
            JOptionPane.showMessageDialog(this, "Name is required");
            return;
        }
        if (email.isEmpty() || !email.contains("@")) {
            JOptionPane.showMessageDialog(this, "Valid email is required");
            return;
        }
        if (gender.isEmpty()) {
            JOptionPane.showMessageDialog(this, "Please select gender");
            return;
        }

        // Update model
        model.setName(name);
        model.setEmail(email);
        model.setGender(gender);
        model.setHobbies(hobbies.toString().trim());

        JOptionPane.showMessageDialog(this,
                "User saved:\nName: " + model.getName() +
                        "\nEmail: " + model.getEmail() +
                        "\nGender: " + model.getGender() +
                        "\nHobbies: " + model.getHobbies());
    }

    private void clearForm() {
        txtName.setText("");
        txtEmail.setText("");
        rbMale.setSelected(false);
        rbFemale.setSelected(false);
        cbSports.setSelected(false);
        cbMusic.setSelected(false);
        cbReading.setSelected(false);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new UserFormMVC().setVisible(true));
    }
}
```

Output / Result (Q15)

- The form window displays fields for name, email, gender (radio buttons), and hobbies (checkboxes) with Submit and Clear buttons.[13]
- On Submit, if any required field is invalid, a dialog shows an error; with valid data, a dialog displays the saved user information from the model.[13]

Viva notes (Q15)

- MVC separates responsibilities: Model holds data, View displays UI, Controller handles user actions and validation logic.[13]
- Swing components (`JTextField`, `JRadioButton`, `JCheckBox`, `JButton`) form the View; event handlers act as the Controller.[13]
- Form validation ensures only correct and complete data is accepted, improving reliability of GUI applications.[13]

⁂

1. Advanced-Java.pdf
2. Advanced-Java.pdf
3. Advanced-Java.pdf
4. Advanced-Java.pdf
5. Advanced-Java.pdf
6. Advanced-Java.pdf
7. Advanced-Java.pdf
8. Advanced-Java.pdf
9. Advanced-Java.pdf
10. Advanced-Java.pdf
11. Advanced-Java.pdf
12. Advanced-Java.pdf
13. Advanced-Java.pdf