Ajp (4351603) - Problem Solver

Antigravity AI

June 4, 2026

Ajp

First Edition: 2026

Author: Antigravity AI
Website: milav.in
YouTube: @milav.dabgar

To my students,
who constantly inspire me to find new analogies,
break down complex theories, and make learning
an engineered science.

Preface

About this Book

Welcome to **Computer Networks**! This textbook is meticulously designed to serve as a comprehensive problem solver and study companion. It provides a robust and intuitive foundation in the core principles of this subject, bridging the gap between theoretical models and practical applications. A unique feature of this book is its focus on decoding complex concepts using clear, step-by-step methodologies and real-world analogies.

Target Audience

This book is intended for students and professionals looking to master the concepts of Computer Networks. Whether you are revising for exams, seeking to solidify your fundamental understanding, or looking for practical examples to clarify abstract concepts, this book has been structured to support your learning journey. It serves as an accessible guide designed to remove the fear of complex topics, replacing it with an appreciation for the underlying mechanics.

Scope and Coverage

The coverage is divided logically to follow standard academic curriculums. Each chapter focuses on key topics, breaking them down into digestible sections:

- **Theoretical Insights** – Core concepts explained intuitively.
- **Method Blocks** – Standardized, step-by-step procedures for solving specific types of problems.
- **Worked Examples** – Fully solved problems that apply the methods in practice.

We hope this resource proves invaluable in your studies and helps you achieve excellence in **Computer Networks**. Happy learning!

Acknowledgments

Writing a comprehensive book that connects the fundamental forces of Chemistry with practical engineering applications requires more than just academic knowledge—it requires a community of support. I would like to express my sincere gratitude to everyone who contributed to the realization of this project.

Specially, I extend my heartfelt gratitude to the **Department of Technical Education (DTE), Government of Gujarat**, and **Government Polytechnic, Palanpur**, for providing an environment that fosters innovation, academic rigor, and continuous professional development.

I am deeply thankful to my family for their unwavering patience and support during the countless hours spent researching, formatting, and refining these chapters.

Finally, my deepest appreciation goes to my students. Your curiosity, challenging questions, and continuous feedback are the true driving force behind the unique analogies and streamlined structure of this book. This textbook was engineered for you.

Antigravity AI

About the Author

Milav Jayeshkumar Dabgar is an engineering educator and R&D professional with over 9 years of experience in electronics hardware development, embedded systems, AI/ML, and full-stack software engineering. He currently serves as a **Lecturer (Class - II) & Innovation Leader** at Government Polytechnic, Palanpur, under the Education Department of the Government of Gujarat.

Milav holds a Master of Engineering (ME) in Communication Systems from L.D. College of Engineering and a Bachelor of Engineering (BE) in Electronics & Communication. He is also currently pursuing a **BS in Data Science and Applications from IIT Madras**, where he has completed both the **Diploma in Programming** and the **Diploma in Data Science** with distinction.

Most recently, he completed the **AICTE QIP PG Certificate Program in Deep Learning from SVNIT Surat** with a **perfect 10/10 CGPA**, topping his batch.

A dedicated mentor, Milav has guided numerous student teams in securing over **INR 45 lakhs** in innovation funding, including INR 25 lakhs from Shark Tank India and INR 20 lakhs through iHub Gujarat, resulting in two student patents. He is recognized as an **NPTEL Evangelist** and **NPTEL Discipline Star** in Computer Science, reflecting his commitment to continuous learning and academic excellence.

His technical expertise spans from embedded systems (8051, STM32, Arduino) to modern web technologies (Next.js, Python, PostgreSQL) and Data Science (TensorFlow, PyTorch). Through this textbook, he aims to simplify complex Engineering Chemistry concepts by integrating relatable, real-world analogies drawn from modern tech and engineering landscapes.

Milav is also an active content creator, running a popular **YouTube channel** (@milav.dabgar) where he shares technical tutorials and academic resources. He maintains a personal blog and resource hub at milav.in and can be reached for professional collaborations on LinkedIn.

Contents

List of Figures

List of Tables

CHAPTER 1

Java Swing

1.1 Differentiate between AWT and Swing

Methodology:

Comparison Criteria between AWT and Swing The differences between Abstract Window Toolkit (AWT) and Swing are based on underlying architecture and features. Understanding these distinctions helps in choosing the right GUI toolkit.

Feature	AWT	Swing
Platform Dependency	Heavyweight components that rely on the native operating system GUI.	Lightweight components written entirely in Java and platform-independent.
Execution Speed	Slower execution due to native peer dependency.	Faster execution since components do not rely on native peers.
MVC Pattern	Does not follow Model-View-Controller architecture.	Follows Model-View-Controller (MVC) architecture.
Look and Feel	Assumes the look and feel of the host operating system.	Supports pluggable look and feel.
Package	Components belong to `java.awt` package.	Components belong to `javax.swing` package.
Component Names	Button, TextField, Label.	JButton, JTextField, JLabel.

1.2 Develop GUI programs using Swing Components

Methodology:

Steps to Create a Swing Application To develop a graphical user interface using Swing components:

1. **Import the package:** Import `javax.swing.*` to access Swing classes.
2. **Create a Container:** Instantiate a top-level container such as `JFrame`.
3. **Set Frame Properties:** Define properties like size, layout manager and default close operation.
4. **Instantiate Components:** Create objects for components like `JButton`, `JLabel` and `JTextField`.
5. **Add Components:** Use the `add()` method to insert components into the frame.
6. **Make Frame Visible:** Call `setVisible(true)` on the frame to display the GUI.

Worked Example:

Create a Basic Login Form Develop a Swing application that displays a login form with username and password fields along with a login button.

Step 1: Import necessary packages

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

Step 2: Create the main class and frame

```
public class LoginForm {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Login");
        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new GridLayout(3, 2));
```

Step 3: Instantiate components

```
        JLabel userLabel = new JLabel("Username:");
        JTextField userText = new JTextField();

        JLabel passLabel = new JLabel("Password:");
        JPasswordField passText = new JPasswordField();

        JButton loginButton = new JButton("Login");
```

Step 4: Add components to the frame

```
        frame.add(userLabel);
        frame.add(userText);
        frame.add(passLabel);
        frame.add(passText);
        frame.add(new JLabel("")); // Empty label for spacing
        frame.add(loginButton);
```

Step 5: Display the frame

```
        frame.setVisible(true);
    }
}
```

1.3 Develop simple event driven programs using event class and event listener interface

Methodology:

Event Handling Architecture Event handling in Java follows the Delegation Event Model. The steps to handle an event are:

1. **Identify the Event Source:** Determine which component will generate the event (e.g. a `JButton`).
2. **Implement the Listener Interface:** Implement the appropriate listener interface (e.g. `ActionListener`) in a class.
3. **Override the Event Method:** Provide the logic to be executed when the event occurs inside the overridden method (e.g. `actionPerformed()`).

4. **Register the Listener:** Attach the listener object to the source component using methods like `addActionListener()`.

Worked Example:

Addition of Two Numbers on Button Click Write a program containing two text fields for inputs and a button. When the button is clicked the sum of the two numbers should be displayed in a third text field.

Step 1: Create the UI layout and components

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

public class AdditionApp implements ActionListener {
    JTextField tf1, tf2, tf3;
    JButton btnAdd;

    public AdditionApp() {
        JFrame frame = new JFrame("Addition Application");
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());

        tf1 = new JTextField(10);
        tf2 = new JTextField(10);
        tf3 = new JTextField(10);
        tf3.setEditable(false);

        btnAdd = new JButton("Add");
```

Step 2: Register the listener

```
        btnAdd.addActionListener(this);
```

Step 3: Add components and show frame

```
        frame.add(new JLabel("Number 1:")); frame.add(tf1);
        frame.add(new JLabel("Number 2:")); frame.add(tf2);
        frame.add(new JLabel("Result:"));   frame.add(tf3);
        frame.add(btnAdd);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
```

Step 4: Implement the event handling logic

```
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == btnAdd) {
            try {
                int a = Integer.parseInt(tf1.getText());
                int b = Integer.parseInt(tf2.getText());
                int sum = a + b;
                tf3.setText(String.valueOf(sum));
            } catch (NumberFormatException ex) {
                tf3.setText("Invalid Input");
            }
        }
    }
```

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

Worked Example:

Handling Mouse Events Develop a Swing program that changes the background color of a frame when the mouse enters and exits the frame area.

Step 1: Set up the Frame and implement MouseListener

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

public class MouseColorApp extends JFrame implements MouseListener {

    public MouseColorApp() {
        setTitle("Mouse Event App");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Step 2: Register the MouseListener

```
        addMouseListener(this);
        setVisible(true);
    }
```

Step 3: Override the mouse event methods

```
    public void mouseEntered(MouseEvent e) {
        getContentPane().setBackground(Color.GREEN);
    }

    public void mouseExited(MouseEvent e) {
        getContentPane().setBackground(Color.RED);
    }

    // Unused interface methods must be overridden with empty bodies
    public void mouseClicked(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}

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

CHAPTER 2

Java Database Connectivity

2.1 Basics of JDBC and Connectivity

The Java Database Connectivity (JDBC) API provides universal data access from the Java programming language. Using the JDBC API, you can access virtually any data source, from relational databases to spreadsheets and flat files.

Methodology:

Steps for JDBC Connectivity To establish a database connection and execute queries in Java, follow these sequential steps:

1. **Import Packages:** Include the necessary SQL packages (`java.sql.*`).
2. **Load and Register Driver:** Load the database-specific JDBC driver class using `Class.forName()`.
3. **Establish Connection:** Use `DriverManager.getConnection()` with the database URL username and password.
4. **Create Statement:** Instantiate a `Statement` or `PreparedStatement` object from the `Connection`.
5. **Execute Query:** Run SQL statements using methods like `executeQuery()` for SELECT or `executeUpdate()` for DML operations.
6. **Process Results:** Iterate through the `ResultSet` returned by `executeQuery()`.
7. **Close Connections:** Close the `ResultSet Statement` and `Connection` objects in a `finally` block or use try-with-resources.

Worked Example:

Establish a JDBC Connection to a MySQL Database Write a Java program to load the MySQL JDBC driver and establish a connection to a database named `studentdb`.
Solution:

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

public class DatabaseConnection {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/studentdb";
        String user = "root";
        String password = "password123";

        try {
            // Step 1: Load and Register the Driver
```

```
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Step 2: Establish the Connection
            Connection conn = DriverManager.getConnection(url, user, password);
            System.out.println("Database connection established successfully!");

            // Step 3: Close the Connection
            conn.close();
        } catch (ClassNotFoundException e) {
            System.out.println("MySQL JDBC Driver not found.");
            e.printStackTrace();
        } catch (SQLException e) {
            System.out.println("Connection failed.");
            e.printStackTrace();
        }
    }
}
```

2.2 Types of JDBC Drivers

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

Methodology:

Selecting a JDBC Driver Type Choose the appropriate driver based on the application requirements:

1. **Type 1 (JDBC-ODBC Bridge Driver):** Translates JDBC calls to ODBC calls. Useful for legacy systems but slow and platform-dependent.
2. **Type 2 (Native-API Driver):** Uses client-side libraries of the database. Requires database-specific native libraries on the client machine.
3. **Type 3 (Network Protocol Driver):** Uses a middleware server to translate JDBC calls into database-specific protocols. Best for a three-tier architecture.
4. **Type 4 (Thin Driver):** Converts JDBC calls directly into the database-specific network protocol. Written entirely in Java platform-independent and offers the best performance. *Highly recommended for modern Java applications.*

2.3 Develop Programs using JDBC to Query and Modify a Database

The `Statement` and `PreparedStatement` interfaces are used to execute SQL queries. `PreparedStatement` is preferred for dynamic queries as it prevents SQL injection and improves performance through pre-compilation.

Methodology:

Executing CRUD Operations with PreparedStatement

1. **Create (Insert):** Use `INSERT INTO` SQL. Call `executeUpdate()`.
2. **Read (Select):** Use `SELECT` SQL. Call `executeQuery()` and iterate over the `ResultSet` using `rs.next()`.
3. **Update (Modify):** Use `UPDATE` SQL. Set parameters using `setInt()` `setString()` etc. Call `executeUpdate()`.

4. **Delete (Remove):** Use `DELETE FROM` SQL. Set parameters and call `executeUpdate()`.

Worked Example:

Query and Modify Employee Records Write a Java program to insert a new employee update their salary retrieve the details and then delete the record using `PreparedStatement`. Assume an `employee` table with columns `id` (int) `name` (varchar) and `salary` (double).

Solution:

```
import java.sql.*;

public class EmployeeManagement {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/companydb";
        String user = "root";
        String password = "password123";

        try (Connection conn = DriverManager.getConnection(url, user, password)) {
            // 1. Insert Record
            String insertSql = "INSERT INTO employee (id, name, salary) VALUES (?, ?, ?)";
            try (PreparedStatement pstmt = conn.prepareStatement(insertSql)) {
                pstmt.setInt(1, 101);
                pstmt.setString(2, "John Doe");
                pstmt.setDouble(3, 50000.0);
                pstmt.executeUpdate();
                System.out.println("Employee inserted.");
            }

            // 2. Update Record
            String updateSql = "UPDATE employee SET salary = ? WHERE id = ?";
            try (PreparedStatement pstmt = conn.prepareStatement(updateSql)) {
                pstmt.setDouble(1, 55000.0);
                pstmt.setInt(2, 101);
                pstmt.executeUpdate();
                System.out.println("Employee salary updated.");
            }

            // 3. Select Record
            String selectSql = "SELECT * FROM employee WHERE id = ?";
            try (PreparedStatement pstmt = conn.prepareStatement(selectSql)) {
                pstmt.setInt(1, 101);
                try (ResultSet rs = pstmt.executeQuery()) {
                    while (rs.next()) {
                        System.out.println("ID: " + rs.getInt("id") +
                                           ", Name: " + rs.getString("name") +
                                           ", Salary: " + rs.getDouble("salary"));
                    }
                }
            }

            // 4. Delete Record
            String deleteSql = "DELETE FROM employee WHERE id = ?";
            try (PreparedStatement pstmt = conn.prepareStatement(deleteSql)) {
                pstmt.setInt(1, 101);
                pstmt.executeUpdate();
```

```
                System.out.println("Employee deleted.");
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

2.4 Object Relational Mapping

Object-Relational Mapping (ORM) is a programming technique used to convert data between incompatible type systems in object-oriented programming languages and relational databases. It creates a "virtual object database" that can be used from within the programming language.

Methodology:

Implementing ORM Concepts To map a Java class to a database table conceptually:

1. **Class to Table:** A Java class represents a database table.
2. **Object to Row:** An instance (object) of the class represents a single row in the table.
3. **Attribute to Column:** The properties (fields) of the class represent the columns of the table.
4. **Data Types:** Ensure Java data types map correctly to SQL data types (e.g. `String` to `VARCHAR` `int` to `INT`).

Worked Example:

Mapping a Java Class to a Relational Table Demonstrate the conceptual mapping of a `Student` Java class to a relational database table.
Solution:
Java Class Representation:

```
public class Student {
    private int rollNo;      // Maps to Primary Key column
    private String name;     // Maps to VARCHAR column
    private double cpi;      // Maps to DECIMAL column

    // Getters and Setters omitted for brevity
}
```

Equivalent SQL Table Schema:

```
CREATE TABLE Student (
    rollNo INT PRIMARY KEY,
    name VARCHAR(100),
    cpi DECIMAL(4, 2)
);
```

In an ORM framework like Hibernate, this mapping is explicitly defined using XML configuration or Java Annotations (e.g. `@Entity` `@Table` `@Id` `@Column`).

2.5 Hibernate Architecture and Environment Setup

Hibernate is a powerful high-performance ORM framework for Java. Its architecture consists of several core interfaces such as `Configuration` `SessionFactory` `Session` `Transaction` and `Query`.

Methodology:

Steps for Hibernate Environment Setup

1. **Add Dependencies:** Include Hibernate core and JDBC driver JAR files in the project classpath or `pom.xml`.
2. **Create Configuration File:** Create `hibernate.cfg.xml` to define database connection properties (URL username password dialect) and mapping resources.
3. **Create Entity Class:** Define a Java POJO class and annotate it with `@Entity` and `@Id`.
4. **Initialize Configuration:** Load the configuration file using `new Configuration().configure()`.
5. **Build SessionFactory:** Create a `SessionFactory` which is a thread-safe factory for `Session` objects.
6. **Open Session:** Obtain a `Session` to interact with the database.
7. **Manage Transaction:** Begin a `Transaction` perform CRUD operations using the `Session` and commit the transaction.

Worked Example:

Basic Hibernate Setup and Save Operation Write the required configuration and a Java application to save a `Product` entity using Hibernate.
Solution:
Step 1: Entity Class (`Product.java`)

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Product {
    @Id
    private int productId;
    private String productName;

    public Product() {}

    public Product(int productId, String productName) {
        this.productId = productId;
        this.productName = productName;
    }
    // Getters and setters omitted
}
```

Step 2: Configuration File (`hibernate.cfg.xml`)

```
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property
```

```
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/shopdb</propert
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">password123</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
        <property name="hibernate.hbm2ddl.auto">update</property>

        <mapping class="Product"/>
    </session-factory>
</hibernate-configuration>
```

Step 3: Main Application (`Main.java`)

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class Main {
    public static void main(String[] args) {
        // Load configuration and build SessionFactory
        Configuration cfg = new Configuration().configure("hibernate.cfg.xml");
        SessionFactory factory = cfg.buildSessionFactory();

        // Open session
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();

        // Create Entity object
        Product p = new Product(1, "Laptop");

        // Save the object
        session.save(p);

        // Commit transaction and close resources
        tx.commit();
        session.close();
        factory.close();

        System.out.println("Product saved successfully.");
    }
}
```

CHAPTER 3

Servlets

3.1 Describe Life Cycle of Servlet

A Servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model. The Servlet life cycle defines how a servlet is created, handles requests, and is destroyed.

Methodology:

Servlet Life Cycle Stages The life cycle of a Servlet is managed by the Servlet Container (like Apache Tomcat) and consists of the following sequential stages:

1. **Loading and Instantiation**: The servlet container loads the servlet class during startup or when the first request is made. It then creates an instance of the servlet.
2. **Initialization**: The container calls the `init(ServletConfig config)` method exactly once after instantiation. This method is used for one-time initialization setups such as database connections.
3. **Request Handling**: For every client request, the container spawns a new thread and calls the `service(ServletRequest req, ServletResponse res)` method. This method reads the request and generates the corresponding response.
4. **Destruction**: When the servlet container decides to unload the servlet (e.g. during server shutdown), it calls the `destroy()` method exactly once. This allows the servlet to release any held resources.

Worked Example:

Tracing the Servlet Life Cycle **Problem Statement:** Create a simple Servlet that overrides all life cycle methods and prints messages to the server console to demonstrate the sequence of execution.
Solution:
Step 1: Write the Servlet Class We create a class extending `GenericServlet` or implementing the `Servlet` interface directly.

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.GenericServlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebServlet;

@WebServlet("/LifeCycleDemo")
public class LifeCycleServlet extends GenericServlet {

    @Override
```

```
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        System.out.println("1. init() method is called.");
    }

    @Override
    public void service(ServletRequest req, ServletResponse res)
            throws ServletException, IOException {
        System.out.println("2. service() method is called.");
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html><body>");
        out.println("<h2>Check server console for Life Cycle messages</h2>");
        out.println("</body></html>");
    }

    @Override
    public void destroy() {
        System.out.println("3. destroy() method is called.");
    }
}
```

Step 2: Execution and Observation

1. When the URL `http://localhost:8080/App/LifeCycleDemo` is accessed for the **first time**, the server console prints:

```
1. init() method is called.
2. service() method is called.
```

2. When the same URL is accessed **subsequent times**, only the `service()` method is invoked. The console prints:

```
2. service() method is called.
```

3. When the server is stopped or the application is undeployed, the container removes the instance and the console prints:

```
3. destroy() method is called.
```

3.2 Develop Web Application Using javax.servlet Package

Developing a web application involves using the classes and interfaces defined in the `javax.servlet` and `javax.servlet.http` packages. The most common approach is to extend the `HttpServlet` class.

Methodology:

Steps to Develop an HttpServlet Application

1. **Setup Directory Structure**: Ensure your application has a `WEB-INF` folder, inside which you have a `classes` folder for compiled Java files and `lib` for external JARs.
2. **Create the Servlet Class**: Create a Java class that extends `javax.servlet.http.HttpServlet`.

3. **Override HTTP Methods**: Override `doGet()` to handle HTTP GET requests or `doPost()` for HTTP POST requests. Both methods take `HttpServletRequest` and `HttpServletResponse` as arguments.
4. **Set Content Type**: In the overridden method, specify the response type using `response.setContentType("text/html")`.
5. **Generate Response**: Obtain a `PrintWriter` object using `response.getWriter()` and write the HTML response output.
6. **Configure Routing**: Map the servlet to a specific URL pattern using the `@WebServlet("/urlPattern")` annotation or by configuring the `web.xml` deployment descriptor.

Worked Example:

Building a Simple Welcome HttpServlet **Problem Statement:** Develop an HTTP Servlet that responds to a GET request and displays a welcome message along with the current server time.
Solution:
Step 1: Write the HttpServlet Class

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/WelcomeServlet")
public class WelcomeServlet extends HttpServlet {

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

        // 1. Set response content type
        response.setContentType("text/html");

        // 2. Get the PrintWriter to write the response
        PrintWriter out = response.getWriter();

        // 3. Write HTML content
        out.println("<html>");
        out.println("<head><title>Welcome Servlet</title></head>");
        out.println("<body>");
        out.println("<h1>Welcome to Servlets!</h1>");
        out.println("<p>The current server date and time is: " + new Date() + "</p>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Step 2: Deployment and Testing

1. Compile the Java class ensuring the Servlet API JAR is in the classpath.

2. Deploy the application to a Servlet container like Apache Tomcat.
3. Access the URL: `http://localhost:8080/YourAppName/WelcomeServlet`.
4. The browser displays the welcome message and the current date and time generated dynamically.

Worked Example:

Processing Form Data using HTTP POST **Problem Statement:** Create an HTML form that accepts a user's name and email. Develop an HTTP Servlet to process this data using the POST method and display a confirmation page.

Solution:

Step 1: Create the HTML Form (index.html)

```
<!DOCTYPE html>
<html>
<head><title>User Registration</title></head>
<body>
    <h2>Enter Your Details</h2>
    <form action="RegisterServlet" method="POST">
        Name: <input type="text" name="userName"><br><br>
        Email: <input type="email" name="userEmail"><br><br>
        <input type="submit" value="Register">
    </form>
</body>
</html>
```

Step 2: Write the HttpServlet Class

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/RegisterServlet")
public class RegisterServlet extends HttpServlet {

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

        // 1. Retrieve form parameters
        String name = request.getParameter("userName");
        String email = request.getParameter("userEmail");

        // 2. Set response content type
        response.setContentType("text/html");

        // 3. Generate the response
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Registration Success</title></head>");
        out.println("<body>");
        out.println("<h2>Registration Successful!</h2>");
```

```
        out.println("<p>Welcome, " + name + "</p>");
        out.println("<p>Your registered email is: " + email + "</p>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Step 3: Execution Steps

1. Access the `index.html` page in the browser.
2. Enter a name (e.g. "John Doe") and email (e.g. "john@example.com").
3. Click the "Register" button. The form data is sent to `RegisterServlet` via an HTTP POST request.
4. The Servlet processes the data using `request.getParameter()` and generates a customized HTML response confirming the registration.

CHAPTER 4

Java Server Pages

4.1 JSP Syntax and Semantics

Methodology:

JSP Scripting Elements JSP pages use several scripting elements to embed Java code within HTML.

1. **Declaration Tag** (`<%! ... %>`): Used to declare variables, methods, or classes that are available to the entire JSP page. Example: `<%! int count = 0; %>`
2. **Scriptlet Tag** (`<% ... %>`): Used to embed any valid Java code. The code is executed each time the page is requested. Example: `<% count++; %>`
3. **Expression Tag** (`<%= ... %>`): Used to evaluate a Java expression and convert the result to a String, which is then inserted directly into the output. Example: `<%= count %>`
4. **Directive Tag** (`<%@ ... %>`): Provides global instructions to the JSP container (e.g., importing packages). Example: `<%@ page import="java.util.*" %>`

Worked Example:

Computing Factorial using JSP Create a simple JSP page that calculates the factorial of a given number using JSP scripting elements.
Step 1: Write the Declaration Declare a method to calculate the factorial using the `<%! %>` tag.

```
<%!
    long factorial(int n) {
        if (n == 0 || n == 1) return 1;
        else return n * factorial(n - 1);
    }
%>
```

Step 2: Write the Scriptlet Define the number for which the factorial needs to be computed using the `<% %>` tag.

```
<%
    int number = 5;
%>
```

Step 3: Write the Expression Output the result using the `<%= %>` tag inside standard HTML.

```
<html>
<body>
    <h2>Factorial Calculation</h2>
    <p>The factorial of <%= number %> is <%= factorial(number) %>.</p>
</body>
</html>
```

4.2 JSP Form Input Elements and Validation

Methodology:

Handling and Validating Form Data Processing form data in JSP involves fetching parameters sent from an HTML form and applying validation logic before proceeding.

1. **Client-Side Form**: Create an HTML form using the `<form>` tag with `method="post"` or `method="get"` pointing to the target JSP.
2. **Fetch Parameters**: In the destination JSP, use `request.getParameter("fieldName")` to retrieve input values as strings.
3. **Validation Logic**: Check if the retrieved strings are null, empty, or match specific constraints (e.g., length, numeric parsing).
4. **Feedback**: Output success messages or validation error messages back to the client using `out.println()` or Expression tags.

Worked Example:

Student Registration Form with Server-Side Validation Implement an HTML form for student registration and a JSP page to validate the input data (name and age).

Step 1: Create the HTML Form (index.html)

```
<form action="validate.jsp" method="post">
    Name: <input type="text" name="studentName"><br>
    Age: <input type="text" name="studentAge"><br>
    <input type="submit" value="Register">
</form>
```

Step 2: Create the Validation JSP (validate.jsp) Retrieve the parameters and validate them. The name must not be empty, and the age must be a valid integer between 18 and 35.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<html>
<body>
<%
    String name = request.getParameter("studentName");
    String ageStr = request.getParameter("studentAge");

    boolean isValid = true;
    String errorMessage = "";

    // Validate Name
    if (name == null || name.trim().isEmpty()) {
        isValid = false;
        errorMessage += "Name cannot be empty.<br>";
    }

    // Validate Age
    int age = 0;
    if (ageStr == null || ageStr.trim().isEmpty()) {
        isValid = false;
        errorMessage += "Age cannot be empty.<br>";
    } else {
        try {
```

```
            age = Integer.parseInt(ageStr);
            if (age < 18 || age > 35) {
                isValid = false;
                errorMessage += "Age must be between 18 and 35.<br>";
            }
        } catch (NumberFormatException e) {
            isValid = false;
            errorMessage += "Age must be a valid number.<br>";
        }
    }

    if (isValid) {
        out.println("<h3>Registration Successful!</h3>");
        out.println("Welcome, " + name + ". Your age is " + age + ".");
    } else {
        out.println("<h3 style='color:red;'>Registration Failed</h3>");
        out.println(errorMessage);
    }
%>
</body>
</html>
```

4.3 JSP Cookies and Session Tracking

Methodology:

Managing Cookies and Sessions Sessions and Cookies allow a web application to maintain state and track user interactions across multiple HTTP requests.

1. **Session Tracking**:
    - Create or retrieve a session using the implicit `session` object.
    - Store data using `session.setAttribute("key", value)`.
    - Retrieve data using `session.getAttribute("key")`.
    - Destroy the session using `session.invalidate()`.
2. **Cookies**:
    - Create a cookie: `Cookie c = new Cookie("key", "value");`
    - Set expiration (in seconds): `c.setMaxAge(60 * 60);`
    - Add to response: `response.addCookie(c);`
    - Read cookies: `Cookie[] cookies = request.getCookies();`

Worked Example:

Implementing a Visit Counter using Sessions and Cookies Build a JSP page that uses a session to track hits during the current active session, and a cookie to track all-time visits from the user's browser.
Step 1: Initialize Variables Retrieve the session attribute and the cookie array.

```
<%
    // 1. Session Tracking
    Integer sessionHits = (Integer) session.getAttribute("hits");
    if (sessionHits == null) {
        sessionHits = 1;
```

```
    } else {
        sessionHits += 1;
    }
    session.setAttribute("hits", sessionHits);

    // 2. Cookie Tracking
    int cookieHits = 1;
    Cookie[] cookies = request.getCookies();
    boolean newVisitor = true;

    if (cookies != null) {
        for (Cookie c : cookies) {
            if (c.getName().equals("totalHits")) {
                cookieHits = Integer.parseInt(c.getValue()) + 1;
                c.setValue(String.valueOf(cookieHits));
                c.setMaxAge(365 * 24 * 60 * 60); // 1 year
                response.addCookie(c);
                newVisitor = false;
                break;
            }
        }
    }

    if (newVisitor) {
        Cookie c = new Cookie("totalHits", "1");
        c.setMaxAge(365 * 24 * 60 * 60);
        response.addCookie(c);
    }
%>
```

Step 2: Display Results Output the hit counts.

```
<html>
<body>
    <h2>Website Hit Counter</h2>
    <p>Hits this session: <%= sessionHits %></p>
    <p>All-time hits: <%= cookieHits %></p>
</body>
</html>
```

4.4 JSTL

Methodology:

Using JSTL Core Tags JSTL simplifies JSP by replacing scriptlets with concise, XML-like tags.

1. **Importing JSTL**: Add the `taglib` directive at the top of the JSP. `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>`
2. **Output (`<c:out>`)**: Displays the result of an expression. `<c:out value="${variable}" />`
3. **Setting Variables (`<c:set>`)**: Sets a variable within a specific scope. `<c:set var="name" value="Value" scope="session" />`
4. **Conditionals (`<c:if>` and `<c:choose>`)**: `<c:if test="${condition}">...</c:if>`

5. **Iteration (`<c:forEach>`)**: Loops over collections or arrays. `<c:forEach var="item" items="${collection}">...</c:forEach>`

Worked Example:

Iterating Over a List with JSTL Create a list of strings in a Servlet or Scriptlet and use JSTL to display the list items conditionally.

Step 1: Setup the List and Import Taglib Use a scriptlet to initialize a list for demonstration purposes, then expose it to the page scope.

```
<%@ page import="java.util.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%
    List<String> cities = new ArrayList<>();
    cities.add("Ahmedabad");
    cities.add("Surat");
    cities.add("Vadodara");
    cities.add("Rajkot");
    pageContext.setAttribute("cityList", cities);
%>
```

Step 2: Use JSTL Tags to Render Output Iterate through the `cityList` and highlight "Ahmedabad".

```
<html>
<body>
    <h3>Major Cities in Gujarat</h3>
    <ul>
        <c:forEach var="city" items="${cityList}">
            <c:choose>
                <c:when test="${city == 'Ahmedabad'}">
                    <li><b><c:out value="${city}" /> (Mega City)</b></li>
                </c:when>
                <c:otherwise>
                    <li><c:out value="${city}" /></li>
                </c:otherwise>
            </c:choose>
        </c:forEach>
    </ul>
</body>
</html>
```

4.5 JSP Database Connection

Methodology:

Connecting to a Database via JSP JDBC (Java Database Connectivity) allows JSP pages to interact with relational databases.

1. **Load Driver**: Use `Class.forName("com.mysql.cj.jdbc.Driver")` to load the database driver.
2. **Establish Connection**: Call `DriverManager.getConnection(url, user, password)`.
3. **Create Statement**: Create a `Statement` or `PreparedStatement` object using the connection.
4. **Execute Query**: Use `executeQuery()` for `SELECT` statements or `executeUpdate()` for `INSERT`, `UPDATE`, `DELETE`.

5. **Process Results**: Iterate over the `ResultSet` (for queries) to extract and display data.
6. **Close Resources**: Always close `ResultSet`, `Statement`, and `Connection` in a finally block or at the end of the scriptlet.

Worked Example:

Retrieving Employee Records using JDBC in JSP Connect to a MySQL database named `company_db` and display records from the `employees` table in an HTML table.
Step 1: Database and Table Setup (SQL Context) Assume the following table exists in MySQL:

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

Step 2: Write the JSP Page (employees.jsp)

```
<%@ page import="java.sql.*" %>
<html>
<body>
    <h2>Employee List</h2>
    <table border="1" cellpadding="5">
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Department</th>
        </tr>
<%
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    try {
        // 1. Load the Driver
        Class.forName("com.mysql.cj.jdbc.Driver");

        // 2. Establish Connection
        String url = "jdbc:mysql://localhost:3306/company_db";
        String user = "root";
        String password = "password";
        conn = DriverManager.getConnection(url, user, password);

        // 3. Create Statement
        stmt = conn.createStatement();

        // 4. Execute Query
        String sql = "SELECT * FROM employees";
        rs = stmt.executeQuery(sql);

        // 5. Process Results
        while (rs.next()) {
            int id = rs.getInt("id");
            String name = rs.getString("name");
            String dept = rs.getString("department");
```

```
%>
            <tr>
                <td><%= id %></td>
                <td><%= name %></td>
                <td><%= dept %></td>
            </tr>
<%
        }
    } catch (Exception e) {
        out.println("<tr><td colspan='3'>Error: " + e.getMessage() + "</td></tr>");
    } finally {
        // 6. Close Resources
        if (rs != null) try { rs.close(); } catch(SQLException e) {}
        if (stmt != null) try { stmt.close(); } catch(SQLException e) {}
        if (conn != null) try { conn.close(); } catch(SQLException e) {}
    }
%>
    </table>
</body>
</html>
```

CHAPTER 5

Web MVC Framework

5.1 Explain Web Application Framework

Methodology:

Web Application Framework Workflow A web application framework provides a structured approach and libraries for building dynamic websites and web services. The standard algorithmic workflow includes:

1. **Routing:** The framework receives an HTTP request and maps the URL to a specific handler or controller method.
2. **Processing:** The handler executes business logic and interacts with the database or calls external services.
3. **Data Encapsulation:** Results are wrapped in a structured format (like Model objects).
4. **View Resolution:** The framework identifies the appropriate presentation template (View).
5. **Response Generation:** Data is rendered into the template (HTML or JSON) and returned to the client.

Worked Example:

Identify Framework Components **Problem:** Given an HTTP GET request to `/users/profile`, identify the step-by-step process a web framework takes to handle it.
Solution:

1. **Request Reception:** The web server intercepts the HTTP GET request for `/users/profile`.
2. **Route Matching:** The framework's router finds the controller mapped to the `/users/profile` path (e.g. a `UserProfileController` class).
3. **Execution:** The matched controller method executes. It fetches user data from the database using a dedicated service class.
4. **Model Population:** The retrieved user data is added to a Model object representing the current state.
5. **View Rendering:** The framework merges the Model data with a pre-defined view template (e.g. `profile.jsp` or `profile.html`).
6. **Response Transmission:** The final HTML string is sent back as an HTTP response to the client's browser.

5.2 Describe MVC Architecture Layers

Methodology:

MVC Architecture Execution Steps The Model-View-Controller (MVC) pattern separates a web application into three distinct interconnected components to manage data, interface, and control flow:

1. **Model (Data Layer):** Represents the application's data and business rules. It executes database queries, retrieves data, and updates state.
2. **View (Presentation Layer):** Renders the user interface. It reads data from the Model and displays it in a format suitable for user interaction.
3. **Controller (Logic Layer):** Intercepts incoming user requests from the View, processes inputs, triggers changes to the Model, and determines the next View to display.

Execution Workflow:

1. The user interacts with the View (e.g. submitting a web form).
2. The Controller receives the request and determines the required business action.
3. The Controller interacts with the Model to update data or retrieve new information.
4. The Controller selects the appropriate View and passes the updated Model data to it.
5. The View renders the final output and sends it to the user.

5.3 Implement web application using Spring Framework

Methodology:

Spring MVC Setup and Implementation Steps To implement a robust web application using Spring MVC, follow these procedural steps:

1. **Configure DispatcherServlet:** Declare the `DispatcherServlet` in `web.xml` to act as the Front Controller handling all incoming HTTP requests.
2. **Create Spring Configuration:** Define a `spring-servlet.xml` file to enable component scanning and configure the ViewResolver.
3. **Create Model Class:** Define Plain Old Java Objects (POJOs) to represent data entities.
4. **Create Controller:** Annotate a Java class with `@Controller` and its methods with `@RequestMapping` to handle specific URLs and request types.
5. **Create Views:** Write JSP or HTML pages to format and display the data passed by the Controller.

Worked Example:

Spring MVC Web Application Implementation **Problem:** Implement a Spring MVC application that takes a user's name from an input form and displays a personalized greeting message.
Solution:
Step 1: Configure `web.xml` Define the `DispatcherServlet` to intercept web requests.

```
<web-app>
  <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>
```

```
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

Step 2: Create `spring-servlet.xml` Enable annotations and configure the ViewResolver to locate JSP files.

```
<beans>
  <context:component-scan base-package="com.example.controller" />
  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
  </bean>
</beans>
```

Step 3: Create Controller Class

```
package com.example.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class GreetingController {

    @RequestMapping("/greet")
    public String showGreeting(@RequestParam("name") String name, Model model) {
        // Business logic to format message
        model.addAttribute("message", "Hello " + name + "!");
        // Returns the view name "result"
        return "result";
    }
}
```

Step 4: Create Views Input form (`index.jsp`):

```
<form action="greet">
  Enter Name: <input type="text" name="name" />
  <input type="submit" value="Greet" />
</form>
```

Output page (`/WEB-INF/views/result.jsp`):

```
<h2>${message}</h2>
```

5.4 Describe Spring Boot

Methodology:

Spring Boot Setup and Execution Spring Boot is an opinionated extension of the Spring framework that eliminates boilerplate configuration by providing auto-configuration and embedded servers.

1. **Initialize Project:** Use Spring Initializr to create a base project structure with the `spring-boot-starter-web` dependency.
2. **Main Application Class:** Annotate the main class with `@SpringBootApplication` to enable auto-configuration.
3. **Create REST Controller:** Annotate a class with `@RestController` and its methods with `@GetMapping` or `@PostMapping` to handle HTTP requests directly.
4. **Run Application:** Execute the `main` method. Spring Boot automatically starts an embedded Tomcat server and deploys the application.

Worked Example:

Create a Spring Boot REST API **Problem:** Create a simple Spring Boot REST API that returns user details in JSON format when the `/api/user` endpoint is accessed.
Solution:
Step 1: Set up Dependencies (`pom.xml`) Ensure the Spring Boot Web Starter dependency is included in the project configuration.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Step 2: Create Main Application Class This serves as the entry point of the Spring Boot application.

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Step 3: Create User Model (POJO) Defines the structure of the data object.

```
package com.example.demo;

public class User {
    private int id;
    private String name;

    public User(int id, String name) {
        this.id = id;
        this.name = name;
    }
```

```
    public int getId() { return id; }
    public String getName() { return name; }
}
```

Step 4: Create REST Controller Maps the HTTP request to the corresponding method and automatically serializes the returned Java object to JSON.

```
package com.example.demo;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

    @GetMapping("/api/user")
    public User getUser() {
        return new User(1, "John Doe");
    }
}
```

Step 5: Execution and Result Run `DemoApplication.main()`. The embedded Tomcat server starts automatically on port 8080. Accessing `http://localhost:8080/api/user` in a browser or API client returns the following JSON response:

```
{
  "id": 1,
  "name": "John Doe"
}
```

Appendix: Key Reference Points

This section typically contains key formulas. However, as this course primarily focuses on theoretical, conceptual, or programming methodologies, mathematical formulae are not applicable.