(Naima
IT22001)

1| Demonstrate how a child class can access a protected member of its parent class within the same package. Explain with example what happens when the child class is in a different package.

Ans: In Java, protected members of a class can be accessed:

(1) Within the same package - by any class

(2) From a subclass - even if the subclass is in a different package.

But there's a difference in how they are accessed.

case-1: child class in the same package:

When the child class is in the same package, it can directly access the protected member of the parent.

* myfamily/Naima.java:

```
package myfamily;
public class Naima {
    protected String message = "Bye, Bye";
}
```

P.T.O

* myfamily/Rafatul.java:

```
package myfamily;
public class Rafatul extends Naima {
    public void showMessage() {
        // Direct access to protected member
        System.out.println(message);
    }
}
```

* myfamily/Main.java:

```
Package myfamily;
public class Main {
    public static void main (String[] args) {
        Rafatul n = new Rafatul();
        n.showMessage(); // Output: Bye, Bye
    }
}
```

Case-2: child class in a different package:

In this case, the child can access the protected member, but only through inheritance using 'this' or inherited reference, not by using a parent reference.

P.T.O.

* parentpkg/Naima.java:

```
package parentpkg;
public class Naima {
    protected String message = "Life is short";
}
```

* childpkg/Kuldip.java:

```
package childpkg;
import parentpkg.Naima;
public class Kuldip extends Naima {
    public void showMessage() {
        System.out.println(this.message);
    }
}
```

* childpkg/Main.java:

```
package childpkg;
public class Main {
    public static void main(String[] args) {
        ~~Rafatul n = new P~~
        Kuldip n = new Kuldip();
        n.showMessage(); // output: Life is short
    }
}
```

Lab:1:

2|| Compare abstract classes and interfaces in terms of multiple inheritance. When would you prefer to use an abstract class and when an interface?

Ans:

Comparison of abstract classes and interfaces in terms of multiple inheritance:

Feature	Abstract Class	Interface
Multiple inheritance	Not supported	Fully supported
Method types	Can have abstract and concrete methods	Only abstract methods (Java 7), default/static (Java 8+)
Constructor	Can have constructors	Can not have constructors.
State (fields/variables)	Can have instance variables (with any access)	Only public static final constants
Access Modifiers	Can use any (private, protected etc)	Methods are public by default.

hen to use an abstract class:

① For want to provide base functionality that all child classes share.
② For need to define state (fields) or constructors.
③ For want to control over method access (public, protected, etc).
④ For creating a tight "is-a relationship (e.g. Naima likes is a girl).

Example:

```
abstract class Naima {
protected String name;
abstract void makeSound();
void eat(){
    System.out.println("Naima is eating");
  }
}
```

hen to use an Interface:

① For need to achieve multiple intercitance
② For defining a contract for what a class should do (capabilities).
③ The methods are behavioral declaration, not implementation.
④ For working with unrelated classes that share capabilities.

Example:

```
interface Flyable{
    void fly();
}
interface Swimmable{
    void swim();
}
class Duck implements Flyable, Swimmable{
    public void fly(){
        System.out.println("Duck flying");}
    public void swim(){
        system.out.println("Duck swimming");
    }
}
```

Lab-2:

3] How does encapsulation ensure data security and integrity? Show with BankAccount class using private variables and validated methods such as setAccountNumber (String), setInitialBalance (double) that rejects null, negative or empty values.

Ans:

Encapsulation is a object-oriented principle where class variable are made private and accessed only through public methods (getters/setters). This protects data from unauthorized access or invalid modification.

It ensures:

* Data security → by hiding internal variables from outside access.
* Data Integrity → by validating input value before setting them.

*"BankAccount" class using encapsulation:

```
public class BankAccount {
    private String accountNumber;
    private double balance;
```

p.T.O

```
public void setAccountNumber(String accountNumber){
    if(accountNumber == null || accountNumber.
        trim().isEmpty()){
        System.out.println("Invalid account Number!");
    }else{
        this.accountNumber = accountNumber;
    }
}
public void setInitialBalance(double amount){
    if(amount < 0){
        system.out.println("Initial balance can't be
                            negative!");
    }else{
        this.balance = amount;
    }
}
public String getAccountNumber(){
    return accountNumber;
}
public double getBalance(){
    return balance;
}
}
```

*Test code:

```
public class Main {
    public static void main (String [] args) {
        BankAccount acc = new Bankaccount();
        acc.setAccountNumber (" ");  // Invalid
        acc.setAccountNumber ("Acc001"); // Valid.
        acc.setInitialBalance (-200);  // Invalid
        acc.setInitialBalance (1500);  // Valid
        System.out.println ("Account :"+ acc.getAccountNumber
                                                    ());
        System.out.println ("Balance : $" + acc.getbalance());
    }
}
```

4] Write Program to- Any (3)

Ans:

(i) Find the kth smallest element in an arrayList:

```
import java.util.*;
public class KthSmallest{
  public static int findKthSmallest(
      ArrayList<Integer> list, int k){
    Collections.sort(list);
    return list.get(k-1);
  }
}
```

(v) Check if two LinkedLists are equal:

```
class LinkedListEquality{
  public static boolean areEqual(LinkedList<
      Integer> List1, LinkedList<Integer> list2){
    return list1.equals(list2);
  }
}
```

(vi) HashMap for Employee ID to Department:

```
class EmployeeDepartment{
    public static HashMap<Integer, String> getEmployeeMap(){
        HashMap<Integer, String> map = new HashMap<>();
        map.put(1, "HR");
        map.put(2, "IT");
        map.put(3, "Finance");
        return map;
    }
}
```

ab-3:

// Multi threaded Java simulation of a "Car Parking Management System".

RegistrarParking.java:

```
ublic class RegistrarParking{
 private final String carNumber;
ublic RegistrarParking (String carNumber){
 this.carNumbere = carNumbere;
ublic String getCarNumber(){
return carNumber;
```

ingPool.java:

```
- java.util.LinkedList;
- java.util.Queue;
lass ParkingPool{
- final Queue<RegistrarParking>
arkingQueue = new LinkedList<>();
nchronized void addRequest(RegistrarParking
                                car){
Queue.add(car);
ut.println("Car" + car.getCarNumber() +
);
```

```
public synchronized RegistrantParking getNextCar(){
  while (parkingQueue.isEmpty()){
   try {
     wait();
   } catch(InterruptedException e){
     Thread.currentThread().interrupt();
     return null;
   }
  }
  return parkingQueue.poll();
 }
}
```

ParkingAgent.java:

```
public class ParkingAgent extends Thread {
  private final int agentId;
  private final ParkingPool pool;

  public ParkingAgent(int agentId, ParkingPool pool){
    this.agentId = agentId;
    this.pool = pool;
  }
@Override
public void run(){
  while (true){
   RegistrantParking car = pool.getNextCar();
   if (car != null){
     System.out.println("Agent" + agentId + "parked car" +
                        car.getCarNumber() + ".");
```

```
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

MainClass.java :

```
public class MainClass {
    public static void main(String[] args) {
        ParkingPool = new ParckingPool();

        parckingAgent agent1 = new ParkingAgent(1, pool);
        parkingAgent agent2 = new ParckingAgent(2, pool);

        agent1.start();
        agent2.start();

        String[] cardNumbers = {"ABC123", "XY2456",
                                "CAR789", "NRK321"};
        for (String number : cardNumber) {
            new Thread(() -> {
                RegistrarParking car = new RegistrarParding
                                        (number);
                pool.addRequest(car);
            }).start();
```

```
try {
    Thread.sleep(300);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
}
}
}
```

6) How does Java handel XML data using Dom and SAX parsers? Compare both approaches with respect to memory usage, processing speed and use cases. Provide a seenario where SAX would be preferred over DOM.

Ans:

Java handeling XML data using Dom (Document Object Model) Parcserc:

- Loads the entire XML file into memory as a tree structure.
- Allows random access and modification of nodes.
- Part of javax.xml.parsers package.

Handeling XML data using SAX (Simple API for XML) Parsers:

- Uses an event-driven approach.
- Does not load entire XML into memory.
- Triggers callbacks like startElement, endElement, etc
- More memory-efficient for Large files.

Comparison of DOM and SAX:

Feature	DOM Parser	SAX Parser
Memory Usage	High (loads entire XML into memory)	Low (reads one tag at a time)
Processing speed	Slower for large files	Faster for large files
Access Type	Random access to any node	Sequential, forward only
Modifiability	Can modify XML tree	Cannot modify (read only)
Ease of Use	Easier, object-based tree	Harder, callback/event-based.
Use case	Small XML with frequent access	Large XML; read only processing

A scenario where SAX would be preferred over DOM:

If processing a log file in XML with millions of entries - Like a server log or transaction log.

→ We only need to count the number of failed transactions, not load or modify the whole file.

SAX is better because:

① SAX reads the file line by-line (low memory).

② DOM would crash or become very slow due to high memory usage.

* Use DOM ⟶ when we need to modify, navigate or access the full structure.

* Use SAX ⟶ for efficient reading of large XML files, especially when you only need specific parts.

7|| How does the virtual DOM in React improve performance? Compare it with the traditional DOM and explain the diffing algorithm with a simple component update example.

Ans:

The Virtual DOM (VDOM) is a lightweight in-memory representation of the real DOM. Instead of updating the actual DOM directly whenever changes occure (which is slow) React:

1. Maintains a virtual copy of the DOM in memory
2. Updates this virtual DOM first.
3. Uses a diffing algorithm to compare the old and new virtual DOM.
4. Efficiently updates only the parts of the real DOM that actually changed.

Comparison of Virtual DOM vs Traditional DOM:

Feature	Traditional DOM	Virtual DOM (React)
Update type	Directly manipulates the real DOM	Uses a virtual copy and updates only diffs.
Speed	Slower for large updates	Faster due to minimal changes
Efficiency	Rerender entire elements/trees	Only updates what has changed.
Memory Usage	Lower but more DOM manipulation	Slightly more memory (for VDOM) but efficient
Code Maintenance	Manual DOM handling	Declarative updates using JSX/components.

React uses a diffing algorcithm to:

- Compare old and new virtual DOM trees.
- Identify the smallest number of operations to update the real DOM.

Example:

We have:

```
function Message(){
  return <h1>Hellow</h1>;
}
```

Now update to:

```
function Message(){
  return <h1>Hello, world!</h1>;
}
```

8|| What is event delegation in JavaScript and how does it optimize performance? Explain with an example of a click event on dynamically added elements.

Ans:

Event Delegation is a technique in JavaScript where a single event listener is added to a parent element, instead of attaching separate listeneres to each child elements. It uses event bubbling, where event propagate (bubble up) from the target element to its ancestors.

Performance optimization:

① Reduces memory usage by minimizing the number of event listeneres.

② Especially beneficial when dealing with many child elements or dynamically added elements.

☐ Click Event on Dynamically added buttons.

Html:

```
<div id = "button-container">
</div>
<button onclick = "addButton()" > Add New Button
                                  </button>
```

JavaScript:

```
document.getElementById("button-container").addEventListener("click", function(e)
if(e.target && e.target.matches("button.dynamic-btn")){
alert("Button clicked:" + e.target.textContent);
}
});
function addButton(){
const container = document.getElementById("button-container");
const newBtn = document.createElement("button");
nextBtn.textContent = "dynamic button";
nextBtn.className = "dynamic-btn";
container.appendChild(newBtn);
}
```

9|| Explain how Java Regular Expressions can be used for input validation. Write a regex pattern to validate an email address and describe how it works using the Pattern and Matcher classes.

Ans: In Java regular expressions (regex) are used to define search patterns for strings commonly used for input validation such as checking email formats, phone numbers, passwords etc.

Email validation with Java Regex:

1. Regex Pattern for Email validation:

```
^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$
```

Pattern Breakdown:

Part	Meaning
^	start of the string
[A-Za-z0-9+_.-]+	Username part: letters, digits, +, _, ., - (at least one).
@	Literal @ symbol
[A-Za-z0-9.-]+	Domain name: letters, digits, ., -
$	end of the string.

Java code Example using pattern and Matcher:

```
import java.util.regex.*;
public class EmailValidator {
    public static void main (String[] args) {
        String email = "username123@example.com";
        String regex = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$";
        Pattern pattern = Pattern.compile (regex);
        Matcher matcher = pattern.matcher (email);
        if (matcher.matches()) {
            System.out.println ("Valid email address.");
        } else {
            System.out.println ("Invalid email address.");
        }
    }
}
```

How it works:

① Pattern.compile (regex)
→ Compiles the regex into a Pattern object

② pattern.matcher (input)
→ Creates a Matcher that will match the input against the pattern.

(5) matcher.matches()
→ Returns true if the entire string matches the regex.

10] What are custom annotations in Java and how can they be used to influence program behavior at runtime using reflection? Design a simple custom annotation and show how it can be processed with annotated elements.

Ans:

Custom Annotations in Java allow developers to define their own metadata tags that can be attached to classes, methods, fields etc. These annotations don't affect program execution directly - but they can be processed using reflection at runtime to influence program behavior.

They can use —

(1) To add metadata for configuration.
(2) To support custom framework, validation, logging etc.
(3) To enable declarative programming style.

1. Define Annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface RunMe {}
```

2. Use it:

```
public class MyService {
    @RunMe
    public void greet() {
        System.out.println("Hello!");
    }
}
```

3. Process with Reflection:

```
for (Method m : MyService.class.getDeclaredMethods()) {
    if (m.isAnnotationPresent(RunMe.class)) {
        m.invoke(new MyService());
    }
}
```

Output:

Hello!

11|| Discuss the singleton design patteren in Java. What problem does it solve and how does it ensure only one instance of a class is created? Extent your answer to explain how thread safety can be achieved in a singleton implementation.

Ans:

The singleton pattern ensures that only one instance of a class exists throughout the application.

Problem it solves:

- Prevents creation of multiple instances of a class
- Useful when a global access point to the object is needed

Singleton ensures one instance by —

(1) Private constructor — prevents direct object creation using 'new'.

(2) Static instance variable — holds the one-and-only instance.

(3) Public static method — returns the instance.

When multiple threads try to access the singleton class at the same time, they may create multiple instance. To prevent this, we must make it thread-safe.

① Synchronized Method (Simple but slower):

```
public class Singleton {
    private static Singleton instance;
    private Singleton() { }
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

② Double-checked locking (Efficient):

```
public class Singleton {
    private static volatile Singleton instance;
    private Singleton() { }
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

- Fast and thread-safe.
- Only locks on first creation.
- Volatile prevents memory issue.

(1) Synchronized Method (Simple but slower):

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

(2) Double-Checked locking (efficient):

```
public class Singleton {
    private static volatile Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Lab: 4:

12] Describe how JDBC manages communication between a Java application and a relational database. Outline the steps involved in executing a SELECT query and fetching results. Include error handling with try-catch and finally blocks.

Ans: JDBC (Java Database Connectivity) is an API that allows a Java application to interact with a relational database (like MySQL, PostgreSQL etc)

It provides methods to:

- Connect to the database.
- Execute SQL queries (SELECT, INSRT, UPDATE, DELETE)
- Fetch result.
- Handle exceptions and close connections.

Steps to Execute a SELECT Query Using JDBC:

① Import JDBC Package:

```
import java.sql.*;
```

② Register the JDBC Driver:

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

③ Create a Connection:

```
Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/mydb","username",
                                    "password");
```

④ Create a Statement:

```
Statement stmt = conn.createStatement();
```

⑤ Execute a SELECT Query:

```
String query = "SELECT id, name FROM students";
ResultSet rs = stmt.executeQuery(query);
```

⑥ Process the Results:

```
while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    System.out.println(id + " - " + name);
}
```

⑦ Error handling and Resource Management:

Use 'try-catch-finally' to handle exceptions and close resource safety.

13] How do Servlets and JSPs work together in a web application following the MVC (Model-View-Controller) architecture? Provide a brief use case showing the servlet as a controller, JSP as a view and a Java class as the model.

Ans: In a Java web application, Servlets and JSPs using MVC (Model-View-Controller):

① Model: Java classes that contain business logic or data.

② View: JSP (JavaServer Pages) used to present data to user (HTML + embedded Java).

③ Controller: Servlet that handles user input, updates the model and forwards data to the JSP.

Use case: Display user info:

① Model - User.java

```
public class User {
    private String name;
    private int age;
public User(String name, int age) {
    this.name = name;
    this.age = age;
}
public String getName() { return name; }
public int getAge() { return age; }
}
```

② Controller (Servlet) – UserServlet.java

```
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.IOException;
public class UserServlet extends HttpServlet {
    protected void doGet(HttpServletRequest
        request, HttpServletResponse response)
    throws ServletException, IOException {
        User user = new User("Naima", 22);
        request.setAttribute("user", user);
        RequestDispatcher dispatcher = request.getRequestDispatcher("user.jsp");
        dispatcher.forward(request, response);
    }
}
```

③ View (JSP) – user.jsp

```
<%@ page import = "Naima.package.User" %>
<%
    User user = (User) request.getAttribute("user");
%>
<html>
<body>
    <h2> User info </h2>
    <p> Name: <%= user.getName() %> </p>
    <p> Age : <%= user.getAge() %> </p>
</body> </html>
```

14|| Explain the life cycle of a Java Servlet. What are the roles of the init(), service() and destroy() methods? Discuss how servlets handle concurrent request and how thread safety issues may arise.

Ans: Life cycle of a Java Servlet:

A servlet's life cycle is managed by the servlet container (like Tomcat) and consists to 3 main stage

Stage	Method	Purpose.
Initialization	init()	Called once when the servlet is created.
Request Handling	service()	Called for every client request
Destruction	destroy()	Called once before the servlet is destroyed

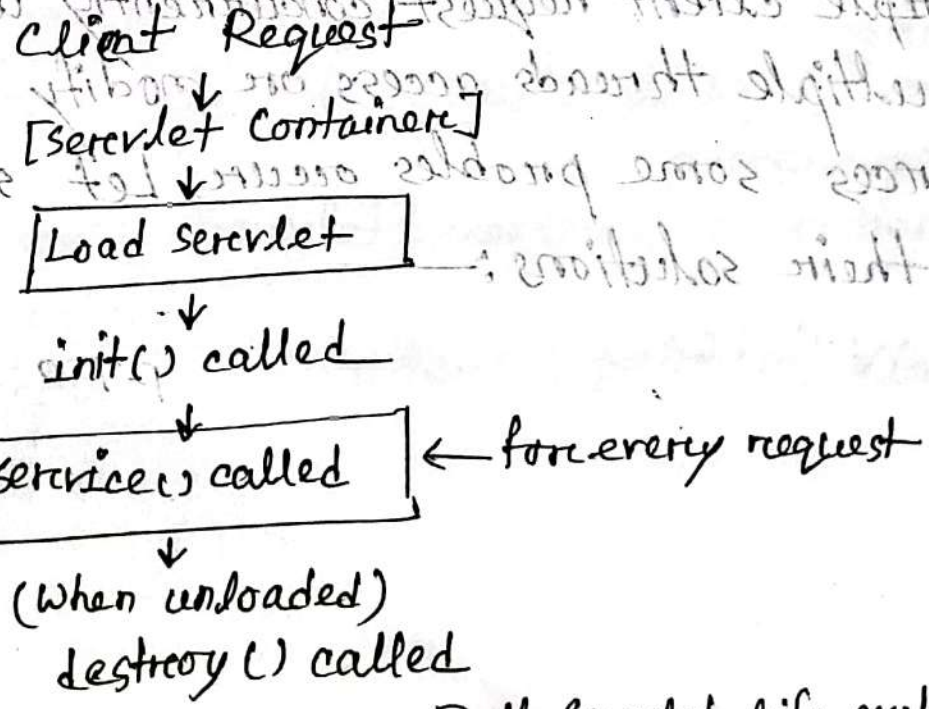


Fig: Full servlet life cycle diagram

How Servlets Handle Requests:

① One instance of the servlet handles multiple requests.

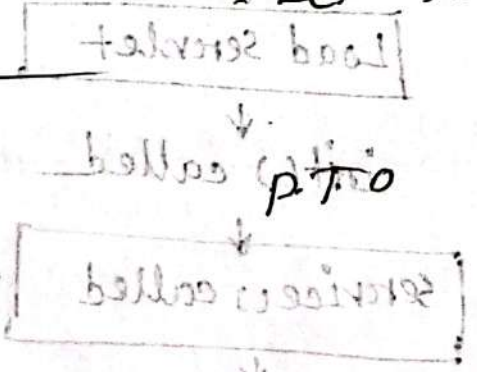
② Each request is served by a separate thread.

Thread Safety issues:

If multiple threads access shared data, race conditions or data inconsistency can occur.

15|| A single instance of a Servlet handles multiple request using threads What problems can occur if shared resources are accessed by multiple threads? Illustrate your answer with an example and suggest a solution using synchronization

Ans: In Java servlets a single instance handle multiple client request concurrently using threads. If multiple threads access or modify shared resources some probles occure. Let see them and their solutions:



Problem	Cause	Solution
① Race condition	Shared mutable state in threads	Use synchronized or AtomicInteger.
② Inconsistent data	Non-atomic operations Like ++	Ensure atomic operations.
③ Application errors	Overlapping access to variables	Avoid using instance variables or protect them.

Example of a Thread-Safety Issue:

```
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.*;
public class CounterServlet extends HttpServlet {
    private int counter = 0;
    protect void doGet(HttpServletRequest request, HttpServ-
                                                letResponse response)
        throws ServletException, IOException {
        counter++;
        response.getWriter().println("Visitor count: " +
                                                counter);
    }
}
```

Solution: Using "synchronized" Block:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
throws ServletException, IOException {
    synchronized (this) {
        counter++;
        response.getWriter().println("Visitor
                count: " + counter);
    }
}
```

16 || Describe how the MVC pattern separates concerns in a Java web Application. Explain the advantages of this structure in terms of maintainability and scalability, using a student registration system as an example.

Ans: MVC pattern separates concerns in a Java web application:

Concern	Handled by	Description
Data/Business	Model	Manages database, validation and rule
Display/UI	View (JSP/HTML)	Renders output to the user (forms, tables).
User interaction	Controller (Servlet)	Handles requests, updates model forwards to view.

① Model - Student.java:

```
public class Student {
    private String name;
    private int age;
}
```

② View - User Interface (JSP):

```
<!---- StudentFrom.jsp --->
<form action="register" method="post">
Name: <input name="name" />
Age: <input name="age" />
<input type="submit" value="Register" />
</form>
```

③ Controller - Request Handling (Servlet):

```
@WebServlet("/register")
public class StudentController extends HttpServlet {
protected void doPost (HttpServletRequest request,
                       HttpServletResponse response)
throws ServletException, IOException {
String name = request.getParameter("name");
int age = Integer.parseInt (request.getParameter("age");
Student student = new Student (name, age);
StudentDAO.save(student);
```

```
request.setAttribute("student", student);
request.getRequestDispatcher("success.jsp").
        forward(request, response);
    }
}
```

Advantages of MVC:

Benefit	Description
Separation of Concerns	Logic (Controller), data (Model) and UI (View) are clearly separated.
Maintainability	Easy to update UI or business logic independently
Scalability	Components can grow independently more developers can work in parallel.
Reusability	Models can be reused in other views or controllers.

Lab-5:

17|| In a Java EE application, how does a servlet controller manage the flow between the model and the view? Provide a brief example that demonstrates forwarding data from a servlet to a JSP and rendering a response.

Ans: In a Java EE application using MVC, the servlet acts as the controller.

① Recieves the request from the user (browser).
② Calls the Model (business logic / data layer).
③ Sets data as request attributes.
④ Forwards the request to a JSP (View) to render the response.

Example: Displaying a Student's Info:

1. Model - Student.java.

```
public class Student {
    private String name;
    private int age;
    public Student (String name, int age){
        this.name = name;
        this.age = age;
    } public String getName () { return name; }
      public int getAge () { return age; }
}
```

2. Servelet Controller - StudentServlet.java

```
@WebServlet("/student")
public class StudentServlet extends HttpServlet {
    protected void doGet(HttpServletRequest
        request, HttpServletResponse response)
    throws ServletException, IOException {
    Student student = new Student("Naima", 23);
    request.setAttribute("student", student);
    RequestDispatcher dispatchere = request.
                getRequestDisaptcher("student.jsp");
    dispatchere.forward(request, response);
}
}
```

3. View (JSP) - student.jsp

```
<%@ page import="Naima.Student" %>
<%
    Student student = (Student) request.getAttribute
                        ("student");
%>
<html>
<body>
    <h2>Student Info</h2>
    <P> Name: <%= student.getName() %></P>
```

Age : <%= student.getAge() %>
</body>
</html>

18|| Compare and contrast cookies, URL rewriting and HttpSession as methods for session tracking in servelets. Discuss their advantages, limitations and ideal use cases.

Ans: Comparison table:

Feature	Cookies	URL Rewriting	HttpSession.
Stored in	Client (browser)	URL as parameter	Server (in memory
Supports Objects	Only text	Only session ID	Any Java Object
Security	Visible and editable	(Session ID exposed in URL)	(Secure, server-side).
Ease of use	Moderate	Hard	Very easy
Advantages	* simple * Persistent if needed	* Works if cookies are disabled * No client storage	* stores full objects. * Fully hidden from user.
Limitations	* Size limit (~4KB) * can be disabled	* Insecure URLs * Must rewrite every link manually	* Server memory usage * Session expires with time.
Ideal use cases	Save theme, language, user perferences	Backup for session tracking when cookies are off	Login sessions, shopping carts, user specific data

19] A web application stores user login information using HttpSession. Explain how the session works across multiple requests and how session timeout or invalidation is handled securely.

Ans:

HttpSession works across multiple requests by

① Session creation:

When a user logs in, the servlet calls:

```
HttpSession session = request.getSession();
```

This creates a new session or returns the existing one

② Storing User Data:

```
session.setAttribute("username", "Naima01");
```

③ Session Tracking:

- A unique session ID is generated
- Sent to the client via a cookie
- On further requests, the browser sends the session ID, allowing the server to retrieve the same session.

Session Timeout and Invalidation:

① Automatic timeout:

The session expires after a period of inactivity

In web.xml:

```
<session-config>
    <session-timeout>15</session-timeout>
</session-config>
```

② Manual Invalidation:

```
session.invalidate();
```

Secure Session Handling:

1. Set timeout properly.
2. Use Https
3. Invalidate on logout
4. Regenerate ID on login
5. Don't store sensitive data.

Scanned with CamScanner

20| Explain how spring MVC handles an HTTP request from a browser. Describe the role of the @Controller, @RequestMapping and Model objects in separating business logic from presentation. Provide a brief flow example of a login form submission?

Ans: Spring MVC follows the Model-View-Controller pattern to handle web requests in a clean, layered way:

Browser Request
↓
DispatcherServlet (Front Controller)
↓
@Controller → Business Logic (Service/Model)
↓
View (JSP/HTML/Thymeleaf)

Key components and their roles:

① @Controller → Marks a class as a Controller to handle HTTP requests.

② @RequestMapping → Maps a URL to a specific method inside the controller.

③ Model object ⟶ Passes data from the controller to the view.

④ View (JSP/HTML) ⟶ Renders the final output to the user using model data.

Example: Login form submission flow:

1. Login form - login.jsp:

```
<form action="/login" method="post">
    Username: <input type="text" name="username" />
    <br />
    password: <input type="password" name="password" /><br />
    <input type="submit" value="Login" />
</form>
```

2. Controller - LoginController.java:

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
public class LoginController {
    @GetMapping("/login")
    public String showLoginForm() {
        return "login";
    }
```

p.7.0

```
@PostMapping("/login")
public String handleLogin(@RequestParam
        String username,
    @RequestParam String password,
    Model model) {
if ("admin".equals(username) && "1234".equals
                                (password)) {
    model.addAttribute("message", "Login
                                successful!");
    return "welcome";
} else {
    model.addAttribute("error", "Invalid
                                credentials!");
    return "login";
}
}
}
```

3. SuccessView — welcome.jsp:

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

21|| Spring MVC uses the DispatcherServlet as a front controller. Describe its role in the request processing workflow. How does it interact with view resolvers and handler mappings?

Ans:

Role of DispatcherServlet:

Task	Description
① Acts as Front controller	Central entry point for all incoming web requests.
② Delegates to Handler Mappings	Finds the correct @Controller method based on URL
③ Invokes Controller Logic	Executes the controller method to process the request.
④ Uses View Resolver	Determines the correct view to return as a response.
⑤ Returns the Response	Renders the view using the Model and sends output to the client.

P.T.O

Client Request

↓

DispatcherServlet

↓

[HandleMapping] → Finds @Controller + @RequestMapping

↓

Invokes controller method

↓

Controller returns view name + data in Model

↓

[ViewResolver] → Maps view name to JSP or HTML

↓

Renders View with Model data

↓

Sends response to browser.

Fig: Workflow

Lab-6:

22) How does Prepared Statement improve performance and security over statement in JDBC? Write a short example to insert record into a MySQL table using Prepared Statement.

Ans:

Prepared Statement improve performance and security over statement in JDBC because:

Feature	Statement	PreparedStatement
① Security	Phone to SQL Injection	Prevent SQL injection via parameter binding.
② Performance.	Compiles SQL each time.	SQL is precompiled and reused (faster in loops).
③ Readability	Concatenates strings manually	Uses ? placeholders for parameters.
④ Error-Prone	Manual escaping of quotes / specials	Handles all data types and escaping internally.

Syntax of Prepared Statement:

```
String sql = "Insert Into users (name, email) values (?, ?)";
PreparedStatement ps = conn.prepareStatement(sql);
ps.setString(1, "Naima");
ps.setString(2, "naima@gmail.com");
ps.executeUpdate();
```

Example: Insert using PreparedStatement in JDBC

```
import java.sql.*;
public class InsertUsingPreparedStatement {
    public static void main (String []args) {
        String url = "jdbc:mysql://localhost:3306/abdb";
        String user = "root"
        String password = "1234";
        try {
            Class.forName
```

Lab-27

23|| In JDBC a resultset is an object that stores the result of SQL select Query executed using a statemen or prepared Statement. It act like a table in memory, where each row can be accessed one by one.

The next() moves the cursse to the next row in the result set and return true if another row exists.

The getString ("ColumnName") method retondere Is a column's value as a string.

The getInt ("Column Name") methot reteieves an integer value from the specified colum

24|| Ans: How JPA manages mapping between Java Objects and Relational table:

JPA is a specification that allows you to map Java classes to relational database tables, handling all CRUD operations automatically.

@Entity → Declares that the class is a table in the Database.

@ID → Marks the primary key field.

@Generate value → Auto generate primary key values.

Feature	JPA	JDBC
Abstraction	High level, Object Oriented	Low Level, Manual SQL need.
Caching	Supported	Not built-in
Query Support	JPQL	Raw SQL Only
Code size	Less boilerplate	Require verbose code

25|| Ans:

	persist()	merge()	remove()
(i) Action	Insert	Update / attach	Delete
(ii) Entity must be	New	Detached	Managed
(iii) Use case	Save a new object	Update or attach a detached object	Delete an existing object

Lab-8:

27|| Ans:

How spring boot simplifies Restful Services:

- Auto configure everything
- Embedded servers (Tomcat)
- Easy annotation for REST API's
- Built in Json handling via Jackson.

Example: REST Controller:

```
@RestController
@RequestMapping("/api/users")
public class UserController {
    @GetMapping
    public List<User> getUsers() {
    return List.of(new User(1, "Naima"),
        new(2, "Saman"));
    }
@PostMapping
public User createUser(@RequestBody User user) {
    return user;
}
}
```

JSON example:

Request (Post):

```
{
  "id": 1,
  "name": "Naima"
}
```

Response (Get / Post):

```
[
  { "id": 1, "name": "Naveen" }
]
```

28|| Ans:

Aspect	@RestController	@Controller
(i) Use case	Rest API's	Web pages
(ii) Returns	Direct JSON	View page (HTML)
(iii) Example return	return List<Book> → JSON	return "home" → Loads home.html.

p.T.O

02. RESTful endpoints for Books in a Library System

Operation	HTTP method	Endpoint	Description
Create Book	Post	/books	Add a new book
Read all books	Get	/books	Get list of books
Read one book	Get	/books/{id}	Get book by ID
Update book	Put	/books/{id}	Update book info by ID
Delete Book	Delete	/book/{id}	Delete book by id.

29|| Ans:

How MAVEN manages Dependencies and build lifecycle:

Dependencies:

- Maven uses the pom.xml file to declare libraries
- It auto downloads and manages versions from Maven central
- Ensures all libraries are compitable and avoid "JAR hell".

Build lifecycle:

Standard phase line:

compile → Compiles Java code
test → Runs tests
Package → Builds .jar or .war
install → installs to local repo.
deploy → Deploys to remt repo

How Spring boot starters Dependencies help :

- spring-boot-starter-web → brings Tomcat + Jason + REST + MVC
- spring-boot-starter-data-jpa → brings Hibernate + JPA
- spring-boot-starter-test → brings JUnit + Mockito

These are no need to add manually

Lab-9:

32|| Demonstrate the project you developed with the important codes and Graphical Users interface.

Ans: My SpringBoot project : medi_doctor

A REST API for Patient Disease & Medicine Suggestion System.

To design and implement a RESTful API using Java SpringBoot and MySQL that:

- Accepts patient details
- Accepts disease Input.
- Suggests appropriate medicine based on the disease.
- Supports full CRUR operations on patients and diseases.

P.T.O

System Modules:

(1) Patient Module:

Fields: id, name, gender, age, mobileNumber.

(2) Endpoints:

- Post/patients → Add a new patient
- Get/patients → View all patients
- Get/patients/{id} → View single patient.
- Put/patients/{id} → Update patient
- Delete/patients/{id} → Delete patient.

(3) Disease Module:

Fields: id, diseaseName, medicineSuggestion.

Endpoints:

- POST/diseases/{patientId} → Submit disease and auto-suggest medicine.
- GET/diseases → View all diseases.

Database Design (MySQL):

- Patients table:
  - id (PK), name, gender, age, mobile_number.
- diseases table:
  - id (PK), disease_name, medicine_suggetion, patient_id (FK)

- Relation:
  - One patient → Many diseases

Medicine Suggestion Logic:

```
String medicine = switch(disease.getDiseas
    Name().toLowerCase()) {
case "feren" -> "Paracetamol";
case "cough" -> "Benadryl";
case "headache" -> "Aspirin";
default -> "Consult Doctor";
};
```

Simple Test Case via Postman:

⇒ Add Patient (POST /patients)

```
{
"name" : "Naima",
"gender" : "Female",
"age" : 23,
"mobileNumber" : "017!!!!!!!"
}
```

⇒ Submit Disease (POST/diseases/1)

```
{
  "diseaseName" : "fevere"
}
```

⇒ Response:

```
{
  "diseaseName" : "fever",
  "medicine Suggestion : "Paracetamal";
  "patient" : {
    "id" : 1,
   "name" :"Naima",
    ....
  }
}
```