ENDTERM PAPER SOLUTION 2024 - ADV JAVA EXPANDED

Q1. All Question are compulsory

(a) Differentiate between core java and advanced java. (5)

Feature	Core Java (J2SE - Java Platform, Standard Edition)	Advanced Java (JEE - Java Enterprise Edition)
Definition & Purpose	The foundation of the Java platform. Provides basic language features, fundamental data types, OOP concepts, basic I/O, collections. Used for standard desktop and command-line applications.	A set of specifications and APIs built *on top of* Core Java. Designed for building large-scale, distributed, multi-tiered enterprise applications.
Scope	Single-tier, desktop-focused applications.	Multi-tier, web/network-focused applications, server-side programming.
Technologies Included	Basic language syntax, `java.lang`, `java.io`, `java.util`, `java.net` (basic networking), Swing, AWT, JavaBeans, basic JDBC.	Includes Core Java plus technologies like Servlets, JSP, EJB (historically), RMI, JPA (Hibernate as implementation), Web Services, JMS, etc.
Application Type	Standalone applications, applets (historically).	Web applications, enterprise applications, distributed systems, web services.
Complexity Management	Deals with fundamental programming constructs.	Simplifies complex tasks like managing thousands of users, data security across networks, integrating different software systems, n-tier applications.
Key Idea	Provides the language basics.	Provides tools and frameworks to apply basics to complex, real-world scenarios like websites and online services.

Core Java is the essential programming language and a set of fundamental libraries, while Advanced Java (specifically JEE) extends Core Java with APIs and specifications tailored for developing robust, scalable, and secure enterprise-level applications, often involving web interfaces, distributed components, and database interactions.

(b) State and explain the types of cookies in servlets. (5)

Types of Cookies based on Persistence:

1. **Non-Persistent Cookie (Session Cookie):**
    - **State:** Default behavior if `setMaxAge()` is not called or set to `-1`.
    - **Explanation:** Stored in browser memory, valid only for the current browser session. Discarded when the browser closes.
    - **Usage:** Temporary session information.
2. **Persistent Cookie:**
    - **State:** Created when `setMaxAge()` is called with a positive integer (seconds).
    - **Explanation:** Stored on the user's hard drive, valid until its expiration time, even if the browser is closed and reopened.
    - **Usage:** Remembering preferences, login information, tracking across sessions.
3. **Deleting a Cookie:**
    - **State:** Achieved by creating a cookie with the same name and setting `setMaxAge(0)`.
    - **Explanation:** Browser immediately deletes the cookie.
    - **Usage:** Explicitly removing a cookie (e.g., on logout).

The primary distinction discussed is based on their lifespan, controlled by the `maxAge` attribute.

(c) List out and explain the features of JSP. (5)

1. **Extension/Wrapper over Servlet Technology:** JSPs are compiled into Servlets, inheriting their power (platform independence, robustness, scalability, access to Java APIs).
2. **Simplified Dynamic Web Page Creation:** HTML-like structure with embedded Java code using special tags, easier for page layout than pure Servlets.
3. **Separation of Presentation and Logic (Improved):** Encourages separating HTML/XML (presentation) from Java code (logic), especially with EL and JSTL.
4. **Use of Reusable Components:** Integrates with JavaBeans and Custom Tag Libraries (like JSTL) for reusable logic and presentation components.
5. **Platform Independence:** Inherited from Java, deployable on any server with a compatible JSP/Servlet container.
6. **Access to Java APIs:** Full access to Java APIs (JDBC, JNDI, RMI, etc.) for server-side processing.
7. **Implicit Objects:** Provides predefined objects (`request`, `response`, `session`, `application`, `out`, etc.) for easy access to web environment details.
8. **Extensibility through Custom Tags:** Allows developers to create custom tags to encapsulate complex logic, making JSPs cleaner.

(d) How JSP is more advantageous than Servlet. (5)

1. **Easier Page Layout and Design:** JSP's HTML-like structure is more intuitive for designing web page layouts compared to generating HTML programmatically in Servlets using `out.println()`.

2. **Better Separation of Presentation and Logic:** JSP (especially with EL & JSTL) facilitates a clearer distinction between the visual presentation (HTML) and the dynamic data generation logic (Java), leading to cleaner and more maintainable code. Servlets tend to mix these heavily.
3. **Reduced Java Code in View Layer:** EL and JSTL in JSP allow common tasks (displaying data, iteration, conditionals) to be done with declarative tags, reducing the amount of Java scriptlet code.
4. **Faster Development for Presentation Tasks:** Modifying the visual aspects of a page is often quicker in JSP (editing HTML-like structures) than in Servlets (modifying Java code and recompiling).
5. **Reusability through Standard Actions and Custom Tags:** JSP provides standard actions and supports custom tags that encapsulate functionality into reusable components directly usable in the page structure.

JSPs don't replace Servlets but offer a more convenient way for the presentation layer, often used with Servlets (as controllers) in an MVC pattern.

(e) Explain Hibernate framework and how it is related to ORM tool. (5)

Hibernate Framework:
Hibernate is an open-source Java framework that simplifies database interaction by providing an **Object-Relational Mapping (ORM)** implementation. It automates the mapping between Java objects (Entities) and relational database tables, reducing manual JDBC code. Hibernate implements the JPA (Java Persistence API) standard, offers database independence through HQL (Hibernate Query Language), and includes features like caching and automatic table creation.

How Hibernate is related to ORM tool:
Hibernate **is an ORM tool**.

- **Object-Relational Mapping (ORM):** A programming technique that maps object-oriented language constructs (like Java objects, inheritance, relationships) to relational database structures (tables, rows, foreign keys). It addresses the "Object-Relational Impedance Mismatch."
- **ORM Tool:** A framework (like Hibernate) that implements ORM. It handles defining mappings, generating SQL for CRUD operations, managing transactions, and abstracting low-level database access.

Hibernate provides a concrete implementation of ORM principles, allowing developers to work with databases in a more object-oriented fashion.

UNIT-I

Q2. (a) Write a java program to demonstrate the concept of socket programming. (6.5)

```
// SimpleServer.java
import java.io.BufferedReader;
import java.io.IOException;
```

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

public class SimpleServer {
    private static final int PORT = 5000;

    public static void main(String[] args) {
        System.out.println("Server started. Listening on port " + PORT);
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("Waiting for a client connection...");
            Socket clientSocket = serverSocket.accept();
            System.out.println("Client connected: " +
clientSocket.getInetAddress().getHostAddress());

            try (PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true);
                 BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()))) {
                String clientMessage = in.readLine();
                System.out.println("Received from client: " +
clientMessage);
                String serverResponse = "Hello back, Client! You said: " +
clientMessage;
                out.println(serverResponse);
                System.out.println("Sent to client: " + serverResponse);
                System.out.println("Client connection closed.");
            } catch (IOException e) {
                System.err.println("Error during client communication: " +
e.getMessage());
            }
        } catch (IOException e) {
            System.err.println("Error in server operation: " +
e.getMessage());
        }
        System.out.println("Server exiting.");
    }
}

// SimpleClient.java
import java.io.BufferedReader;
import java.io.IOException;
```

```
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

public class SimpleClient {
    private static final String SERVER_ADDRESS = "localhost";
    private static final int PORT = 5000;

    public static void main(String[] args) {
        try (Socket socket = new Socket(SERVER_ADDRESS, PORT);
             PrintWriter out = new PrintWriter(socket.getOutputStream(),
true);
             BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()))) {
            System.out.println("Connected to server at " + SERVER_ADDRESS +
":" + PORT);
            String messageToSend = "Hello Server from Client!";
            out.println(messageToSend);
            System.out.println("Sent to server: " + messageToSend);
            String serverResponse = in.readLine();
            System.out.println("Received from server: " + serverResponse);
        } catch (UnknownHostException e) {
            System.err.println("Server not found: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("Error in client operation: " +
e.getMessage());
        }
        System.out.println("Client exiting.");
    }
}
```

(b) Discuss the advantages, disadvantages and hierarchy of applets. (6)

Applets: Small Java programs embedded in HTML, run in browsers (with plugin).

Advantages:

1. Dynamic/Interactive web content (historically).
2. Cross-Platform (with plugin).
3. Rich GUI (AWT/Swing).
4. Client-Side Processing.
5. Access to Java's power.

Disadvantages:

1. Requires Java Plugin.
2. Security Concerns & Sandbox Restrictions.
3. Slow Startup Time.
4. Deprecated, no modern browser support.
5. Complex for simple tasks (vs. JavaScript).
6. Integration issues with HTML/JS.

Hierarchy (Class Hierarchy):

1. `java.lang.Object`
2. `java.awt.Component`
3. `java.awt.Container`
4. `java.awt.Panel`
5. `java.applet.Applet` (Base class for applets, providing lifecycle methods: `init()`, `start()`, `paint()`, `stop()`, `destroy()`).

Q3. (a) Explain the basic steps of implementing a server with basic methods used in each step. (6.5)

1. **Create a ServerSocket Object:**
   - Purpose: Establish a listening point for client requests.
   - Method: `ServerSocket serverSocket = new ServerSocket(int port);`
2. **Wait for a Client Connection (Accepting):**
   - Purpose: Pause until a client connects.
   - Method: `Socket clientSocket = serverSocket.accept();` (blocking call, returns a `Socket` for the client).
3. **Get Input and Output Streams:**
   - Purpose: Obtain streams for communication with the connected client.
   - Methods: `clientSocket.getInputStream();`, `clientSocket.getOutputStream();` (Often wrapped in `BufferedReader`/`PrintWriter`).
4. **Communicate with the Client:**
   - Purpose: Exchange data.
   - Methods: `in.readLine();` (to read), `out.println(String);` (to write).
5. **Close the Connection:**
   - Purpose: Release resources for the specific client.

- Methods: `in.close(); out.close(); clientSocket.close();`

6. **Close the ServerSocket (On server shutdown):**
   - Purpose: Stop listening for new connections.
   - Method: `serverSocket.close();`

(b) Write a program in java to demonstrate the concept of applets. (6)

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;

/*
<applet code="SimpleDrawingApplet.class" width="300" height="150">
</applet>
*/
public class SimpleDrawingApplet extends Applet {
    @Override
    public void init() {
        setBackground(Color.lightGray);
        System.out.println("Applet Initialized");
    }

    @Override
    public void start() {
        System.out.println("Applet Started");
    }

    @Override
    public void paint(Graphics g) {
        g.setColor(Color.blue);
        g.fillRect(20, 20, 100, 50); // Blue rectangle
        g.setColor(Color.red);
        g.drawString("Hello from Applet!", 140, 50); // Red text
        g.setColor(Color.green);
        g.drawOval(50, 80, 60, 40); // Green oval outline
    }

    @Override
    public void stop() {
        System.out.println("Applet Stopped");
    }

    @Override
```

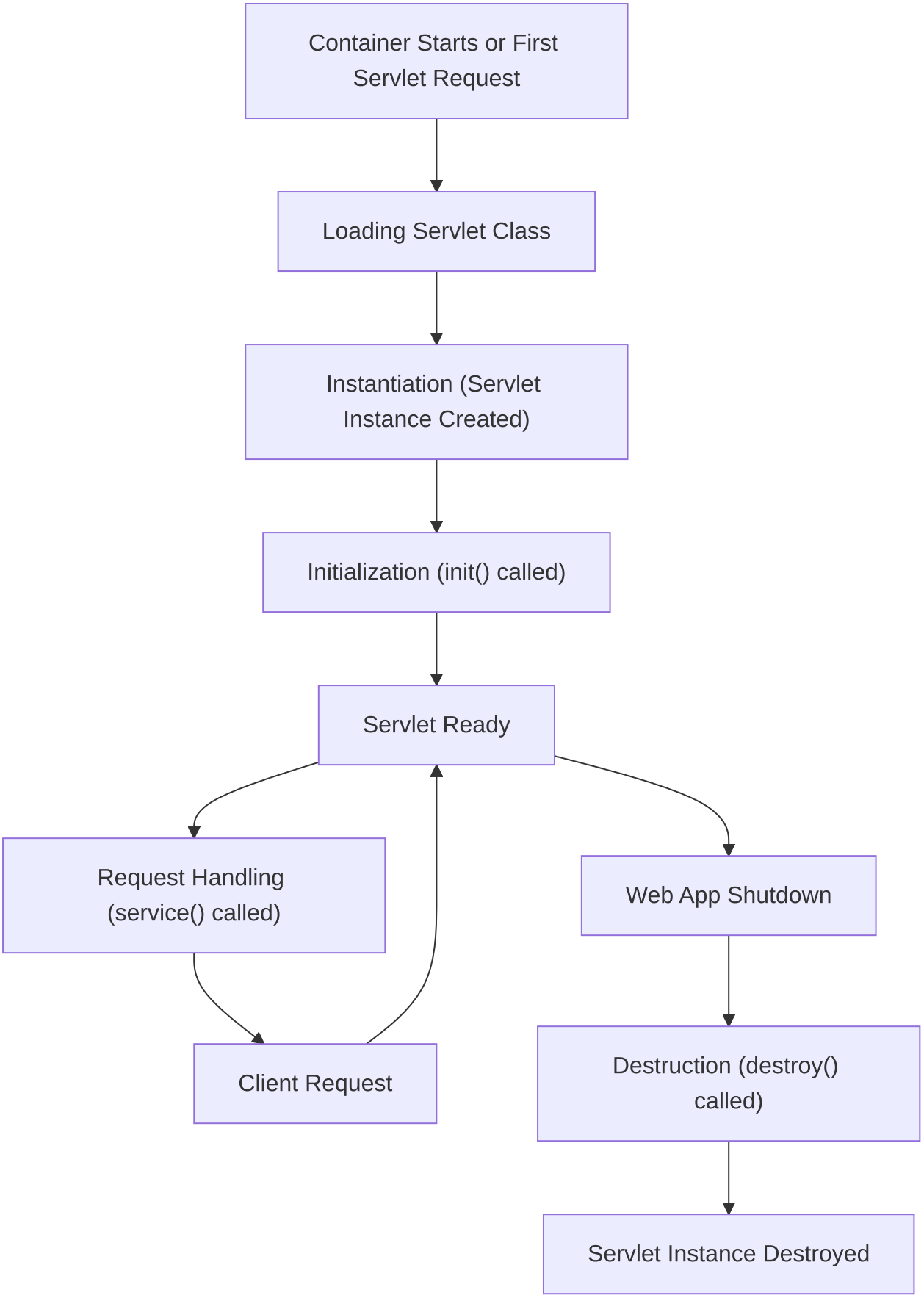
```
    public void destroy() {
        System.out.println("Applet Destroyed");
    }
}
```

UNIT-II

Q4. (a) Explain the lifecycle of a servlet with an example. (6.5)

The servlet lifecycle is managed by the web container:

1. **Loading:** Container loads the servlet class (first request or startup).
2. **Instantiation:** Container creates one instance of the servlet (using no-arg constructor).
3. **Initialization (`init(ServletConfig config)`):** Called once after instantiation for one-time setup (e.g., read init params, DB connections).
4. **Request Handling (`service(ServletRequest req, ServletResponse res)`):** Called for each client request. In `HttpServlet`, this dispatches to `doGet()`, `doPost()`, etc. This is where request processing logic resides.
5. **Destruction (`destroy()`):** Called once when the servlet is taken out of service (app shutdown) for cleanup (e.g., close DB connections).



Example:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;
import java.io.PrintWriter;
```

```
import java.util.Date;

public class LifecycleServlet extends HttpServlet {
    private int hitCount;
    @Override
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        hitCount = 0;
        log("LifecycleServlet Initialized at " + new Date());
    }
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        hitCount++;
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body><h1>Servlet Lifecycle</h1>" +
                    "<p>Accessed " + hitCount + " times.</p>" +
                    "<p>Instance: " + this.hashCode() + "</p>" +
                    "</body></html>");
        log("doGet called. Hit count: " + hitCount);
    }
    @Override
    public void destroy() {
        log("LifecycleServlet Destroyed at " + new Date() + ". Final hits: "
+ hitCount);
        super.destroy();
    }
}
```

(b) Write a java program to demonstrate the use of Java Beans. (6)

JavaBean (`Employee.java`):

```
package mypack;
import java.io.Serializable;
public class Employee implements Serializable {
    private int id;
    private String name;
    public Employee() { } // No-arg constructor
    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 void displayInfo() {
        System.out.println("ID: " + id + ", Name: " + name);
    }
}
```

Using Class (`TestBean.java`):

```
package mypack;
public class TestBean {
    public static void main(String[] args) {
        Employee emp1 = new Employee();
        emp1.setId(101);
        emp1.setName("Arjun Sharma");
        System.out.println("Employee Name: " + emp1.getName());
        emp1.displayInfo();
    }
}
```

Okay, let's refocus the answer for Q5 specifically on the distinction between standard JavaBeans (as per Unit 2, Section A of your notes) and Enterprise JavaBeans (EJBs) (as per Unit 2, Section B), detailing the types of EJBs mentioned.

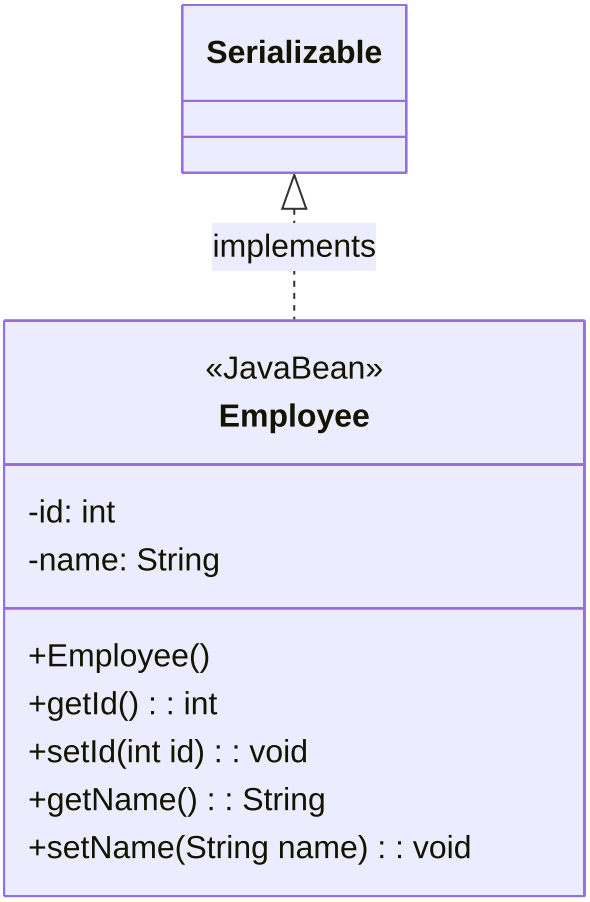
Q5. Discuss the types of Java Beans with a diagram of each type. (12.5)

The term "Java Beans" can refer to two related but distinct concepts: standard **JavaBeans** used as reusable software components (often for data encapsulation), and **Enterprise JavaBeans (EJBs)** which are server-side components for building robust, scalable enterprise applications. Your notes cover both.

I. Standard JavaBeans (Unit 2 Notes, Section A)

These are simple Java classes following specific design conventions to make them reusable. They are not typically categorized into formal "types" but rather serve various roles.

- **Definition:** A JavaBean is a Java class that is:
  1. `Serializable` (implements `java.io.Serializable`).
  2. Has a public no-argument constructor.
  3. Has private properties exposed through public getter and setter methods following naming conventions (e.g., `getPropertyName()`, `setPropertyName()`).
- **Primary Use:** Often used as data carriers (Data Transfer Objects - DTOs or Value Objects - VOs), encapsulating data and logic.
- **Diagram (Generic Data Bean - e.g., `Employee` from notes):**

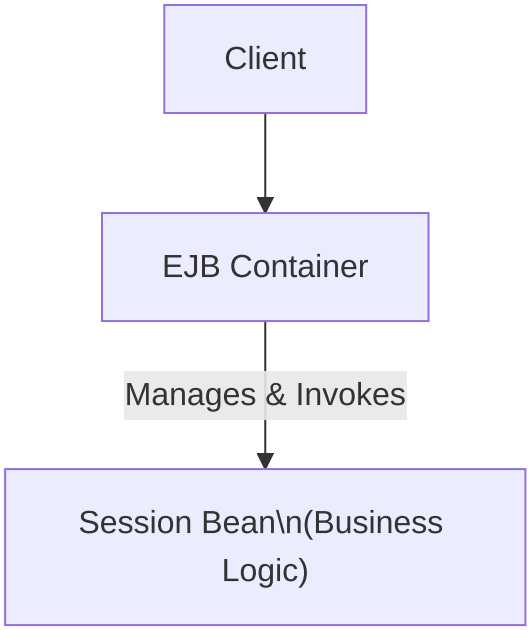


- **Description:** Represents a simple data-holding bean. It's serializable, has a no-arg constructor, private fields, and public getters/setters.

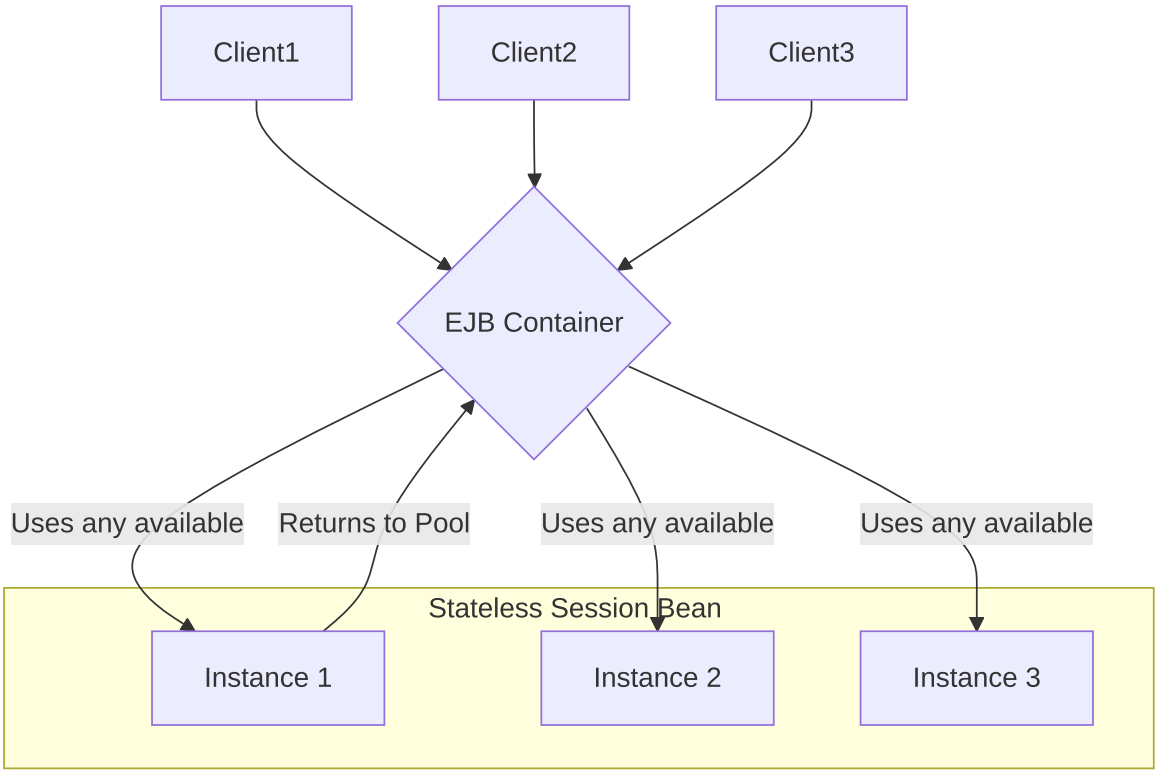
II. Enterprise JavaBeans (EJBs) (Unit 2 Notes, Section B)

EJBs are server-side components that run in an EJB container within a Java EE application server. They handle complex aspects like transactions, security, and concurrency automatically. Your notes detail three main types (Image 13):

1. **Session Beans:**
   - **Purpose:** Represent business logic or tasks performed for a client. They encapsulate business processes and are typically not persistent themselves (their state usually doesn't survive server restarts).
   - **Diagram (Conceptual Interaction):**



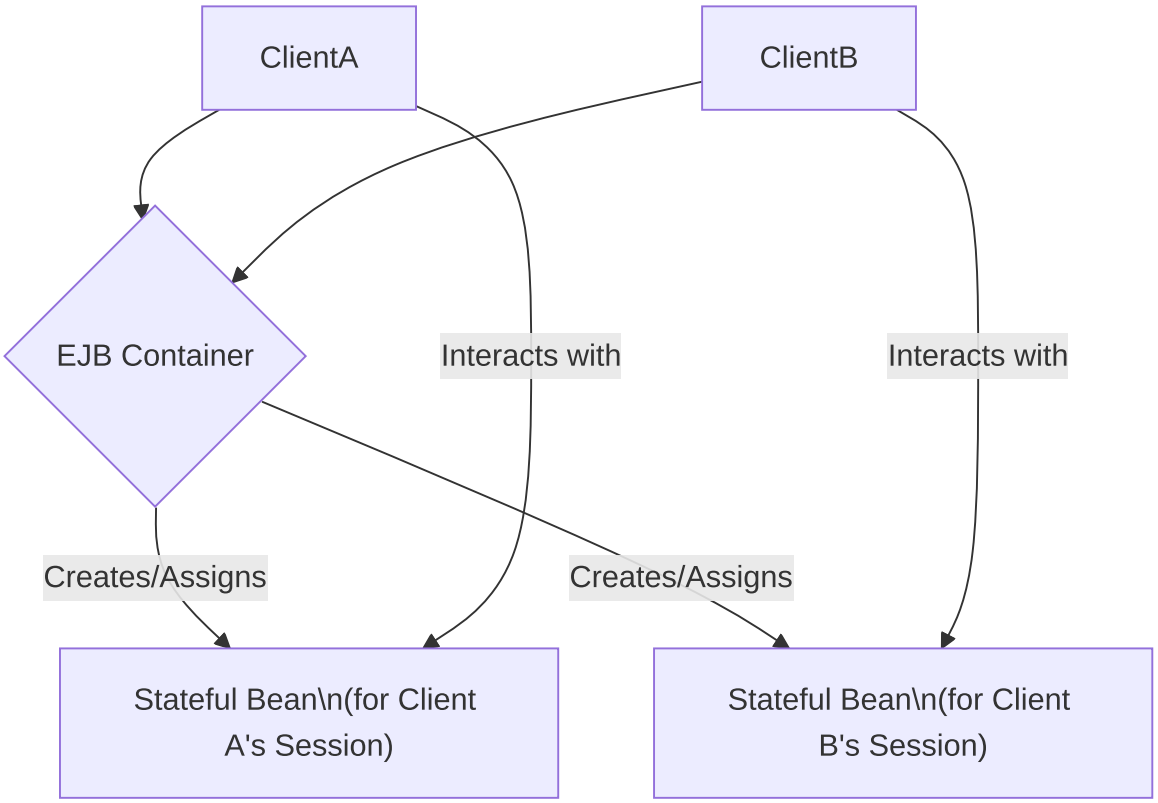
- **Types of Session Beans (Images 14-18):**
  - **a. Stateless Session Bean:**
    - **Description:** Does *not* maintain client-specific conversational state between method calls. Any instance from a pool can serve any client request. Efficient due to pooling. Often used for general business logic or web services.
    - **Diagram (Conceptual Pooling):**



    - **Key:** No conversational state, pooled instances.
  - **b. Stateful Session Bean:**
    - **Description:** *Maintains* client-specific conversational state across multiple method calls within a specific client-bean session. There's a one-to-one relationship between a client

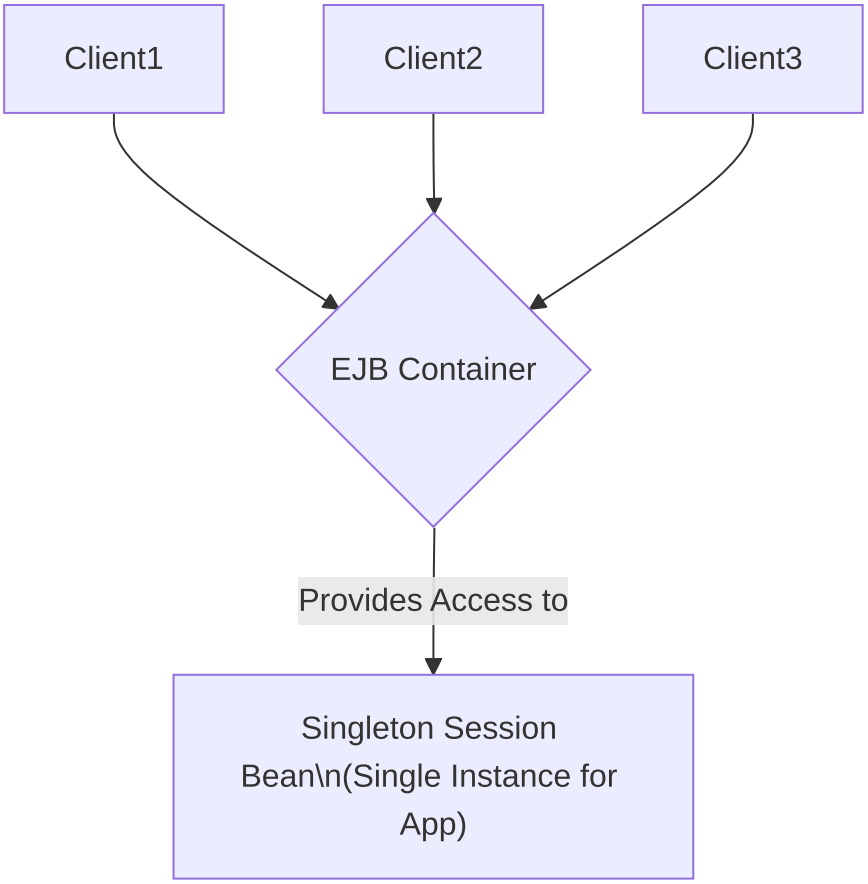
and a bean instance during the session. Suitable for multi-step tasks (e.g., shopping cart). Less scalable than stateless.

- **Diagram (Conceptual Client-Bean Affinity):**



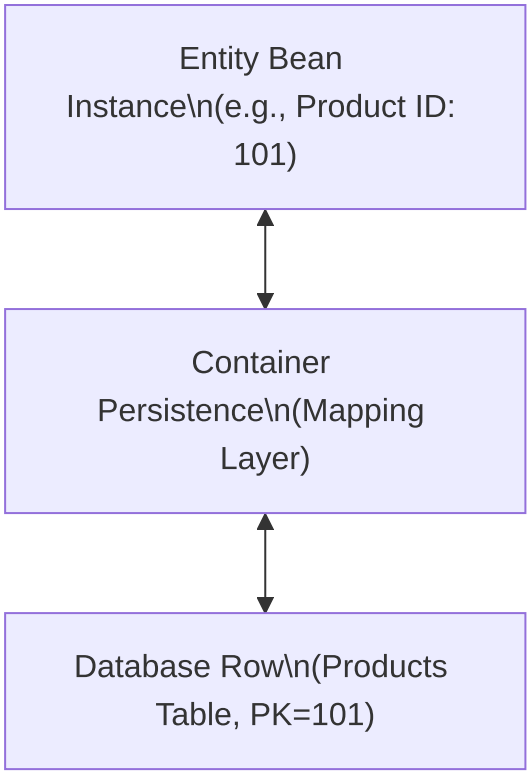
- **Key:** Conversational state per client, dedicated instance per session.

- **c. Singleton Session Bean:**
  - **Description:** A single instance of the bean exists for the entire application, shared by all clients. Suitable for shared resources, application-wide configuration, or tasks needing to run on startup/shutdown.
  - **Diagram (Conceptual Shared Instance):**



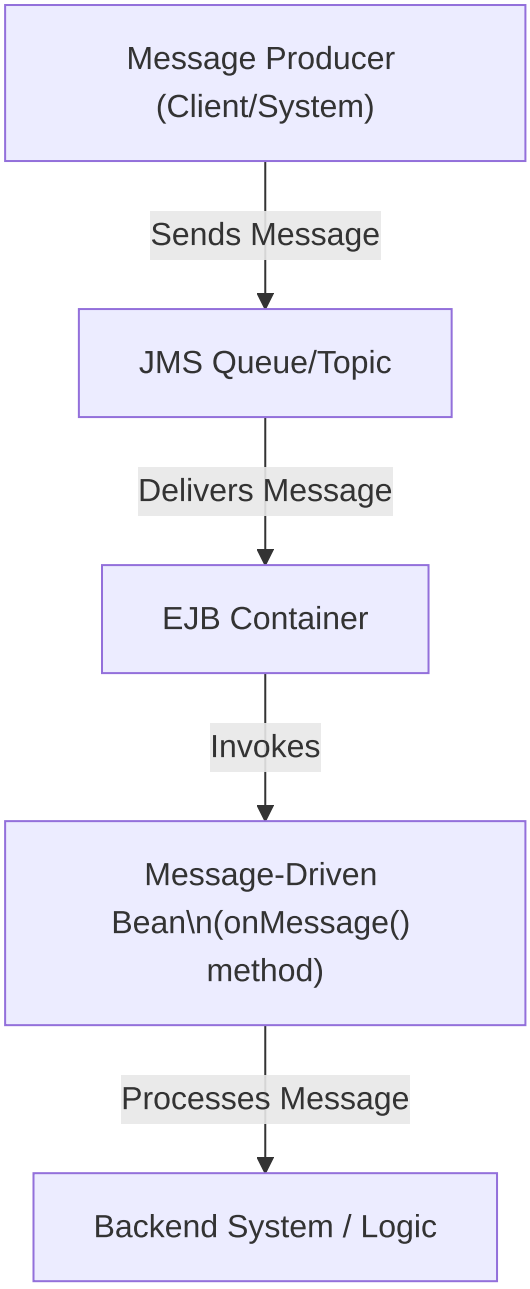
- **Key:** One instance for the whole application.

2. **Entity Beans (Largely replaced by JPA - Unit 2 Notes, Image 19, 22):**
    - **Purpose (Historically):** Represented persistent data stored in a database. Each Entity Bean instance typically corresponded to a row in a database table. They had a primary key.
    - **Persistence Management (Image 21):**
        - **Bean-Managed Persistence (BMP):** Bean code contained explicit database access calls (JDBC).
        - **Container-Managed Persistence (CMP):** EJB container automatically handled data loading/saving based on mappings.
    - **Diagram (Conceptual Mapping):**



- **Note:** Modern Java EE uses JPA with POJOs (@Entity annotated classes) for persistence, often implemented by frameworks like Hibernate, rather than traditional EJB Entity Beans.

3. **Message-Driven Beans (MDBs) (Unit 2 Notes, Images 23, 25, 26):**
   - **Purpose:** Act as asynchronous message consumers (listeners). They listen for messages arriving on a Java Messaging Service (JMS) queue or topic.
   - **Operation:** When a message arrives, the EJB container invokes the MDB's `onMessage(Message msg)` method to process it.
   - **Characteristics:** Asynchronous, decoupled from message senders.
   - **Diagram (Conceptual Message Flow):**



Summary of Differences:

- **Standard JavaBeans** are general-purpose, often client-side or simple data objects, defined by conventions.
- **Enterprise JavaBeans (EJBs)** are server-side components managed by an EJB container, designed for specific roles in enterprise applications (business logic, persistence (historically), asynchronous messaging) with container-provided services.

UNIT-III

Q6. Explain the lifecycle of a JSP page with a diagram. (12.5)

The lifecycle of a JavaServer Page (JSP) is a sequence of events that occurs from the time a JSP page is created until it is destroyed. Because JSP pages are ultimately converted into Java Servlets, their lifecycle is an extension of the Servlet lifecycle, incorporating additional steps for translation and compilation. The web container (e.g., Tomcat, Jetty) manages this entire lifecycle.

JSP Lifecycle Phases in Detail:

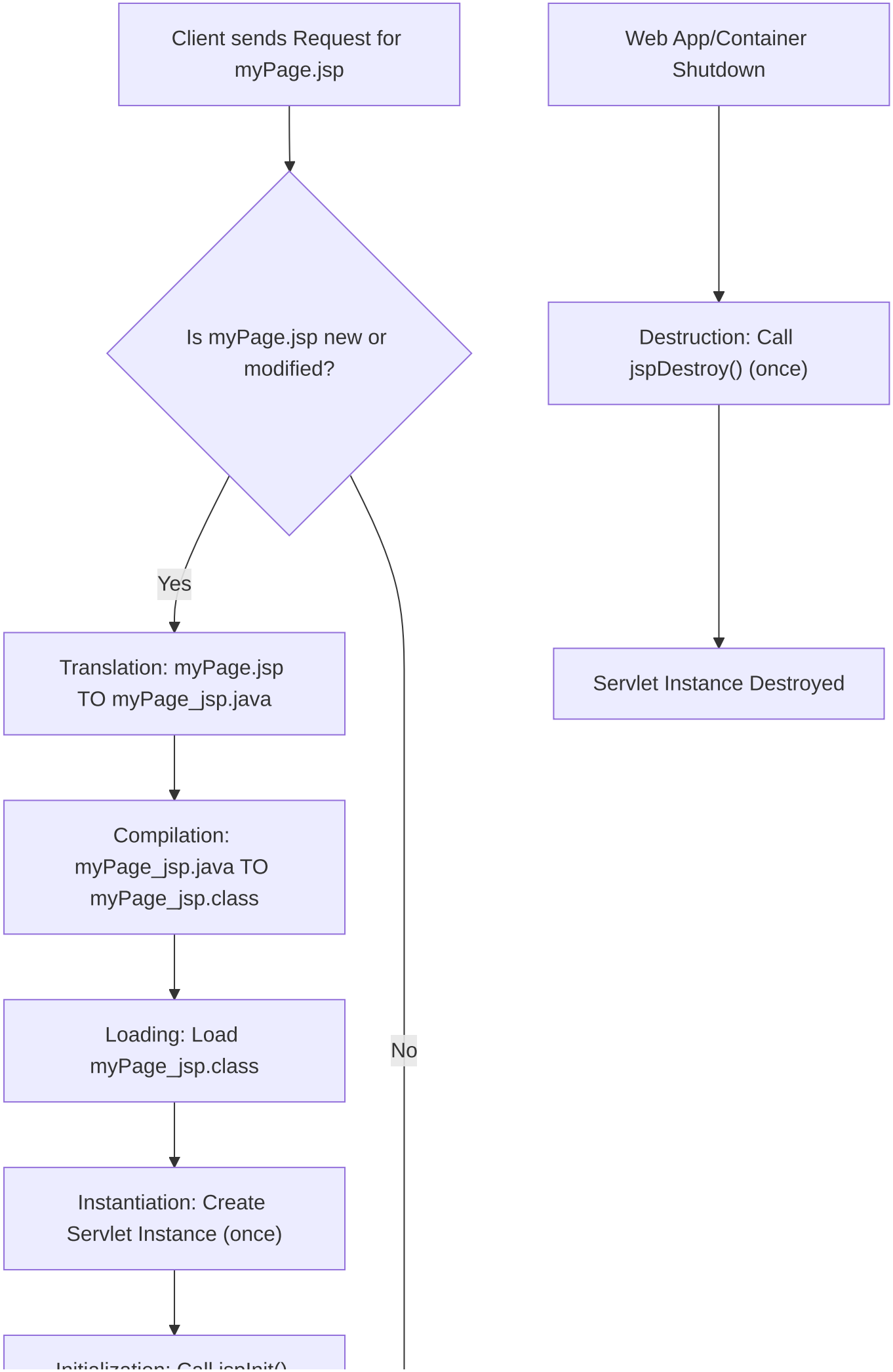
1. **Translation Phase (JSP to Java Servlet Source Code):**
    - **Trigger:** This phase is initiated under two conditions:
        1. When the `.jsp` file is requested by a client for the very first time.
        2. When the `.jsp` file has been modified since its last translation and a new request comes in (the container typically checks the timestamp of the JSP file).
    - **Process:** The JSP engine (a specialized component within the web container) parses the `.jsp` file.
        - **Static Content:** Standard HTML, XML, or other text content in the JSP file is converted into Java code that will write this content directly to the response stream (e.g., using `out.println("<html>");`).
        - **JSP Directives (`<%@ ... %>`):** These (like `page`, `include`, `taglib`) are processed first. They provide instructions to the translator on how to generate the servlet. For example, `@page import` adds Java import statements to the generated servlet. The `@include` directive causes the content of another file to be textually inserted at this stage.
        - **JSP Scripting Elements:**
            - **Declarations (`<%! ... %>`):** Java code within declaration tags is placed outside the main service method (e.g., `_jspService()`) of the generated servlet, becoming member variables or methods of the servlet class.
            - **Scriptlets (`<% ... %>`):** Java code within scriptlet tags is inserted directly into the `_jspService()` method of the generated servlet.
            - **Expressions (`<%= ... %>`):** Java expressions are evaluated, converted to strings, and written to the response stream, typically via `out.print(...)` within the `_jspService()` method.
        - **JSP Standard Actions (`<jsp:... />`):** These XML-like tags are converted into Java code that performs specific actions at request time (e.g., `<jsp:useBean>` might involve code to instantiate or locate a JavaBean).
    - **Output:** A Java Servlet source code file (e.g., `myPage_jsp.java`).
    - **Notes Reference:** JAVA UNIT 3 NOTES, Image 4, 10, 15, 25; Part 3, Step 1.
2. **Compilation Phase (Java Servlet Source to Bytecode):**
    - **Trigger:** Occurs immediately after the translation phase if a new `.java` file was generated.
    - **Process:** The web container invokes the Java compiler (javac) to compile the generated servlet source file (`myPage_jsp.java`) into a Java servlet class file (`myPage_jsp.class`), which contains platform-independent bytecode.
    - **Error Handling:** If there are any syntax errors in the Java code (either directly written in scriptlets/declarations or generated due to incorrect JSP syntax), compilation will fail, and an error page is typically sent to the client.

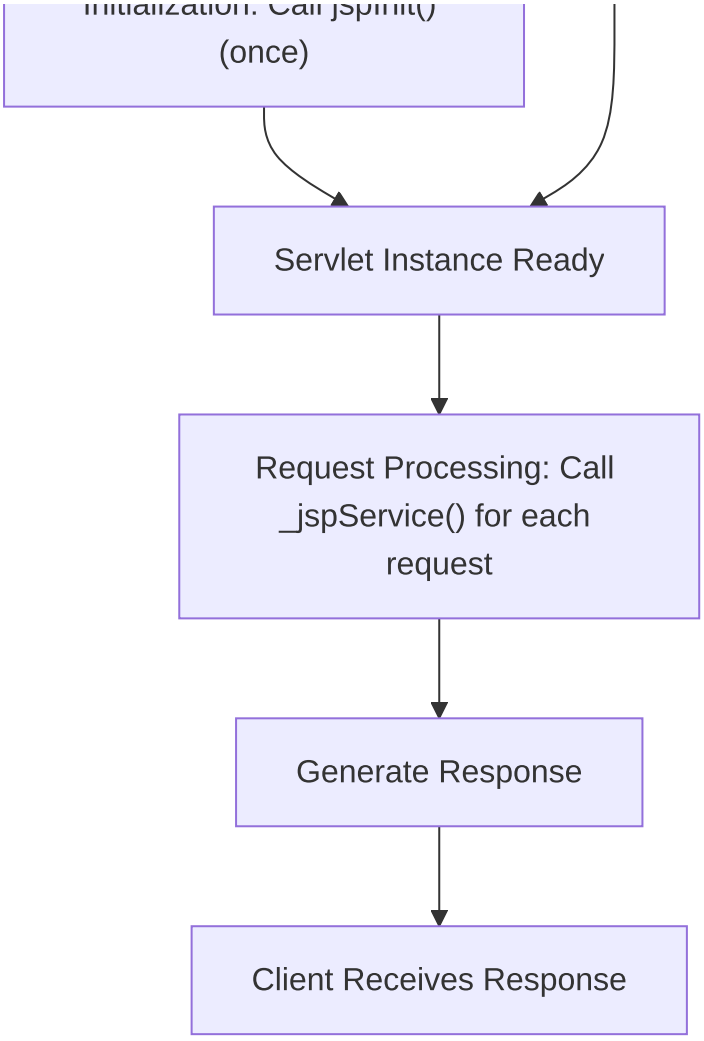
- **Output:** A compiled servlet `.class` file.
- **Notes Reference:** JAVA UNIT 3 NOTES, Image 4, 10; Part 3, Step 2.

3. **Loading Phase (Loading the Servlet Class):**
   - **Trigger:** After successful compilation of the servlet class file.
   - **Process:** The web container's class loader loads the `myPage_jsp.class` file into the Java Virtual Machine (JVM).
   - **Notes Reference:** JAVA UNIT 3 NOTES, Part 3, Step 3.
4. **Instantiation Phase (Creating the Servlet Instance):**
   - **Trigger:** After the servlet class is loaded.
   - **Process:** The web container creates an instance (object) of the compiled servlet class. Typically, only **one instance** of the servlet is created per JSP page definition in the application (unless specific, now deprecated, configurations like `SingleThreadModel` were used). This instantiation happens only once during the lifecycle of that JSP's servlet. The container uses the servlet's public no-argument constructor.
   - **Notes Reference:** JAVA UNIT 3 NOTES, Image 4; Part 3, Step 4 (refers to Unit 2 Servlet lifecycle).
5. **Initialization Phase (Calling `jspInit()`):**
   - **Trigger:** Called by the container **once** immediately after the servlet instance is created and before any requests are handled by it.
   - **Method Signature (defined by developer in JSP):** `public void jspInit() { /* initialization code */ }` (This method must be placed within JSP declaration tags: `<%! ... %>`).
   - **Process:** This method provides a hook for the developer to perform one-time setup tasks specific to the JSP. Examples include:
     - Loading database drivers.
     - Establishing database connections (though modern applications often use connection pools managed outside the JSP/servlet).
     - Initializing expensive resources or loading configuration settings.
     - Reading servlet initialization parameters (though less common directly in JSP `jspInit` than in servlet `init`).
   - The underlying generated servlet's `init(ServletConfig config)` method (inherited from the Servlet API) is called by the container, and this `init` method, in turn, calls the `jspInit()` method defined by the JSP author.
   - If `jspInit()` throws an exception, the JSP (servlet) will be marked as unavailable.
   - **Notes Reference:** JAVA UNIT 3 NOTES, Part 3, Step 5.
6. **Request Processing Phase (Calling `_jspService()`):**
   - **Trigger:** Called by the container for **every client request** that is mapped to this JSP page.

- **Method Signature (generated by container, not written by developer):** Typically `protected void _jspService(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException;`
- **Process:** This is the heart of the JSP's execution for each request. The container creates new `request` and `response` objects for each request and passes them to this method. The `_jspService()` method contains the Java code translated from the JSP page's body, including:
  - Code to write out static HTML portions.
  - Execution of Java code from scriptlets.
  - Evaluation of JSP expressions and writing their results to the output stream.
  - Execution of JSP standard actions and custom tag handlers.
- **Implicit Objects:** Within `_jspService()`, all JSP implicit objects (like `request`, `response`, `session`, `application`, `out`, `pageContext`, `config`, `page`, `exception`) are available for use.
- The container usually handles multiple concurrent requests by creating a new thread for each request to execute the `_jspService()` method on the single servlet instance. Therefore, code within scriptlets and declarations must be thread-safe if it accesses shared member variables.
- **Notes Reference:** JAVA UNIT 3 NOTES, Image 4, 25; Part 3, Step 6.

7. **Destruction Phase (Calling `jspDestroy()`):**
   - **Trigger:** Called by the container **once** just before the servlet instance is taken out of service and destroyed. This usually happens when:
     - The web application is being undeployed.
     - The web container (server) is being shut down.
   - **Method Signature (defined by developer in JSP):** `public void jspDestroy() { /* cleanup code */ }` (This method must be placed within JSP declaration tags: `<%! ... %>`).
   - **Process:** This method provides a hook for the developer to perform cleanup tasks and release any resources that were acquired during the `jspInit()` phase or during the servlet's lifetime. Examples include:
     - Closing database connections.
     - Releasing file handles or network sockets.
     - Saving any persistent state if necessary.
   - After `jspDestroy()` completes, the servlet instance is eligible for garbage collection by the JVM.
   - **Notes Reference:** JAVA UNIT 3 NOTES, Part 3, Step 7.


Client sends Request for myPage.jsp
Is myPage.jsp new or modified?
Yes
No
Translation: myPage.jsp TO myPage_jsp.java
Compilation: myPage_jsp.java TO myPage_jsp.class
Loading: Load myPage_jsp.class
Instantiation: Create Servlet Instance (once)
Initialization: Call jspInit()
Web App/Container Shutdown
Destruction: Call jspDestroy() (once)
Servlet Instance Destroyed



Q7. Illustrate about any five implicit objects of JSP with example. (12.5)

JSP implicit objects are pre-defined Java objects that the JSP container makes automatically available to developers within JSP pages (specifically in scriptlets and expressions) without needing explicit declaration. They provide convenient access to request, response, session, and application-level information and functionalities.

1. **`request` Object:**
   - **Java Type:** `javax.servlet.http.HttpServletRequest`
   - **Scope:** Request (available for the duration of a single client HTTP request).
   - **Purpose:** Represents the HTTP request sent by the client to the server. It provides methods to access request information such as:
     - Parameters sent with the request (e.g., form data, URL query strings).
     - HTTP headers (e.g., User-Agent, Accept-Language).
     - Cookies sent by the client.
     - The request method (GET, POST, etc.).
     - Client's IP address.
   - **Concise Example:**

```
<%-- Assuming URL: myPage.jsp?userName=Alice --%>
<%@ page contentType="text/html" %>
<html><body>
    <% String name = request.getParameter("userName"); %>
    User Name: <%= (name != null ? name : "Guest") %> <br/>
    Request Method: <%= request.getMethod() %> <br/>
    Your IP Address: <%= request.getRemoteAddr() %>
</body></html>
```

- This example retrieves a URL parameter "userName", gets the HTTP method, and displays the client's IP.

2. **`response` Object:**
   - **Java Type:** `javax.servlet.http.HttpServletResponse`
   - **Scope:** Request (used to construct the response for the current request).
   - **Purpose:** Represents the HTTP response that the server will send back to the client. It's used to:
     - Set the content type of the response (e.g., "text/html", "application/json").
     - Add cookies to the client's browser.
     - Redirect the client to a different URL.
     - Set HTTP status codes.
     - (Note: Direct output to the client is typically done via the `out` implicit object in JSP).
   - **Concise Example:**

```
<%@ page contentType="text/html" %>
<%
    response.setContentType("text/html;charset=UTF-8"); // Set content type and encoding
    Cookie visitCookie = new Cookie("lastVisit", new java.util.Date().toString().replace(" ","_"));
    response.addCookie(visitCookie); // Add a cookie to the response
    // response.sendRedirect("anotherPage.jsp"); // Uncomment to test redirection
%>
<html><body>
    Response content type set. Check your browser's cookies for 'lastVisit'.
</body></html>
```

   - This sets the response content type and adds a cookie. Redirection is commented out as it would prevent further output.

3. **`out` Object:**

- **Java Type:** `javax.servlet.jsp.JspWriter`
- **Scope:** Page (used to write content to the current page's output stream).
- **Purpose:** Provides methods to send character data to the client's browser. This is the primary object used to generate the HTML (or other textual) content of the JSP page. JSP expressions (`<%= ... %>`) automatically write their evaluated result to this `out` object.
- **Concise Example:**

```
<%@ page contentType="text/html" %>
<html><body>
    <%
        out.println("<h1>Welcome!</h1>"); // Using out object in a
scriptlet
        String dynamicMessage = "This is a dynamic message.";
    %>
    <p><%= dynamicMessage %></p> <%-- Using a JSP expression, which
uses 'out' implicitly --%>
</body></html>
```

  - Demonstrates writing HTML using `out.println()` from a scriptlet and an expression.

4. **`session` Object:**
   - **Java Type:** `javax.servlet.http.HttpSession`
   - **Scope:** Session (available for the duration of a user's browsing session, across multiple requests from the same client).
   - **Purpose:** Used to store and retrieve user-specific information that needs to persist across multiple page requests from the same client. This enables features like user login tracking, shopping carts, or remembering user preferences during a visit. A session is typically identified by a session ID (often managed via a cookie). Available by default, unless disabled by `<%@ page session="false" %>`.
   - **Concise Example:**

```
<%@ page contentType="text/html" session="true" %>
<html><body>
    <%
        String username = (String)
session.getAttribute("loggedInUser");
        if (request.getParameter("loginName") != null) {
            username = request.getParameter("loginName");
            session.setAttribute("loggedInUser", username); // Store in
session
        }
    %>
    Welcome, <%= (username != null ? username : "Guest") %>! <br/>
```

```
    Session ID: <%= session.getId() %>
    <%-- To test: myPage.jsp?loginName=Bob --%>
</body></html>
```

- Stores a username in the session upon a "login" (simulated by URL param) and retrieves it on subsequent visits within the same session.

5. **`application` Object:**
    - **Java Type:** `javax.servlet.ServletContext`
    - **Scope:** Application (available to all users and all components of the web application; lasts as long as the web application is running on the server).
    - **Purpose:** Used to store and retrieve information that is global to the entire web application, shared among all users and sessions. Also provides access to application initialization parameters (from `web.xml`), server information, and methods for logging messages.
    - **Concise Example:**

```
<%@ page contentType="text/html" %>
<html><body>
    <%
        Integer hitCount = (Integer)
application.getAttribute("appHitCounter");
        if (hitCount == null) {
            hitCount = 0;
        }
        hitCount++;
        application.setAttribute("appHitCounter", hitCount); // Store
application-wide counter
    %>
    This web application has been hit: <%= hitCount %> times by all
users. <br/>
    Server Info: <%= application.getServerInfo() %>
</body></html>
```

- Implements a simple application-wide hit counter, shared across all users accessing this JSP.

These implicit objects significantly simplify JSP development by providing direct, easy access to core web application functionalities and data scopes.

UNIT-IV

Q8. Discuss the steps to write a RMI program with an example of each step. (12.5)

1. **Define Remote Interface:** Extends `java.rmi.Remote`; methods throw `RemoteException`.

```java
// Adder.java
import java.rmi.*;
public interface Adder extends Remote {
    int add(int x, int y) throws RemoteException;
}
```

2. **Implement Remote Interface:** Extends `UnicastRemoteObject`, implements interface, constructor throws `RemoteException`.

```java
// AdderImpl.java
import java.rmi.*;
import java.rmi.server.*;
public class AdderImpl extends UnicastRemoteObject implements Adder {
    public AdderImpl() throws RemoteException { super(); }
    public int add(int x, int y) { return x + y; }
}
```

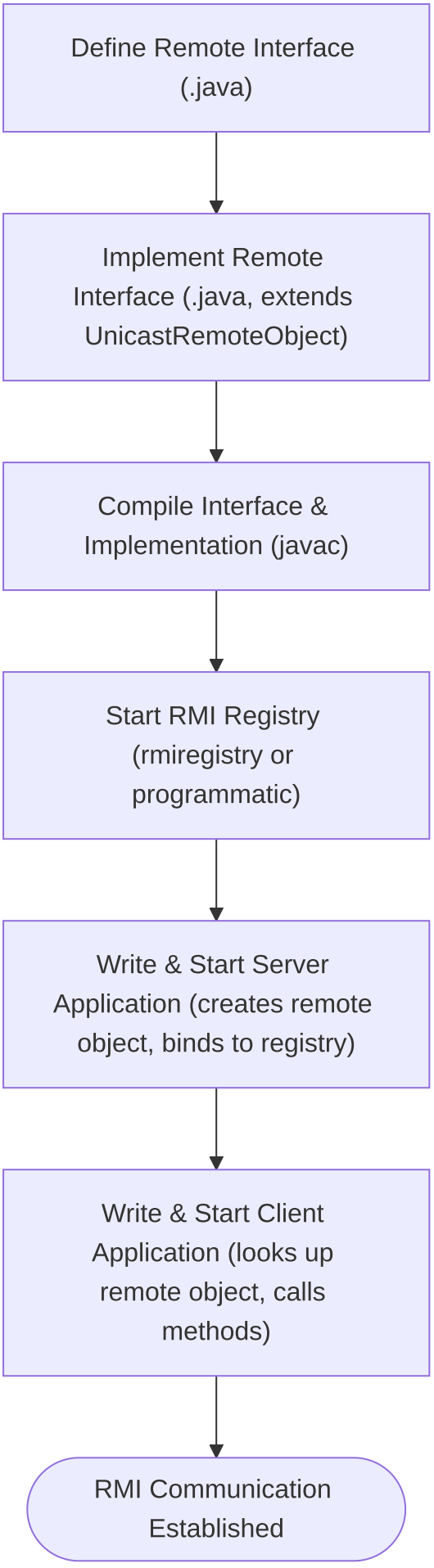
3. **Compile Interface & Implementation:** `javac Adder.java AdderImpl.java`. (Historically, `rmic` for stubs/skeletons).
4. **Start RMI Registry:** `rmiregistry` (command line) or `LocateRegistry.createRegistry(1099);` (programmatic).
5. **Write & Start Server Application:** Creates remote object, binds to registry.

```java
// AdderServer.java
import java.rmi.*;
import java.rmi.registry.*;
public class AdderServer {
    public static void main(String args[]) {
        try {
            LocateRegistry.createRegistry(1099);
            AdderImpl obj = new AdderImpl();
            Naming.rebind("rmi://localhost/AdderService", obj);
            System.out.println("Server ready.");
        } catch (Exception e) { System.out.println(e); }
    }
}
```

6. **Write & Start Client Application:** Looks up remote object, calls methods.

```java
// AdderClient.java
import java.rmi.*;
public class AdderClient {
    public static void main(String args[]) {
        try {
            Adder stub =
```

```
(Adder)Naming.lookup("rmi://localhost/AdderService");
            System.out.println("Sum: " + stub.add(34, 4));
        } catch (Exception e) { System.out.println(e); }
    }
}
```



Q9. Draw the architecture diagram of Hibernate framework and also explain its elements. (12.5)

Hibernate is a powerful Object-Relational Mapping (ORM) framework that simplifies database interaction in Java applications. Its architecture is designed in layers, with several key objects facilitating the persistence process.

Explanation of Hibernate Elements (Core Objects and Components):

1. **Configuration Object (`org.hibernate.cfg.Configuration` or JPA's `persistence.xml` processing):**
    - **Role:** This is the very first object created in a Hibernate application (typically at startup). Its primary responsibility is to gather all necessary configuration information and mapping details that Hibernate needs to operate.
    - **Details:**
        - It reads properties from configuration files like `hibernate.cfg.xml` (for native Hibernate) or `persistence.xml` (for JPA standard). These files specify database connection details (URL, username, password, JDBC driver), the database dialect (e.g., `MySQLDialect`, `OracleDialect` which helps Hibernate generate appropriate SQL), connection pool settings, caching strategies, transaction factory class, and other Hibernate-specific properties (like `hbm2ddl.auto` for schema generation).
        - It also processes mapping information, either from XML mapping files (`.hbm.xml`) or from annotations (`@Entity`, `@Table`, `@Id`, etc.) within the persistent Java classes.
    - **Lifecycle:** Created once at application startup. Used to build the `SessionFactory`.
2. **SessionFactory (`org.hibernate.SessionFactory` or JPA's `javax.persistence.EntityManagerFactory`):**
    - **Role:** A heavyweight, thread-safe factory object responsible for creating `Session` instances. It is immutable once created.
    - **Details:**
        - It's built from the `Configuration` object and is typically created **once per application** during startup because it's expensive to create. This single instance is then shared across all parts of the application that need to interact with the database.
        - It holds compiled mapping metadata (how Java classes map to database tables and columns).
        - It manages database connection properties (often via a `ConnectionProvider`).
        - It's the entry point for the (optional) **Second-Level Cache**, which is a cache shared across multiple sessions and can significantly improve performance by caching frequently accessed, rarely changed data.
    - **Lifecycle:** Created once at application startup and lives for the duration of the application.
3. **Session (`org.hibernate.Session` or JPA's `javax.persistence.EntityManager`):**
    - **Role:** A lightweight, **single-threaded, short-lived** object representing a "unit of work" or a conversation with the database. It's the primary interface used by the application to interact with

the database.

- **Details:**
  - Obtained from the `SessionFactory` whenever a database interaction is needed (e.g., at the beginning of a business transaction or web request).
  - It is **not thread-safe**; a `Session` instance should only be used by a single thread.
  - Provides methods for CRUD (Create, Read, Update, Delete) operations on persistent objects (e.g., `save()`, `update()`, `delete()`, `get()`, `load()`).
  - Acts as a factory for `Transaction` and `Query` objects.
  - Manages the **First-Level Cache** (also known as the session cache). This is a mandatory cache specific to the current `Session`. Any object loaded or saved within a session is cached here. If the same object is requested again within the same session, Hibernate retrieves it from this cache, avoiding a redundant database call. This cache is cleared when the session is closed.
- **Lifecycle:** Opened when needed, used for a set of operations, and then closed (e.g., `session.close()`).

4. **Transaction (`org.hibernate.Transaction` or JPA's `javax.persistence.EntityTransaction`):**
   - **Role:** Represents an atomic unit of work with the database. It ensures that a series of database operations either all succeed (commit) or all fail (rollback), maintaining data consistency and integrity.
   - **Details:**
     - Obtained from the `Session` object (e.g., `session.beginTransaction()`).
     - The application performs one or more database operations within the scope of a transaction.
     - The transaction is then ended by calling `commit()` (to make changes permanent in the database) or `rollback()` (to undo all changes made within the transaction if an error occurs).
     - While the notes mention it as "optional," for any write operations (insert, update, delete), using transactions is crucial for data integrity.
   - **Lifecycle:** Begins, then either committed or rolled back, associated with a `Session`.
5. **Persistent Objects (Entities):**
   - **Role:** These are plain old Java objects (POJOs) that represent the data your application works with and that needs to be persisted in the database.
   - **Details:**
     - Each instance of a persistent class typically corresponds to a row in a database table.
     - Fields (instance variables) in the class map to columns in the table.
     - They are marked for persistence using annotations (e.g., `@Entity`, `@Id` from JPA) or defined in XML mapping files.

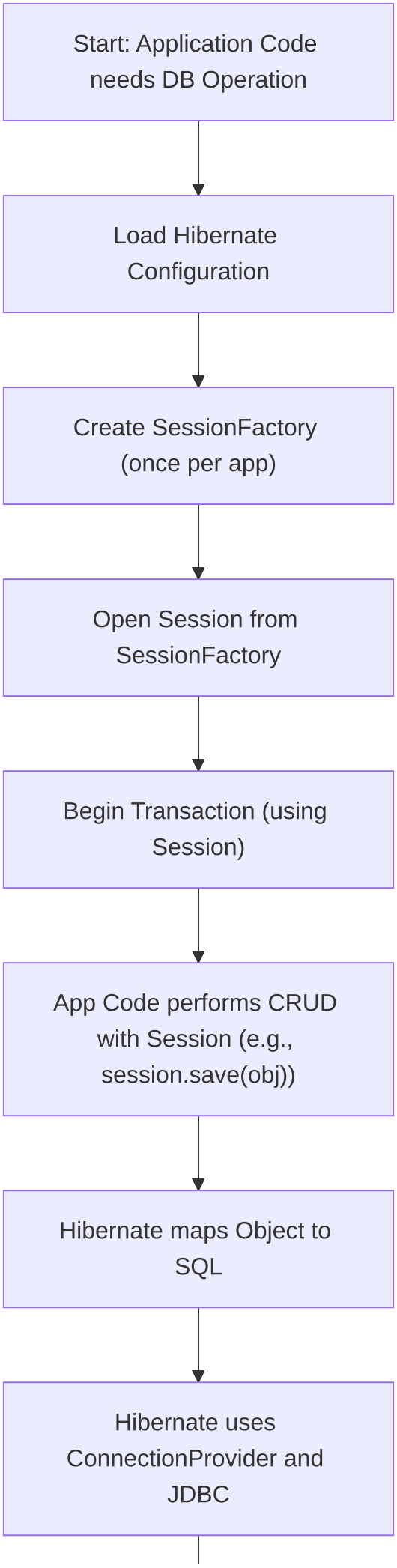
    - Hibernate manages the lifecycle of these objects (transient, persistent, detached states) in relation to a `Session`.
6. **ConnectionProvider (`org.hibernate.connection.ConnectionProvider`):**
    - **Role:** An abstraction layer for obtaining JDBC connections to the database.
    - **Details:**
        - Hibernate needs JDBC connections to communicate with the database. The `ConnectionProvider` is responsible for supplying these connections.
        - It can be configured to use a built-in connection pool, a third-party connection pool (like C3P0, HikariCP), or obtain connections from a `DataSource` managed by an application server (via JNDI).
        - This abstracts the Hibernate core from the specifics of connection management.
    - **Lifecycle:** Managed by the `SessionFactory`.
7. **Query Objects (`org.hibernate.Query`, `javax.persistence.Query`, `org.hibernate.Criteria`):**
    - **Role:** Used to retrieve data from the database or perform bulk update/delete operations.
    - **Details:**
        - Hibernate supports several ways to query:
            - **HQL (Hibernate Query Language):** An object-oriented query language, similar to SQL but uses entity and property names instead of table and column names.
            - **JPQL (Java Persistence Query Language):** The standard query language defined by JPA, very similar to HQL.
            - **Native SQL:** Allows executing raw SQL queries directly.
            - **Criteria API:** A programmatic, type-safe way to build queries dynamically.
        - Query objects are created from the `Session`.

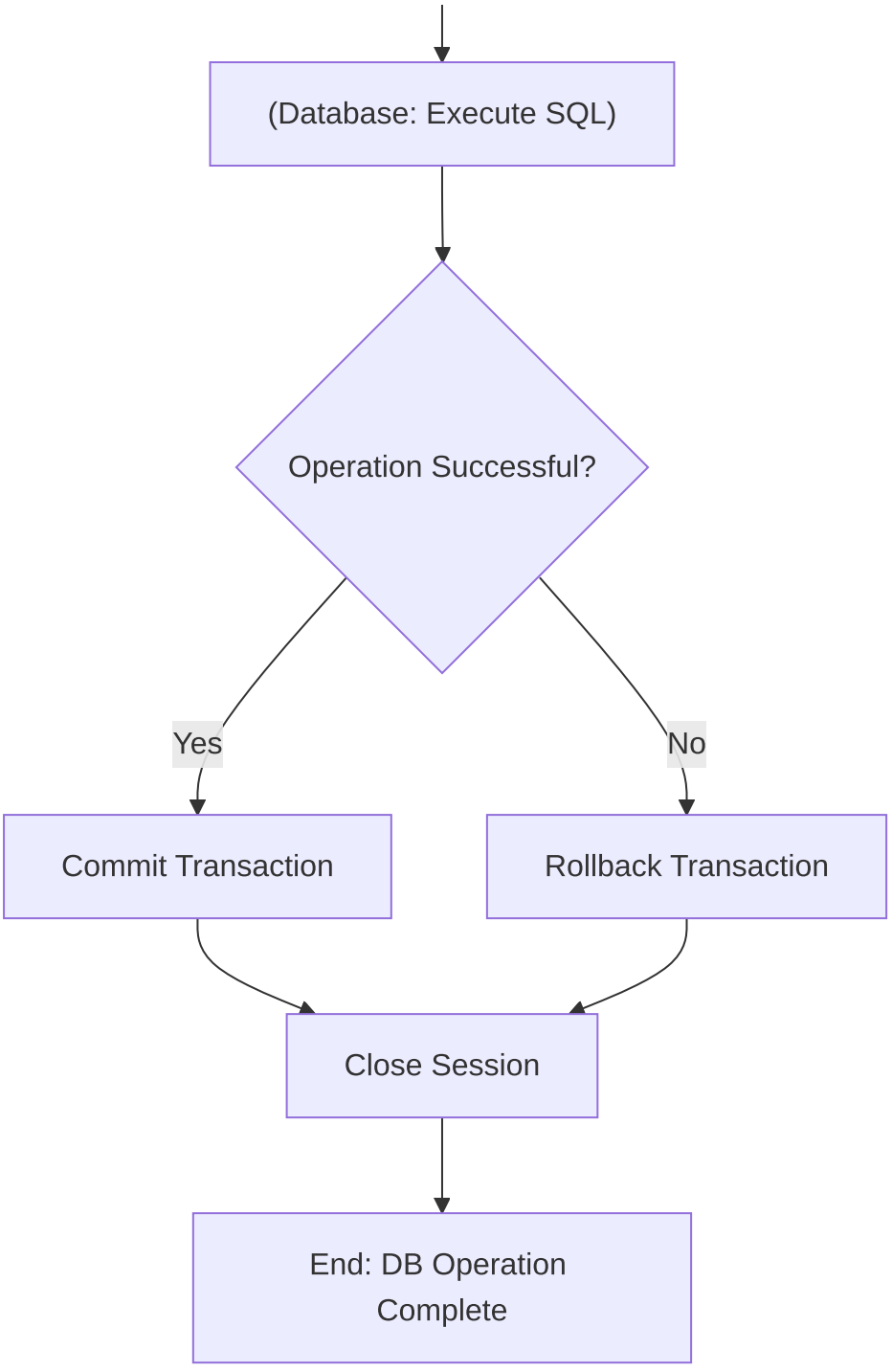
Hibernate Layers (Conceptual):

- **Java Application Layer:** This is your application code (e.g., Servlets, business logic classes, Data Access Objects - DAOs) that uses the Hibernate/JPA APIs to perform persistence operations.
- **Hibernate Framework Layer:** This is the core of Hibernate. It includes the `SessionFactory`, `Session`, transaction management, cache management, mapping processing, and the query engine. It translates object-oriented operations into database interactions.
- **Backend API Layer:** Hibernate itself doesn't directly talk to the database drivers. It uses standard Java APIs for this:
    - **JDBC (Java Database Connectivity):** The fundamental API for all database communication in Java.
    - **JTA (Java Transaction API) (Optional):** Used for managing distributed transactions, typically in an enterprise application server environment.

    - **JNDI (Java Naming Directory Interface) (Optional):** Can be used to look up `DataSource` objects configured in an application server.
- **Database Layer:** The actual relational database system (e.g., MySQL, PostgreSQL, Oracle, SQL Server) where the data is stored. Hibernate generates SQL specific to this database (based on the configured dialect).

This layered architecture with well-defined components allows Hibernate to provide a robust, flexible, and database-independent persistence solution for Java applications.


Start: Application Code needs DB Operation
Load Hibernate Configuration
Create SessionFactory (once per app)
Open Session from SessionFactory
Begin Transaction (using Session)
App Code performs CRUD with Session (e.g., session.save(obj))
Hibernate maps Object to SQL
Hibernate uses ConnectionProvider and JDBC


(Database: Execute SQL)
Operation Successful?
Yes
No
Commit Transaction
Rollback Transaction
Close Session
End: DB Operation Complete