J2EE

UNIT-1

1-mark type

Define J2EE.
J2EE (Java 2 Platform, Enterprise Edition) is a Java platform specification for building and deploying multi-tier, distributed, enterprise applications (web apps, business logic, and backend integration). It provides APIs and runtime services such as Servlets, JSP, EJB, JMS, JNDI, JDBC, and security/transaction management.

Note (brief): J2EE was later rebranded as Java EE and now Jakarta EE in newer ecosystems.

Expand JDBC.
JDBC = Java Database Connectivity.
(It's the Java API that enables Java applications to connect to and interact with relational databases using SQL.)

What is Tomcat?
Apache Tomcat is an open-source Java Servlet container and web server that implements the Java Servlet and JSP specifications. It runs web applications (Servlets/JSP) and provides the web tier of a Java EE-like stack. Tomcat is commonly used as a lightweight application server for web components (it does not provide a full EJB container).

2-mark type

Explain two-tier architecture.

Two-tier (client–server) architecture has two layers:

1. **Client Tier (Presentation):** The user interface — desktop or thin client (GUI/browser).
2. **Server Tier (Data/Business):** Database server (and sometimes business logic) — handles data storage and processing.

Simple diagram (two boxes):

[Client (UI)] <----TCP/SQL----> [Server (Database / Business Logic)]

Key points:

- Client directly communicates with the database server.
- Simpler, faster for small systems.
- **Advantages:** Easy to implement, lower latency for simple queries, fewer layers.
- **Disadvantages:** Poor scalability, business logic duplication on clients, tight coupling between client and DB, harder to maintain and secure for large systems.

(Contrast: three-tier separates presentation, business logic, and data for better scalability/maintenance.)

Explain difference between Statement and PreparedStatement.

Aspect	Statement	PreparedStatement
SQL compilation	Compiled each time when executed	SQL precompiled once; can reuse with different parameters
Use case	Simple, ad-hoc SQL (no parameters)	Repeated queries, parameterized SQL
Parameter support	No placeholders	Supports ? placeholders
Performance	Slower for repeated execution (no caching)	Faster for repeated execution (precompiled, DB caches execution plan)
Security	Vulnerable to SQL injection if concatenating user input	Safer — parameters are bound, preventing most SQL injection
Example	`stmt.executeQuery("SELECT * FROM users WHERE name = '" + name + "'");`	`ps = con.prepareStatement("SELECT * FROM users WHERE name = ?");` `ps.setString(1, name);`

Short example (PreparedStatement safer):

```
// Statement (unsafe)
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM users WHERE username = '" + user + "'");

// PreparedStatement (safe)
PreparedStatement ps = con.prepareStatement("SELECT * FROM users WHERE username = ?");
ps.setString(1, user);
ResultSet rs = ps.executeQuery();
```

Exam keywords: parameter binding, precompilation, SQL injection prevention, reusability.

5-mark type

Explain J2EE architecture with a neat diagram (detailed)

High-level J2EE multi-tier architecture separates responsibilities into layers for scalability, maintainability, and manageability.

Neat simple diagram (text):

```
[Client Tier]
  - Browser / Mobile / Desktop
        |
        v
[Web Tier (Web Container)]
  - Web Server / Servlet Container (Tomcat, WebLogic web tier)
  - Servlets, JSP, JSF
        |
        v
[Business Tier (EJB / Application Container)]
  - EJBs (Session, Entity*, Message-Driven)
  - Business services, transactions, security
        |
        v
[Integration / EIS Tier]
  - Database (via JDBC), Legacy systems, Messaging systems (JMS)
```

Detailed explanation (pointwise):

1. **Client Tier (Presentation):**
    - Includes web browsers, mobile apps, or thick clients.
    - Responsible for rendering UI and collecting user input. Communicates over HTTP (or other protocols) to the web tier.
2. **Web Tier (Web Container):**
    - Hosts **Servlets**, **JSP**, **JSF**; manages HTTP requests and responses.
    - Handles presentation logic and forwards requests to business tier components.
    - Web container provides lifecycle management, concurrency, security for web components.
3. **Business Tier (EJB / Application Container):**
    - Contains business logic implemented as **Enterprise JavaBeans (EJBs)** or POJO services.
    - Responsibilities: transactions (via JTA), business rules, security, concurrency, remote access.
    - EJB types: Session beans (stateless/stateful), Message-driven beans (asynchronous JMS consumers). (*Entity beans are legacy; JPA replaced them.*)
4. **Integration / EIS Tier (Enterprise Information Systems):**
    - Persistent storage (RDBMS) accessed via **JDBC** or JPA.
    - Messaging systems (JMS), legacy systems, and third-party services.
5. **Supporting Services / APIs (Across tiers):**
    - **JNDI** for naming and lookup (DataSource, EJBs).
    - **JMS** for messaging and asynchronous communication.
    - **JTA** for distributed transaction management.
    - **Security services** for authentication/authorization.
    - **Connection pooling, clustering** for scalability/high availability.
6. **Deployment descriptors & containers:**
    - Application server (e.g., WebLogic, JBoss/WildFly) provides full container services (web + EJB + resource management). Tomcat is a web container (servlets/JSP) but not a full EJB container.

Why multi-tier?

- Separation of concerns → easier maintenance and development.
- Scalability: each tier can be scaled independently.
- Reusability: business logic reused by different clients.

Exam keywords to write: web container, EJB container, JNDI, JMS, JDBC, JTA, deployment descriptor, session beans, stateless/stateful, message-driven beans, connection pooling.

Explain JDBC architecture and types of JDBC drivers (detailed)

JDBC architecture (diagram + explanation)

Diagram (text):

[Java Application] --JDBC API--> [JDBC Driver Manager/Drivers] --DB Protocol--> [Database]

Components & flow:

1. **JDBC API (in Java app):** Application uses classes/interfaces such as `DriverManager`/`DataSource`, `Connection`, `Statement`/`PreparedStatement`/`CallableStatement`, `ResultSet`, and `SQLException`. These are part of `java.sql` package.
2. **JDBC Driver Manager / Driver:**
    - `DriverManager` locates appropriate JDBC driver for the database URL and obtains a `Connection`.
    - A **JDBC driver** implements the JDBC API for a particular DB (Oracle, MySQL, etc.). There can be multiple drivers loaded.
3. **Database / DBMS:**
    - Driver translates JDBC calls into DB-specific network protocol and SQL commands; sends to database and returns results.
4. **Optional: DataSource & Connection Pooling:**
    - Modern apps use `DataSource` (javax.sql) for container-managed connections and connection pooling (improves performance). App servers provide pooled DataSource via JNDI.

Typical JDBC workflow (steps):

1. Load driver (older): `Class.forName("com.mysql.cj.jdbc.Driver");`
2. Get connection: `Connection con = DriverManager.getConnection(url, user, pass);` (or via DataSource lookup)
3. Create statement or prepared statement: `Statement st = con.createStatement();` or `PreparedStatement ps = con.prepareStatement(sql);`
4. Execute query/update: `ResultSet rs = ps.executeQuery();` or `ps.executeUpdate();`
5. Process `ResultSet` (iterate over rows).
6. Close `ResultSet`, `Statement`, `Connection` (return to pool).

Key interfaces: `Connection`, `Statement`, `PreparedStatement`, `CallableStatement` (stored procedures), `ResultSet`, `DatabaseMetaData`.

Types of JDBC drivers (Type 1 to Type 4) — explain & compare

There are **four driver types** defined by JDBC.

Type 1 — JDBC-ODBC Bridge Driver

- **How:** Java calls converted to ODBC calls via a bridge; ODBC driver then talks to DB.
- **Pros:** Quick to prototype on platforms with ODBC.
- **Cons:** Requires ODBC on client, platform dependent, poor performance, not recommended for production. (Deprecated/obsolete in modern Java.)

Type 2 — Native-API (partly Java) driver

- **How:** Java calls mapped to native DB client library (vendor-specific native code) using JNI.
- **Pros:** Better performance than Type 1.
- **Cons:** Requires native binary library on client; platform dependent; complex deployment.

Type 3 — Network Protocol / Middleware driver

- **How:** JDBC calls sent to a middleware server (translation server) which converts to DB protocol. Client uses Java-only driver; middleware handles DB specifics.
- **Pros:** Database-independent client, middleware can do load balancing and connection pooling.
- **Cons:** Extra network hop and middleware to manage.

Type 4 — Native Protocol / Pure Java driver

- **How:** 100% Java driver that converts JDBC calls directly into DB's native network protocol. (e.g., MySQL Connector/J, PostgreSQL driver)
- **Pros:** Platform independent, good performance, easy deployment (single JAR), most widely used in production.
- **Cons:** Driver must be implemented for each DB vendor (but vendors provide them).

Which to use?

- **Type 4** is preferred today for performance, portability, and ease of deployment. Use `DataSource` + connection pooling in app servers for production.

Additional useful points (exam boosters)

- **Connection pooling:** Managed by app server or libraries (e.g., HikariCP, c3p0). Improves performance by reusing connections. Usually exposed as a `DataSource` via JNDI.
- **Transactions:** JDBC supports transaction control via `con.setAutoCommit(false)`, `con.commit()`, `con.rollback()`. For distributed transactions across multiple resources, JTA is used in J2EE containers.
- **SQLException:** Handle nested exceptions, inspect SQLState and vendor error codes.
- **CallableStatement:** For calling stored procedures: `CallableStatement cs = con.prepareCall("{call procname(?, ?)}");`

Quick code snippet: JDBC basic usage

```
// 1. Load driver (optional for modern drivers)
// Class.forName("com.mysql.cj.jdbc.Driver");

String url = "jdbc:mysql://localhost:3306/mydb";
try (Connection con = DriverManager.getConnection(url, "user", "pass");
     PreparedStatement ps = con.prepareStatement("SELECT id, name FROM students WHERE grade = ?")) {

    ps.setInt(1, 12);
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) {
            int id = rs.getInt("id");
            String name = rs.getString("name");
            // process...
        }
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

UNIT-2

1-Mark Type

Q1. Define Servlet.
A **Servlet** is a Java program that runs on a web server or application server and acts as a **server-side component** to handle client requests (mostly HTTP) and generate dynamic responses (such as HTML, JSON, XML, etc.).

- It extends the capabilities of servers hosting applications accessed by request–response programming models.
- Servlets are part of the Java EE platform.

Q2. Full form of API.
API = **Application Programming Interface**

- It is a set of classes, methods, and protocols that allow software components to communicate with each other.
- In Java, APIs (like Servlet API, JDBC API) provide predefined interfaces and classes for developers.

Q3. Servlet life cycle stages.
The **three stages** of a servlet life cycle are:

1. **Initialization (`init()`):** Servlet instance created once; used for one-time setup.
2. **Service (`service()`):** Called for each client request; processes request and generates response.
3. **Destruction (`destroy()`):** Called before servlet is unloaded from memory; cleanup activities.

2-Mark Type

Q1. Difference between GenericServlet and HttpServlet.

Feature	GenericServlet	HttpServlet
Base class	Abstract class implementing `Servlet` interface	Subclass of `GenericServlet`
Protocol support	Protocol-independent (can handle any type)	Specifically designed for **HTTP protocol**
Methods to override	Must override `service()`	Override doGet(), doPost(), doPut(), doDelete(), etc.
Common usage	Rare, used for custom protocols	Most commonly used for web applications (HTTP)

Q2. Role of Deployment Descriptor (web.xml).

- The **deployment descriptor (web.xml)** is an XML configuration file located in the `WEB-INF` directory of a web application.
- **Purpose:** Defines how the web application should be deployed and configured.
- **Key roles:**
    1. Define servlet names and their class mappings.
    2. Define URL patterns for servlets (`<servlet-mapping>`).
    3. Configure initialization parameters.

4. Configure session timeout, error pages, security constraints, filters, etc.
5. Acts as a bridge between servlet classes and the application server.

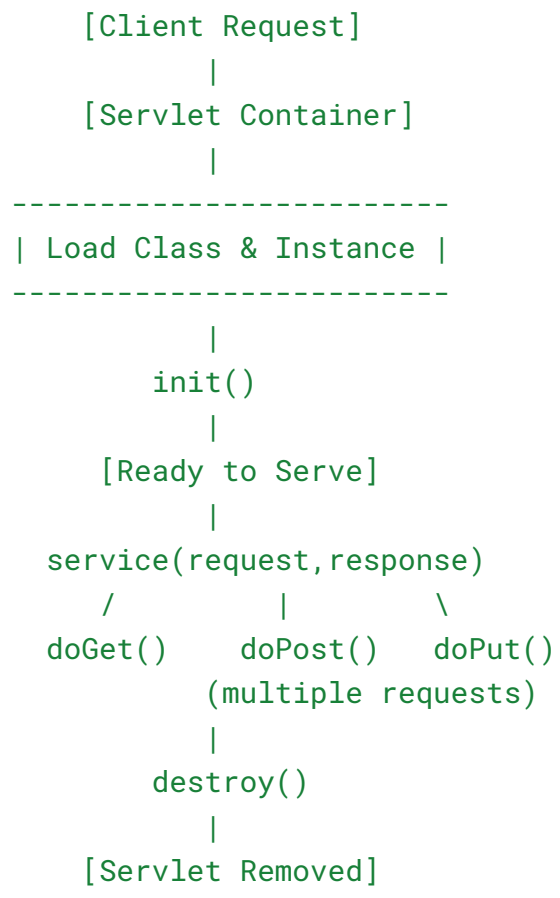
5-Mark Type

Q1. Explain servlet life cycle with diagram.

A servlet's life cycle is managed by the **Servlet Container** (like Tomcat). It has 3 main stages:

1. **Loading & Instantiation**
    - Container loads servlet class and creates its instance.
2. **Initialization (`init()` method)**
    - Called **once** when the servlet is first loaded.
    - Used for allocating resources (DB connections, log setup).
3. **Request handling (`service()` method)**
    - Called **for every client request**.
    - Dispatches to `doGet()`, `doPost()`, etc. depending on HTTP method.
4. **Destruction (`destroy()` method)**
    - Called once when the servlet is unloaded or server shuts down.
    - Used for cleanup (close DB connections, free resources).

Diagram (Servlet Life Cycle):



Q2. Explain servlet request/response handling.

Servlet Request Handling:

- Managed by `HttpServletRequest` object.
- Provides data about:
    - **Client request information** (headers, method type, URL, IP).
    - **Form data** via `getParameter("name")`.
    - **Session data** via `getSession()`.
    - **Attributes** using `setAttribute()` / `getAttribute()`.

Servlet Response Handling:

- Managed by `HttpServletResponse` object.
- Provides methods to:
    - Set **content type**: `response.setContentType("text/html");`
    - Write output: `PrintWriter out = response.getWriter(); out.println("<h1>Hello</h1>");`
    - Redirect response: `response.sendRedirect("home.jsp");`
    - Set cookies, headers, and HTTP status codes.

Flow of request/response:

1. Client (Browser) sends request (HTTP GET/POST).
2. Servlet container converts it into `HttpServletRequest`.
3. Servlet processes it via `doGet()` or `doPost()`.
4. Servlet creates a response using `HttpServletResponse`.
5. Container sends the response back to client.

🔑 **Keywords for exam:**

- Servlet = server-side Java program.
- Life cycle: init → service → destroy.
- Deployment Descriptor (web.xml) maps servlet to URL.
- Request/Response = HttpServletRequest + HttpServletResponse.
- GenericServlet is protocol-independent, HttpServlet is HTTP-specific.

UNIT-3

1-Mark Type

Q1. Expand JSP.
JSP = **Java Server Pages**

- A technology used to create dynamic web pages using Java code embedded in HTML.
- Part of the Java EE platform.

Q2. List any two implicit objects.
In JSP, the container automatically provides **implicit objects**. Examples:

1. `request` → Represents the client's request (HttpServletRequest).

2. `response` → Used to generate response to the client (HttpServletResponse).
   (Other examples: `session`, `application`, `out`, `config`, `pageContext`.)

Q3. Define directive elements.

- **Directive elements** in JSP are instructions for the JSP engine that affect the overall structure of the servlet generated from JSP.
- Syntax: `<%@ directive attribute="value" %>`
- Types:
    1. **page directive** (`<%@ page ... %>`)
    2. **include directive** (`<%@ include ... %>`)
    3. **taglib directive** (`<%@ taglib ... %>`).

✅ 2-Mark Type

Q1. Difference between JSP and Servlet.

Feature	JSP	Servlet
Nature	HTML with embedded Java code	Pure Java class with HTML in `out.println()`
Focus	Presentation (view)	Logic (controller)
Ease of coding	Easier (write HTML + small Java)	Harder (HTML inside Java code)
Translation	JSP is first translated into Servlet	Directly written as Servlet
Usage	Front-end page generation	Request handling, business logic

---

Q2. Explain scripting elements in JSP.
Scripting elements allow embedding Java code inside a JSP page. There are three types:

1. **Declarations (`<%! ... %>`)**
    - Declare variables and methods.

```
<%! int count = 0; public int getCount(){ return count++; } %>
```

2. **Scriptlets (`<% ... %>`)**
    - Java code fragments executed for every request.

```
<% out.println("Current Time: " + new java.util.Date()); %>
```

3. **Expressions (`<%= ... %>`)**
    - Short form to output values directly.

```
<h1>Welcome <%= request.getParameter("name") %></h1>
```

✅ 5-Mark Type

Q1. Explain JSP architecture and life cycle.

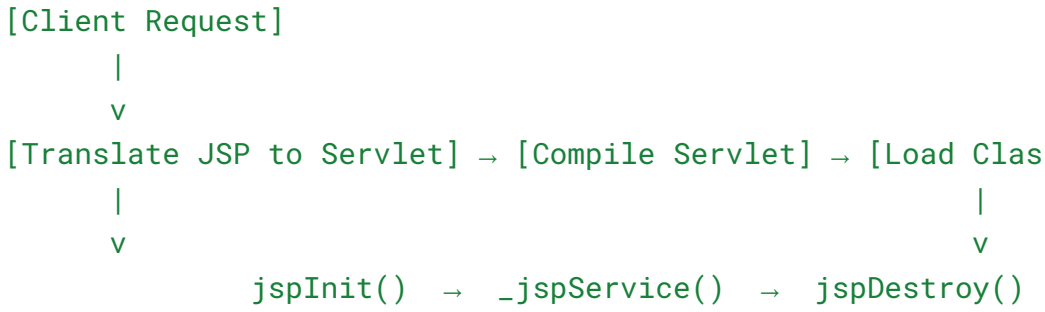
JSP Architecture

1. **Client (Browser):** Sends HTTP request to server.
2. **Web Server / JSP Container (e.g., Tomcat):**
    - Finds the JSP page.
    - If JSP is new or modified → it is compiled into a Servlet.
3. **Servlet Class:** The generated servlet handles requests using `service()` and produces response.
4. **Database / Backend (optional):** JSP/Servlet can interact with DB via JDBC.
5. **Response (HTML/JSON/XML):** Sent back to client.

JSP Life Cycle

1. **Translation Phase** – JSP is translated into a Servlet by the container.
2. **Compilation Phase** – The servlet source is compiled into bytecode.
3. **Class Loading** – The servlet class is loaded into JVM.
4. **Instantiation** – An instance of the servlet is created.
5. **Initialization (`jspInit()`)** – Called once when JSP is initialized.
6. **Request Processing (`_jspService()`)** – For every request, this method executes and generates response.
7. **Destroy (`jspDestroy()`)** – Called once when JSP is being unloaded from container.

Lifecycle Diagram (text-based):



Q2. Explain JSP elements in detail (Directives, Scripting, Actions).

JSP has **three types of elements**:

1. Directives

- Provide global instructions to the JSP container.
- Types:
    - **Page Directive** (`<%@ page ... %>`) → defines page settings (import classes, session, error pages).
      Example: `<%@ page language="java" import="java.util.*" %>`

- **Include Directive** (`<%@ include file="header.jsp" %>`) → static include (file content added at translation time).
- **Taglib Directive** (`<%@ taglib uri="..." prefix="..." %>`) → declares custom tags and libraries.

---

2. Scripting Elements

- Allow embedding Java code.
- **Declaration (`<%! ... %>`), Scriptlet (`<% ... %>`), Expression (`<%= ... %>`)**.
- Used for variables, logic, or printing values.

---

3. Action Elements

- XML-like tags that control JSP behavior at runtime.
- Examples:
  - `<jsp:include page="header.jsp" />` → dynamic include at request time.
  - `<jsp:forward page="home.jsp" />` → forward request to another resource.
  - `<jsp:param name="user" value="Harshal" />` → pass parameters.
  - `<jsp:useBean id="obj" class="mypackage.MyClass" scope="session" />` → use/create JavaBeans.
  - `<jsp:setProperty>` and `<jsp:getProperty>` for bean properties.

Summary (for 5 marks):

- **Directives** → instructions for container (compile-time).
- **Scripting** → embed Java code (runtime).
- **Actions** → predefined tags for dynamic behavior and JavaBean usage.

🔑 **Keywords for exam:**

- JSP = Java Server Pages.
- Life cycle: translation → compilation → loading → instantiation → jspInit() → _jspService() → jspDestroy().
- Implicit objects (`request`, `response`, `session`, `out`, etc.).
- Elements: Directives, Scripting, Actions.