🧭 Roadmap for Advanced Java (J2EE)

#	Topic
1	Servlet Basics (Lifecycle, API)
2	JSP (Java Server Pages)
3	HTTP and Session Management
4	Filters and Listeners
5	JDBC (Java Database Connectivity)
6	MVC Architecture (without Spring)
7	JSTL and Expression Language (EL)
8	Web.xml vs Annotation Config
9	Deployment on Tomcat
10	Interview Questions & Real Projects

🔍 Should You Learn J2EE as a Full-Stack Developer?

- **YES**, but focus only on **required and interview-relevant topics**.
- Most real-world projects now use **Spring Boot** instead of raw Servlets or JSP.
- However, interviewers may still ask **servlet lifecycle**, **session handling**, or **JDBC basics**.

✅ Topic 1: Servlet Basics (Servlet API + Lifecycle)

🔷 What is a Servlet?

A **Servlet** is a Java class that handles **HTTP requests** and generates **dynamic web responses** (usually HTML).
It's a core part of the Java EE (Jakarta EE) platform.

🔶 Why do we use Servlets?

Reason	Explanation
Handle HTTP requests	Used to build server-side web applications
Platform-independent	Write once, run anywhere (Java-based)
Fast and efficient	Lives in memory (no process startup/shutdown per request like CGI)
Base of modern frameworks	Spring MVC is internally based on servlet principles

---

🔷 Servlet Lifecycle

There are **3 main methods** involved:

Method	Purpose
init()	Called once when servlet is first loaded
service()	Called for every request
destroy()	Called once before the servlet is destroyed

Lifecycle Flow:

scss

CopyEdit

Browser Request → Servlet Container → init() → service() → destroy()

🔸 Code Example

java

CopyEdit

```
@WebServlet("/hello")
public class HelloServlet extends HttpServlet {

  @Override
  public void init() {
    System.out.println("Servlet initialized");
  }

  @Override
  public void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws IOException {
    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter();
    out.println("<h1>Hello from Servlet</h1>");
  }

  @Override
  public void destroy() {
    System.out.println("Servlet destroyed");
  }
}
```

@WebServlet is used instead of web.xml in modern servlet development.

🔶 **Real-Life Example**

Imagine you're logging into a website:

1. You enter credentials and hit submit.

2. The form data is sent to a **Servlet** (LoginServlet) via HTTP POST.
3. The servlet reads the data, checks it against the database, and sends back a response (success/failure).

🔴 Drawbacks of Servlets

Issue	Explanation
Verbose Code	Need to manage HTML with Java (mixing logic + view)
Manual session management	Must handle cookies and session manually
Difficult to scale UI	Can't build modern UIs easily (no JS, no component system)
JSP and frameworks preferred	JSP, JSF, or Spring Boot preferred for cleaner separation

---

✅ Summary

Key Concept	Description
Servlet	Java class to handle HTTP requests
Lifecycle Methods	init(), service(), destroy()
Runs in	Servlet Container (Tomcat, Jetty, etc.)
Example Use Case	Login, Registration, API endpoints
Modern Replacement	Spring Boot (with embedded servlet support)

✅ Topic 2: JSP (Java Server Pages)

🔷 What is JSP?

JSP (Java Server Pages) is a technology that helps you write **HTML pages with embedded Java code**.

It's mainly used to **generate dynamic web content** — like displaying user data, creating forms, showing product details, etc.

🔶 Why Do We Use JSP?

Purpose	Explanation
Simplify HTML + Java mixing	Easier than writing HTML in Java code (like in Servlets)
Cleaner UI layer	Separates presentation (HTML) from logic (Java)
Supports dynamic content	Can fetch values from DB, session, etc.
Replaces Servlet HTML writing	Easier alternative to PrintWriter in servlets

---

🔷 Basic JSP Syntax

jsp

CopyEdit

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

- <%= %>: Outputs the result of an expression.
- <%! %>: Declares variables/methods.
- <% %>: Executes Java code (not recommended in modern usage).

🔷 JSP Lifecycle (Similar to Servlets)

Step	Description
Translation	JSP is converted into a servlet class
Compilation	Servlet is compiled into a .class
Initialization	init() method is called
Request Handling	service() method is called
Destruction	destroy() method is called

🔶 How JSP Works Internally?

arduino

CopyEdit

Client Request → JSP Page → Translated to Servlet → Compiled → Response to Client

JSP is just **syntactic sugar** — behind the scenes, it's **converted to a servlet**.

🔸 Real-Life Use Case

Let’s say you’re building a website that shows user profile details.

- You fetch the user data in the servlet or controller.
- Forward that data to a **JSP page** for display.

jsp

CopyEdit

<!-- profile.jsp -->

<h2>User Details</h2>

Name: ${user.name}

Email: ${user.email}

🔶 Tags in JSP

JSP supports three main types of tags:

1. **Directive Tags** (<%@ ... %>) – page level info
2. **Scriptlet Tags** (<% ... %>) – write Java code (not recommended)
3. **Expression Tags** (<%= ... %>) – output data

✅ Benefits of JSP

Benefit	Description
Easy to write views	Just HTML + simple Java
MVC support	Acts as View layer in MVC pattern
Reusable templates	Use JSP includes and tag libraries (JSTL)
Standardized	Supported in all servlet containers

❌ Drawbacks of JSP

Drawback	Explanation
Logic and view can mix	Not good practice for modern MVC
Hard to debug	Java errors in HTML files are difficult to trace
Outdated for REST-based apps	Better to use Thymeleaf, React, Angular, etc. now
Poor frontend separation	Not ideal for component-based frontend architectures

---

✅ Summary

Key Concept	Details
JSP	Java + HTML for dynamic web pages
Behind scenes	Translated into servlet
Syntax	<% %>, <%= %>, <%@ %>
Good for	UI layer of traditional web apps
Replaced by	Thymeleaf, React, Angular in modern apps

✅ Topic 3: HTTP & Session Management

🔷 What is HTTP?

HTTP (HyperText Transfer Protocol) is the protocol used by web browsers and servers to communicate.

- It's **stateless**, meaning **each request is independent**.
- Runs on **port 80** (HTTPS on port 443).
- Used for **GET, POST, PUT, DELETE**, etc.

🔸 Common HTTP Methods:

Method	Purpose	Example Use
GET	Fetch data	View a user profile
POST	Submit new data	Register a new user
PUT	Update existing data	Update user details
DELETE	Delete data	Remove a product

🔷 Why Session Management is Needed?

Since HTTP is **stateless**, it doesn't remember users across requests.

👉 **Session management** solves this by **tracking users between requests**, like:

- Logged-in user sessions
- Shopping cart info
- Temporary user preferences

🔶 Types of Session Management

Technique	Description
Cookies	Small pieces of data stored in the browser and sent with each request

Technique	Description
URL Rewriting	Session ID is passed as a query parameter in the URL
Hidden Fields	Session data stored in hidden form fields
HttpSession	Server-side session tracking (most used in Java)

🔷 HttpSession (in Java EE)

Java provides **HttpSession** object to manage session data.

🔸 How to use it:

java

CopyEdit

```
HttpSession session = request.getSession();
session.setAttribute("username", "Viraj");
```

🔸 Retrieve it in another request:

java

CopyEdit

```
HttpSession session = request.getSession(false);
String username = (String) session.getAttribute("username");
```

🔸 Session Configuration

xml

CopyEdit

```
<!-- web.xml -->
<session-config>
   <session-timeout>30</session-timeout> <!-- in minutes -->
</session-config>
```

🔷 Real-Life Example

1. User logs in: credentials verified → server creates a session
2. Session stores username or userId
3. In every request, server checks session info
4. When user logs out, session is destroyed

✅ Benefits of Session Management

Benefit	Description
Maintains user state	Helps web apps remember who the user is
Reduces DB calls	Data like "username" can be stored in session
Secure	Server-side sessions are safer than cookies

❌ Drawbacks of Sessions

Drawback	Explanation
Memory overhead	Sessions are stored on server → more users = more memory
Not scalable for large apps	Need session replication in clustered environments
Session Hijacking	If session ID is stolen, user identity can be misused

---

✅ Summary

Concept	Details
HTTP	Stateless protocol for web communication
GET/POST/etc.	Methods for data exchange
Session	Mechanism to track users across requests
HttpSession	Java EE's built-in session object
Best Practice	Use sessions for minimal, essential data only

✅ Topic 4: Filters and Listeners in Servlet (J2EE)

🔷 What is a Filter?

A **Filter** is a reusable piece of code in Java EE that can:

- Intercept **requests before they reach servlets/JSPs** (Pre-processing)
- Modify **responses before they go back to the client** (Post-processing)

🔸 Why Use Filters?

Use Case	Example
Logging	Log every request URI, method, etc.
Authentication/Authorization	Check if a user is logged in before accessing a resource
Compression	GZIP compress the response data
Input validation	Check parameters before sending to servlet
CORS & headers	Modify response headers (e.g., add Access-Control-Allow-Origin)

🔷 Filter Lifecycle

1. **init()** – Runs once when the filter is created
2. **doFilter()** – Called for each request
3. **destroy()** – Called once when the filter is removed

🔸 Filter Code Example

java

CopyEdit

@WebFilter("/secure/*")

public class AuthFilter implements Filter {

```
    public void init(FilterConfig config) {}

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
            throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpSession session = req.getSession(false);

        if (session == null || session.getAttribute("user") == null) {
            ((HttpServletResponse) response).sendRedirect("/login.jsp");
        } else {
            chain.doFilter(request, response); // Continue to servlet
        }
    }

    public void destroy() {}
}
```

🔷 What is a Listener?

Listeners are used to listen for important **events** in the web application lifecycle like:

- App startup/shutdown
- Session creation/destruction
- Attribute addition/removal

🔶 Types of Listeners

Listener Type	Trigger Event
ServletContextListener	App startup or shutdown

Listener Type	Trigger Event
HttpSessionListener	Session creation or destruction
ServletRequestListener	Every HTTP request to the server
Attribute Listeners	Track when attributes are added/removed/changed

---

🔸 **ServletContextListener Example**

java

CopyEdit

```
@WebListener
public class AppStartupListener implements ServletContextListener {
  public void contextInitialized(ServletContextEvent sce) {
    System.out.println("Web application started");
  }

  public void contextDestroyed(ServletContextEvent sce) {
    System.out.println("Web application stopped");
  }
}
```

🔸 **HttpSessionListener Example**

java

CopyEdit

```
@WebListener
public class SessionTracker implements HttpSessionListener {
  public void sessionCreated(HttpSessionEvent se) {
    System.out.println("New Session Created: " + se.getSession().getId());
  }
```

```
    public void sessionDestroyed(HttpSessionEvent se) {
        System.out.println("Session Destroyed: " + se.getSession().getId());
    }
}
```

🔷 Real-Life Examples

Feature	Uses Filter / Listener?
Login check	Filter (validate before controller)
Count logged-in users	Listener (sessionCreated/destroyed)
Load config on startup	Listener (contextInitialized)
Logging API hits	Filter (doFilter())

✅ Benefits

Feature	Filters	Listeners
Centralized	Apply logic across multiple URLs	Listen across lifecycle events
Reusable	Modular (use in multiple apps)	No changes in existing logic
Clean Design	No need to duplicate logic	Great for monitoring and setup

---

❌ Drawbacks

Filters	Listeners
Can be complex if overused	Too many can make tracking hard
Slight performance overhead	Adds memory usage if misused

✅ Summary

Concept	Description
Filter	Pre/post processing for requests/responses
Listener	Responds to lifecycle events (app, session, request)
Example Use	Auth, Logging, Init config, Session tracking
Declared in	@WebFilter, @WebListener or web.xml

✅ Topic 5: JDBC (Java Database Connectivity) and Connection Pooling

---

🔷 What is JDBC?

JDBC (Java Database Connectivity) is an API in Java that allows you to **connect and interact with relational databases like MySQL, PostgreSQL, Oracle**, etc.

🔸 Why Use JDBC?

- To **connect Java applications to databases**
- To perform operations like:
  - Insert (INSERT)
  - Read (SELECT)
  - Update (UPDATE)
  - Delete (DELETE)

🔷 Steps in JDBC

Step	Code Example	Description
1	Class.forName("com.mysql.cj.jdbc.Driver");	Load the DB driver
2	Connection conn = DriverManager.getConnection(...)	Create DB connection
3	PreparedStatement pstmt = conn.prepareStatement(...)	Prepare SQL statement
4	ResultSet rs = pstmt.executeQuery(); / executeUpdate()	Run SQL
5	conn.close();	Close the connection

🔶 JDBC Example

java

CopyEdit

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

```
Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test",
"root", "pass");


PreparedStatement ps = con.prepareStatement("SELECT * FROM users WHERE id=?");
ps.setInt(1, 1);


ResultSet rs = ps.executeQuery();
while (rs.next()) {
   System.out.println(rs.getString("username"));
}


con.close();
```

🔷 What is Connection Pooling?

Connection Pooling means **reusing a pool of database connections** instead of opening and closing a new one every time.

🔸 Why Use Connection Pooling?

Reason	Benefit
Faster performance	Avoids repeated creation of connections
Scalable	Supports many users efficiently
Resource management	Controls number of active DB connections

🔶 Libraries for Connection Pooling

Library	Description
HikariCP	Fast, modern, default in Spring Boot
Apache DBCP	Popular and stable

Library	Description
C3P0	Another classic pooling library

🔸 **Spring Boot Example using HikariCP (Default)**

properties

CopyEdit

application.properties

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=pass

spring.datasource.hikari.maximum-pool-size=10

No need to configure manually; Spring Boot auto-configures it.

🔷 **Real-Life Example**

- 🔹 In a login system:
  1. User submits login form
  2. Java app uses JDBC to check user in DB
  3. With pooling, the same DB connection is reused
  4. App quickly responds without opening new connections every time

---

✅ **Benefits of JDBC**

Feature	Description
Simple API	Easy to connect Java and DB
Vendor independent	Works with most relational databases
PreparedStatement	Prevents SQL injection

✅ **Benefits of Connection Pooling**

Feature	Description
Performance	Reuses existing DB connections
Stability	Prevents overload from too many connections
Scalability	Suitable for large traffic applications

❌ Drawbacks

JDBC Limitation	Description
Too much boilerplate	Requires a lot of repetitive code
Manual connection mgmt	Forgetting to close can lead to memory leaks
No OOP-style mapping	You handle ResultSet manually

✅ Summary

Concept	Description
JDBC	Java API for database connectivity
Connection Pooling	Reuse of DB connections for better performance
Libraries	HikariCP (default in Spring), Apache DBCP

✅ Topic 6: DAO (Data Access Object) Design Pattern & JDBC Best Practices

---

🔷 What is DAO (Data Access Object)?

The **DAO pattern** separates **persistence logic (database operations)** from **business logic**.

In simple terms:

✅ DAO handles DB access
✅ Service/Controller handles business rules and user interactions

---

🔶 Why Use DAO?

Reason	Benefit
Separation of concerns	Clean and modular code
Easy maintenance	Changes in DB logic don't affect other layers
Reusability	Common DAO methods can be reused
Unit testing made easy	Logic is isolated and mockable

🔷 DAO Architecture (Layered Style)

css

CopyEdit

[Controller Layer] → [Service Layer] → [DAO Layer] → [DB]

🔶 Basic DAO Structure

Let's say we are working with a Student entity.

1. Student Model (Entity)

java

CopyEdit

public class Student {

```
    private int id;
    private String name;
    private int age;

    // Getters and Setters
}
```

---

2. StudentDAO Interface

```
java
CopyEdit
public interface StudentDAO {
    void save(Student student);
    Student getById(int id);
    List<Student> getAll();
    void update(Student student);
    void delete(int id);
}
```

---

3. StudentDAO Implementation

```
java
CopyEdit
public class StudentDAOImpl implements StudentDAO {
    private Connection conn;

    public StudentDAOImpl(Connection conn) {
        this.conn = conn;
    }
```

```
  public void save(Student student) {
    try {
      PreparedStatement ps = conn.prepareStatement("INSERT INTO students (name, age) VALUES (?, ?)");
      ps.setString(1, student.getName());
      ps.setInt(2, student.getAge());
      ps.executeUpdate();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public Student getById(int id) {
    Student student = null;
    try {
      PreparedStatement ps = conn.prepareStatement("SELECT * FROM students WHERE id=?");
      ps.setInt(1, id);
      ResultSet rs = ps.executeQuery();
      if (rs.next()) {
        student = new Student();
        student.setId(rs.getInt("id"));
        student.setName(rs.getString("name"));
        student.setAge(rs.getInt("age"));
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
    return student;
```

```
    }

    // Other methods (getAll, update, delete) would follow similar pattern
}
```

🔷 JDBC Best Practices

Best Practice	Explanation
Always close Connection, ResultSet, PreparedStatement	Prevents memory leaks
Use try-with-resources	Automatically closes resources
Use PreparedStatement instead of Statement	Avoids SQL injection
Keep connection logic in DAO only	Avoids clutter in business logic
Use connection pooling	Improves performance in production apps

---

✅ Example: Try-With-Resources

java

CopyEdit

```
try (Connection conn = DriverManager.getConnection(...);
     PreparedStatement ps = conn.prepareStatement("SELECT * FROM users")) {

    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
        System.out.println(rs.getString("name"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

🔷 Real-Life Example

Let's say you're building a **School Management System**:

- You can have:
    - StudentDAO
    - TeacherDAO
    - CourseDAO

Each one will handle their own DB operations without affecting controller or service logic.

✅ Benefits of DAO

Benefit	Description
Clean architecture	Follows good design principles
Reusable	DAO methods can be used across the app
Testable	Can be easily mocked during unit testing
Scalable	New features (e.g., logging, transaction mgmt) can be added easily

---

❌ Drawbacks

Drawback	Description
More code upfront	Needs interfaces and implementation
Complex for small apps	Might feel like overkill for simple apps

---

✅ Summary

Concept	Description
DAO Pattern	Manages database operations cleanly
Benefits	Reusability, separation of concerns, testability

Concept	Description
Best Practice	Use try-with-resources, PreparedStatement, pooling

✅ Topic 7: Introduction to ORM and Hibernate Basics

🔷 What is ORM?

ORM (Object Relational Mapping) is a technique that allows you to **map Java objects to database tables**.

In simpler terms:

It converts database **rows into Java objects** and **Java objects into rows**.

🔸 Why Use ORM?

Without ORM (using JDBC):

- You manually write SQL queries.
- You manually map each column to a field.
- Error-prone and time-consuming.

With ORM:

- Object mapping is automatic.
- You use Java code to interact with the DB.
- Easy to maintain and understand.

🔷 Hibernate: A Popular ORM Framework

Hibernate is the most widely used ORM framework in Java.

🔸 Benefits of Hibernate

Feature	Description
Automatic Mapping	Converts Java classes to DB tables
HQL Support	Hibernate Query Language (like SQL, but for objects)
Caching	Reduces DB hits by caching objects
Lazy Loading	Loads data only when needed
Transaction Mgmt	Handles transactions internally

🔷 How Hibernate Works (Simple Flow)

java

CopyEdit

Java Class → Hibernate → SQL Query → Database

When you save a Java object:

- Hibernate translates it into SQL.
- Executes it in the DB.
- Automatically maps the result back into Java.

🔶 Key Components of Hibernate

Component	Purpose
SessionFactory	Creates sessions (like a connection factory)
Session	Interface between app and DB (like a DB connection)
Transaction	Manages DB transactions
Query	HQL/SQL query executor
Configuration	Loads DB properties and mappings

🔷 Sample Entity Class (Java → Table)

java

CopyEdit

```
@Entity
@Table(name = "students")
public class Student {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int id;

   @Column(name = "student_name")
   private String name;

   private int age;

   // Getters and setters
}
```

🔷 **Hibernate Configuration (hibernate.cfg.xml)**

xml

CopyEdit

```
<hibernate-configuration>
 <session-factory>
   <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
   <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mydb</property>
   <property name="hibernate.connection.username">root</property>
   <property name="hibernate.connection.password">pass</property>
```

```
  <property
name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
  <property name="show_sql">true</property>
  <property name="hbm2ddl.auto">update</property>

  <mapping class="com.example.Student"/>
 </session-factory>
</hibernate-configuration>
```

🔷 Saving an Object

java

CopyEdit

```
SessionFactory factory = new Configuration().configure().buildSessionFactory();
Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Student s = new Student();
s.setName("Viraj");
s.setAge(22);

session.save(s);
tx.commit();

session.close();
factory.close();
```

✅ Real-Life Example

Imagine you're building an **Online Bookstore**:

- You create a Book class.
- Hibernate maps it to the books table.
- You save a new book using session.save(book) instead of writing SQL.

✅ Benefits of Hibernate

Benefit	Description
Saves Development Time	No need to write SQL manually
Cleaner Code	Works with Java objects
Portable	Can work with many databases
Built-in Caching	Faster performance with less DB hits

❌ Drawbacks

Drawback	Description
Learning Curve	More difficult than plain JDBC for beginners
Debugging Queries	Errors are sometimes hidden behind abstraction
Overhead	Not ideal for very small/simple apps

✅ Summary

Concept	Description
ORM	Maps Java objects to database tables
Hibernate	A powerful ORM framework for Java
Components	SessionFactory, Session, Transaction, etc.
Advantage	Saves time, clean code, automatic mapping

✅ Topic 8: Hibernate Annotations & HQL (Hibernate Query Language)

🔷 What are Hibernate Annotations?

Hibernate annotations are used to define **mapping metadata directly in Java classes**, instead of XML configuration.

Annotations make the code cleaner and easier to maintain.

🔶 Common Hibernate Annotations

Annotation	Description
@Entity	Marks the class as a persistent entity
@Table(name = "")	Maps the class to a table
@Id	Marks the primary key
@GeneratedValue	Auto-generates ID values
@Column(name = "")	Maps a field to a table column
@OneToOne, @OneToMany, @ManyToOne, @ManyToMany	Used for relationships

---

✅ Example: Annotated Entity

java

CopyEdit

```
@Entity
@Table(name = "students")
public class Student {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int id;
```

```
    @Column(name = "student_name", nullable = false)
    private String name;

    private int age;

    // Getters and setters
}
```

🔷 Hibernate Query Language (HQL)

HQL is an object-oriented query language similar to SQL, but it works with Java **objects** and **entities**, not table names.

✅ SQL → works with tables
✅ HQL → works with Java classes

🔶 Why Use HQL?

Reason	Benefit
Database independent	You don't need to worry about DB-specific SQL
Cleaner and readable queries	Query Java objects, not tables
Object-oriented	Works with class names and field names

✅ Basic HQL Syntax

java

CopyEdit

```
String hql = "from Student";
Query query = session.createQuery(hql);
List<Student> list = query.list();
```

🔷 HQL Query Types

Type	Example
Select	from Student where age > 20
Insert	Not supported in HQL (use native SQL)
Update	update Student set name = 'John' where id = 1
Delete	delete from Student where id = 5

🔶 Parameterized Queries in HQL

To avoid SQL injection and reuse queries:

java

CopyEdit

Query query = session.createQuery("from Student where name = :name");

query.setParameter("name", "Viraj");

List<Student> list = query.list();

🔷 Real-Life Example

In a **Library Management System**:

- Instead of writing SELECT * FROM books WHERE author='John', you can write:

java

CopyEdit

Query q = session.createQuery("from Book where author = :author");

q.setParameter("author", "John");

This makes the code **clean**, **object-oriented**, and **DB-independent**.

✅ Benefits of HQL & Annotations

Benefit	Description
Easy to use	No separate XML files required
Object-oriented queries	Queries written using class names
More readable	Reduces boilerplate code
Portable and flexible	Independent of database

❌ Drawbacks

Drawback	Description
Limited SQL Support	Cannot use complex joins like native SQL
Insert not supported	Need native SQL for insert
Not always performant	Abstracted layer may reduce fine-grained control

---

✅ Summary

Concept	Description
Hibernate Annotations	Replace XML configuration with Java annotations
HQL	Object-oriented query language for Hibernate
Benefits	Simpler, cleaner, reusable, DB-independent

✅ Topic 9: Relationships in Hibernate (OneToOne, OneToMany, ManyToOne, ManyToMany)

🔷 Why Relationships?

In real-world applications, entities are **related to each other**. For example:

- A Student has one Address
- A Customer has many Orders
- Many Students can enroll in many Courses

Hibernate makes it easy to map such relationships using annotations.

🔶 1. @OneToOne Relationship

🔸 Meaning:

One entity is related to **exactly one** entity.

Example: One Person has one Passport.

✅ Java Example

java

CopyEdit

```
@Entity
public class Person {
   @Id
   private int id;
   private String name;

   @OneToOne
   private Passport passport;
}
```

java

CopyEdit

```
@Entity
public class Passport {
   @Id
   private int id;
   private String number;
}
```

🔶 2. @OneToMany Relationship

🔸 **Meaning:**

One entity is related to **many** other entities.

Example: One Customer has many Orders.

✅ **Java Example**

java

CopyEdit

```
@Entity
public class Customer {
   @Id
   private int id;
   private String name;

   @OneToMany
   private List<Order> orders;
}
```

java

CopyEdit

```
@Entity
public class Order {
   @Id
```

```
    private int id;
    private String product;
}
```

🔶 3. @ManyToOne Relationship

🔸 Meaning:

Many entities relate to **one** entity.

Example: Many Employees work in one Department.

✅ Java Example

java

CopyEdit

```
@Entity
public class Employee {
    @Id
    private int id;
    private String name;

    @ManyToOne
    private Department department;
}
```

java

CopyEdit

```
@Entity
public class Department {
    @Id
    private int id;
    private String deptName;
}
```

🔶 **4. @ManyToMany Relationship**

🔸 **Meaning:**

Many entities relate to **many** other entities.

Example: Many Students can enroll in many Courses.

✅ **Java Example**

java

CopyEdit

```
@Entity
public class Student {
   @Id
   private int id;
   private String name;

   @ManyToMany
   private List<Course> courses;
}
```

java

CopyEdit

```
@Entity
public class Course {
   @Id
   private int id;
   private String courseName;
}
```

🔷 **Relationship Annotations Explained**

Annotation	Used For	Example
@OneToOne	1 entity → 1 other	Person → Passport
@OneToMany	1 entity → Many others	Customer → Orders
@ManyToOne	Many entities → 1	Employees → Department
@ManyToMany	Many entities → Many entities	Students ↔ Courses

🔷 Optional Annotations

- @JoinColumn: Defines the foreign key column.
- mappedBy: Indicates the owning side of the relationship.
- cascade: Helps in cascading operations like persist, delete, etc.
- fetch: Controls when the data is loaded (LAZY, EAGER).

✅ Real-Life Use Case Example

A **College Management System**:

- One Professor teaches many Courses.
- Many Students enroll in many Courses.
- Each Course is assigned to one Department.

These relationships help model real-world rules in code using Hibernate.

✅ Benefits of Relationship Mapping

Benefit	Description
No manual foreign keys	Automatically handled by Hibernate
Maintains DB consistency	Supports cascading and transactional ops
Clean and readable code	Easy-to-understand object navigation

❌ Drawbacks

Drawback	Description
Complexity	Relationships can get complicated to manage
Lazy loading confusion	Improper fetching strategy can hurt performance
Cascading risks	Careless cascades can delete related data

✅ Summary

Concept	Description
OneToOne	One entity is linked to one
OneToMany	One entity is linked to many
ManyToOne	Many entities link to one
ManyToMany	Many entities link to many

✅ Topic 10: Hibernate Caching & Performance Tuning

🔷 Why Caching in Hibernate?

When we retrieve data from the database repeatedly, it causes **performance issues**. Caching helps store frequently accessed data in memory to **reduce database calls** and improve performance.

🔶 Types of Hibernate Caching

Hibernate provides **two levels of caching**:

Cache Level	Description	Scope
1st Level	Enabled by default (Session-level cache)	Per Hibernate Session
2nd Level	Optional (SessionFactory-level cache)	Shared across sessions

---

✅ 1. First-Level Cache (Session-Level)

- Hibernate **automatically caches** objects in the current session.
- If you request the same entity again, it is **fetched from memory**, not DB.

🔸 **Example:**

java

CopyEdit

```
Session session = sessionFactory.openSession();

Student s1 = session.get(Student.class, 1); // From DB

Student s2 = session.get(Student.class, 1); // From Cache
```

🟢 Benefit: Avoids repeated DB calls
❌ Limitation: Works only within the **same session**

✅ 2. Second-Level Cache (Across Sessions)

- Must be **explicitly enabled**.
- Stores entities across sessions using **providers** like:

- EHCache
- Infinispan
- Redis
- OSCache

🔸 **Configuration Example:**

xml

CopyEdit

<property name="hibernate.cache.use_second_level_cache">true</property>

<property name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>

🔸 **Annotate Entity:**

java

CopyEdit

@Entity

@Cacheable

@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)

public class Student {

  ...

}

🔶 **Query Cache (Optional)**

Hibernate also supports **caching the result of queries**.

java

CopyEdit

Query query = session.createQuery("from Student");

query.setCacheable(true);

✅ Requires 2nd level cache to be enabled.

🔷 **Hibernate Performance Tuning Techniques**

Technique	Description
Enable caching	Reduces DB round-trips
Use batch-size	Optimizes fetching collections
Use fetch strategy wisely	Choose between LAZY and EAGER fetching
Use pagination	Avoid loading large datasets at once (setFirstResult, setMaxResults)
Use projections (select)	Fetch only needed fields instead of full objects
Avoid N+1 problem	Use JOIN FETCH or batch fetching

✅ **Example of Lazy vs Eager Fetching**

java

CopyEdit

@OneToMany(fetch = FetchType.LAZY)

private List<Order> orders;

java

CopyEdit

@OneToMany(fetch = FetchType.EAGER)

private List<Order> orders;

Use LAZY loading unless you absolutely need immediate fetching.

🔷 **Real-Life Example**

In an **e-commerce system**, product details and categories rarely change, but are read frequently.

✅ So we enable **2nd level caching** for them using **EHCache**.

✅ Benefits of Caching

Benefit	Description
Reduced DB load	Fewer queries go to the database
Faster data retrieval	Data served from memory
Better scalability	Handles more users efficiently

❌ Drawbacks of Caching

Drawback	Description
Stale data risk	Data in cache might not reflect latest DB changes
Extra configuration	Needs setup for 2nd level and query caches
More memory usage	Stores data in memory, which increases RAM usage

---

✅ Summary

Feature	Description
1st Level Cache	Default cache per session
2nd Level Cache	Shared cache across sessions (needs config)
Query Cache	Caches HQL query results
Tuning Tips	Batch fetching, pagination, lazy loading

✅ Topic 11: JPQL vs HQL vs SQL

(Java Persistence Query Language, Hibernate Query Language, Structured Query Language)

🔷 Why Different Query Languages?

Hibernate and JPA provide custom query languages to:

- Work with **objects/entities**, not just database tables.
- Make queries database-independent.
- Allow ORM (Object-Relational Mapping) features like polymorphism and inheritance in queries.

🔶 1. What is SQL?

- **SQL (Structured Query Language)** is the **standard** query language for databases.
- Works directly with **tables and columns**.
- Completely **database-dependent**.

🔸 SQL Example:

sql

CopyEdit

```
SELECT * FROM student;
```

🔶 2. What is HQL?

- **HQL (Hibernate Query Language)** is a query language provided by **Hibernate**.
- Works with **Java objects/entities**, not DB tables.
- Case-sensitive (especially entity names).

🔸 HQL Example:

java

CopyEdit

Query q = session.createQuery("FROM Student WHERE name = 'John'");

✅ It uses **class names and property names**, not table or column names.

🔶 3. What is JPQL?

- **JPQL (Java Persistence Query Language)** is defined by **JPA (Java Persistence API)**.
- It's almost the **same as HQL**, but **standardized** and **vendor-neutral**.
- Works with JPA entities and annotations (@Entity).

🔸 **JPQL Example:**

java

CopyEdit

TypedQuery<Student> q = entityManager.createQuery("SELECT s FROM Student s WHERE s.name = 'John'", Student.class);

✅ It also uses **entity names** and **Java field names**.

🔷 Comparison Table

Feature	SQL	HQL	JPQL
Based On	Tables & Columns	Entities & Fields	Entities & Fields
Provided By	Database	Hibernate	JPA
Portability	DB-specific	Portable with Hibernate	Portable across JPA providers
Syntax	SELECT * FROM table	FROM Entity	FROM Entity
Supports ORM	❌ No	✅ Yes	✅ Yes

✅ Real-Life Analogy

Imagine working in a company:

- **SQL**: You directly contact the **HR database** and pull employee records.
- **HQL/JPQL**: You talk to the **HR system** (Java app) and ask for **Employee objects** instead.

✅ Benefits of HQL / JPQL over SQL

Benefit	Description
Object-oriented	Works with entities instead of tables
Auto-mapping	Hibernate handles mapping of objects to DB
Portable	Works across databases
Inheritance/polymorphism	Can query base and subclass entities

---

❌ Drawbacks

Drawback	Description
Not as powerful as SQL	Complex native queries might need SQL
Case sensitivity	Entity names and fields are case-sensitive
Learning curve	Slightly different syntax from SQL

✅ Summary

Language	Purpose	Usage Scope
SQL	Low-level database query language	Native DB access
HQL	Hibernate object query language	Hibernate-only apps
JPQL	Standard JPA query language	All JPA apps

🔹 1. Filters in Servlet

➤ What is it?
A Filter is used to **intercept requests/responses** before they reach servlets or after they leave servlets.

➤ Why we use it?

- Logging
- Authentication check
- Compression
- Input validation

➤ How it works?

- Defined in web.xml or using @WebFilter
- It implements javax.servlet.Filter and overrides doFilter()

java

CopyEdit

```
@WebFilter("/secure/*")
public class AuthFilter implements Filter {
  public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
    throws IOException, ServletException {
    // pre-processing
    chain.doFilter(req, res); // pass to next
    // post-processing
  }
}
```

➤ Real-life example:
Before accessing a secure page, check if the user is logged in.

🔹 2. Listeners

➤ **What is it?**
Listeners in servlet track **events like app start/stop, session creation, or attribute changes**.

➤ **Types:**

- ServletContextListener – App-level lifecycle
- HttpSessionListener – Session-level lifecycle
- ServletRequestListener – Request-level lifecycle

java

CopyEdit

```
@WebListener
public class MyAppListener implements ServletContextListener {
  public void contextInitialized(ServletContextEvent e) {
    System.out.println("App Started");
  }
}
```

➤ **Real-life example:**
Logging the number of active sessions in a web app.

🔹 **3. Cookie vs Session**

Feature	Cookie	Session
Stored In	Browser (client-side)	Server-side
Size Limit	4KB	No specific limit
Security	Less secure	More secure
Lifetime	Can be set manually	Till session timeout
Use	Track users across pages	Store user data per session

Interview Tip:
👉 Use Session for sensitive data. Use Cookies for preferences.

🔹 4. HttpSession API

➤ Used to:
Store user data across multiple requests.

➤ Key Methods:

java

CopyEdit

HttpSession session = request.getSession();

session.setAttribute("username", "Viraj");

String name = (String) session.getAttribute("username");

session.invalidate(); // destroys session

➤ Real-life use:
Shopping cart, login session, etc.

🔹 5. Hidden Form Fields

➤ What is it?
A way to **send data** from one form/page to another **without displaying it to the user**.

html

CopyEdit

<input type="hidden" name="userId" value="12345"/>

➤ Used when:
You don't want to use session but still send data between pages.

🔹 6. URL Rewriting

➤ What is it?
Passing information via the URL when cookies/sessions are not used.

html

CopyEdit

Next

➤ **Drawback:**
Visible to users. Not secure for sensitive info.

✅ **Summary of Remaining Topics**

Topic	Use
Filter	Intercept & modify requests
Listener	Track lifecycle events
Cookie vs Session	User data storage options
HttpSession	Manage user sessions securely
Hidden Fields	Pass data invisibly via form
URL Rewriting	Pass data in URL