ADVANCED JAVA – COMPREHENSIVE GUIDE

UNIT I: Java Database Connectivity (JDBC)

1.1 Introduction to JDBC

JDBC is Java's API for connecting and executing queries on databases. It provides a standard interface independent of database vendors. Imagine a universal translator allowing Java to communicate with MySQL, Oracle, PostgreSQL, etc., using the same methods.

Real-time Example: A banking application uses JDBC to connect to a database storing customer accounts, transactions, and balances.

1.2 JDBC Architecture

Two-tier architecture (Java App → JDBC → Database) and three-tier (Client → Middleware/JDBC → Database). JDBC sits between Java application and DBMS, converting Java calls to DB-understandable statements.

1.3 JDBC Drivers

Type 1: JDBC-ODBC Bridge - Uses ODBC driver. Requires ODBC configuration. Deprecated. *Example*: Legacy Windows applications connecting to Excel/Access.

Type 2: Native API Driver - Uses client-side libraries. Faster but platform-dependent. *Example*: Oracle OCI driver requiring Oracle client installation.

Type 3: Network Protocol Driver - Pure Java driver translating JDBC to middleware protocol. *Example*: Application server pooling connections for multiple clients.

Type 4: Thin Driver - Direct Java-to-DB protocol (most common). Platform independent. *Example*: MySQL Connector/J connecting directly to MySQL on port 3306.

1.4 JDBC API Components

DriverManager: Manages driver list and connections.

```
Class.forName("com.mysql.cj.jdbc.Driver");
Connection con = DriverManager.getConnection(url, user, pass);
```

Connection: Session with database. Can create statements and manage transactions.

Statement: Executes static SQL queries.

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

PreparedStatement: Precompiled SQL with parameters. Prevents SQL injection.

```
PreparedStatement pstmt = con.prepareStatement(
    "INSERT INTO users VALUES(?, ?)");
pstmt.setString(1, "John");
pstmt.setInt(2, 30);
pstmt.executeUpdate();
```

CallableStatement: Executes stored procedures.

```
CallableStatement cstmt = con.prepareCall("{call getEmployee(?, ?)}");
cstmt.setInt(1, 101);
cstmt.registerOutParameter(2, Types.VARCHAR);
```

ResultSet: Tabular data returned from queries. Scrollable and updatable in newer versions.

```
while(rs.next()) {
    System.out.println(rs.getString("name"));
}
```

1.5 Steps to Connect Java Application with Database

1. Load driver: `Class.forName("driver class")`
2. Establish connection: `DriverManager.getConnection()`
3. Create statement
4. Execute query
5. Process ResultSet
6. Close resources (in finally block or try-with-resources)

1.6 Executing SQL Queries using JDBC

```
// DDL Example
stmt.execute("CREATE TABLE products(id INT, name VARCHAR(50))");

// DML Example
int rows = stmt.executeUpdate(
    "UPDATE accounts SET balance=5000 WHERE acc_no='ACC123'");

// DQL Example
ResultSet rs = stmt.executeQuery(
    "SELECT * FROM orders WHERE date > '2024-01-01'");
```

1.7 CRUD Operations using JDBC

```
// CREATE
pstmt = con.prepareStatement("INSERT INTO employees VALUES (?, ?)");
pstmt.setInt(1, 101);
pstmt.setString(2, "Alice");

// READ
rs = stmt.executeQuery("SELECT * FROM employees WHERE id=101");

// UPDATE
pstmt = con.prepareStatement("UPDATE employees SET salary=? WHERE
id=?");
pstmt.setDouble(1, 75000);
pstmt.setInt(2, 101);
```

```
// DELETE
pstmt = con.prepareStatement("DELETE FROM employees WHERE id=?");
pstmt.setInt(1, 101);
```

1.8 Transaction Management

Auto-commit: Default true. Each statement commits immediately.

```
con.setAutoCommit(false); // Start transaction
try {
    stmt.executeUpdate("UPDATE account SET balance=balance-1000 WHERE
id=1");
    stmt.executeUpdate("UPDATE account SET balance=balance+1000 WHERE
id=2");
    con.commit(); // Both succeed
} catch(Exception e) {
    con.rollback(); // Revert both if any fails
}
```

1.9 Batch Processing

Execute multiple queries in single DB call.

```
pstmt = con.prepareStatement("INSERT INTO logs VALUES (?, ?)");
pstmt.setString(1, "Error1");
pstmt.setTimestamp(2, new Timestamp(System.currentTimeMillis()));
pstmt.addBatch();

pstmt.setString(1, "Error2");
pstmt.addBatch();

int[] counts = pstmt.executeBatch(); // Executes all at once
```

1.10 Exception Handling in JDBC

```
try (Connection con = DriverManager.getConnection(url, user, pass);
     Statement stmt = con.createStatement()) {
```

```
    // JDBC operations
} catch (SQLException e) {
    System.out.println("SQL State: " + e.getSQLState());
    System.out.println("Error Code: " + e.getErrorCode());
    System.out.println("Message: " + e.getMessage());
}
```

`SQLException` provides database-specific error information.

UNIT II: Servlets

2.1 Introduction to Web Applications

Dynamic applications accessed via browsers. Servlet is Java's server-side technology for web applications.

Real-time Example: Amazon.com - servlets handle product search, cart management, checkout.

2.2 Client-Server Architecture

Browser (client) sends HTTP request → Web Server → Servlet Container → Servlet → Response

2.3 What is a Servlet?

Java class extending server capabilities. Runs inside servlet container (Tomcat, Jetty).

2.4 Servlet Architecture

```
HttpServletRequest → Servlet → HttpServletResponse
        ↑                           ↑
    Client Browser             Response to Browser
```

2.5 Servlet Life Cycle

1. **Loading and Instantiation**: Container loads servlet class, creates instance.
2. **Initialization**: `init()` method called once.
3. **Request Handling**: `service()` method called for each request.
4. **Destruction**: `destroy()` called before removing from container.

```
public class MyServlet extends HttpServlet {
    public void init() { // One-time setup
        log("Servlet initialized");
    }

    protected void doGet(HttpServletRequest req, HttpServletResponse
res) {
        // Handle GET request
    }

    public void destroy() { // Cleanup
        log("Servlet destroyed");
    }
}
```

2.6 Types of Servlets

- **GenericServlet**: Protocol independent
- **HttpServlet**: HTTP specific (most common)

2.7 Servlet API

HttpServlet: Provides `doGet()`, `doPost()`, `doPut()`, `doDelete()` methods. **ServletRequest**: Contains client request data. **ServletResponse**: Used to send response to client.

2.8 Handling HTTP Requests

```
protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
    String username = req.getParameter("user"); // Get query parameter
    PrintWriter out = res.getWriter();
    out.println("<h1>Welcome " + username + "</h1>");
}

protected void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
    // Handle form submission
}
```

2.9 GET vs POST

GET: Parameters in URL, bookmarkable, limited length. `/login?user=john&pass=1234`

POST: Parameters in body, secure for sensitive data, no length limit.

```
<form method="post">
    <input name="creditcard" type="text">
</form>
```

2.10 Session Tracking Techniques

Cookies: Store data on client side.

```
Cookie ck = new Cookie("user", "John");
ck.setMaxAge(60*60); // 1 hour
response.addCookie(ck);

// Retrieving
Cookie[] cookies = request.getCookies();
```

HttpSession: Store data on server side.

```
HttpSession session = request.getSession();
session.setAttribute("cartItems", itemsList);
// Retrieval
List items = (List) session.getAttribute("cartItems");
```

URL Rewriting: Append session ID to URLs.

```
String url = response.encodeURL("products.jsp");
// Becomes: products.jsp;jsessionid=1234
```

Hidden Form Fields:

```
<input type="hidden" name="sessionid" value="1234">
```

2.11 Deployment Descriptor (web.xml)

XML file configuring servlets, mappings, parameters.

```
<web-app>
    <servlet>
        <servlet-name>LoginServlet</servlet-name>
        <servlet-class>com.example.LoginServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>LoginServlet</servlet-name>
        <url-pattern>/login</url-pattern>
    </servlet-mapping>
</web-app>
```

2.12 Servlet Annotations

Java EE 5+ alternative to web.xml.

```
@WebServlet(
    name = "LoginServlet",
    urlPatterns = {"/login", "/doLogin"},
    initParams = {
        @WebInitParam(name = "timeout", value = "30")
    }
)
public class LoginServlet extends HttpServlet {
    // Servlet code
}
```

UNIT III: JavaServer Pages (JSP)

3.1 Introduction to JSP

Servlet extension simplifying dynamic content creation. JSP = HTML + Java code.

Real-time Example: E-commerce product display page where HTML structure is fixed but product data is dynamic.

3.2 JSP Architecture

JSP file → JSP Container translates to Servlet → Servlet compiled → Executed

3.3 Life Cycle of JSP

1. Translation (JSP to Servlet)
2. Compilation (Servlet to bytecode)
3. Class loading
4. Instantiation
5. Initialization (`jspInit()`)
6. Request processing (`_jspService()`)
7. Destruction (`jspDestroy()`)

3.4 JSP Scripting Elements

Scriptlet: `<% Java code %>`

```
<%
    String name = request.getParameter("name");
    out.println("Hello " + name);
%>
```

Declaration: `<%! method or variable %>`

```
<%!
    private int counter = 0;
    public void increment() { counter++; }
%>
```

Expression: `<%= expression %>` (outputs value)

```
<p>Current time: <%= new java.util.Date() %></p>
```

3.5 JSP Directives

page: Defines page attributes.

```
<%@ page language="java" contentType="text/html"
         import="java.util.*, java.sql.*" errorPage="error.jsp" %>
```

include: Includes file during translation phase.

```
<%@ include file="header.jsp" %>
```

taglib: Declares custom tag library.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

3.6 JSP Implicit Objects

Predefined variables available in JSP:

- **request**: `HttpServletRequest`
- **response**: `HttpServletResponse`
- **session**: `HttpSession`
- **application**: `ServletContext`
- **out**: `JspWriter`
- **config**: `ServletConfig`
- **pageContext**: Access to all objects
- **page**: `this` reference
- **exception**: `Throwable` (error pages only)

```
<%
    String user = request.getParameter("user");
    session.setAttribute("loggedUser", user);
    application.setAttribute("visitorCount", 1000);
    out.println("Welcome " + user);
%>
```

3.7 JSP Actions

jsp:include: Includes at runtime.

```
<jsp:include page="navigation.jsp" />
```

jsp:forward: Forwards to another page.

```
<jsp:forward page="welcome.jsp">
    <jsp:param name="message" value="Login successful" />
</jsp:forward>
```

jsp:useBean: Creates/accesses JavaBean.

```
<jsp:useBean id="user" class="com.example.UserBean" scope="session">
    <jsp:setProperty name="user" property="*" />
</jsp:useBean>
```

3.8 JavaBeans

Reusable Java classes following conventions:

- No-arg constructor
- Private properties
- Getter/setter methods

```
public class UserBean {
    private String name;
    private String email;
    public UserBean() {}

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    // ... other getters/setters
}
```

3.9 Expression Language (EL)

Simplifies JSP output: `${expression}`

```
<p>Welcome ${sessionScope.user.name}</p>
<p>Total: ${cart.total * 1.18}</p> <!-- VAT included -->
<p>Empty cart: ${empty cart.items}</p>
```

3.10 JSTL (JSP Standard Tag Library)

Tag libraries for common tasks:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<c:if test="${user.role == 'admin'}">
    <p>Admin panel</p>
</c:if>

<c:forEach var="product" items="${products}">
    <tr>
        <td>${product.name}</td>
        <td><fmt:formatNumber value="${product.price}"
type="currency"/></td>
    </tr>
</c:forEach>
```

UNIT IV: Enterprise JavaBeans (EJB) & Web Services

4.1 Introduction to Enterprise Applications

Large-scale, distributed, transactional, secure applications. *Example*: Banking system with thousands of concurrent users.

4.2 EJB Architecture

Server-side component architecture for modular enterprise apps.

4.3 Types of EJB

Session Beans:

- **Stateless**: No client state between calls. Pooled instances.

```
@Stateless
public class PaymentService {
  public boolean processPayment(Payment p) {
      // Process payment
      return true;
  }
}
```

- **Stateful**: Maintains client conversation state. ```java @Stateful public class ShoppingCart { private List items = new ArrayList<>();

public void addItem(Item item) { items.add(item); }

}

~~~~
**Message Driven Beans**: Processes JMS messages asynchronously.
```java
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue")
~~~~

```
})
public class OrderProcessor implements MessageListener {
    public void onMessage(Message msg) {
        // Process order message
    }
}
```

4.4 Life Cycle of EJB

Stateless: Does not exist → Ready pool → Destroyed **Stateful**: Does not exist → Ready → Passive (passivated) → Destroyed **Message Driven**: Does not exist → Ready pool → Destroyed

4.5 Advantages of EJB

- Transaction management
- Security
- Concurrency handling
- Resource pooling
- Distributed computing support

4.6 Introduction to Web Services

Software system for interoperable machine-to-machine communication.

4.7 Types of Web Services

SOAP: XML-based protocol with WSDL contract.

```
<soap:Envelope>
    <soap:Body>
        <getProduct>
            <productId>123</productId>
        </getProduct>
    </soap:Body>
</soap:Envelope>
```

REST: Architectural style using HTTP methods.

```
GET /products/123
POST /orders
PUT /products/123
DELETE /products/123
```

4.8 RESTful Web Services

```
@Path("/products")
public class ProductService {

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Product getProduct(@PathParam("id") int id) {
        return productDAO.find(id);
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createProduct(Product product) {
        productDAO.save(product);
        return Response.status(201).build();
    }
}
```

4.9 JSON and XML

JSON: Lightweight, JavaScript compatible.

```
{
    "id": 101,
    "name": "Laptop",
    "price": 999.99,
    "inStock": true
}
```

XML: Verbose, schema validation.

```
<product>
    <id>101</id>
    <name>Laptop</name>
    <price>999.99</price>
    <inStock>true</inStock>
</product>
```

4.10 Web Service Security Basics

- HTTPS/SSL encryption
- Authentication tokens
- OAuth 2.0
- API keys
- Rate limiting

UNIT V: Advanced Java Frameworks & Security

5.1 Introduction to Java Frameworks

Pre-built structures solving common problems.

5.2 MVC Architecture

Model: Business logic and data **View**: Presentation layer **Controller**: Handles user input

```
User → Controller → Model → View → User
```

5.3 Introduction to Spring Framework

Lightweight, comprehensive framework for enterprise Java.

5.4 Spring Core

Dependency Injection: Objects given dependencies rather than creating them.

```
@Component
public class OrderService {
    @Autowired
    private PaymentService paymentService;
    // paymentService injected by Spring
}
```

5.5 Spring MVC Overview

```
@Controller
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    private OrderService orderService;

    @GetMapping("/{id}")
    public String getOrder(@PathVariable int id, Model model) {
        Order order = orderService.getOrder(id);
        model.addAttribute("order", order);
        return "orderDetails"; // view name
    }
}
```

5.6 Hibernate ORM

Object-Relational Mapping framework.

ORM Concepts: Maps Java objects to database tables.

```
@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

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

    @OneToMany(mappedBy = "employee")
    private List<Address> addresses;
}
```

5.7 Hibernate Architecture

```
Java App → Hibernate → JDBC → Database
         (SessionFactory, Session, Transaction)
```

5.8 HQL (Hibernate Query Language)

Object-oriented SQL.

```
Session session = sessionFactory.openSession();
Query<Employee> query = session.createQuery(
    "FROM Employee WHERE department = :dept", Employee.class);
query.setParameter("dept", "Sales");
List<Employee> employees = query.list();
```

5.9 Java Web Application Security

Authentication: Verifying user identity.

```
public boolean authenticate(String username, String password) {
    User user = userDAO.findByUsername(username);
    return passwordEncoder.matches(password, user.getPassword());
}
```

Authorization: Checking access rights.

```
@PreAuthorize("hasRole('ADMIN')")
public void deleteUser(int userId) {
    // Only admins can execute
}
```

5.10 Filters and Listeners

Filter: Intercepts requests/responses.

```
@WebFilter("/*")
public class LoggingFilter implements Filter {
    public void doFilter(ServletRequest req, ServletResponse res,
                         FilterChain chain) {
        System.out.println("Request URI: " +
            ((HttpServletRequest)req).getRequestURI());
        chain.doFilter(req, res); // Continue to next filter/servlet
    }
}
```

Listener: Reacts to events.

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

5.11 Logging (Log4j / SLF4J)

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class OrderService {
    private static final Logger logger =
LoggerFactory.getLogger(OrderService.class);

    public void processOrder(Order order) {
        logger.info("Processing order {}", order.getId());
        try {
            // Process order
        } catch (Exception e) {
            logger.error("Failed to process order {}", order.getId(), 
e);
        }
    }
}
```

5.12 Performance Optimization

- Connection pooling
- Caching (Ehcache, Redis)
- Lazy loading
- Query optimization
- CDN for static resources

5.13 Advanced Java Application Deployment

Containerization:

```
FROM tomcat:9-jdk11
COPY target/myapp.war /usr/local/tomcat/webapps/
EXPOSE 8080
CMD ["catalina.sh", "run"]
```

Cloud Deployment: AWS Elastic Beanstalk, Azure App Service, Google App Engine.

Microservices Architecture: Breaking monolith into independently deployable services.

This comprehensive guide covers all topics in your index with practical examples. Each section builds upon previous knowledge, creating a complete Advanced Java learning path suitable for enterprise application development.