Unit_3_And_Unit_4_Advanced_Java

UNIT III: JavaServer Pages (JSP) - Broken Down

Part 1: What is JSP and Why Use It?

- **What is JSP?**
  - JSP stands for **JavaServer Pages**.
  - Think of it as an **HTML page that can have Java code inside**.
  - It's used to create **dynamic web pages** (pages that change content).
- **Why Not Just Use HTML?**
  - Plain HTML (`.html`) is static – it always looks the same.
  - JSP lets you show different content based on user input, time of day, database info, etc. (e.g., "Hello, [User Name]!").
- **Why Not Just Use Servlets?**
  - Servlets are Java classes that generate HTML using `out.println("<html>...")`.
  - This gets very messy and hard to read/design.
  - **JSP is easier for designing the page layout** because it looks mostly like HTML. Java code is added only where needed.
- **Key Idea:** JSP separates the **presentation** (how the page looks - HTML) from the **logic** (what data to show - Java).

Part 2: How JSP Works (The Magic Behind the Scenes)

1. **You write:** A file ending in `.jsp` (looks like HTML with special Java tags).
2. **User asks for the page:** Someone visits `yourpage.jsp` in their browser.
3. **Server Translates (First time only):** The web server (like Tomcat) turns your `.jsp` file into a regular Java Servlet file (`.java`). All the HTML becomes `out.println()` statements in Java.
4. **Server Compiles (First time only):** The server compiles the generated `.java` file into a `.class` file (Java bytecode).
5. **Server Executes:** The server runs the compiled Servlet (`.class` file). This Java code generates the final HTML output.
6. **Server Sends HTML:** The server sends the *pure HTML* result back to the user's browser. The browser never sees your original JSP code.
7. **Faster Next Time:** If the `.jsp` file hasn't changed, the server skips steps 3 & 4 on future requests and just runs the existing `.class` file (step 5).

Part 3: JSP Life Cycle (Birth, Life, Death of a JSP)

- This is similar to the Servlet life cycle, plus the translation steps:
  1. **Translation:** `.jsp` to `.java` (only when needed).
  2. **Compilation:** `.java` to `.class` (only when needed).
  3. **Loading:** Getting the `.class` file ready.
  4. **Instantiation:** Creating an object (instance) of the JSP's Servlet.
  5. **Initialization (`jspInit()`):** A special method called **once** when the JSP is first loaded. Good for setup tasks (like loading data). You can write your own `jspInit` using `<%! ... %>`.
  6. **Request Processing (`_jspService()`):** This method is called **for every user request**. This is where all the HTML and Java code inside `<% ... %>` runs to create the response. You don't write this method directly; the server generates it from your JSP content.
  7. **Destruction (`jspDestroy()`):** A special method called **once** when the server shuts down or unloads the JSP. Good for cleanup tasks (like releasing resources). You can write your own `jspDestroy` using `<%! ... %>`.

Part 4: JSP Scripting Elements (Putting Java in JSP)

- **WARNING:** Using lots of Java code directly in JSP is **old style** and makes pages hard to read. Modern ways (EL, JSTL - explained later) are much better! But you need to know these basics:
- **1. Scriptlets:** `<% ... Java code ... %>`
  - Use this to put blocks of regular Java code (statements, loops, if/else).
  - Example: `<% String name = "Guest"; if (user != null) { name = user.getName(); } %>`
- **2. Expressions:** `<%= ... Java expression ... %>`
  - Use this to **print** the result of a single Java expression directly into the HTML.
  - The expression's result is turned into a String.
  - **NO semicolon (`;`)** inside!
  - Example: `<p>Welcome, <%= name %>!</p>` (Prints the value of the `name` variable).
  - Example: `<p>Time: <%= new java.util.Date() %></p>` (Prints the current date/time).
- **3. Declarations:** `<%! ... Java declarations ... %>`
  - Use this to declare **member variables** (variables belonging to the JSP object) or **methods**.
  - Code here goes *outside* the main `_jspService` method in the generated Servlet.
  - Use carefully! Member variables can cause problems if multiple users access the JSP at the same time (thread-safety issues).
  - Example:

```
<%!
    private int hitCounter = 0; // Member variable
    private String getGreeting() { // Helper method
        return "Hello from Declaration!";
    }
%>
Page Hits: <%= ++hitCounter %> <br>
Greeting: <%= getGreeting() %>
```

- **4. JSP Comments: `<%-- ... comment ... --%>`**
  - Notes for developers reading the JSP code.
  - **Completely ignored** by the server. Does NOT appear in the final HTML sent to the browser.
  - Example: `<%-- TODO: Fix this calculation later --%>`
- **(Remember HTML Comments: `<!-- ... comment ... -->`)**
  - These ARE sent to the browser. Users can see them if they "View Source".

Part 5: Implicit Objects (Built-in Variables in JSP)

- These are special objects the server automatically creates and makes available for you to use directly in scriptlets (`<%...%>`) and expressions (`<%=...%>`). You don't need to declare them.
- **The Main Ones:**
  - `request`: Info **from** the user's browser (form data, URL parameters, headers). Lives for one request. Use `request.getParameter("formFieldName")`.
  - `response`: Used to send info **back** to the browser (like redirecting: `response.sendRedirect("otherpage.jsp")`). Lives for one page execution.
  - `out`: The writer used to **print** HTML content to the page. Expressions (`<%= ... %>`) use this behind the scenes. Lives for one page execution. Use `out.println("Some text");`.
  - `session`: A place to store info about a **specific user** across multiple requests (like login status or a shopping cart). Lives for the user's entire visit (or until timeout). Use `session.setAttribute("cart", myCart)` and `session.getAttribute("cart")`.
  - `application`: A place to store info shared by **all users** of the web application (like global counters or shared data). Lives as long as the web app is running. Use `application.setAttribute(...)` and `application.getAttribute(...)`.
  - `pageContext`: A central object that can access all other implicit objects and manage data in different "scopes" (page, request, session, application).
  - `config`: Configuration info for this specific JSP (less commonly used).
  - `page`: Represents the JSP object itself (`this`). Rarely used directly.
  - `exception`: Holds the error/exception details. **Only available on special "error pages"**.

Part 6: JSP Directives (Instructions for the JSP Engine)

- These tags start with `<%@ ... %>` and give instructions to the server on how to handle the JSP page *during translation*. They don't produce direct output.
- **1. `page` Directive: `<%@ page ... %>`**
  - Sets page-level properties. Usually at the top.

- **Key Attributes:**
    - `import="java.util.List, com.mypackage.*"`: Like Java's `import`, lets you use classes without full package names in scriptlets. Comma-separated.
    - `contentType="text/html; charset=UTF-8"`: **Very Important!** Tells the browser the type of content (HTML) and the character encoding (UTF-8 is best for handling different languages).
    - `pageEncoding="UTF-8"`: Tells the server how the `.jsp` file itself is saved. Best to match `contentType`'s charset.
    - `session="true"` (default) or `session="false"`: Controls if the page uses the `session` implicit object. Set to `false` if you don't need sessions for that page.
    - `errorPage="path/to/ErrorHandler.jsp"`: If an error happens on *this* page, automatically forward the request to `ErrorHandler.jsp`.
    - `isErrorPage="true"` or `false` (default): Set to `true` on `ErrorHandler.jsp` itself. This makes the `exception` implicit object available there.

- **2. `include` Directive: `<%@ include file="path/to/header.jspf" %>`**
    - Acts like a **copy-paste** during translation.
    - The *content* of `header.jspf` is directly inserted into the main JSP file *before* it's turned into a Servlet.
    - The `file` path is relative to the current JSP.
    - Good for including **static** content like headers, footers, menus that are the same for many pages.
    - Changes in the included file (`.jspf`) require the main JSP to be re-translated.

- **3. `taglib` Directive: `<%@ taglib uri="..." prefix="..." %>`**
    - Declares that you want to use a **Custom Tag Library** (like JSTL).
    - `uri`: A unique name identifying the library.
    - `prefix`: A short nickname you'll use for the tags from that library (e.g., `c` for JSTL core tags).
    - Example: `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>`

Part 7: JSP Standard Actions (XML-like Tags for Common Tasks)

- These look like XML/HTML tags (`<jsp:actionName ...>`) and provide a cleaner way to do things than scriptlets. They run at **request time**.
- **1. `<jsp:include page="path/to/dynamicContent.jsp" />`**
    - Includes the output of *another* resource (JSP, Servlet) dynamically when the page is requested.
    - The included page is run separately, and its output is merged into the main page.
    - Good for including content that might change per request (e.g., personalized banner, current news).

  - **Difference vs. `@include` directive:** `@include` is static copy-paste at translation time. `jsp:include` is dynamic execution at request time.
- **2. `<jsp:forward page="path/to/nextPage.jsp" />`**
  - Stops processing the current page and **transfers the request completely** to another page/Servlet on the server.
  - The URL in the browser **does not change**.
  - Often used in MVC pattern: Servlet does processing, then forwards to a JSP to display results.
- **3. `<jsp:param name="paramName" value="paramValue" />`**
  - Used *inside* `<jsp:include>` or `<jsp:forward>` to add extra parameters to the request being sent to the included/forwarded page.
  - Example: `<jsp:include page="userInfo.jsp"><jsp:param name="userId" value="123"/></jsp:include>`
- **4. `<jsp:useBean id="myBean" class="com.example.MyBean" scope="request" />`**
  - The core action for working with **JavaBeans** (reusable Java components).
  - `id`: The variable name you'll use to refer to the bean.
  - `class`: The full Java class name of the bean. Must have a public no-argument constructor.
  - `scope`: Where the bean object lives:
    - `page`: Just this page execution.
    - `request`: This request (can be accessed by forwarded pages). (Common)
    - `session`: This user's session (across multiple requests). (Common for user data)
    - `application`: Shared by all users. (Rarely used for beans).
  - **How it works:**
    1. Looks for a bean named `id` in the specified `scope`.
    2. If found, uses that existing bean.
    3. If not found, creates a **new instance** of the `class` and stores it in the `scope` with the name `id`.
- **5. `<jsp:setProperty name="myBean" property="propertyName" value="someValue" />`**
  - Sets a property on a bean previously created/found by `useBean`.
  - `name`: Matches the `id` from `useBean`.
  - `property`: The name of the property (e.g., `userName`). Calls the bean's `setUserName(...)` method.
  - `value`: The specific value to set. Can use expressions: `value="<%= someVariable %>"`.
  - **Special Uses:**
    - `property="*"`: Automatically sets all bean properties that match request parameter names (useful for forms!).

        - `param="requestParamName"`: Sets the property using the value from a specific request parameter. `<jsp:setProperty name="user" property="email" param="emailAddr"/>`
- **6. `<jsp:getProperty name="myBean" property="propertyName" />`**
    - Gets a property's value from a bean and prints it to the page.
    - `name`: Matches the `id` from `useBean`.
    - `property`: The name of the property (e.g., `userName`). Calls the bean's `getUserName()` method.
    - Example: `<p>User: <jsp:getProperty name="currentUser" property="name"/></p>`

Part 8: Custom Tag Libraries (Making JSP Cleaner!)

- **Problem:** Scriptlets (`<% ... %>`) make JSPs messy and mix Java logic with HTML.
- **Solution:** Use **Custom Tags**! These look like HTML tags but perform actions defined in Java code.
- **Benefits:**
    - **Cleaner JSPs:** Replace Java code with descriptive tags like `<myapp:ifUserLoggedIn>`.
    - **Reusable Logic:** Write Java code once in a "Tag Handler" class, use the tag in many JSPs.
    - **Easier Maintenance:** Update logic in one Java class, not in many JSPs.
    - **Better for Designers:** Designers work with tags, developers work with Java handlers.
- **JSTL (JSP Standard Tag Library): The Most Important Tag Library**
    - A library of standard, pre-built custom tags provided by Oracle/Jakarta EE for common tasks. **Strongly recommended** over scriptlets.
    - **How to Use JSTL:**
        1. **Add JARs:** Put the JSTL `.jar` files (like `jakarta.servlet.jsp.jstl-api.jar` and `jakarta.servlet.jsp.jstl.jar`) into your web app's `WEB-INF/lib` folder.
        2. **Add `taglib` Directive:** Add `<%@ taglib uri="..." prefix="..." %>` to your JSP (see Part 6).
        3. **Use Tags:** Use the tags with the prefix you defined.
    - **Expression Language (EL): `${...}`**
        - JSTL tags almost always work with EL.
        - EL is a simple language to access data without Java code.
        - `${user.name}` gets `user.getName()`.
        - `${param.id}` gets request parameter `id`.
        - `${sessionScope.cart}` gets `cart` object from session.
        - Much cleaner and safer than scriptlet expressions (`<%= ... %>`). Handles nulls better.
    - **Common JSTL Libraries (Prefixes are conventional):**
        - **Core (`c`):** Most used. Variables, loops, conditional logic.

        - `<c:set var="isAdmin" value="${user.role == 'admin'}" />` (Set variable)
        - `<c:if test="${isAdmin}">...</c:if>` (If condition)
        - `<c:forEach var="item" items="${myList}"><li>${item.name}</li></c:forEach>` (Loop)
        - `<c:out value="${user.description}" />` (Print value, safely escaping HTML) **Use this instead of `${...}` directly for user data to prevent security issues (XSS)!**
        - `<c:url value="/product"><c:param name="id" value="${p.id}"/></c:url>` (Create URLs correctly)
    - **Formatting (`fmt`):** Formatting numbers, dates, times; internationalization (multi-language support).
        - `<fmt:formatNumber value="${price}" type="currency" />`
        - `<fmt:formatDate value="${orderDate}" pattern="yyyy-MM-dd" />`
    - **SQL (`sql`):** (Don't use this! Database code belongs in Java classes, not JSPs).
    - **XML (`x`):** For working with XML data.
    - **Functions (`fn`):** String manipulation, collection length, etc. Used inside EL: `${fn:length(myList)}`, `${fn:toUpperCase(name)}`.

UNIT IV: RMI & Hibernate - Broken Down

Part 9: Client and Server Roles

- **Client:** A program or computer that **requests** services or data. (Your web browser is a client).
- **Server:** A program or computer that **provides** services or data when requested. (The web server hosting a website is a server).
- **Distributed Computing:** Making programs on different computers work together over a network. Client/Server is the most common way to organize this.

Part 10: Remote Method Invocation (RMI) - What is it?

- **RMI = Remote Method Invocation.**
- It's Java's way for code in one Java program (the **Client**) to call a method on an object living in *another* Java program (the **Server**), possibly on a different machine.
- **Goal:** Make calling a remote method look almost the same as calling a local method in your code. RMI hides the complex network stuff.
- **Think of it like:** Making a phone call to ask someone else (the server object) to do a specific task (execute a method) for you.

Part 11: RMI - Key Pieces

1. **Remote Interface:**
    - A Java `interface` that extends `java.rmi.Remote`.

    - Declares the methods the Client will be allowed to call remotely.
    - **Crucial:** Every method *must* declare `throws java.rmi.RemoteException`. This warns the client that network errors can happen.
2. **Remote Object Implementation:**
    - A Java `class` that `implements` the Remote Interface.
    - Contains the actual code that runs on the **Server** when a remote method is called.
    - Usually extends `java.rmi.server.UnicastRemoteObject` (this helps make it available remotely).
3. **Stub (Client-side Proxy):**
    - An object that lives on the **Client**.
    - It looks just like the Remote Interface to the client code.
    - When the client calls a method on the Stub, the Stub:
        - Packages up the method call and parameters.
        - Sends them over the network to the Server.
        - Waits for the result.
        - Unpacks the result and returns it to the client code.
    - The client interacts *only* with the Stub.
4. **Skeleton (Server-side Helper - mostly invisible now):**
    - An object on the **Server** that:
        - Receives the request from the Stub.
        - Unpacks the method call.
        - Calls the *actual* method on the real Remote Object Implementation.
        - Packages the result and sends it back to the Stub.
    - In modern Java, this is often handled automatically by RMI, you don't usually see an explicit Skeleton class.
5. **RMI Registry (The Phonebook):**
    - A simple naming service.
    - The **Server** "registers" its remote object with the Registry using a unique name (like "CalculatorService").
    - The **Client** "looks up" that name in the Registry to get the **Stub** object.
    - You can run the `rmiregistry` command or create one inside your server code.

Part 12: RMI - How to Set it Up (The Steps)

1. **Write the Remote Interface:** (`MyService.java`) Define methods, extend `Remote`, add `throws RemoteException`.

2. **Write the Implementation Class:** (`MyServiceImpl.java`) Implement the interface, extend `UnicastRemoteObject`, write method bodies, add constructor throwing `RemoteException`.
3. **Write the Server Code:** (`Server.java`)
    - Create an instance of your Implementation class (`MyServiceImpl service = new MyServiceImpl();`).
    - Start or find the RMI Registry (`LocateRegistry.createRegistry(1099);`).
    - Bind/Register your service object with a name (`Naming.rebind("//localhost/MyServiceName", service);`).
    - Keep the server running.
4. **Write the Client Code:** (`Client.java`)
    - Look up the service in the Registry using the same name (`MyService stub = (MyService) Naming.lookup("//server_address/MyServiceName");`).
    - Call methods on the `stub` object (`stub.doSomething();`). Handle potential `RemoteException`.
5. **Compile:** `javac *.java`
6. **Run:**
    - Start `rmiregistry` (if not created in server code).
    - Start the Server: `java Server`
    - Start the Client: `java Client`

Part 13: RMI - Sending Data (Parameter Passing)

- How arguments and return values travel between Client and Server.
- **Method 1: Pass By Value (Like Sending a Photocopy)**
    - **Applies to:** Primitives (int, float, boolean...) and Objects that implement `java.io.Serializable` (like String, Date, ArrayList, or your own marked `Serializable`).
    - **How:** The object is copied, sent over the network, and a new copy is created on the other side.
    - **Effect:** Changes made to the copy on the receiving end **do not** affect the original object on the sending end.
- **Method 2: Pass By Reference (Like Sending a Remote Control)**
    - **Applies to:** Objects that are themselves `Remote` objects (i.e., they implement a Remote Interface and have been exported).
    - **How:** Instead of sending the object, RMI sends its **Stub**.
    - **Effect:** Both client and server now have references (directly or via stub) pointing back to the **single, original** remote object instance on the server. Calling methods on the received stub executes them on the original object.

Part 14: Hibernate - What is it? (Object-Relational Mapping)

- **Problem:** Java uses Objects (classes, fields, references). Databases use Tables (rows, columns, foreign keys). Translating between these two worlds manually using JDBC (Java Database Connectivity) is repetitive, boring, and error-prone. This difference is called the **Object-Relational Impedance Mismatch**.
- **Hibernate Solution:** Hibernate is an **ORM (Object-Relational Mapping)** framework.
- **It acts like a smart translator:**
    - You tell Hibernate how your Java Objects map to your Database Tables (using annotations or XML files).
    - Hibernate automatically generates the SQL (`INSERT`, `SELECT`, `UPDATE`, `DELETE`) needed to save, load, and modify your objects in the database.
    - You work mostly with your Java objects, Hibernate handles the messy SQL/JDBC stuff.

Part 15: Hibernate - Why Use It? (Benefits)

- **Less Code:** Writes the boring JDBC code for you. You write less, do more.
- **Database Independence:** Easily switch databases (e.g., MySQL to PostgreSQL) by changing configuration. Hibernate adapts the SQL it generates.
- **Focus on Objects:** Work naturally with your Java classes and relationships.
- **Caching:** Speeds things up by keeping frequently used data in memory (Session cache, optional Second-level cache), reducing database hits.
- **HQL (Hibernate Query Language):** Write database queries using your Java class and property names instead of table/column names (e.g., `FROM User WHERE name = 'John'`).

Part 16: Hibernate Architecture - Key Players

- Think of these as the main components you work with:
    1. **Configuration (`hibernate.cfg.xml` or `persistence.xml` / Annotations):**
        - Settings file(s) or annotations that tell Hibernate:
            - How to connect to the DB (driver, URL, user, password).
            - Which database type it is (the SQL "Dialect").
            - *Where to find the mappings* (which classes are Entities, how they map to tables).
            - Other settings (caching, SQL logging, etc.).
    2. **SessionFactory (`SessionFactory` / `EntityManagerFactory`):**
        - Created **once** when your application starts (using the Configuration).
        - It's expensive to create, so only do it once.
        - It's **thread-safe** (multiple threads can use it).
        - Its job is to **create `Session` objects**. Think of it as a factory for sessions.
    3. **Session (`Session` / `EntityManager`):**

    - Created by the SessionFactory for a **single unit of work** (like handling one web request or one transaction).
    - It's **NOT thread-safe**. Each thread needs its own Session.
    - It's **short-lived** – open it, do your database work, then close it.
    - This is the **main object you use** to interact with Hibernate:
        - `session.save(myObject);` // Saves an object
        - `MyClass obj = session.get(MyClass.class, id);` // Loads an object by ID
        - `session.delete(myObject);` // Deletes an object
        - `session.createQuery(...)` // Creates a query
    - Has a **First-Level Cache:** Remembers objects loaded within *this specific session* to avoid re-fetching from the DB immediately.
4. **Transaction (`Transaction` / `EntityTransaction`):**
    - Manages database transactions (groups of operations that must all succeed or all fail together).
    - Get it from the Session: `Transaction tx = session.beginTransaction();`
    - Use `tx.commit();` if everything went okay.
    - Use `tx.rollback();` if an error occurred, to undo changes in that transaction.
5. **Persistent Objects (Entities):**
    - Your Java classes (POJOs) that are mapped to database tables.
    - Marked with `@Entity` annotation (or defined in `hbm.xml`).
    - Have an `@Id` property for the primary key.
    - Have getters/setters for fields you want to save.
6. **Query Objects (`Query`, `CriteriaQuery`, `NativeQuery`):**
    - Objects created by the Session to fetch data based on criteria.
    - Can use HQL/JPQL (object-oriented), Criteria API (programmatic, type-safe), or Native SQL.

Part 17: Hibernate - Basic Workflow

1. **Startup:** Load Configuration -> Create `SessionFactory` (once).
2. **Request/Task Begins:** Get a `Session` from the `SessionFactory`.
3. **Start Transaction:** `Transaction tx = session.beginTransaction();`
4. **Do Work:** Use `session` methods (`save`, `get`, `createQuery`, etc.) to work with your Entity objects.
5. **End Transaction:**
    - If OK: `tx.commit();`
    - If Error: `tx.rollback();`

6. **Cleanup:** `session.close();` (Always close the session!).
