JAVA UNIT 3 NOTES

Advanced Java - Unit III: Java Server Pages (JSP)

This unit shifts focus to Java Server Pages (JSP), a technology built on top of Servlets that simplifies the creation of dynamic web content. It's primarily concerned with the presentation layer of web applications, allowing developers and designers to easily embed Java code within HTML or XML pages. It builds upon the core Java and basic networking concepts covered in Unit 1 and the Servlet technology introduced in Unit 2.

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). (Notes Images 2, 25).
  - JSP files are text-based documents, typically with a `.jsp` extension. (Notes Image 2).
  - They combine **static content** (standard HTML/XML markup) and **dynamic content** (generated by JSP tags, scriplets, etc., written in Java). (Notes Image 2).
- **Why Use JSP?**
  - **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]!"). This is the core purpose of dynamic web pages.
  - **Why Not Just Use Servlets?**
    - Servlets are Java classes that generate HTML using `out.println("<html>...")`. While powerful, this approach mixes presentation logic (HTML structure) heavily with Java logic (data retrieval, conditionals).
    - Generating complex HTML layouts, especially those involving tables, forms, or extensive styling, becomes very messy, hard to read, and difficult to maintain in a Servlet's Java code. (Notes Image 25 mentions that before JSP, CGI/Servlets handled this, but JSP simplifies it).
    - **JSP is easier for designing the page layout** because it looks mostly like HTML. Java code is added only where needed using special tags, making the code clearer for web designers.
  - **Key Idea:** JSP helps separate the **presentation** (how the page looks - HTML/XML) from the **logic** (what data to show - Java/Server-side processing). This separation is a core principle of good web application design (often seen in the Model-View-Controller pattern, where JSP serves as the View).

- **Relationship to Servlets:**
  - JSP technology is an **extension/wrapper over the Java servlet technology**. (Notes Images 2, 25).
  - Every JSP page ultimately gets **translated and compiled into a standard Java Servlet** by the web container. (Notes Images 4, 10, 15, 25).
  - This means JSPs benefit from all the features and performance of Servlets (platform independence, robustness, scalability, access to the full Java API and Servlet API). (Notes Image 29, 35).
- **References:** Notes Images 2-4, 10, 15, 25 provide a general overview of JSP, its relationship to Servlets, and historical context (CGI vs Servlets vs JSP).

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

Java Server Pages execution involves a two-phase process managed by the web container:

1. **Translation Phase (When the `.jsp` is first requested or modified):**
   - You write your web page logic and design in a `.jsp` file (looks like HTML with special Java tags). (Notes Image 4).
   - When a user requests the `.jsp` page for the first time, or if the original `.jsp` file has been changed since the last request, the web container's JSP engine (or translator) reads the `.jsp` file.
   - It converts (translates) the `.jsp` file into a regular Java Servlet source file (`.java`). All the static HTML content becomes `out.println()` statements in the generated `.java` file, and the JSP dynamic elements (scripting elements, actions, directives) are converted into corresponding Java code within the Servlet. (Notes Images 4, 10, 25).
2. **Request Processing Phase (For every user request):**
   - **Compilation (First request or after modification):** The web container compiles the generated Servlet `.java` file into a Java Servlet `.class` file (bytecode). (Notes Images 4, 10).
   - **Loading:** The server loads the compiled Servlet `.class` file.
   - **Instantiation:** The server creates an object (instance) of the loaded Servlet class. This typically happens only *once*. (Notes Image 4).
   - **Execution:** The server runs the compiled Servlet instance's methods. The core request handling is done in a method automatically generated by the JSP engine (often called `_jspService()`). This method contains the logic derived from your original JSP code, which generates the final HTML/XML output. (Notes Images 4, 25).
   - **Sending Response:** The web container captures the output generated by the Servlet (the HTML/XML) and sends it back to the user's browser as an HTTP response. The browser receives and displays standard HTML, never seeing the original JSP source code. (Notes Image 4).

- **Faster Subsequent Requests:** If the `.jsp` file hasn't changed, the server skips the translation and compilation steps on future requests and just runs the existing compiled Servlet (`.class` file), making it efficient. (Notes Image 15).

- **Simplified Flow:**
  `Your_Page.jsp` --(JSP Engine, Translation)--> `Your_Page_jsp.java` --(Javac Compiler)--> `Your_Page_jsp.class` --(JVM, Execution)--> `Generated HTML/XML` --(Web Server)--> `Browser`
- **References:** Notes Images 4, 10, 15, 25 explain the translation and execution process.

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

Since a JSP page is translated into a Servlet, its lifecycle is based on the Servlet lifecycle, with added steps for translation and compilation. The JSP specification also defines methods you can implement to hooks into certain points:

1. **Translation:** `.jsp` to `.java` (Happens when the `.jsp` is first requested or modified).
2. **Compilation:** `.java` to `.class` (Happens after translation).
3. **Loading:** The web container loads the compiled Servlet `.class` file.
4. **Instantiation:** The container creates an instance of the JSP's Servlet class. This happens only **once**. (Notes Images 44, 45, 56 in Unit 2 notes covered this for Servlets, which applies here).
5. **Initialization (`jspInit()`):**
   - A special method you can define in your JSP using a **declaration** (`<%! public void jspInit() { ... } %>`).
   - Called **once** when the JSP's generated Servlet instance is first loaded and initialized by the container (analogous to the Servlet's `init()` method). (Notes Image 45 from Unit 2 notes shows `public void init(...)`).
   - Good for performing one-time setup tasks that are expensive or should only happen once, like loading configuration data or establishing database connections (though database connections are often managed by application servers or connection pools outside the JSP).
   - The Servlet's `init(ServletConfig)` method calls your `jspInit()` method.
6. **Request Processing (`_jspService()`):**
   - This method is automatically generated by the JSP container from the content of your `.jsp` file. **You do NOT write this method yourself**.
   - It's called **for every user request** (analogous to the Servlet's `service()` or `doGet`/`doPost` methods). (Notes Images 44, 46, 56, 57 from Unit 2 notes covered this).
   - It contains the code to process the request, execute scriptlets and actions, evaluate expressions, and output the final HTML/XML response.
   - It receives the implicit `request` (`HttpServletRequest`) and `response` (`HttpServletResponse`) objects.

7. **Destruction (`jspDestroy()`):**
    - A special method you can define in your JSP using a **declaration** (`<%! public void jspDestroy() { ... } %>`).
    - Called **once** when the web application is undeployed, or the container is shutting down, before the JSP's generated Servlet instance is destroyed and garbage collected (analogous to the Servlet's `destroy()` method). (Notes Image 46 from Unit 2 notes covered this).
    - Good for performing cleanup tasks, such as closing database connections or releasing other resources acquired in `jspInit()`.
    - The Servlet's `destroy()` method calls your `jspDestroy()` method.

- **References:** The provided text explicitly mentions `jspInit`, `_jspService`, `jspDestroy`. The Unit 2 notes (Images 44-46) cover the underlying Servlet lifecycle (`init`, `service`, `destroy`) which these map to.

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

- **WARNING:** Using lots of Java code directly in JSP (**scriplets, declarations**) is **old style** and makes pages hard to read and maintain. Modern approaches using **Expression Language (EL)** and **JSTL** (explained later) are strongly preferred for cleaner separation of presentation and logic. However, understanding scripting elements is fundamental as they are part of the JSP core. (Notes Images 13, 14, 18, 19, 25 discuss these elements).
- **1. Scriptlets:** `<% ... Java code ... %>`
    - Used to embed blocks of regular Java code (statements, control flow like loops and if/else) within the JSP page. (Notes Images 14, 19, 25).
    - The code inside a scriptlet is executed **every time** the JSP page is requested.
    - Code inside scriptlets goes directly into the `_jspService()` method of the generated Servlet.
    - Can contain any valid Java code, including local variable declarations, loops, conditionals, and method calls.
    - Can directly access JSP Implicit Objects (`request`, `response`, `session`, `application`, `out`, etc.).
    - Example: `<% String name = "Guest"; if (request.getParameter("user") != null) { name = request.getParameter("user"); } %>`
- **2. Expressions:** `<%= ... Java expression ... %>`
    - Used to **print** the result of a single Java expression directly into the response HTML/XML at the location where the expression tag appears. (Notes Images 6, 14, 19, 25).
    - The expression is evaluated **every time** the page is requested.
    - The result of the expression is automatically converted to a `String` by the container and written to the `out` implicit object.
    - **NO semicolon (`;`) is allowed or needed** inside the `<%= ... %>` tag.

- Example: `<p>Welcome, <%= name %>!</p>` (Prints the value of the `name` variable).
- Example: `<p>Current Time: <%= new java.util.Date() %></p>` (Evaluates `new java.util.Date()` and prints the result).
- Example: `<p>Browser IP: <%= request.getRemoteAddr() %></p>` (Calls a method on the implicit `request` object and prints the result).

- **3. Declarations: `<%! ... Java declarations ... %>`**
  - Used to declare **member variables** (fields) or **methods** that become part of the generated Servlet class, outside the `_jspService()` method. (Notes Images 14, 18, 25).
  - Code here is executed when the Servlet is initialized (`init()` method).
  - Variables declared here are shared **among all client requests** processed by that specific Servlet instance (thread-safety is a major concern).
  - Methods declared here can be called from scriptlets or expressions (as shown in the simple code example for scripting elements in the previous notes).
  - Can be used to implement the `jspInit()` and `jspDestroy()` lifecycle methods.
  - Example:

```
<%!
    private int hitCounter = 0; // Member variable (shared by all users accessing this JSP instance)
    // A method that can be called from scriptlets/expressions
    private String getSimpleGreeting() {
        return "Hello from Declaration!";
    }
%>
<p>Page Hits: <%= ++hitCounter %> </p><%-- Accessing and incrementing the member variable --%>
<p>Greeting: <%= getSimpleGreeting() %> </p><%-- Calling the declared method --%>
```

- **4. JSP Comments: `<%-- ... comment ... --%>`**
  - Used to add comments within the JSP code that are meant for developers and should **not** appear in the final HTML output. (Explicitly mentioned in the provided text).
  - **Completely ignored** by the JSP container during translation and execution. Does NOT appear in the source code of the generated Servlet or the final HTML sent to the browser.
  - Example: `<%-- TODO: Implement proper error handling here --%>`

- **(Contrast with HTML Comments: `<!-- ... comment ... -->`)**
  - These are standard HTML comments. They *are* included in the final HTML output sent to the browser. Users can see them if they use the "View Source" feature in their browser.

- **References:** Notes Images 6, 13, 14, 18, 19, 25 provide definitions and syntax for scripting elements (Declaration, Scriplet, Expression). The provided text adds the explicit mention of JSP Comments and reinforces the others.

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

- These are predefined Java objects that are automatically available for use directly in JSP scripting elements (scriplets, expressions, declarations). They are created and managed by the web container for each request. (Notes Image 13).
- **Why is it important?**
  - They provide convenient access to essential objects representing the request, response, session, application scope, etc., without needing explicit casting or lookup.
- **Key Concepts & Common Implicit Objects:** There are nine implicit objects in JSP: (Notes Image 13 lists some; the provided text lists the main ones and a few others; the full list is standard JSP).
  1. `request`: An instance of `javax.servlet.http.HttpServletRequest`. Represents the client's HTTP request. Contains data sent *from* the browser (form data, URL parameters, headers, cookies, request method). Lives for the duration of a single request. Use `request.getParameter("name")`, `request.getCookies()`, `request.getMethod()`, etc. (Notes Image 12, later OCR pages show `request.getParameter()`).
  2. `response`: An instance of `javax.servlet.http.HttpServletResponse`. Represents the server's HTTP response being sent *back* to the browser. Used to set headers (`setContentType()`, `addCookie()`), redirect (`sendRedirect()`). Lives for the duration of a single request. (Notes Image 12, later OCR pages show `response.getWriter()`). Note: Writing to the response is usually done via the `out` object in JSP.
  3. `out`: An instance of `javax.servlet.jsp.JspWriter`. Used to write character data directly to the response output stream. Expressions (`<%= ... %>`) automatically write to `out`. Scriptlets can use `out.println(...)` or `out.print(...)`. Lives for the duration of a single request.
  4. `session`: An instance of `javax.servlet.http.HttpSession`. Represents the user's session on the server. Used to store and retrieve data specific to a **single user** across multiple requests (`setAttribute("key", value)`, `getAttribute("key")`). Lives for the duration of the user's session (until session timeout or invalidation). (Covered in Unit 2 session tracking and mentioned in Notes Image 13). Only available if the `@page session="true"` directive is used (which is the default).
  5. `application`: An instance of `javax.servlet.ServletContext`. Represents the entire web application running on the server. Used to store and retrieve data shared by **all users** of the application (`setAttribute("key", value)`, `getAttribute("key")`), get application initialization parameters, log messages. Lives as long as the web application is deployed and running. (Mentioned in Notes Image 13).
  6. `pageContext`: An instance of `javax.servlet.jsp.PageContext`. The "god object" of JSP. Provides a single entry point to access attributes stored in any of the four scopes (page, request, session, application) and provides convenient methods to access all other implicit

objects (`pageContext.getRequest()`, `pageContext.getSession()`, etc.). Lives for the duration of a single page execution.

7. `config`: An instance of `javax.servlet.ServletConfig`. Represents the configuration object for the generated Servlet. Used to get servlet initialization parameters. Less commonly used directly in JSP compared to Servlets. Lives as long as the servlet instance lives. (Mentioned in Notes Image 13).
8. `page`: An instance of `java.lang.Object` representing the generated Servlet instance (`this`). Casting is often needed if you use it (`((HttpServlet)page).someMethod()`). Rarely used directly in modern JSP. (Mentioned in Notes Image 13).
9. `exception`: An instance of `java.lang.Throwable`. Represents an uncaught exception that occurred in the JSP page. **Only available on pages designated as error pages** using the `@page isErrorPage="true"` directive.

- **References:** Notes Image 13 lists several implicit objects. Their usage is demonstrated in code examples for scripting elements (scriptlets and expressions). The provided text lists the most common ones and their purpose.

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

- These special tags start with `<%@ ... %>` and provide instructions to the JSP container on how to translate and execute the JSP page. They affect the generated Servlet code itself and do **not** produce direct output on the page. (Notes Images 13, 14, 17, 25). They are processed during the **translation phase**.
- **Why is it important?**
  - They configure essential properties of the JSP page, such as importing Java classes (making them available in scriptlets), including other files at translation time (for code reuse), or specifying custom tag libraries.
- **Key Concepts:** Syntax: `<%@ directiveName attribute="value" attribute2="value2" ... %>`. Can span multiple lines. Common directives:
  1. `page` **Directive (`<%@ page ... %>`):** The most frequently used and versatile. Allows setting numerous page-specific attributes that control the translation and runtime behavior. Usually placed at the top of the `.jsp` file. (Notes Images 14, 17).
     - `import="package.Class, package.other.Class"`: Like Java's `import`, this makes classes or entire packages available for use in JSP scripting elements (scriplets and declarations). You can use multiple `import` attributes separated by commas or use multiple `@page import` directives. (Notes Images 14, 17).
     - `language="java"`: Specifies the scripting language used in scriptlets. `java` is the default and only standard option.
     - `contentType="mimeType; charset=charset"`: **Very Important!** Sets the `Content-Type` HTTP header of the response sent to the browser. This tells the browser how to render the

content (e.g., `text/html`, `application/xml`, `text/plain`). The `charset` specifies the character encoding of the response (e.g., `UTF-8`).

- `pageEncoding="charset"`: Specifies the character encoding of the `.jsp` file itself when saved on the server file system. This helps the container read the `.jsp` file correctly during translation. It is highly recommended to match the `charset` in `contentType`.
- `isErrorPage="true|false"`: Set to `true` on a JSP page intended to handle exceptions. If `true`, the `exception` implicit object is available. Default is `false`.
- `errorPage="url"`: Specifies the URL of a JSP page to forward to if an uncaught `Throwable` occurs on *this* page during request processing.
- `session="true|false"`: Determines whether this JSP page participates in an HTTP session and makes the `session` implicit object available. Default is `true`. Set to `false` if you don't need session state for performance/resource saving.
- `buffer="none|sizekb"`: Controls the size of the output buffer used by the `out` implicit object. `none` means no buffering (output goes directly to the client, potentially less efficient).
- `autoFlush="true|false"`: Applies when `buffer` is set. If `true` (default), the buffer is automatically flushed to the client when it becomes full. If `false`, a `JspException` is thrown if the buffer overflows before it's explicitly flushed.

2. **`include` Directive:** `<%@ include file="path/to/resource" %>`
   - Includes the static or dynamic content of another resource (like HTML, text files, CSS, or even other JSP files) into the current JSP **at translation time**. (Notes Images 14, 17).
   - The content of the `file` is literally copy-pasted into the JSP source file *before* the JSP engine translates it into a Servlet.
   - The `file` attribute's path is relative to the current JSP page.
   - Good for reusing static chunks of code like headers, footers, navigation menus that appear on many pages.
   - Changes to the included file will cause the including JSP to be re-translated the next time it's requested.
   - Less flexible than the `jsp:include` *action* for including content that changes dynamically *per request*.
3. **`taglib` Directive:** `<%@ taglib uri="taglibURI" prefix="prefix" %>`
   - Declares that the JSP page will use a **Custom Tag Library** (like the JSTL). (Notes Images 14, later OCR pages mention this for JSTL).
   - `uri`: A unique identifier (often a URN or URL) that the container uses to locate the Tag Library Descriptor (`.tld`) file.
   - `prefix`: A short string (e.g., `c`, `fmt`, `my`) that you will use as a prefix when using tags from this library in your JSP page.

- **Simple Code Example:** (Same as previous notes, still relevant)

```jsp
<%-- Example 1: Using page directive for import and content type --%>
<%@ page import="java.util.Date, java.text.SimpleDateFormat" %>
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ page pageEncoding="UTF-8" %> <%-- Good practice to match content type
charset --%>
<!DOCTYPE html>
<html>
<head>
    <title>JSP Directives Example</title>
</head>
<body>

    <h1>Demonstrating JSP Directives</h1>

    <%
        // Now we can use Date and SimpleDateFormat because they were imported
        SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        String formattedDate = formatter.format(new Date());
    %>

    <p>
        Current Date and Time (using imported classes): <%= formattedDate %>
    </p>

    <h2>Including a Footer (Translation Time Include):</h2>
    <%-- Example 2: Using include directive --%>
    <%-- Assumes footer.html exists in the same directory with content like
<footer><p>&copy; 2023 My Website</p></footer> --%>
    <%@ include file="footer.html" %>


</body>
</html>
```

(You still need a simple `footer.html` file in the same directory for the include directive example to work.)

- **Explanation of Code:**
  - The `@page import` directive makes `Date` and `SimpleDateFormat` available.
  - `@page contentType` and `@page pageEncoding` configure the response type and file encoding.
  - `@include file="footer.html"` directive statically includes the content of `footer.html` into this JSP during the translation phase.

- **References:** Notes Images 13, 14, 17, 25 provide definitions and syntax for directives.

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

- These are predefined XML-like tags provided by the JSP specification that perform specific, common tasks at **request time**. (Notes Images 13, 14, 20, 21).
- They are processed by the web container when the generated Servlet is executed (`_jspService()` method). They are a cleaner alternative to performing these tasks using scriptlets.
- **Why is it important?**
  - They offer a structured, easy-to-read way to perform common web development tasks directly in the JSP page, promoting code readability and separating presentation logic from embedded Java code.
- **Key Concepts:** Syntax: `<jsp:actionName attribute="value">body</jsp:actionName>` or `<jsp:actionName attribute="value"/>` (for actions without a body). (Notes Image 20). Common actions:
  1. `<jsp:useBean id="myBean" class="com.example.MyBean" scope="request" />`
     - The core action for working with **JavaBeans** (reusable Java components that follow specific conventions). (Notes Images 14, 20, 21, 36, 45).
     - `id`: A scripting variable name you'll use to refer to the bean within this JSP.
     - `class`: The fully qualified Java class name of the JavaBean. The class must be public, have a public no-argument constructor, and typically implement `Serializable`.
     - `scope`: Specifies the scope (storage location) where the bean object should be looked for or stored:
       - `page`: The bean is stored in the `pageContext` and is available only within the current JSP page execution.
       - `request`: The bean is stored in the `HttpServletRequest` object and is available for the current request and any pages included or forwarded to. (Common for data passed from a Controller/Servlet to a View/JSP).
       - `session`: The bean is stored in the `HttpSession` object and is available for the entire duration of the user's session across multiple requests. (Common for user-specific data like login info or a shopping cart).
       - `application`: The bean is stored in the `ServletContext` and is shared by all users of the web application. (Less common for mutable beans due to thread safety).
     - `type`: (Optional) Specifies the type of the scripting variable created. Can be an interface or superclass of the bean's class. Allows using polymorphism.
     - **How it works:** The container first looks for an existing object with the given `id` in the specified `scope`. If found, it casts it to the specified `class` (or `type`) and makes it available as a scripting variable. If not found, it creates a **new instance** of the `class` using its no-

argument constructor, stores it in the specified `scope` with the name `id`, and makes it available as a scripting variable.

2. `<jsp:setProperty name="myBean" property="propertyName" value="someValue" />`
    - Sets one or more properties of a JavaBean previously created/found by `jsp:useBean`. (Notes Images 14, 20, 21, 45).
    - `name`: Matches the `id` of the bean from the `jsp:useBean` action.
    - `property`: The name of the property to set on the bean (e.g., `userName`). The container will automatically call the corresponding public setter method (`setUserName(...)`) on the bean instance.
    - `value`: The static string literal value to set. Cannot use a JSP expression (`<%= ... %>`) directly here. Use EL instead (`value="${someVariable}"` - discussed later).
    - `param`: The name of a request parameter whose value should be used to set the property. If both `value` and `param` are specified, `param` takes precedence.
    - **Special Use:** `<jsp:setProperty name="beanId" property="*"/>` (wildcard) automatically sets all bean properties whose names match request parameter names. This is extremely useful for easily populating a bean with data submitted from an HTML form. (Notes Images 45, 47).
3. `<jsp:getProperty name="myBean" property="propertyName" />`
    - Gets the value of a specific property from a JavaBean and prints it to the page output. (Notes Images 14, 20, 21, 36, 40, 45).
    - `name`: Matches the `id` of the bean from `jsp:useBean`.
    - `property`: The name of the property to get (e.g., `userName`). The container calls the corresponding public getter method (`getUserName()`) and converts the returned value to a String for output.
    - This is equivalent to `<%= myBean.getPropertyName() %>` using a scripting expression, but cleaner.
4. `<jsp:include page="path/to/resource" />`
    - Includes the output generated by another resource (JSP, Servlet, static file) into the current page's response **at request time**. (Notes Images 14, 20, 21, 22, 28, 29).
    - The container executes the specified `page` resource and inserts its output into the current JSP's output stream at the location of the tag.
    - Attributes: `page` (the URL of the resource to include, relative to the current JSP or web application root), `flush` (whether to flush the output buffer before including; `true` is common).
    - Good for including content that might change dynamically *per request* (e.g., dynamic headers/footers, advertisements - as shown in Notes Images 39-41). Can pass parameters using nested `jsp:param` tags.

    - **Difference vs. `@include` directive:** `jsp:include` is dynamic execution at request time, allowing for conditional inclusion or passing dynamic parameters. `@include` is static copy-paste at translation time.
5. **`<jsp:forward page="path/to/nextPage" />`**
    - Terminates the processing of the current JSP page and **transfers the request and response objects completely** to another resource (JSP, Servlet, static file) on the server. (Notes Images 14, 20, 21, 31, 32).
    - The URL in the browser's address bar **does not change**, but the response content comes entirely from the forwarded-to page.
    - Attributes: `page` (the URL of the resource to forward to).
    - Often used in the MVC pattern: a Servlet (Controller) processes user input/business logic, then forwards the request to a JSP (View) to render the results. Can pass parameters using nested `jsp:param` tags.
6. **`<jsp:param name="paramName" value="paramValue" />`**
    - Used *within* the body of `<jsp:include>` or `<jsp:forward>` actions (must be nested inside them) to add or modify request parameters that are sent to the included or forwarded resource. (Notes Images 14, 21, 31, 32).
    - Attributes: `name` (the parameter name), `value` (the parameter value - can be a static string or a JSP expression/EL).
7. `jsp:plugin`: Embeds a Java applet or other plugin component (historically used). (Notes Images 14, 20, 21). Largely deprecated due to applet deprecation.

- **Simple Code Example (Using `jsp:useBean`, `jsp:getProperty`, `jsp:include`, `jsp:forward` - based on previous notes' example):**

```
<%-- Assumes you have a simple JavaBean called 'SimpleBean' --%>
<%-- e.g., package mybeans; public class SimpleBean implements
java.io.Serializable { private String message; public String getMessage() {
return message; } public void setMessage(String msg) { this.message = msg; }
public SimpleBean() {} } --%>

<%@ page contentType="text/html; charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
    <title>JSP Actions Example</title>
</head>
<body>

    <h1>Demonstrating JSP Standard Actions</h1>
```

```
    <%-- Example 1: jsp:useBean and jsp:getProperty --%>
    <%-- Create or find a SimpleBean instance in page scope --%>
    <jsp:useBean id="myBean" class="mybeans.SimpleBean" scope="page"/>
    <%-- Set the message property using scriptlet (or jsp:setProperty) --%>
    <% myBean.setMessage("Hello from the Bean!"); %>

    <p>
        Getting property from bean using jsp:getProperty: <jsp:getProperty
name="myBean" property="message"/>
    </p>

    <%-- Example 2: jsp:include --%>
    <h2>Including another page dynamically:</h2>
    <%-- Assumes includedPage.jsp exists (e.g., with content like "<h2>Included
Content</h2><p>Included from: <%= request.getParameter(\"from\") %></p>") --%>
    <jsp:include page="includedPage.jsp">
        <jsp:param name="from" value="mainPage"/> <%-- Pass a parameter to
includedPage.jsp --%>
    </jsp:include>

    <hr/>

    <%-- Example 3: jsp:forward (This will stop execution here and go to the
next page) --%>
    <%-- Uncomment the block below to test forwarding --%>
    <%--
    <%-- Assumes forwardedPage.jsp exists (e.g., with content like "
<h1>Forwarded Page</h1><p>Status: <%= request.getParameter(\"status\") %></p>")
--%>
    <h2>Forwarding to another page...</h2>
    <jsp:forward page="forwardedPage.jsp">
         <jsp:param name="status" value="success"/> <%-- Pass a parameter to
forwardedPage.jsp --%>
    </jsp:forward>
    <p>This text will NOT be displayed if the forward happens.</p>
     --%>



</body>
</html>
```

(You still need the `SimpleBean.java`, `includedPage.jsp`, and `forwardedPage.jsp` files for this example to work fully when deployed in a web container.)

- **Explanation of Code:**

- The example demonstrates how `jsp:useBean` creates/locates a JavaBean, `jsp:getProperty` reads from it, `jsp:include` dynamically includes another page's output, and `jsp:forward` transfers control. `jsp:param` is shown adding parameters for `jsp:include` and `jsp:forward`.
- The use of these tags keeps the main JSP page cleaner than using only scriptlets.

- **References:** Notes Images 13, 14, 20-22, 31, 32, 36, 40, 45 provide definitions, syntax, and examples for standard actions. The provided text reinforces these concepts.

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

- **Problem:** While standard actions reduce scriplet usage for *some* tasks, many common presentation logic tasks (like complex loops, conditionals, formatting) still end up in messy scriptlets (`<% ... %>`).
- **Solution:** Use **Custom Tag Libraries**! These allow developers to create their own XML-like tags (similar to standard actions but more flexible) to represent reusable chunks of presentation logic directly in JSP pages. (Notes Image 1 lists this topic).
- **Why is it important?**
  - **Cleaner JSPs:** Replace verbose Java code in scriptlets with simple, descriptive tags (e.g., `<myprefix:formatPrice value="${product.price}"/>` instead of complex Java formatting code).
  - **Code Reusability:** Encapsulate complex or frequently used presentation logic into a single Java "Tag Handler" class, which can then be used via its custom tag in many different JSP pages.
  - **Better Separation of Concerns:** Moves presentation-related Java logic out of the view (JSP) and into dedicated Java classes (Tag Handlers), resulting in cleaner, more readable JSP pages that are easier for web designers to understand and modify.
  - **Abstraction:** Hide complex implementation details behind a simple tag interface.
- **Key Concepts:**
  - **Tag Library:** A collection of custom tags.
  - **Tag Library Descriptor (TLD):** An XML file (`.tld` extension), typically located in `WEB-INF/tlds`, that defines the tags in the library, their attributes, and maps each tag to its corresponding Java **Tag Handler Class**.
  - **Tag Handler Class:** A Java class (e.g., implementing `javax.servlet.jsp.tagext.SimpleTag` or extending `javax.servlet.jsp.tagext.SimpleTagSupport` for JSP 2.0+; older versions used `Tag` or `BodyTag` interfaces). This class contains the Java code that is executed by the container when the custom tag is processed in the JSP. It has methods (`doTag()` for `SimpleTag`) that perform the tag's logic, interact with the `JspContext` (similar to `PageContext`), and can write output to the response.
  - **Using a Custom Tag in a JSP:**
    1. **Add Tag Library JARs:** If using external libraries like JSTL, place their `.jar` files (containing the tag handler classes and TLDs) into your web application's `WEB-INF/lib` folder.

2. **Declare the Tag Library:** Use the `<%@ taglib %>` directive at the top of the JSP page: `<%@ taglib uri="taglibURI" prefix="prefix" %>`.
    - `uri`: A unique identifier for the tag library. The container uses this to find the corresponding TLD file (which can be inside a JAR or directly in the web app's `WEB-INF`).
    - `prefix`: A short nickname you choose. All custom tags from this library will be used with this prefix (e.g., `<myprefix:tagName ...>`).
3. **Use the Tag:** Use the custom tags in the page body using the declared prefix, e.g., `<myprefix:helloTag attribute="value"/>` (simple tag) or `<myprefix:loopTag>Body content</myprefix:loopTag>` (tag with body).

- **JSTL (JSP Standard Tag Library): The Most Important Custom Tag Library**
    - JSTL is a standard, widely-used custom tag library that provides tags for common tasks that would otherwise require scriptlets. **Using JSTL (along with EL) is the modern best practice for presentation logic in JSPs and significantly reduces or eliminates the need for scriplets.**
    - **How to Use JSTL:**
        1. **Add JSTL JARs:** Download and place the JSTL implementation JARs (e.g., from the Apache Taglibs project) into your web app's `WEB-INF/lib` folder.
        2. **Add `taglib` Directive:** Add the appropriate `@taglib` directive(s) for the JSTL library you need (see Part 6 example). Common URIs: `http://java.sun.com/jsp/jstl/core` (for `c:` prefix), `http://java.sun.com/jsp/jstl/fmt` (for `fmt:` prefix).
        3. **Use Tags:** Use the JSTL tags with the prefix you defined.
    - **Expression Language (EL): `${...}`**
        - JSTL tags almost always work with EL. **EL is also a standard part of JSP (since 2.0) and is used extensively in modern JSPs, typically alongside JSTL, to access data without scriptlets.**
        - EL expressions start with `${` and end with `}`. They are evaluated by the container and their result is output (similar to `<%= ... %>`, but cleaner and safer).
        - EL provides simple syntax to access:
            - Bean properties (implicitly using getters): `${myBean.message}` is equivalent to `<%= myBean.getMessage() %>`.
            - Map values: `${myMap.key}` or `${myMap['key']}`.
            - Array or List elements: `${myArray[0]}` or `${myList[1]}`.
            - Implicit objects and their properties: `${request.getParameter('name')}`, `${sessionScope.user}`, `${header['User-Agent']}`. `sessionScope` and `requestScope` etc. are EL implicit objects that provide access to attributes stored in different scopes.

- Handles nulls gracefully (usually outputs nothing instead of throwing a `NullPointerException`).
- Much cleaner and safer than scriptlet expressions (`<%= ... %>`).

- **Common JSTL Libraries (Prefixes are conventional):**
  - **Core (`c` prefix):** `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>`. Provides tags for basic control flow and output.
    - `<c:set var="isAdmin" value="${user.role == 'admin'}" scope="request"/>`: Sets a variable in a specified scope (like request).
    - `<c:if test="${isAdmin}"> ... </c:if>`: Simple conditional logic.
    - `<c:choose>`, `<c:when>`, `<c:otherwise>`: More complex conditional logic (like if/else if/else).
    - `<c:forEach var="item" items="${myList}"><li>${item.name}</li></c:forEach>`: Iterates over collections or arrays.
    - `<c:out value="${user.description}" escapeXml="true"/>`: Outputs a value. `escapeXml="true"` (default) prevents Cross-Site Scripting (XSS) by converting HTML special characters (like `<`) to entities (`<`). **Always use `<c:out>` for outputting potentially unsafe data!**
    - `<c:url var="productLink" value="/product"><c:param name="id" value="${p.id}"/></c:url>`: Creates URLs correctly, handling context path and parameters. `<a href="${productLink}">...</a>`.
  - **Formatting (`fmt` prefix):** `<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>`. Provides tags for formatting numbers, dates, times, and handling internationalization (multi-language support).
    - `<fmt:formatNumber value="${price}" type="currency" currencyCode="USD"/>`
    - `<fmt:formatDate value="${orderDate}" pattern="yyyy-MM-dd HH:mm"/>`
    - `<fmt:message key="greeting.message"/>`: Retrieves a localized message for internationalization.
  - **SQL (`sql` prefix): Avoid using this!** This library allows direct database access from JSP. Putting database logic in the presentation layer is bad practice. Database code belongs in Java classes (Servlets, EJBs, DAOs) in the business/data access layer.
  - **XML (`x` prefix):** For processing XML data.
  - **Functions (`fn` prefix):** `<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>`. Provides common functions usable within EL expressions.
    - `${fn:length(myList)}`: Returns the size of a collection/array or length of a string.
    - `${fn:toUpperCase(name)}`: Converts a string to uppercase.
    - `${fn:contains(text, substring)}`: Checks if a string contains a substring.

- **Simple Code Example (Using JSTL and EL):**

```
<%-- Declare JSTL Core and Formatting libraries --%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%-- Assuming JSTL JARs are in WEB-INF/lib --%>

<%@ page contentType="text/html; charset=UTF-8" %>
<%-- Assumes a 'user' object (with properties like name, isAdmin, lastLogin) --
%>
<%-- and a 'products' list (List<Product> where Product has name, price) --%>
<%-- are available in the request scope (e.g., set by a Servlet using
request.setAttribute) --%>

<!DOCTYPE html>
<html>
<head>
    <title>JSP JSTL/EL Example</title>
</head>
<body>

    <h1>Demonstrating JSTL and EL</h1>

    <%-- Using EL to access bean properties and request parameters --%>
    <p>Hello, ${requestScope.user.name}! <%-- Access 'name' property of 'user'
from request scope --%></p>
    <p>Your role is: ${user.role} <%-- Can often drop 'Scope' if unique --%></p>

    <%-- Using JSTL c:if for conditional logic --%>
    <c:if test="${user.admin}"> <%-- Checks user.isAdmin or user.getIsAdmin()
boolean property --%>
        <p>You are an administrator.</p>
    </c:if>

    <%-- Using JSTL fmt:formatDate for formatting --%>
    <c:if test="${not empty user.lastLogin}"> <%-- Checks if user.lastLogin is
not null or empty --%>
        <p>Your last login was: <fmt:formatDate value="${user.lastLogin}"
pattern="MM/dd/yyyy HH:mm"/></p>
    </c:if>

    <%-- Using JSTL c:forEach for loops --%>
    <h2>Products:</h2>
    <c:if test="${not empty products}">
```

```
        <ul>
            <c:forEach var="product" items="${requestScope.products}"> <%-- Loop
over the 'products' list --%>
                <li>
                    <%-- Using c:out for safe output --%>
                    <c:out value="${product.name}"/> -
                    <%-- Using fmt:formatNumber for currency formatting --%>
                    <fmt:formatNumber value="${product.price}" type="currency"
currencyCode="USD"/>
                </li>
            </c:forEach>
        </ul>
    </c:if>
     <c:if test="${empty products}">
         <p>No products available.</p>
     </c:if>

</body>
</html>
```

(This example is conceptual and requires a Servlet to set the `user` and `products` attributes in the request scope before forwarding to this JSP, and the presence of the JSTL JARs in `WEB-INF/lib`)

- **Explanation of Code:**
  - The `@taglib` directives declare the JSTL libraries and their prefixes (`c`, `fmt`).
  - EL expressions (`${...}`) are used to access properties of the `user` object and the `products` list, which are assumed to be placed in the request scope by a controlling Servlet.
  - JSTL tags (`<c:if>`, `<fmt:formatDate>`, `<c:forEach>`, `<c:out>`, `<fmt:formatNumber>`) provide control flow, formatting, and safe output without needing any scriplets.
- **References:** The syllabus lists "Custom Tag Libraries". The provided text explicitly introduces EL and JSTL and explains their purpose and common uses. This section expands on that introduction to show practical JSTL/EL examples and how they replace scripting elements for many tasks.
