WEB TECHNOLOGY
Unit – V

Contents

Servlets

Servlet Overview

A Servlet is a Java program that runs on a web server. It is used to handle requests from users and provide responses, usually in the form of a web page. For example, when you fill out a form on a website and press submit, a servlet might process the data you entered.

Why use Servlets?

- To create dynamic web pages (pages that change based on user input).

- They are faster and more secure compared to older methods like CGI (Common Gateway Interface).

- They are portable, meaning they can run on any server that supports Java.

Servlet Architecture

The architecture of a servlet involves three main components:

1. **Client (Browser)**:

- This is where the user interacts with the website.

- The client sends a request (e.g., when you click a button or type a URL in the browser).

2. **Web Server**:

- The server receives the request from the client.

- It forwards the request to the servlet for processing.

3. **Servlet (Java Program)**:

- This is the core part of the architecture.

- It processes the request (e.g., by reading form data or interacting with a database).

- It generates a response (e.g., an HTML page) and sends it back to the client.

Steps in Servlet Workflow

1. **Client Sends a Request**:

- A user opens a web page or submits a form.

- The browser sends this request to the server.

2. **Server Calls the Servlet**:

- The server finds the correct servlet to handle the request.

3. **Servlet Processes the Request**:

- The servlet reads the request data (like form inputs).

- It performs necessary actions (like saving data to a database).

4. **Servlet Sends a Response**:

- The servlet generates an output (like a webpage or a message).

- The server sends this response back to the user's browser.

Example Scenario:

- A student logs into their school website.

- The browser (client) sends their login details to the web server.

- The server forwards this request to a servlet.

- The servlet checks the login details in the database.

- If the details are correct, the servlet sends a "Welcome" page back to the browser.

Benefits of Using Servlets

1. **Fast and Efficient**:

They are faster than traditional methods because they run in the server's memory.

2. **Secure**:

They can use encryption and authentication to protect user data.

3. **Reusable**:

The same servlet can handle multiple requests.

4. **Platform-Independent**:

They run on any operating system with a Java-enabled server.

Interface Servlet

In Java, an interface is like a blueprint that defines what methods a class must have. The Servlet interface is part of the Java library and is used to create servlets.

What is the Servlet Interface?

- It is an interface in Java that defines basic methods that every servlet must have.

- These methods help in managing the servlet’s tasks, like starting, stopping, and handling user requests.

Methods of the Servlet Interface:

1. **init()**: Called when the servlet is first created. It is used to initialize resources like database connections.

2. **service()**: Called whenever the servlet receives a request from the client. It contains the logic to process the request and generate a response.

3. **destroy()**: Called when the servlet is being removed from the server. It is used to release resources like closing database connections.

4. **getServletConfig()**: Provides configuration details about the servlet.

5. **getServletInfo()**: Returns information about the servlet, like its version or developer details.

Servlet Life Cycle

The life cycle of a servlet refers to the stages it goes through from creation to destruction. There are three main phases in a servlet's life cycle: Initialization, Request Handling, and Destruction.

1. **Initialization (init())**

- This phase occurs when the servlet is loaded for the first time.

- The init() method is called to initialize the servlet.

- Example: Opening a database connection or reading configuration files.

- It runs only once during the servlet's life.

2. **Request Handling (service())**

- This is the main phase where the servlet handles user requests.

- Every time a user sends a request (like clicking a button or submitting a form), the service() method is called.

- The servlet processes the request, performs the required task (like fetching data from a database), and sends a response back to the user.

- This method can be called multiple times, once for each request.

3. **Destruction (destroy())**

- This phase occurs when the servlet is being removed or the server is shutting down.

- The destroy() method is called to clean up resources.

- Example: Closing database connections or freeing up memory.

- It runs only once, just before the servlet is removed.

Steps in the Servlet Life Cycle

1. **Loading the Servlet**:

The server loads the servlet class when it is requested for the first time.

2. **Creating an Instance**:

The server creates an object (instance) of the servlet.

3. **Initializing the Servlet**:

The server calls the init() method to prepare the servlet for use.

4. **Handling Requests**:

For every client request, the server calls the service() method.

5. **Destroying the Servlet**:

When the server no longer needs the servlet, it calls the destroy() method.

Example for Better Understanding

Think of a servlet as a teacher in a classroom:

1. **Initialization (init())**:

When the teacher enters the classroom, they set up their materials (like chalk and books).

2. **Request Handling (service())**:

During the class, students ask questions (requests), and the teacher provides answers (responses).

3. **Destruction (destroy())**:

When the class ends, the teacher cleans up their materials and leaves.

Why is the Servlet Life Cycle Important?

- It helps in managing resources efficiently.

- Developers can control how a servlet starts, processes requests, and stops.

- It ensures that a servlet behaves predictably and reliably.

Handling HTTP GET Requests

In web technology, a GET request is a type of request sent by a web browser to a server to ask for information, like a webpage or data.

What is an HTTP GET Request?

- It is used to request data from a server.

- Example: When you type a URL in the browser and press Enter, the browser sends a GET request to the server.

- Data sent with a GET request is added to the URL as part of the query string (e.g., example.com?name=John&age=18).

How to Handle GET Requests in Servlets?

- In Java servlets, the doGet() method is used to handle GET requests.

- The doGet() method processes the request, retrieves any data from the URL, and sends a response back to the browser.

Steps for Handling GET Requests:

1. The client (browser) sends a GET request to the server.

2. The servlet's doGet() method is called.

3. The servlet reads any data from the URL (if present).

4. The servlet processes the request (e.g., fetches data from a database).

5. The servlet sends the response back to the client (e.g., a webpage).

Example of Handling GET Requests in a Servlet:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    // Reading data from the URL (query string)
    String name = request.getParameter("name");

    // Sending a response back to the client
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
```

```
    out.println("<h1>Hello, " + name + "!</h1>");
}
```

If the URL is example.com?name=John, the servlet will respond with: "Hello, John!"

Handling HTTP POST Requests

A POST request is another type of request sent by the browser, usually to send data to the server, such as when submitting a form.

What is an HTTP POST Request?

- It is used to send data securely to a server.

- Unlike GET requests, the data is not visible in the URL. It is sent in the body of the request.

- Example: When you fill out a login form and press the submit button, the browser sends a POST request to the server with your username and password.

How to Handle POST Requests in Servlets?

- In Java servlets, the doPost() method is used to handle POST requests.

- The doPost() method processes the request, reads the data from the body, and performs actions like saving the data in a database.

Steps for Handling POST Requests:

1. The client sends a POST request to the server.

2. The servlet's doPost() method is called.

3. The servlet reads the data from the request body (e.g., form data).

4. The servlet processes the request (e.g., saves the data to a database).

5. The servlet sends a response back to the client.

Example of Handling POST Requests in a Servlet:

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    // Reading data from the request body
    String username = request.getParameter("username");
    String password = request.getParameter("password");
```

```
    // Sending a response back to the client
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    if (username.equals("admin") && password.equals("1234")) {
        out.println("<h1>Login Successful!</h1>");
    } else {
        out.println("<h1>Invalid Credentials</h1>");
    }
}
```

If the form sends username=admin and password=1234, the servlet will respond with: "Login Successful!"

Redirecting Requests to Other Resources

In web applications, redirecting requests means sending the user to a different webpage or resource, either on the same server or a different one.

What is Request Redirection?

- When a user requests a webpage, the server can send them to another page instead of the one they asked for.

- This is called redirecting a request.

Why Redirect a Request?

1. If a page has moved to a new location.

2. To guide users to a login page when they are not logged in.

3. To send users to a thank-you page after submitting a form.

Types of Redirection:

1. **Client-Side Redirection**:

- The browser is told to request a new URL.

- Example: Redirecting using JavaScript or HTTP status code 302 (temporary redirect).

2. **Server-Side Redirection**:

- The server directly sends the request to another resource.

How to Redirect Requests in Java Servlets?

There are two common ways:

1. **Using sendRedirect() (Client-Side Redirection)**:

- The server tells the browser to request a new URL.

response.sendRedirect("newPage.html");

- Example: If the user tries to visit example.com/dashboard but isn't logged in, they can be redirected to example.com/login.

2. **Using RequestDispatcher (Server-Side Redirection):**

- The server internally forwards the request to another resource.

RequestDispatcher dispatcher = request.getRequestDispatcher("newPage.html");
dispatcher.forward(request, response);

- Example: Sending the user from a login form to the homepage.

Session Tracking

In web applications, session tracking is a way to remember information about a user across multiple pages or interactions.

What is a Session?

- A session represents a user's interaction with a website, starting when they visit and ending when they leave.

- Example: When you log in to an online shopping site, the website remembers your account and shopping cart until you log out.

Why Do We Need Session Tracking?

1. HTTP is stateless, meaning the server forgets who you are after every request.

2. To keep track of user activities like login status, shopping cart items, or preferences.

Methods of Session Tracking:

1. **Cookies**:

- Small pieces of data stored on the user's browser.

- The server sends a cookie to the browser, and the browser sends it back with every request.

Cookie cookie = new Cookie("username", "John");
response.addCookie(cookie);

- Example: Remembering a user's preferred language.

2. **Session API (Java Session):**

- The server creates a session object for each user.

- The session stores data like login status or cart items.

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

- Example: Keeping a user logged in until they close the browser.

3. **URL Rewriting**:

- The session ID is added to the URL if cookies are disabled.

- Example: example.com/dashboard?sessionID=12345.

4. **Hidden Form Fields**:

- Session data is stored in hidden fields within forms.

- Example: Sending the session ID in a hidden form field when navigating pages.

How Session Tracking Works in Servlets:

1. A user logs in to a website.

2. The server creates a session and stores user information.

3. As the user navigates pages, the server identifies them using a session ID (stored in a cookie or URL).

4. The session ends when the user logs out or closes the browser.

Example of Using Sessions in Servlets:

// Creating a session
HttpSession session = request.getSession();
session.setAttribute("username", "John");

```
// Retrieving data from the session
String username = (String) session.getAttribute("username");

// Ending a session
session.invalidate();
```

Cookies

A cookie is a small piece of data that a website stores on a user's browser to remember information about them.

What are Cookies?

- Cookies help websites remember users and their preferences.

- For example, when you visit an online store, cookies remember your shopping cart items, even if you navigate to other pages.

Types of Cookies:

1. **Session Cookies**:

Temporary cookies that are deleted when you close the browser.

Example: A shopping cart on an e-commerce site.

2. **Persistent Cookies**:

Stored on your device even after you close the browser.

Example: A website remembering your login details.

How Cookies Work:

1. A website sends a cookie to your browser.

2. The browser stores the cookie.

3. When you revisit the website, the browser sends the cookie back to the website.

Creating and Reading Cookies in Servlets:

Creating a Cookie:

```
Cookie cookie = new Cookie("username", "John");
response.addCookie(cookie);
```

Reading a Cookie:

```
Cookie[] cookies = request.getCookies();
for (Cookie c : cookies) {
    if (c.getName().equals("username")) {
        String username = c.getValue();
    }
}
```

Session Tracking with HttpSession

A session is a way for a web application to remember a user while they browse multiple pages.

What is a Session?

- A session stores information about a user on the server.

- It allows the website to keep track of the user's actions during their visit.

Why Use Sessions?

- HTTP is stateless, meaning the server forgets who you are after each request.

- Sessions solve this problem by storing information like login status, shopping cart items, or preferences.

How HttpSession Works:

1. When a user visits the website, the server creates a session.

2. A session ID is assigned to the user.

3. The session ID is sent to the browser as a cookie.

4. For each request, the browser sends the session ID back to the server.

5. The server uses the session ID to retrieve the user's data.

Steps in Session Tracking with HttpSession:

1. **Create a Session**:

When a user logs in or interacts with the website, a session is created.

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

2. **Retrieve Data from the Session**:

To get stored data from the session:

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

3. **End the Session**:

When the user logs out or closes the browser, the session can be ended:

session.invalidate();

Example Scenario:

1. A user logs in to a shopping website.

2. The server creates a session and stores the user's login details and cart items.

3. As the user browses different pages, the server uses the session to remember them.

4. When the user logs out, the session is destroyed, and all data is cleared.

Java Server Pages (JSP)

JSP Introduction

Java Server Pages (JSP) is a technology used to create dynamic web pages. It allows developers to write HTML and Java code together in one file.

What is JSP?

- JSP is a part of Java technology that helps in developing web applications.

- It is a simpler way to create web pages compared to writing servlets because it combines HTML (for structure) and Java (for logic).

- JSP files have the extension .jsp.

Why Use JSP?

1. **Dynamic Content**: JSP can generate web pages that change based on user input or data from the server (e.g., showing user-specific information after login).

2. **Separation of HTML and Java**: JSP keeps the presentation (HTML) and logic (Java code) in one file, making it easier to write and manage.

3. **Reusable Code**: Developers can use reusable Java components (like beans or libraries) with JSP to reduce redundancy.

4. **Better than Servlets for Web Pages**: Writing HTML inside servlets is difficult, but JSP makes it easy.

How JSP Works:

1. **Request**: The user (browser) requests a JSP page by clicking a link or entering a URL.

2. **Processing**:

- The server converts the JSP file into a servlet (Java class).

- This servlet processes the user's request.

3. **Response**: The servlet generates an HTML response and sends it back to the user's browser.

Key Features of JSP:

1. **Mix of HTML and Java**:

Developers can embed Java code directly into HTML using special tags.

Example: *<%= "Hello, User!" %>* prints "Hello, User!" on the web page.

2. **Custom Tags and Libraries**:

JSP allows the use of tags to simplify the code (e.g., *<jsp:include>* to include another file).

3. **Integration with Java Servlets**:

JSP works seamlessly with servlets for complex tasks like session management.

JSP Syntax

1. **Scriptlet (<% %>)**: Used to write Java code inside a JSP file.

Example:

<% int num = 10; %>

2. **Expression (<%= %>)**: Prints the value of a Java expression directly in the HTML.

Example:

```
<p>The result is: <%= 5 + 5 %></p>
```

3. **Directive (<%@ %>)**: Gives special instructions to the server.

Example:

```
<%@ page language="java" %>
```

Example of a JSP File

```
<html>
<head>
    <title>Welcome Page</title>
</head>
<body>
    <h1>Welcome to JSP!</h1>
    <%
        String username = "John";
    %>
    <p>Hello, <%= username %>! Thanks for visiting.</p>
</body>
</html>
```

Advantages of JSP

1. **Simpler than Servlets**: Writing HTML in JSP is easier compared to servlets.

2. **Platform-Independent**: Runs on any server that supports Java (e.g., Apache Tomcat).

3. **Reusable Components**: Allows the use of reusable Java beans and tag libraries.

4. **Dynamic and Interactive**: Creates web pages that respond to user actions or data.

Disadvantages of JSP

1. **Performance**: Converting JSP to a servlet takes extra time during the first request.

2. **Complexity**: For very complex applications, managing JSP code can become tricky.

3. **Not Ideal for Pure Logic**: It is better to use servlets for tasks involving only business logic.

Conclusion

Java Server Pages (JSP) is a powerful tool for creating dynamic and interactive web pages. It simplifies web development by allowing developers to mix Java and HTML code, making it an essential part of modern Java-based web applications.

Java Server Pages (JSP) Overview

Java Server Pages (JSP) is a technology used to create dynamic web pages. It allows developers to combine HTML and Java code in one file to make web pages that can change based on user input or data from the server.

Key Features of JSP

1. **Dynamic Web Pages**: JSP can generate web pages that display different content for each user. For example, a shopping website can show a personalized welcome message.

2. **Simplifies Development**: It is easier to write HTML in JSP compared to servlets, as servlets require embedding HTML inside Java code.

3. **Reusable Components**: JSP can include reusable Java components like beans and custom tags to avoid writing repetitive code.

4. **Integration with Servlets**: JSP works with servlets for tasks like handling user requests or managing sessions.

How JSP Works

1. **Client Request**: The browser sends a request for a JSP page.

2. **JSP Conversion**: The server translates the JSP file into a servlet (Java code).

3. **Execution**: The servlet executes the Java code in the JSP file and generates an HTML response.

4. **Response to Browser**: The server sends the HTML response back to the user's browser.

Benefits of JSP

1. **Ease of Use**: Developers can focus more on HTML and design, while Java handles the logic.

2. **Powerful Java Features**: JSP can use Java libraries and APIs, making it versatile.

3. **Portability**: JSP runs on any platform that supports Java, making it highly portable.

4. **Separation of Content and Logic**: JSP allows for a clean separation of design (HTML) and logic (Java), making code easier to maintain.

A First Java Server Page Example

Creating a simple JSP file is straightforward. Here's how you can make a basic web page using JSP.

Steps to Create and Run a JSP File

1. **Create a JSP File**:

A JSP file is just an HTML file with a .jsp extension and some Java code embedded.

2. **Write the Code**:

Save the following content in a file named welcome.jsp:

```
<html>
<head>
    <title>Welcome Page</title>
</head>
<body>
    <h1>Welcome to Java Server Pages (JSP)!</h1>
```

```
    <%
        // Java code inside a scriptlet
        String username = "John";
        out.println("Hello, " + username + "! Have a great day!");
    %>
</body>
</html>
```

3. **Deploy the JSP File**:

Place the welcome.jsp file in the webapps folder of a web server like Apache Tomcat.

4. **Run the File**:

Open a browser and visit http://localhost:8080/welcome.jsp to see the output.

Code Explanation

1. **HTML Content**:

The regular HTML code defines the structure of the webpage (like the title and heading).

2. **Java Code**:

Inside <% %>, you can write Java code. In this example, the scriptlet sets a username variable and displays a message using out.println.

3. **Dynamic Output**:

The message changes based on the Java code, demonstrating how JSP creates dynamic web pages.

Output

When you visit the page in a browser, it will display:

Welcome to Java Server Pages (JSP)!

Hello, John! Have a great day!

Conclusion

This example shows how JSP mixes Java and HTML to create dynamic web pages. With just a few lines of code, you can generate a personalized, interactive experience for users. As students, understanding this concept is key to building modern web applications!

Implicit Objects in JSP

Implicit Objects are pre-defined objects in JSP that are automatically created by the server and made available for use in JSP pages. You can use these objects without declaring or initializing them.

List of Implicit Objects:

Object	Description
request	Represents the client's HTTP request and contains data like form inputs.
response	Represents the server's HTTP response sent to the client.
session	Stores user-specific data (e.g., login info) across multiple pages.
application	Represents the web application and allows sharing data across the whole app.
out	Used to write output to the client's browser.
config	Provides configuration details for the JSP page.
pageContext	Provides access to all other implicit objects and additional page-level info.

Object	Description
page	Refers to the current JSP page as an object.
exception	Used to handle errors or exceptions that occur in the JSP page.

Example: Using Implicit Objects

```
<%
   // Using the request object to get data from a form
   String name = request.getParameter("username");
   out.println("Welcome, " + name + "!");
%>
```

Scripting in JSP

Scripting refers to the way Java code is written directly within a JSP file. JSP scripting elements allow you to embed Java logic into the HTML of a webpage.

Types of Scripting Elements:

1. **Scriptlet (<% %>):**

- Used to write Java code inside JSP.

- Code written here is executed every time the page is requested.

Example:

```
<%
   int number = 10;
   out.println("The number is: " + number);
%>
```

2. **Expression (<%= %>):**

- Used to print the result of a Java expression directly into the HTML.

- It is shorter and easier for outputting values.

Example:

```
<p>The sum is: <%= 5 + 5 %></p>
```

3. **Declaration (<%! %>)**:

- Used to declare methods or variables that can be used across the JSP file.

Example:

```
<%!
   int multiply(int a, int b) {
       return a * b;
   }
%>
<p>The product is: <%= multiply(5, 4) %></p>
```

Note:

Overusing scripting can make JSP code hard to read. It's better to separate logic (Java) from design (HTML) using JavaBeans or MVC frameworks.

Standard Actions in JSP

Standard Actions are predefined tags in JSP that perform specific tasks. They start with <jsp: and make JSP more powerful by reducing the need for Java code.

Common Standard Actions:

1. **<jsp:include>**:

- Includes content from another file into the current JSP page.

- Useful for reusing headers, footers, or navigation bars.

Example:

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

2. **<jsp:forward>**:

- Forwards the request to another resource (like a JSP page, servlet, or HTML file).

- Useful for redirecting users to another page.

Example:

```
<jsp:forward page="login.jsp" />
```

3. **<jsp:useBean>**:

- Creates or uses a JavaBean object in the JSP page.

Example:

```
<jsp:useBean id="user" class="com.example.User" />
```

4. **<jsp:setProperty>**:

- Sets the value of a JavaBean property.

Example:

```
<jsp:setProperty name="user" property="username" value="John" />
```

5. **<jsp:getProperty>**:

- Gets the value of a JavaBean property and displays it on the webpage.

Example:

```
<p>Username: <jsp:getProperty name="user" property="username" /></p>
```

Example JSP File with All Concepts

```
<html>
<head>
    <title>JSP Example</title>
</head>
<body>
    <h1>Welcome to JSP</h1>
```

```
    <%-- Using Implicit Objects --%>
    <%
        String username = request.getParameter("username");
        session.setAttribute("user", username);
    %>
    <p>Hello, <%= username %>! Your session ID is: <%= session.getId() %></p>
    <%-- Using Scripting Elements --%>
    <%
        int a = 5, b = 10;
    %>
    <p>The sum is: <%= a + b %></p>
    <%-- Using Standard Actions --%>
    <jsp:include page="footer.jsp" />
</body>
</html>
```

Conclusion

- Implicit Objects simplify accessing request, response, and session data.

- Scripting Elements allow embedding Java code directly in JSP for logic and calculations.

- Standard Actions enhance reusability and reduce code complexity by using tags for tasks like including files and working with JavaBeans.

Directives in JSP

Directives are special instructions in JSP that guide the server on how to process the JSP file. They do not produce any output on the web page but control the overall behavior of the page.

Types of Directives in JSP

1. **Page Directive**

- Used to define page-level settings such as importing Java classes, error handling, and output content type.

- Syntax: *<%@ page attribute="value" %>*

Common Attributes:

import: Includes Java classes.

<%@ page import="java.util.Date" %>

contentType: Sets the type of content sent to the client (e.g., HTML, XML).

<%@ page contentType="text/html" %>

errorPage: Specifies the page to display if an error occurs.

<%@ page errorPage="error.jsp" %>

2. **Include Directive**

- Includes the content of another file (e.g., header, footer) at compile time.

- Syntax: *<%@ include file="filename" %>*

- Example:

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

- This directive is useful for reusing code.

3. **Taglib Directive**

- Declares a custom tag library that can be used in the JSP page.

- Syntax: *<%@ taglib uri="tag-library-uri" prefix="prefix" %>*

Example:

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

Here, the JSTL (JavaServer Pages Standard Tag Library) is included with the prefix c.

Custom Tag Libraries

Custom Tag Libraries are user-defined or pre-built collections of tags that extend the functionality of JSP. These tags are similar to HTML tags but perform specific tasks.

Why Use Custom Tag Libraries?

1. **Reusable Code**: Write logic once and use it multiple times.

2. **Simplified JSP**: Reduces Java code in JSP, making it cleaner and easier to read.

3. **Enhanced Functionality**: Add advanced features like loops, conditions, or database operations using custom tags.

Example of a Custom Tag Library

Using JSTL (JavaServer Pages Standard Tag Library):

JSTL is a widely used library with ready-made tags for common tasks like looping, conditions, and formatting.

Steps to Use JSTL:

1. Add the JSTL library to your project (commonly available in servers like Apache Tomcat).

2. Declare the library using the taglib directive:

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

3. Use JSTL tags in the JSP page.

Example: Looping with JSTL:

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

<html>

<head>

    <title>JSTL Example</title>

</head>

<body>
```

```
    <h1>List of Fruits</h1>
    <ul>
        <c:forEach var="fruit" items="${fruits}">
            <li>${fruit}</li>
        </c:forEach>
    </ul>
</body>
</html>
```

Explanation:

- ${fruits} is a list of fruits passed from the server.

- <c:forEach> loops through the list and displays each fruit.

Creating a Custom Tag

You can also create your own tag library for specific tasks.

Steps to Create a Custom Tag:

1. Write a Java Class for the tag logic.

2. Create a Tag Library Descriptor (TLD) file to define the tag.

3. Use the custom tag in your JSP page.

Example: Custom Tag to Greet a User

1. **Java Class**:

```
package mytags;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;
```

```
public class GreetTag extends SimpleTagSupport {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public void doTag() throws JspException, IOException {
        JspWriter out = getJspContext().getOut();
        out.print("Hello, " + name + "!");
    }
}
```

2. **TLD File**:

```
<taglib>
    <tlib-version>1.0</tlib-version>
    <short-name>MyTags</short-name>
    <uri>mytags</uri>
    <tag>
        <name>greet</name>
        <tag-class>mytags.GreetTag</tag-class>
        <body-content>empty</body-content>
        <attribute>
            <name>name</name>
```

```
            <required>true</required>
        </attribute>
    </tag>
</taglib>
```

3. **JSP Page**:

```
<%@ taglib uri="mytags" prefix="mt" %>
<html>
<head>
    <title>Custom Tag Example</title>
</head>
<body>
    <mt:greet name="John" />
</body>
</html>
```

Output:

Hello, John!

Conclusion

- Directives control the overall behavior of a JSP page, such as including files or declaring custom tags.

- Custom Tag Libraries simplify JSP development by encapsulating logic into reusable tags, making pages cleaner and more manageable. These concepts enhance productivity and make JSP more powerful for creating dynamic web applications.