MODULE 4

SERVLETS

- A **Servlet** is a small program that executes on the server side of a web connection.
- Just as applets extend the functionality of web browsers, servlets extend the functionality of web servers.
- Servlets are designed to handle requests, process data, and generate dynamic content for web pages.

When a user requests a static page, they enter a URL into the browser. The browser generates an HTTP request for a specific file, which the server maps to the appropriate resource. The server responds with an HTTP response containing the requested file. The response header includes metadata, such as the content type (e.g., text/html), which indicates the MIME type of the source web page.

Advantages of Servlets Over Traditional CGI

Java servlets offer several advantages over traditional Common Gateway Interface (CGI) scripts and other CGI-like technologies. These advantages include efficiency, convenience, power, portability, and cost-effectiveness.

1. Efficiency

- **Process Management**: Traditional CGI scripts start a new process for each HTTP request, which can be inefficient. The overhead of starting a new process can be significant, especially for fast operations. In contrast, servlets run within the Java Virtual Machine (JVM), and each request is handled by a lightweight Java thread rather than a heavyweight operating system process.
- **Memory Usage**: With traditional CGI, if multiple requests are made to the same CGI program simultaneously, the code for the CGI program is loaded into memory multiple times. Servlets, however, use multiple threads within a single JVM instance, meaning only one copy of the servlet class is loaded into memory regardless of the number of requests.

2. Convenience

- **Language Familiarity**: Developers familiar with Java do not need to learn another language, such as Perl, to write server-side programs. This reduces the learning curve and increases productivity.
- **Built-in Utilities**: Servlets provide extensive infrastructure for common tasks such as parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, and tracking sessions. These built-in utilities simplify development and reduce the need for custom code.

3. Power

- **Direct Server Communication**: Servlets can directly interact with the web server, which is not possible with regular CGI programs. This direct communication

simplifies tasks that involve looking up images or other data stored in standard locations.

- **Data Sharing**: Servlets can share data among each other, facilitating features like database connection pools. This capability makes it easier to implement complex functionalities and improve performance.
- **State Management**: Servlets can maintain information across multiple requests, simplifying session tracking and caching of previous computations. This state management capability is essential for creating dynamic, user-specific content.

4. Portability

- **Standardized API**: Servlets are written in Java and follow a well-defined, standardized API. This ensures that servlets developed for one server (e.g., I-Planet Enterprise Server) can run on other servers (e.g., Apache, Microsoft IIS) with minimal changes.
- **Cross-Platform Support**: Servlets are supported by almost every major web server, either directly or through a plugin. This wide support enhances their portability across different server environments.

5. Cost-Effectiveness

- **Inexpensive Deployment**: Many web servers that support servlets are available for free or at a low cost. This makes it economical to add servlet functionality to a web server. While many commercial-quality web servers are expensive, adding servlet support is usually free or inexpensive.
- **Low-Cost Infrastructure**: The use of free or inexpensive servers is suitable for personal use or low-volume websites. However, even for high-volume commercial sites, the cost of adding servlet support is minimal compared to the overall cost of the web server infrastructure.

Phases of the Servlet Life Cycle

The life cycle of a servlet is managed by the web container (e.g., Apache Tomcat) and includes the following phases:

Loading the Servlet Class

The servlet class is loaded when the web container receives the first request for the servlet. This loading happens only once during the servlet's life cycle.

Creating an Instance

The web container creates an instance of the servlet class. This instance creation also happens only once, ensuring a single instance of the servlet is used to handle multiple requests.

Initialization (init)

The web container invokes the init method to initialize the servlet. This method is called only once when the servlet instance is first loaded.

Syntax:

public void init(ServletConfig config) throws ServletException

Request Handling (service)

Each time a request is received, the web container calls the service method of the servlet. If the servlet has not been initialized, the container calls the init method before invoking the service method.

Syntax:

public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException

Destruction (destroy)

Before the servlet instance is removed from service, the web container calls the destroy method. This method allows the servlet to perform any cleanup, such as releasing resources or closing connections.

Syntax:

public void destroy()

Example of a Servlet

Below is an example of a simple servlet that handles HTTP GET requests and responds with a "Hello, World!" message.

```
import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;


public class HelloWorldServlet extends HttpServlet {


   // Initialization method

   @Override

   public void init() throws ServletException {

      // Initialization code here

      System.out.println("Servlet is being initialized");
```

```
    }

    // Service method to handle HTTP GET requests
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        // Set the response content type to text/html
        response.setContentType("text/html");

        // Write the response
        response.getWriter().println("<h1>Hello, World!</h1>");
    }

    // Destruction method
    @Override
    public void destroy() {
        // Cleanup code here
        System.out.println("Servlet is being destroyed");
    }
}
```

Explanation

- **Loading the Servlet Class**: When the servlet is first requested, the web container loads the HelloWorldServlet class.
- **Creating an Instance**: The web container creates an instance of HelloWorldServlet.
- **Initialization**: The init method is called, which can be used to perform initialization tasks.
- **Request Handling**: The doGet method handles HTTP GET requests. It sets the response content type to text/html and writes a simple "Hello, World!" message to the response.
- **Destruction**: When the servlet is about to be unloaded, the destroy method is called, allowing for any necessary cleanup.

Deployment Descriptor

A **Deployment Descriptor (DD)** is an XML file named web.xml located in the WEB-INF directory of a Java web application. This file configures the web application and controls the behavior of Java Servlets and JavaServer Pages (JSP). It defines mappings between URLs and the corresponding servlets that should handle requests to those URLs. It also specifies initialization parameters, servlet filters, listener classes, and other configurations.

Structure of web.xml

The web.xml file includes several important components:

1. **Header**: This includes the XML version and encoding.
2. **DOCTYPE**: Specifies the Document Type Definition (DTD) for the XML file.
3. **<web-app> element**: This is the root element of the deployment descriptor and contains other configuration elements.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/dtd/web-app_2_2.dtd">

<web-app>
  <servlet>
    <servlet-name>MyJavaServlet</servlet-name>
    <servlet-class>myPackage.MyJavaServletClass</servlet-class>
    <init-param>
      <param-name>parameter1</param-name>
      <param-value>735</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>MyJavaServlet</servlet-name>
    <url-pattern>/myServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

Components of web.xml

1. **Header**:

<?xml version="1.0" encoding="ISO-8859-1"?>

This line declares the XML version and the character encoding used in the file.

2. **DOCTYPE**:

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN" "http://java.sun.com/dtd/web-app_2_2.dtd">

This line defines the DTD, ensuring the XML file follows the structure specified by the DTD for web applications.

3. **<web-app> Element**:

```
<web-app>
  ...
</web-app>
```

The root element that contains all the configuration details of the web application.

4. **<servlet> Element**:

```
<servlet>
  <servlet-name>MyJavaServlet</servlet-name>
  <servlet-class>myPackage.MyJavaServletClass</servlet-class>
  <init-param>
    <param-name>parameter1</param-name>
    <param-value>735</param-value>
  </init-param>
</servlet>
```

- **<servlet-name>**: Specifies a name for the servlet. This name is used in the <servlet-mapping> element to map the servlet to a URL pattern.
- **<servlet-class>**: Specifies the fully qualified class name of the servlet.
- **<init-param>**: Specifies initialization parameters for the servlet. Each <init-param> contains a <param-name> and a <param-value>.

5. **<servlet-mapping> Element**:

```
<servlet-mapping>
  <servlet-name>MyJavaServlet</servlet-name>
  <url-pattern>/myServlet</url-pattern>
```

</servlet-mapping>

- **<servlet-name>**: Matches the name specified in the <servlet> element.
- **<url-pattern>**: Defines the URL pattern that will be mapped to the servlet. When a request matches this URL pattern, the specified servlet will handle the request.

Additional Configuration Options

The web.xml file can also include other configurations such as:

- **Servlet Filters**: To intercept requests and responses and perform filtering tasks.
- **Listener Classes**: To handle events in the web application lifecycle.
- **Error Pages**: To define custom error pages for different HTTP error codes.
- **Security Constraints**: To define security constraints for the web application.

Reading Data from Client in a Servlet

To read data sent from a client to a servlet, you need to handle HTTP requests, either using the GET or POST method. The data sent by the client can be accessed using the getParameter() or getParameterValues() methods of the HttpServletRequest object. Here's a detailed explanation and example:

Methods to Handle Client Data

1. **doGet() Method**:
   - Used when the client sends data via the HTTP GET method.
   - Typically handles query parameters appended to the URL.
2. **doPost() Method**:
   - Used when the client sends data via the HTTP POST method.
   - Handles form data sent in the body of the HTTP request.

Retrieving Parameters

- **getParameter(String name)**:
  - Retrieves the value of a single parameter as a String.
  - Returns null if the parameter does not exist.
  - Returns an empty string if the parameter exists but has no value.
- **getParameterValues(String name)**:
  - Retrieves the values of a parameter that has multiple values as a String array.
  - Useful for handling multi-select form elements or checkboxes.

HTML Form to Send Data to the Servlet:

```
<FORM ACTION="/myservlets.js2" METHOD="POST">
  Enter Email Address: <INPUT TYPE="TEXT" NAME="email">
  <INPUT TYPE="SUBMIT" VALUE="Submit">
</FORM>
```

Servlet Code to Handle the Request:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {

  // Handles GET requests
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    handleRequest(request, response);
  }

  // Handles POST requests
  public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    handleRequest(request, response);
  }

  // Common method to handle both GET and POST requests
  private void handleRequest(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    // Retrieve the email parameter from the request
    String email = request.getParameter("email");
```

```
    // Set the content type of the response
    response.setContentType("text/html");

    // Get the writer to write the response
    PrintWriter pw = response.getWriter();

    // Write the HTML response
    pw.println("<html>");
    pw.println("<head><title>Java Servlet</title></head>");
    pw.println("<body>");
    pw.println("<h1>My First Servlet</h1>");
    if (email != null && !email.isEmpty()) {
      pw.println("<p>My Email Address: " + email + "</p>");
    } else {
      pw.println("<p>No Email Address provided.</p>");
    }
    pw.println("</body>");
    pw.println("</html>");
  }
}
```

Explanation of the Servlet Code

1. **Import Statements**:
   - Import necessary classes from java.io, javax.servlet, and javax.servlet.http.
2. **Class Definition**:
   - MyServlet extends HttpServlet, allowing it to handle HTTP requests.
3. **doGet() Method**:
   - Handles HTTP GET requests by calling a common method handleRequest.
4. **doPost() Method**:
   - Handles HTTP POST requests by calling the same common method handleRequest.
5. **handleRequest() Method**:

- Retrieves the parameter "email" from the request using request.getParameter("email").
- Sets the content type of the response to text/html.
- Gets a PrintWriter to write the HTML response.
- Constructs and writes the HTML response, including the email address if provided.

Reading HTTP Request Headers in a Servlet

HTTP request headers are used by the client to provide information about the request context so that the server can tailor the response accordingly. A servlet can read these request headers to process the data component of the request. Here's a detailed explanation and example:

Common HTTP Request Headers and Their Uses

Accept: Specifies the MIME types that the client can handle. Example: Accept: image/jpeg, image/gif, */*.

Accept-Charset: Specifies the character sets that are acceptable. Example: Accept-Charset: utf-8.

Accept-Encoding: Specifies the content encodings that are acceptable. Example: Accept-Encoding: gzip, deflate.

Cookie: Returns stored cookies to the server. Example: Cookie: CustNum=12345.

Host: Specifies the domain name of the server and the TCP port number on which the server is listening. Example: Host: www.example.com.

Referer: Contains the URL of the web page from which the request was made. Example: Referer: http://www.example.com/index.html.

Reading HTTP Request Headers in a Servlet

To read HTTP request headers in a servlet, you use the getHeader() method of the HttpServletRequest object. This method requires the name of the header you want to read as an argument and returns the value of that header as a String.

Example Code

HTML Form to Send Data to the Servlet:

```
<!DOCTYPE html>
<html>
<head>
   <title>HTTP Header Example</title>
</head>
```

```
<body>
   <form action="/myHeaderServlet" method="get">
      <label for="email">Enter Email Address:</label>
      <input type="text" id="email" name="email">
      <input type="submit" value="Submit">
   </form>
</body>
</html>
```

Servlet Code to Handle HTTP Headers:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyHeaderServlet extends HttpServlet {

   // Handles GET requests
   public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      processRequest(request, response);
   }

   // Handles POST requests
   public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      processRequest(request, response);
   }

   // Common method to process both GET and POST requests
   private void processRequest(HttpServletRequest request, HttpServletResponse response)
```

```
        throws ServletException, IOException {

        // Set the response content type
        response.setContentType("text/html");

        // Get the PrintWriter to write the response
        PrintWriter out = response.getWriter();

        // Read some common headers
        String acceptHeader = request.getHeader("Accept");
        String hostHeader = request.getHeader("Host");
        String refererHeader = request.getHeader("Referer");
        String userAgentHeader = request.getHeader("User-Agent");

        // Write the response
        out.println("<html>");
        out.println("<head><title>HTTP Header Example</title></head>");
        out.println("<body>");
        out.println("<h1>HTTP Header Example</h1>");
        out.println("<p>Accept: " + acceptHeader + "</p>");
        out.println("<p>Host: " + hostHeader + "</p>");
        out.println("<p>Referer: " + refererHeader + "</p>");
        out.println("<p>User-Agent: " + userAgentHeader + "</p>");
        out.println("</body>");
        out.println("</html>");

        // Close the writer
        out.close();
    }
}
```

Sending Data to Client and Writing HTTP Response Headers

A servlet responds to a client's request by reading client data and HTTP request headers, processing information based on the request, and then sending an appropriate response. The servlet can send explicit data to the client by writing to the response stream or by sending HTTP response headers.

Sending Data Using the Response Stream

To send data back to the client, create an instance of the PrintWriter object and use the println() method to transmit information to the client.

Example Code for Sending Data

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyResponseServlet extends HttpServlet {

   // Handles GET requests
   public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {

      // Set the response content type
      response.setContentType("text/html");

      // Get the PrintWriter to write the response
      PrintWriter out = response.getWriter();

      // Write the HTML response
      out.println("<html>");
      out.println("<head><title>My Response</title></head>");
      out.println("<body>");
      out.println("<h1>This is my response</h1>");
```

```
        out.println("</body>");
        out.println("</html>");

        // Close the writer
        out.close();
    }
}
```

Writing to the HTTP Response Header

To write to the HTTP response header, use methods such as setStatus() and setHeader() of the HttpServletResponse object.

Example Code for Setting Response Headers

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyHeaderResponseServlet extends HttpServlet {

    // Handles GET requests
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // Set the status code
        response.setStatus(HttpServletResponse.SC_OK);

        // Set some headers
        response.setHeader("Content-Type", "text/html");
        response.setHeader("Custom-Header", "CustomValue");

        // Get the PrintWriter to write the response
        PrintWriter out = response.getWriter();
```

```
        // Write the HTML response
        out.println("<html>");
        out.println("<head><title>My Header Response</title></head>");
        out.println("<body>");
        out.println("<h1>Response with Headers</h1>");
        out.println("</body>");
        out.println("</html>");

        // Close the writer
        out.close();
    }
}
```

Cookies in Servlets

Cookies are small text files stored on the client machine to track information about the user. Servlets can create and manipulate cookies, enabling persistent client-side state.

How Cookies Work

1. **Server sends cookies to the client**: The server sends cookies via HTTP headers.
2. **Client stores cookies**: The browser stores the cookies locally.
3. **Client returns cookies**: On subsequent requests, the browser sends stored cookies to the server.

Setting Cookies in Servlets

To set cookies, follow these steps:

1. **Creating a Cookie Object**: Use the Cookie constructor to create a cookie with a name and a value.

```
Cookie cookie = new Cookie("key", "value");
```

2. **Setting the Maximum Age**: Define how long the cookie is valid.

```
cookie.setMaxAge(60 * 60 * 24); // 24 hours
```

3. **Adding the Cookie to the Response**: Add the cookie to the HTTP response headers.

```
response.addCookie(cookie);
```

Example: Writing a Cookie

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloForm extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      // Create a cookie
      Cookie myCookie = new Cookie("userId", "123");

      // Set the maximum age of the cookie
      myCookie.setMaxAge(60 * 60); // 1 hour

      // Add the cookie to the response
      response.addCookie(myCookie);

      // Set the content type of the response
      response.setContentType("text/html");

      // Write the response
      PrintWriter out = response.getWriter();
      out.println("<html>\n" +
            "<head><title>My Cookie</title></head>\n" +
            "<body>\n" +
            "<h1>My Cookie</h1>\n" +
            "<p>Cookie Written</p>\n" +
            "</body></html>");
   }
}
```

Reading Cookies in Servlets

To read cookies, follow these steps:

1. **Retrieve Cookies**: Call the getCookies() method of HttpServletRequest.

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

2. **Access Each Cookie**: Loop through the array of Cookie objects and use getName() and getValue() methods to access each cookie's name and value.

```
if (cookies != null) {
   for (Cookie cookie : cookies) {
      String name = cookie.getName();
      String value = cookie.getValue();
      // Process the cookie
   }
}
```

Example: Reading Cookies

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ReadCookies extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      // Retrieve all cookies
      Cookie[] cookies = request.getCookies();

      // Set the content type of the response
      response.setContentType("text/html");

      // Write the response
      PrintWriter out = response.getWriter();
      out.println("<html>\n" +
```

```
            "<head><title>Reading Cookies Example</title></head>\n" +
            "<body>\n" +
            "<h2>Found Cookies Name and Value</h2>");

    if (cookies != null) {
      for (Cookie cookie : cookies) {
        out.println("Name: " + cookie.getName() + ", ");
        out.println("Value: " + cookie.getValue() + "<br/>");
      }
    } else {
      out.println("<h2>No cookies found</h2>");
    }

    out.println("</body></html>");
  }
}
```

JavaServer Pages (JSP)

JavaServer Pages (JSP) is a technology for developing web pages that support dynamic content. It provides a simplified way to create and manage content using HTML and Java combined. JSP is designed to provide a high-level abstraction over Java servlets.

Key Characteristics of JSP

- **Server-Side Processing**: JSP runs on the server, generating dynamic content before sending it to the client.
- **Ease of Use**: JSP is easier to write and manage than Java servlets because it embeds Java code directly within HTML.

Lifecycle Methods in JSP

There are three key lifecycle methods in JSP:

1. **jspInit()**:
    - Called once when the JSP is first loaded.
    - Similar to the init() method in servlets.
    - Used for initialization tasks.

```
public void jspInit() {
```

```
    // Initialization code here
}
```

2. **service()**:
    - Handles client requests.
    - This method is called for every client request.
    - Similar to the doGet() and doPost() methods in servlets.

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    // Request handling code here
}
```

3. **jspDestroy()**:
    - Called once when the JSP is unloaded.
    - Similar to the destroy() method in servlets.
    - Used for cleanup tasks.

```
public void jspDestroy() {
    // Cleanup code here
}
```

Example JSP Program

Here's a simple JSP example that demonstrates the use of HTML and embedded Java code to display a message and current date and time.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>Welcome to JSP</title>
</head>
<body>
    <h1>Welcome to JSP</h1>
    <p>Current date and time is: <%= new java.util.Date() %></p>
    <%
```

```
    // Java code block
    String name = "User";
    out.println("<p>Hello, " + name + "!</p>");
  %>
</body>
</html>
```

Explanation

- **Directives (<%@ ... %>)**: The page directive is used to set the language, content type, and page encoding.
- **Scriptlet (<% ... %>)**: This tag is used to embed Java code within HTML. Code inside these tags will be executed when the page is requested.
- **Expression (<%= ... %>)**: This tag is used to output the result of a Java expression directly to the client. In this example, it outputs the current date and time.

JSP Tags

JSP tags are used to embed Java code into HTML content, allowing the dynamic generation of web pages. These tags are processed on the server before the response is sent to the client.

1. **Comment Tags**
    - **Syntax**: <%-- comment --%>
    - **Purpose**: Used to add comments within the JSP code that are not sent to the client.
    - **Example**:

```
<%-- This is a comment --%>
```

2. **Declaration Tags**
    - **Syntax**: <%! declaration %>
    - **Purpose**: Used to declare variables, methods, or classes that are used within the JSP page.
    - **Example**:

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

3. **Directive Tags**
    - **Syntax**: <%@ directive attribute="value" %>
    - **Purpose**: Provide global information about the JSP page, such as importing Java packages, including files, or defining custom tag libraries.

- **Common Directives**:
  - **Page Directive**: Used to import Java classes, set page encoding, etc.

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

  - **Include Directive**: Includes content from another file.

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

  - **Taglib Directive**: Declares a tag library that can be used in the JSP page.

```
<%@ taglib uri="myTags.tld" prefix="mytag" %>
```

4. **Expression Tags**
   - **Syntax**: <%= expression %>
   - **Purpose**: Evaluates the expression and converts the result to a string, which is then included in the output.
   - **Example**:

```
<%= new java.util.Date() %>
```

5. **Scriptlet Tags**
   - **Syntax**: <% code %>
   - **Purpose**: Contains Java code that is executed during the request processing.
   - **Example**:

```
<%
   int number = 10;
   if (number > 5) {
      out.println("Number is greater than 5");
   }
%>
```

Example JSP Page

Below is an example JSP page that demonstrates the use of various JSP tags:

```
<%@ page import="java.util.Date" %>
<html>
<head>
   <title>JSP Example</title>
</head>
```

```
<body>
   <%-- This is a comment --%>
   <%! int age = 25; %>
   <p>Your age is: <%= age %></p>
   <p>Current date and time: <%= new Date() %></p>
   <%
      for (int i = 0; i < 5; i++) {
         out.println("Count: " + i + "<br>");
      }
   %>
</body>
</html>
```

Variables and Objects in JSP

In a JSP page, you can declare variables and objects using declaration tags. These variables and objects are available throughout the JSP page. Here's how you can declare and use them:

Declaration Example

```
<%@ page import="java.util.Date" %>
<html>
<head>
   <title>JSP Programming</title>
</head>
<body>
   <%! int age = 29; %>
   <%! Date currentDate = new Date(); %>
   <p>Your age is: <%= age %></p>
   <p>Current date and time: <%= currentDate %></p>
</body>
</html>
```

Explanation

- **Declaration Tag (<%! ... %>)**: Used to declare the variable age and the object currentDate.

- **Expression Tag (<%= ... %>)**: Used to output the values of age and currentDate directly into the HTML content.

Declaring and Using Methods in JSP

In JSP, methods can be declared in the same way they are in regular Java programs, but they need to be placed within JSP declaration tags (<%! ... %>). These methods can then be called within the JSP using expression tags (<%= ... %>).

Here is an example to illustrate how methods are declared and used in JSP:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

  <title>JSP Programming</title>

</head>

<body>

  <%!

    // Method to add two numbers

    int add(int n1, int n2) {

      int c;

      c = n1 + n2;

      return c;

    }

  %>

  <p>Addition of two numbers: <%= add(45, 46) %></p>

</body>

</html>
```

Explanation:

1. **Page Directive (<%@ ... %>)**: Specifies the language as Java and sets the content type and page encoding.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
```

2. **Declaration Tag (<%! ... %>)**: Used to declare the add method within the JSP page.

```
<%!
   int add(int n1, int n2) {
      int c;
      c = n1 + n2;
      return c;
   }
%>
```

3. **Expression Tag (<%= ... %>)**: Calls the add method and outputs the result within the HTML content.

```
<p>Addition of two numbers: <%= add(45, 46) %></p>
```

Detailed Breakdown:

- **Declaration Tag**:
  - The add method is defined within the declaration tag. This makes the method available to the entire JSP page.
  - The method add takes two integers as parameters, adds them, and returns the result.
- **Expression Tag**:
  - The expression tag is used to call the add method with the values 45 and 46.
  - The result of the method call (91) is embedded directly into the HTML content of the JSP page.

Control Statements in JSP

Control statements in JSP allow you to alter the flow of your program based on conditions. This feature enables the creation of truly dynamic content for web applications. The two main control statements in JSP are the if statement and the switch statement, both of which are also used in Java programming.

if Statement

The if statement is used to execute a block of code only if a specified condition is true. Optionally, an else block can be included to execute code if the condition is false.

Example Using if Statement

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
```

```
<!DOCTYPE html>
<html>
<head>
   <title>JSP Programming</title>
</head>
<body>
   <%! int grade = 26; %> <!-- Declaring a variable in JSP -->


   <% if (grade > 69) { %>
      <p>You Passed!</p>
   <% } else { %>
      <p>Better Luck Next Time</p>
   <% } %>
</body>
</html>
```

Explanation:

1. **Page Directive**: Specifies the page language as Java and sets the content type and page encoding.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
```

2. **Declaration Tag**: Declares an integer variable grade and initializes it with a value of 26.

```
<%! int grade = 26; %>
```

3. **Scriptlet Tags**: Contains the if statement to check the value of grade and output different content based on the condition.

```
<% if (grade > 69) { %>
   <p>You Passed!</p>
<% } else { %>
   <p>Better Luck Next Time</p>
<% } %>
```

switch Statement

The switch statement allows the execution of one block of code among many based on the value of a variable or expression.

Example Using switch Statement

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
  <title>JSP Programming</title>
</head>
<body>
  <%! int grade = 85; %> <!-- Declaring a variable in JSP -->

  <%
  String result;
  switch (grade / 10) {
    case 10:
    case 9:
      result = "A";
      break;
    case 8:
      result = "B";
      break;
    case 7:
      result = "C";
      break;
    case 6:
      result = "D";
      break;
    default:
      result = "F";
```

```
        break;
    }
    %>

    <p>Your grade is: <%= result %></p>
</body>
</html>
```

Explanation:

1. **Page Directive**: Specifies the page language as Java and sets the content type and page encoding.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
```

2. **Declaration Tag**: Declares an integer variable grade and initializes it with a value of 85.

```
<%! int grade = 85; %>
```

3. **Scriptlet Tags**: Contains the switch statement to determine the letter grade based on the value of grade.

```
<%
String result;
switch (grade / 10) {
    case 10:
    case 9:
        result = "A";
        break;
    case 8:
        result = "B";
        break;
    case 7:
        result = "C";
        break;
    case 6:
        result = "D";
```

```
        break;
    default:
        result = "F";
        break;
}
%>
```

4. **Expression Tag**: Outputs the letter grade to the HTML content.

```
<p>Your grade is: <%= result %></p>
```

Looping Statements in JSP

Loops in JSP are almost identical to loops in Java programming, with the added capability of repeating HTML tags within the loop. The three common types of loops used in JSP are:

1. **For Loop**
2. **While Loop**
3. **Do-While Loop**

These loops are essential, especially in JSP database programs where repetitive tasks such as displaying rows from a database are required.

Example Using for Loop

The following example demonstrates a simple for loop in JSP that outputs "Hello World" ten times:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>For Loop Example</title>
</head>
<body>
    <%
    for (int i = 0; i < 10; i++) {
    %>
    <p>Hello World</p>
    <%
```

```
}
%>
</body>
</html>
```

Explain Requset String generated by browser. how to read a request string in jsp?

1. A browser generate requst string whenever the submit button is selected. The user requests the string consists of URL and the query the string. Example of request string: http://www.jimkeogh.com/jsp/?fname=" Bob" & lname ="Smith"

2. Your jsp programneeds to parse the query string to extract the values of fields that are to be processed by your program. You can parse the query string by using the methods of the request object.

3. getParameter(Name) method used to parse a value of a specific field that are to be processed by your program

4. code to process the requset string

5. Copying from multivalued field such as selection list field can be tricky multivalued fields are handled by using getParameterValues()

6. Other than requset string url has protocols, port no, the host name

1. Request Parameters (Query String)

When a form is submitted, the browser sends a request to the server with parameters encoded in the URL (query string) or sometimes in the request body. Here's an example of a request string:

http://www.example.com/servlet?fname=Bob&lname=Smith

To retrieve these parameters in a JSP, you use the request.getParameter("parameterName") method.

2. Parsing Request Parameters in JSP

To parse and use the parameters sent in the request, you typically do something like this in your JSP:

```
<%@ page language="java" %>
<%
String firstName = request.getParameter("fname");
String lastName = request.getParameter("lname");
%>
```

Here, fname and lname are the names of the parameters as specified in the query string.

3. Handling Multivalued Parameters

If a parameter can have multiple values (like checkboxes or multi-select dropdowns), you use request.getParameterValues("parameterName"):

String[] selectedValues = request.getParameterValues("fieldName");

4. Cookies in JSP

Cookies are small pieces of data stored on the client-side by the browser. In JSP, you can create and read cookies using the Cookie class and methods provided by the HttpServletRequest object.

Creating a Cookie:

```
<%@ page language="java" %>
<%
String cookieName = "EMPID";
String cookieValue = "AN2536";

Cookie myCookie = new Cookie(cookieName, cookieValue);
response.addCookie(myCookie);
%>
```

Reading a Cookie:

```
<%@ page language="java" %>
<%
String cookieName = "EMPID";
String cookieValue = "";

Cookie[] cookies = request.getCookies();
if (cookies != null) {
   for (Cookie cookie : cookies) {
      if (cookie.getName().equals(cookieName)) {
         cookieValue = cookie.getValue();
         break;
      }
   }
}
```

%>

In this example, request.getCookies() returns an array of Cookie objects sent by the client. You iterate through these cookies to find the one with the name "EMPID".

Step-by-Step Guide to Configure Tomcat

1. Download Tomcat

- Visit the Apache Tomcat website (jakarta.apache.org or tomcat.apache.org).
- Navigate to the download section and select the latest stable release.
- Choose the "Binaries" distribution, which includes the executable files you need.

2. Install Tomcat

- Create a folder for Tomcat. For example, C:\tomcat (on Windows) or /opt/tomcat (on Unix-like systems).
- Download the zip file for Tomcat and unzip it into your chosen folder (C:\tomcat or /opt/tomcat).

3. Configure Environment Variables (Windows)

- Set JAVA_HOME environment variable:
  - Right-click on "My Computer" or "This PC" and select "Properties".
  - Go to "Advanced system settings" -> "Environment Variables".
  - Under "System Variables", click "New" and add JAVA_HOME with the path to your JDK installation (e.g., C:\Program Files\Java\jdk1.8.0_291).

4. Configure Environment Variables (Unix-like systems)

- Edit your shell profile file (e.g., .bash_profile, .bashrc, .profile):

export JAVA_HOME=/path/to/your/jdk

export PATH=$PATH:$JAVA_HOME/bin

- Source the profile file to apply changes: source ~/.bash_profile or source ~/.bashrc.

5. Modify Tomcat Startup Script

- Navigate to C:\tomcat\bin or /opt/tomcat/bin.
- Edit the startup script (startup.bat for Windows, startup.sh for Unix-like systems).
- Set the JAVA_HOME variable to your JDK installation path:

set JAVA_HOME=C:\path\to\your\jdk (for Windows)

JAVA_HOME=/path/to/your/jdk (for Unix-like systems)

6. Start Tomcat

- Open a command prompt (Windows) or terminal (Unix-like systems).

- Navigate to C:\tomcat\bin or /opt/tomcat/bin.
- Run startup.bat (Windows) or ./startup.sh (Unix-like systems) to start Tomcat.

7. Verify Tomcat Installation

- Open your web browser.
- Enter http://localhost:8080 in the address bar.
- If Tomcat is configured correctly, you should see the Tomcat home page indicating that Tomcat is running.

Notes:

- Ensure that your firewall allows connections to Tomcat on port 8080.
- Tomcat can be further configured by editing files in C:\tomcat\conf or /opt/tomcat/conf.
- Stop Tomcat using shutdown.bat (Windows) or ./shutdown.sh (Unix-like systems) in the bin directory.

Creating and Using Session Objects in JSP

1. Creating a Session Object and Setting Attributes

In JSP, a session object is automatically created the first time a JSP page calls request.getSession() to obtain a session. Here's how you can set session attributes:

```
<html>
<head>
<title>JSP Session</title>
</head>
<body>
<%
String attributeName = "Product";
String attributeValue = "1234";

// Set session attribute
session.setAttribute(attributeName, attributeValue);
%>
<p>Session attribute set successfully.</p>
</body>
</html>
```

- **Explanation**:
    - request.getSession(): Retrieves the current session associated with the request, creating a new one if necessary.
    - session.setAttribute(attributeName, attributeValue): Sets an attribute (Product with value 1234) in the session object.

2. Reading Session Attributes

To read session attributes in subsequent requests or pages within the same session, you can use methods like session.getAttributeNames() and session.getAttribute(attributeName):

```
<html>
<head>
<title>JSP Session</title>
</head>
<body>
<%
// Get all attribute names in the session
Enumeration<String> attributeNames = session.getAttributeNames();

while (attributeNames.hasMoreElements()) {
   String attributeName = attributeNames.nextElement();
   String attributeValue = (String) session.getAttribute(attributeName);
%>
   <p>Attribute Name: <%= attributeName %></p>
   <p>Attribute Value: <%= attributeValue %></p>
<%
}
%>
</body>
</html>
```

- **Explanation**:
    - session.getAttributeNames(): Returns an Enumeration of all attribute names stored in the session.

- session.getAttribute(attributeName): Retrieves the value of the attribute specified by attributeName from the session.

Important Points:

- **Session Creation**: Sessions are managed by the servlet container (like Apache Tomcat) and are created automatically when request.getSession() is called for the first time in a session.
- **Session ID**: Each session is identified by a unique session ID, typically stored as a cookie in the user's browser. This allows the server to associate subsequent requests from the same client with the correct session.
- **Session Attributes**: You can store any Java object as a session attribute (setAttribute) and retrieve it (getAttribute) across multiple requests until the session expires or is invalidated.

Session Management Best Practices:

- Always handle session attributes securely, especially when dealing with sensitive information.
- Invalidate or expire sessions (session.invalidate()) when they are no longer needed to free up resources.
- Be mindful of session timeout settings (<session-timeout> in web.xml) to manage session lifecycle effectively.