Module 8) Web Technologies in Java

HTML Tags: Anchor, Form, Table, Image, List Tags, Paragraph, Break, Label

Theory:

QUE.1 Introduction to HTML and its structure.

ANS. **Introduction to HTML:**

- HTML stands for **HyperText Markup Language**.
- It is the **standard language** used to create and design webpages.
- HTML defines the **structure and content** of a webpage using **tags** (e.g., <p>, <h1>, <a>).
- Web browsers read HTML files and display them as formatted text, images, links, forms, etc.

Structure of an HTML Document:

```
<!DOCTYPE html>
<html>
<head>
    <title>Page Title</title>
</head>
<body>
    <h1>Heading</h1>
    <p>This is a paragraph</p>
</body>
</html>
```

QUE.2 Explanation of key tags:

 <a>: Anchor Tag

- Used to create **hyperlinks**.
- Syntax: Link Text
- Example: Google

 <form>: Form Tag

- Used to **collect user input**.
- Contains form elements like input fields, checkboxes, buttons, etc.
- Example:

  <form action="submit.php" method="post">

<input type="text" name="username">

<input type="submit" value="Submit">

</form>

 <table>: Table Tag

- Used to **display data in rows and columns**.
- Contains <tr> (row), <th> (header cell), <td> (data cell).
- Example:

<table border="1">

<tr><th>Name</th><th>Age</th></tr>

<tr><td>Ayush</td><td>20</td></tr>

</table>

 <img>: Image Tag

- Used to **embed images** in a webpage.
- Requires src (source) and alt (alternate text) attributes.
- Example: <img src="image.jpg" alt="Sample Image" width="200">

 **List Tags: <ul>, <ol>, **

- : Creates an **unordered (bulleted) list**.
- : Creates an **ordered (numbered) list**.
- : Defines each **list item**.
- Example:

Apple

Mango

Step 1

Step 2

 <p>: Paragraph Tag

- Used to define a **paragraph of text**.
- Automatically adds line breaks before and after the text.

- Example: This is a paragraph.

 <**br>: Line Break Tag**

- Inserts a **single line break**.
- It is an **empty tag** (no closing tag).
- Example: Hello
World

 <**label>: Label Tag**

- Provides a **caption** for form input elements.
- Improves **accessibility** (can be clicked to focus the input).
- Example:

  <label for="name">Name:</label>

  <input type="text" id="name" name="name">

Lab Exercise:

QUE.1 Create a webpage that includes:

o A navigation menu with anchor tags.

o A form with input fields, labels, and a submit button.

o A table that displays user data.

o Images with appropriate alt text.

o Both ordered and unordered lists.

ANS.

```
<!DOCTYPE html>
<html>
<head>
   <title>Sample Webpage</title>
</head>
<body>

   <! -- Navigation Menu with Anchor Tags -->
   <h2>Navigation Menu</h2>
   <a href="#home">Home</a> |
   <a href="#about">About</a> |
   <a href="#contact">Contact</a>
   <hr>

   <!-- Form with Labels, Input Fields & Submit Button -->
   <h2>User Form</h2>
   <form>
      <label for="name">Name:</label>
      <input type="text" id="name" name="name"><br><br>
```

```
    <label for="email">Email:</label>
    <input type="email" id="email" name="email"><br><br>

    <input type="submit" value="Submit">
  </form>
  <hr>

  <!-- Table for User Data -->
  <h2>User Data</h2>
  <table border="1">
    <tr>
      <th>Name</th>
      <th>Email</th>
    </tr>
    <tr>
      <td>Ayush</td>
      <td>ayush@example.com</td>
    </tr>
    <tr>
      <td>Buddy</td>
      <td>buddy@example.com</td>
    </tr>
  </table>
  <hr>

  <!-- Image with Alt Text -->
  <h2>Sample Image</h2>
  <img src="https://via.placeholder.com/150" alt="Sample Image">
  <hr>

  <!-- Ordered and Unordered Lists -->
  <h2>Lists Example</h2>
  <p>Unordered List:</p>
  <ul>
    <li>Pizza</li>
    <li>Burger</li>
    <li>Dabeli</li>
  </ul>

  <p>Ordered List:</p>
  <ol>
    <li>Step One</li>
    <li>Step Two</li>
    <li>Step Three</li>
  </ol>

</body>
</html>
```

==========

CSS: Inline CSS, Internal CSS, External CSS

Theory:

QUE.1 Overview of CSS and its importance in web design.

ANS. **Overview of CSS:**

- CSS stands for **Cascading Style Sheets**.
- It is used to **style and format** HTML elements (colors, fonts, spacing, layouts, etc.).
- CSS separates **content (HTML)** from **presentation (design)**.
- It supports **inline, internal, and external** styles.
- Modern CSS also provides features like **media queries, flexbox, grid, animations**.

Importance of CSS in Web Design:

1. **Improves Appearance** → Makes webpages attractive with colors, fonts, and layouts.
2. **Consistency** → Same stylesheet can be applied to multiple pages for a uniform design.
3. **Separation of Concerns** → Keeps design (CSS) separate from structure (HTML).
4. **Responsive Design** → Helps in creating websites that adapt to mobiles, tablets, and desktops.
5. **Saves Time** → External CSS can style thousands of pages by editing just one file.
6. **Accessibility & User Experience** → Enhances readability and navigation for users.

QUE.2 Types of CSS:

o Inline CSS: Directly in HTML elements.

ANS. Applied **directly inside an HTML element** using the style attribute.

Affects only that specific element.

Example:

```
<p style="color:blue; font-size:18px;">This is inline CSS</p>
```

o Internal CSS: Inside a <style> tag in the head section.

ANS. Written inside a <style> tag in the **<head> section** of the HTML document.

Styles apply to the whole page but only within that file.

Example:

```
<head>
  <style>
```

```
    p { color: green; font-size: 20px; }
  </style>
</head>
```

o External CSS: Linked to an external file.

ANS. Written in a **separate .css file** and linked to the HTML file using <link>.

Best for large projects and multiple pages.

Example:

```
<head>
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
```

style.css file:

```
p { color: red; font-size: 22px; }
```

Lab Exercise:

QUE.1 Create a webpage where:

o You apply inline CSS to an element.

o Use internal CSS for another element.

o Link an external CSS file to style other elements.

ANS.

📁 File 1: **index.html**

```
<!DOCTYPE html>
<html>
<head>
  <title>CSS Types Example</title>

  <!-- Internal CSS -->
  <style>
    h2 {
      color: green;
      text-align: center;
    }
  </style>

  <!-- External CSS -->
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
```

```
    <!-- Inline CSS -->
    <h1 style="color: blue; font-size: 28px;">This is Inline CSS Example</h1>

    <!-- Internal CSS Applied -->
    <h2>This heading is styled using Internal CSS</h2>

    <!-- External CSS Applied -->
    <p>This paragraph is styled using External CSS file.</p>

</body>
</html>
```

📁 File 2: **style.css**

```
p {
    color: red;
    font-size: 20px;
    font-family: Arial, sans-serif;
}
```

==========

CSS: Margin and Padding

Theory:

QUE.1 Definition and difference between margin and padding.

ANS. **Margin**:

- The **outer space** around an element.
- Creates distance between the element and other elements.

Padding:

- The **inner space** between an element's content and its border.
- Pushes the content **inside** the element.

Aspect	Margin (Outside)	Padding (Inside)
Position	Space **outside** the border	Space **inside** the border
Effect	Pushes element away from others	Pushes content away from border
Example Use	To create gap between two divs	To create space around text inside a box

QUE.2 How margins create space outside the element and padding creates space inside.

ANS. **Margins**:

- Control the space **outside** an element.
- Example:

  div {

    margin: 20px; /* Creates 20px gap around the element */

  }

Padding:

- Controls the space **inside** an element (between content and border).
- Example:

  div {

    padding: 20px; /* Creates 20px space inside the element */

  }

Lab Exercise:

QUE.1 Create a webpage and use CSS to demonstrate:

o Margin applied to an element.

o Padding applied to a div.

o The effect of different margin and padding values on the layout.

```
<!DOCTYPE html>
<html>
<head>
  <title>Margin and Padding Example</title>
  <style>
    /* Box with margin */
    .margin-box {
      background-color: lightblue;
      margin: 30px;   /* space outside the box */
      padding: 10px;
      border: 2px solid blue;
    }

    /* Box with padding */
    .padding-box {
      background-color: lightgreen;
      margin: 10px;
      padding: 40px;  /* space inside the box */
      border: 2px solid green;
    }
```

```
      /* Comparison box */
      .comparison {
         background-color: lightcoral;
         margin: 50px;
         padding: 50px;
         border: 2px solid red;
      }
   </style>
</head>
<body>

   <h1>Margin and Padding Demonstration</h1>

   <!-- Margin Example -->
   <div class="margin-box">
      This box has <b>30px margin</b> outside and <b>10px padding</b> inside.
   </div>

   <!-- Padding Example -->
   <div class="padding-box">
      This box has <b>40px padding</b> inside, making content push away from border.
   </div>

   <!-- Comparison Example -->
   <div class="comparison">
      This box has <b>50px margin</b> and <b>50px padding</b> together.
   </div>

</body>
</html>
```

==========

CSS: Pseudo-Class

Theory:

QUE.1 Introduction to CSS pseudo-classes like :hover, :focus, :active, etc.

ANS. **Pseudo-classes** define the **special state of an element**.

- They are used to style elements based on user interaction.
- Common pseudo-classes:
    1. **:hover** → Applies style when the user hovers the mouse over an element.
    2. **:focus** → Applies style when an element (like input) is selected or active.
    3. **:active** → Applies style when an element (like a button or link) is being clicked.
    4. **:visited** → Styles a link after it has been visited.

5. **:first-child, :last-child** → Style based on element position in hierarchy.

QUE.2 Use of pseudo-classes to style elements based on their state.

ANS. Pseudo-classes change element styles dynamically based on **user actions**.

Examples:

- **:hover** → Highlight links or buttons when mouse is over them.
- **:focus** → Highlight form fields when selected for typing.
- **:active** → Give feedback when a button/link is pressed.

This improves **interactivity** and **user experience** in web design.

Lab Exercise:

QUE.1 Create a navigation menu and use pseudo-classes to:

o Change the color of links on hover.

o Style form inputs when they are focused.

```
<!DOCTYPE html>
<html>
<head>
  <style>
    nav a { text-decoration:none; color:black; padding:5px; }
    nav a:hover { color:red; }   /* Hover effect */
    input:focus { border:2px solid blue; } /* Focus effect */
  </style>
</head>
<body>
  <nav>
    <a href="#">Home</a> |
    <a href="#">About</a> |
    <a href="#">Contact</a>
  </nav>

  <form>
    <input type="text" placeholder="Enter name">
    <input type="submit" value="Go">
  </form>
</body>
</html>
```

==========

CSS: ID and Class Selectors

Theory:

QUE.1 Difference between id and class in CSS.

ANS.

Feature	id	class
Definition	A unique identifier used to style a single element.	A reusable name that can be applied to multiple elements.
Selector in CSS	Uses # (hash). Example: #header { }	Uses . (dot). Example: .title { }
Uniqueness	Each element should have **only one id**.	Multiple elements can share the same class.
Specificity	Higher specificity than class.	Lower specificity compared to id.
Usage	For **unique styling** like logo, navigation bar, footer.	For **common styling** like headings, buttons, paragraphs.

QUE.2 Usage scenarios for id (unique) and class (reusable).

ANS. **id (unique use cases):**

- Assigning styles to a unique element like #main-header, #footer, #login-button.
- Useful in JavaScript for targeting a single element.

class (reusable use cases):

- Styling multiple elements with the same look (e.g., .btn, .card, .title).
- Grouping elements for consistent design (e.g., all product boxes).

Lab Exercise:

QUE.1 Create a webpage where:

o You apply an id to an element and style it uniquely.

o Use class to apply the same style to multiple elements.

ANS.

Code Example (HTML + CSS):

```
<!DOCTYPE html>
<html>
<head>
```

```
  <title>ID and Class Example</title>
  <style>
    /* id selector (unique) */
    #main-heading {
      color: blue;
      font-size: 28px;
      text-align: center;
    }

    /* class selector (reusable) */
    .highlight {
      color: green;
      font-weight: bold;
    }
  </style>
</head>
<body>

  <!-- id applied (unique style) -->
  <h1 id="main-heading">Welcome to My Webpage</h1>

  <!-- class applied (reusable style) -->
  <p class="highlight">This paragraph is highlighted using class.</p>
  <p class="highlight">Another paragraph with the same class style.</p>
  <p>This paragraph has no special class.</p>

</body>
</html>
```

==========

Introduction to Client-Server Architecture

Theory:

QUE.1 Overview of client-server architecture.

Ans. Client-Server architecture is a network model where two entities, client and server, interact with each other.

The **client** requests services or resources, while the **server** provides those services or resources.

Communication usually happens over the internet using protocols like **HTTP/HTTPS, FTP, SMTP, etc.**

Example: A web browser (client) requests a webpage from a web server, and the server sends back the webpage data.

QUE.2 Difference between client-side and server-side processing.

Ans.

Aspect	Client-Side	Server-Side
Execution	Code runs on the client's device (browser).	Code runs on the server machine.
Languages Used	HTML, CSS, JavaScript	PHP, Java, Python, Node.js, etc.
Speed	Faster as no server request is needed.	Slower due to network and server response.
Security	Less secure (code is visible to user).	More secure (hidden from user).
Examples	Form validation using JavaScript.	Login authentication using database.

QUE.3 Roles of a client, server, and communication protocols.

Ans. ➔**Client**:

- Sends request for service/data.
- Acts as a user interface (e.g., browser, app).

➔**Server**:

- Processes client requests.
- Retrieves or stores data.
- Sends the response back to the client.

➔**Communication Protocols**:

- Define rules for communication between client and server.
- Examples: **HTTP/HTTPS (web), FTP (file transfer), SMTP/IMAP (emails), TCP/IP (data transfer)**.

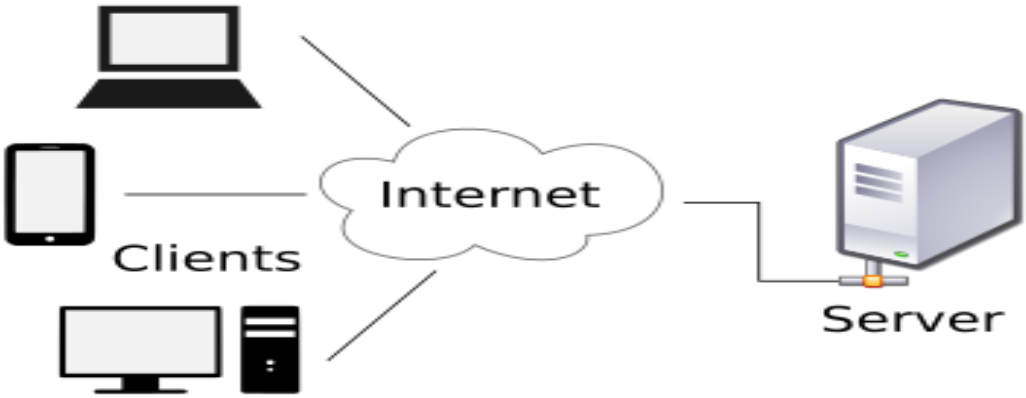
Lab Exercise:

QUE.1 Create a diagram explaining client-server communication flow and explain how a request is processed by the server and sent back to the client.

Ans.

Explanation of Flow

1. **Client initiates request** – Example: User enters a URL in browser.
2. **Request sent to server** – The request travels via internet using protocols (HTTP/HTTPS).
3. **Server processes request** – The server fetches data from application logic or database.
4. **Server sends response** – Response is prepared (HTML, JSON, images, etc.) and sent back.
5. **Client displays response** – Browser renders the received content to the user.



==========

HTTP Protocol Overview with Request and Response Headers

Theory:

QUE.1 Introduction to the HTTP protocol and its role in web communication.

Ans. **HTTP (Hypertext Transfer Protocol)** is the standard protocol used for communication between a **web client (browser)** and a **web server**.

➔It follows a **request-response model**:

- The client sends an **HTTP request** to the server.
- The server processes it and returns an **HTTP response**.

➔HTTP is **stateless**: every request is independent (does not store user state by default).

➔Role in Web Communication:

- Enables transfer of text, images, audio, video, and other resources.
- Uses methods like **GET, POST, PUT, DELETE** for different operations.
- Works on **TCP/IP** and usually on port **80 (HTTP)** or **443 (HTTPS)**.

QUE.2 Explanation of HTTP request and response headers.

Ans. **HTTP Request Headers**:

- Sent by the **client** to provide information about the request.
- Examples:
    - Host: Domain name of the server.
    - User-Agent: Browser or client making the request.

- Accept: Types of content client can handle (e.g., text/html).
- Cookie: Data stored by the client for session tracking.

➔**HTTP Response Headers**:

- Sent by the **server** to provide information about the response.
- Examples:
  - Content-Type: Type of data being sent (e.g., text/html, application/json).
  - Content-Length: Size of the response body.
  - Set-Cookie: Server instructs client to store cookies.
  - Cache-Control: Defines caching rules.
  - Custom headers can be added (e.g., X-App-Version: 1.2).

Lab Exercise:

QUE.1 Create a Java servlet that:

o Displays the HTTP request headers.

o Sends an HTTP response with custom headers.

Ans.

Java Servlet Program

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

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

      // Set content type
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      // Display HTTP Request Headers
      out.println("<html><body>");
      out.println("<h2>HTTP Request Headers:</h2>");
      Enumeration<String> headerNames = request.getHeaderNames();
      while (headerNames.hasMoreElements()) {
         String headerName = headerNames.nextElement();
         String headerValue = request.getHeader(headerName);
         out.println("<b>" + headerName + ":</b> " + headerValue + "<br>");
      }
```

```
        // Sending HTTP Response with custom headers
        response.setHeader("X-Custom-Header", "Buddy123");
        response.setHeader("X-Powered-By", "Java Servlet");

        out.println("<h2>Custom Response Headers Sent!</h2>");
        out.println("</body></html>");
    }
}
```

==========

J2EE Architecture Overview

Theory:

QUE.1 Introduction to J2EE and its multi-tier architecture.

- Ans. **J2EE** (now called **Jakarta EE**) is a platform for developing and deploying **enterprise-level applications** using Java.
- It supports building **distributed, scalable, secure, and platform-independent** applications.
- J2EE is based on a **multi-tier (layered) architecture**, which separates the application into layers for better maintainability and scalability.
- The main tiers are:
    1. **Client (Presentation) Tier** – User interface (web browsers, mobile apps).
    2. **Middle Tier** – Contains **Web Tier** (Servlets, JSP) and **Business Tier** (EJBs, business logic).
    3. **Data Tier** – Database or legacy systems storing persistent data.

QUE.2 Role of web containers, application servers, and database servers.

Ans. **Web Container (Servlet/JSP Container)**

- Manages execution of Servlets and JSPs.
- Provides services like request handling, session management, security.
- Example: Apache Tomcat.

➔**Application Server**

- Hosts and manages business logic (EJBs, services).
- Provides transaction management, security, load balancing.
- Examples: JBoss, GlassFish, WebLogic.

➔**Database Server**

- Stores and manages application data.
- Supports queries using SQL.
- Examples: MySQL, Oracle, PostgreSQL.

Lab Exercise:

QUE.1 Draw and explain the J2EE architecture, labeling the layers like the presentation layer, business logic layer, and data layer.

Ans.

Explanation of Layers

1. **Presentation Layer (Client Tier)**
    - Interacts with the user.
    - Browser/mobile apps send requests to the server.
2. **Web Layer (Web Container)**
    - Processes user requests using **Servlets/JSP**.
    - Acts as a controller to forward requests to business logic.
3. **Business Logic Layer (Application Server)**
    - Executes application-specific operations (e.g., processing orders, validating transactions).

- Implemented using **EJBs, JavaBeans, services**.

4. **Data Layer (Database Server)**
    - Responsible for storing, retrieving, and managing data.
    - Communicates with RDBMS through **JDBC or JPA**.

==========

Web Component Development in Java (CGI Programming)

Theory:

QUE.1 Introduction to CGI (Common Gateway Interface).

Ans. **CGI** is a standard protocol used to enable interaction between a **web server** and **external programs/scripts**.

➔ It allows a web server to execute programs (written in Java, Python, Perl, C, etc.) and dynamically generate web pages based on user input.

➔ Example: When a user submits an HTML form, CGI processes the data and returns a dynamic response.

QUE.2 Process, advantages, and disadvantages of CGI programming.

Ans. **Process:**

1. User submits an HTML form (client-side).
2. Web server forwards the request to a CGI script.
3. The CGI script executes, processes the input, and generates output.
4. The server sends this output back to the client browser as an HTML page.

Advantages:

- Simple and language-independent (can be written in many languages).
- Portable across different systems.
- Useful for creating dynamic and interactive web pages.

Disadvantages:

- **Performance issues**: Each request spawns a new process, which is resource-heavy.
- **Not scalable** for high-traffic websites.
- Newer technologies like Servlets, JSP, and PHP are more efficient.

Lab Exercise:

QUE.1 Write a simple CGI script using Java to accept user input from a form and display it on a webpage.

Ans.

👉 First, you need an **HTML form** to accept user input:

```html
<!DOCTYPE html>
<html>
<head>
    <title>CGI Form Example</title>
</head>
<body>
    <form method="GET" action="/cgi-bin/SimpleCGI.class">
        Enter your name: <input type="text" name="username">
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

👉 Now, the **Java CGI program** (SimpleCGI.java):

```java
import java.io.*;
import java.util.*;

public class SimpleCGI {
    public static void main(String[] args) throws IOException {
        // Read query string from environment variable
        String query = System.getenv("QUERY_STRING");
        String username = "";

        if (query != null && query.contains("username=")) {
            username = query.split("=")[1];
        }

        // Output HTTP header
        System.out.println("Content-Type: text/html\n");

        // Output HTML response
        System.out.println("<html><body>");
        System.out.println("<h2>Hello, " + username + "!</h2>");
        System.out.println("<p>This is a response from Java CGI script.</p>");
        System.out.println("</body></html>");
    }
}
```

==========

Servlet Programming: Introduction, Advantages, and Disadvantages

Theory:

QUE.1 Introduction to servlets and how they work.

Ans. A **Servlet** is a Java program that runs on a **web server** or **application server**.

➔ It is used to handle requests and responses in **web applications**.

➔**Working of a Servlet:**

1. A client (browser) sends a request to the server.
2. The web server forwards this request to the **Servlet container**.
3. The container loads and executes the servlet.
4. The servlet processes the request (using doGet() or doPost()) and generates a response.
5. The response (usually HTML) is sent back to the client.

QUE.2 Advantages and disadvantages compared to other web technologies.

Ans. **Advantages:**

- Written in **Java**, so portable, secure, and platform-independent.
- **Efficient**: A single servlet instance handles multiple requests using threads (unlike CGI, which spawns a process per request).
- Provides access to Java APIs like JDBC, EJB, etc.
- Easily maintainable and extensible.

➔**Disadvantages:**

- Requires a **Java-enabled server** (Servlet container).
- More complex to learn compared to scripting-based technologies like PHP or JSP.
- For very simple tasks, servlets may be “overkill.”

Lab Exercise:

QUE.1 Write a simple Java servlet that accepts parameters from a user and displays a response.

Ans. 👉 First, create an **HTML form** for user input:

```
<!DOCTYPE html>
<html>
<head>
    <title>Servlet Example</title>
</head>
<body>
    <form action="HelloServlet" method="GET">
        Enter your name: <input type="text" name="username">
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

👉 Now, the **Servlet Code (HelloServlet.java)**

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

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

        // Set response type
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Get user input from request
        String name = request.getParameter("username");

        // Display response
        out.println("<html><body>");
        out.println("<h2>Hello, " + name + "!</h2>");
        out.println("<p>Welcome to the Servlet Example.</p>");
        out.println("</body></html>");
    }
}
```

QUE.2 Discuss the advantages of using servlets over CGI.

Ans.

1. **Performance** – Servlets use multithreading instead of creating a new process for each request like CGI.
2. **Platform Independence** – Servlets are Java-based and run on any platform with a servlet container.
3. **Security** – Java's strong security model protects applications.
4. **Integration** – Can use JDBC, RMI, EJB, and other Java APIs easily.
5. **Maintainability** – Code is modular and object-oriented, easier to update.

==========

Servlet Versions, Types of Servlets

Theory:

QUE.1 History of servlet versions.

Ans.

- **Servlet 2.0 (1997)** – Initial version, basic request/response handling.
- **Servlet 2.1 (1999)** – Introduced deployment descriptor (web.xml).
- **Servlet 2.2 (1999)** – Separated web server and container; WAR packaging.

- **Servlet 2.3 (2001)** – Added filters, listeners, request dispatchers.
- **Servlet 2.4 (2003)** – XML schema for deployment descriptors.
- **Servlet 2.5 (2005)** – Annotation support, dependency on Java EE 5.
- **Servlet 3.0 (2009)** – Major update: annotation-based configuration (@WebServlet), asynchronous support.
- **Servlet 3.1 (2013)** – Non-blocking I/O, protocol upgrade mechanism.
- **Servlet 4.0 (2017)** – HTTP/2 support, push resources.
- **Servlet 5.0 (2020)** – Jakarta EE 9 namespace (jakarta.servlet.*).
- **Servlet 6.0 (2022)** – Jakarta EE 10 features, improved APIs.

QUE.2 Types of servlets: Generic and HTTP servlets.

Ans. ➔ **GenericServlet**:

- Protocol-independent servlet.
- Must override the service() method to handle requests.
- Can be used for custom protocols beyond HTTP.

➔ **HttpServlet**:

- Subclass of GenericServlet.
- Designed for **HTTP protocol only**.
- Provides methods like doGet(), doPost(), doPut(), doDelete().
- Easier for web applications because most use HTTP.

Lab Exercise:

QUE.1 Create a Java servlet program using both GenericServlet and HttpServlet and compare their implementation.

Ans.

==========

Difference between HTTP Servlet and Generic Servlet

Theory:

QUE.1 Detailed comparison between HttpServlet and GenericServlet.

Ans.

Basis	**GenericServlet**	**HttpServlet**
Package	javax.servlet	javax.servlet.http
Protocol Supported	Protocol-independent	HTTP protocol specific
Methods	Only service() method	Has doGet(), doPost(), doPut(), doDelete() etc.
Usage	Used for general purpose servlets	Used for web-based applications
Request/Response Object	Uses ServletRequest and ServletResponse	Uses HttpServletRequest and HttpServletResponse
Preferred For	Non-HTTP protocols	HTTP-based web apps
Example	Email or FTP servlet	Login or form processing servlet

Lab Exercise:

QUE.1 Write a program using HttpServlet to handle HTTP-specific requests like GET and POST.

Ans.

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

public class MyHttpServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws IOException, ServletException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<h3>This is GET request</h3>");
  }
```

```
    public void doPost(HttpServletRequest req, HttpServletResponse res)
            throws IOException, ServletException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        String name = req.getParameter("uname");
        out.println("<h3>Hello, " + name + "! This is POST request.</h3>");
    }
}
```

==========

Servlet Life Cycle

Theory:

QUE.1 Explanation of the servlet life cycle: init(), service(), and destroy() methods.

Ans. Servlet life cycle consists of **three main methods**:

1. **init()** – Called once when servlet is first loaded into memory. Used for initialization.
2. **service()** – Called every time a request is received. Handles client requests and responses.
3. **destroy()** – Called once when servlet is about to be unloaded. Used for cleanup.

Lab Exercise:

QUE.1 Write a servlet program and override all life cycle methods to log messages when each method is called.

Ans.

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

public class LifeCycleDemo extends HttpServlet {

    public void init() {
```

```
        System.out.println("Servlet initialized!");
    }

    public void service(ServletRequest req, ServletResponse res)
            throws IOException, ServletException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<p>Service method called</p>");
    }

    public void destroy() {
        System.out.println("Servlet destroyed!");
    }
}
```

==========

Creating Servlets and Servlet Entry in web.xml

Theory:

QUE.1 How to create servlets and configure them using web.xml.

Ans. Servlets are Java classes that extend HttpServlet.

They must be **configured in web.xml** to define servlet name, class, and URL pattern.

web.xml Entry Example:

```
<web-app>
  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>com.demo.HelloServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>hello</servlet-name>
```

```
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>
```

Lab Exercise:

QUE.1 Create a servlet and configure it in web.xml for deployment.

Ans.

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

public class HelloServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws IOException {
        res.setContentType("text/html");
        res.getWriter().println("<h2>Hello from Servlet!</h2>");
    }
}
```

==========

Logical URL and ServletConfig Interface

Theory:

QUE.1 Explanation of logical URLs and their use in servlets.

Ans. A **logical URL** is a user-friendly path that maps to a servlet in web.xml.

Example: /login → actually points to LoginServlet class.

QUE.2 Overview of ServletConfig and its methods.

Ans. Used to read **initialization parameters** defined for a particular servlet in web.xml.

Methods:

- getInitParameter(String name)
- getInitParameterNames()
- getServletName()

Lab Exercise:

1. Write a servlet that uses ServletConfig to fetch initialization parameters.

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

public class ConfigDemo extends HttpServlet {
   public void doGet(HttpServletRequest req, HttpServletResponse res)
         throws IOException {
      res.setContentType("text/html");
      PrintWriter out = res.getWriter();
      ServletConfig config = getServletConfig();
      String user = config.getInitParameter("username");
      out.println("Username: " + user);
   }
}
```

==========

RequestDispatcher Interface: Forward and Include Methods

Theory:

QUE.1 Explanation of RequestDispatcher and the forward() and include() methods.

- ANS. RequestDispatcher allows one servlet/JSP to **forward** or **include** another resource.
- **Methods:**

- forward(request, response) → transfers control to another resource.
- include(request, response) → includes output of another resource.

Lab Exercise:

Ans.

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

public class LoginServlet extends HttpServlet {
   public void doPost(HttpServletRequest req, HttpServletResponse res)
         throws IOException, ServletException {
      String user = req.getParameter("user");
      String pass = req.getParameter("pass");
      RequestDispatcher rd;
      if (user.equals("admin") && pass.equals("123")) {
         rd = req.getRequestDispatcher("welcome.jsp");
         rd.forward(req, res);
      } else {
         rd = req.getRequestDispatcher("login.jsp");
         rd.include(req, res);
      }
   }
}
```

==========

ServletContext Interface and Web Application Listener

Theory:

QUE.1 Introduction to ServletContext and its scope.

Ans. **ServletContext**:

- Provides info about the entire web application.
- Used to share data among all servlets.
- Methods: getInitParameter(), setAttribute(), getAttribute()

QUE.2 How to use web application listeners for lifecycle events.

Ans. **Web Application Listener:**

- Listens to events like app start, stop, session creation, etc.
- Implemented using interfaces like ServletContextListener.

Lab Exercise:

QUE.1 Use ServletContext to share data across multiple servlets.

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

public class ContextDemo extends HttpServlet {
   public void doGet(HttpServletRequest req, HttpServletResponse res)
         throws IOException {
      res.setContentType("text/html");
      ServletContext ctx = getServletContext();
      String college = ctx.getInitParameter("collegeName");
      res.getWriter().println("College: " + college);
   }
}
```

QUE.2 Create a web application listener that logs application start and stop events.

Ans.
```
import javax.servlet.*;
```

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

    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("Application stopped!");
    }
}
```

Java Filters: Introduction and Filter Life Cycle

Theory:

Filters in Java (specifically in the Servlet API, under **javax.servlet.Filter**) are components that intercept and process HTTP requests and responses before they reach the target servlet or JSP, or after the response is generated. They act as a middleware layer in the web application, allowing you to perform tasks like logging, authentication, data compression, encoding, or validation without modifying the core servlet/JSP code.

Filters are needed when you want to:

- Apply common logic across multiple resources (e.g., checking user permissions for all pages).
- Modify requests/responses (e.g., sanitizing input to prevent XSS attacks).
- Handle cross-cutting concerns like caching, rate limiting, or auditing.
  They promote separation of concerns and reusability, especially in large web apps. Without filters, you'd have to duplicate code in every servlet.

QUE.2 Filter lifecycle and how to configure them in web.xml.
Ans.
The filter lifecycle in Java Servlets follows these stages:

1. **Initialization (init() method)**: Called once when the filter is loaded (e.g., during application startup). Use this to allocate resources like database connections. It receives a **FilterConfig** object for configuration parameters.
2. **Request Processing (doFilter() method)**: Called for every matching request. It wraps the request/response in a chain and invokes the next filter or target resource. You can inspect/modify the request before/after forwarding.
3. **Destruction (destroy() method)**: Called once when the filter is unloaded (e.g., app shutdown). Use this to release resources.

Filters are configured in **web.xml** (or via annotations in Servlet 3.0+).

Lab Exercise:

QUE.1 Implement a filter to perform server-side validation of user input.

Ans. package com.example;

```
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebFilter(urlPatterns = {"/submitForm"})  // Apply to specific URL
public class ValidationFilter implements Filter {
   @Override
   public void init(FilterConfig config) throws ServletException {
      // Initialization logic (e.g., load validation rules)
   }

   @Override
   public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
         throws IOException, ServletException {
      HttpServletRequest req = (HttpServletRequest) request;
      HttpServletResponse res = (HttpServletResponse) response;

      String username = req.getParameter("username");
      if (username == null || username.trim().isEmpty() || username.length() < 3) {
```

```
            res.sendRedirect("error.jsp?msg=Invalid username");  // Forward to error
            return;
        }

        // If valid, proceed to next in chain
        chain.doFilter(request, response);
    }

    @Override
    public void destroy() {
        // Cleanup logic
    }
}
```

Step 2: Create a simple form servlet/JSP (**submitForm.jsp**) to test:

RunCopy code

```
<form method="POST" action="submitForm">
    Username: <input type="text" name="username">
    <input type="submit" value="Submit">
</form>
```

==========

Practical Example: Server-Side Validation Using Filters

Lab Exercise:

QUE.1 Write a filter that checks whether form input fields are empty. If they are, forward back to the input form; otherwise, proceed with the request.

Ans.

This builds on the previous filter but focuses on multiple fields (e.g., username and email). If any field is empty, it forwards back to the input form with an error message; else, proceeds.

Step 1: Create the filter (**EmptyFieldFilter.java**):

```
package com.example;
```

```
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebFilter(urlPatterns = {"/processForm"})
public class EmptyFieldFilter implements Filter {
   @Override
   public void init(FilterConfig config) throws ServletException {}

   @Override
   public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
         throws IOException, ServletException {
      HttpServletRequest req = (HttpServletRequest) request;
      HttpServletResponse res = (HttpServletResponse) response;

      String username = req.getParameter("username");
      String email = req.getParameter("email");

      if (username == null || username.trim().isEmpty() ||
         email == null || email.trim().isEmpty()) {
         // Set error attribute and forward back to form
         req.setAttribute("errorMsg", "All fields are required!");
```

```
            RequestDispatcher dispatcher = req.getRequestDispatcher("inputForm.jsp");
            dispatcher.forward(req, res);
            return;
        }

        // Proceed if valid
        chain.doFilter(request, response);
    }

    @Override
    public void destroy() {}
}
```

Step 2: Create the input form (**inputForm.jsp**):

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head><title>Input Form</title></head>
<body>
    <% if (request.getAttribute("errorMsg") != null) { %>
        <p style="color:red;"><%= request.getAttribute("errorMsg") %></p>
    <% } %>
    <form method="POST" action="processForm">
        Username: <input type="text" name="username"><br>
        Email: <input type="text" name="email"><br>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

==========

JSP Basics: JSTL, Custom Tags, Scriplets, and Implicit Objects

Theory:

QUE.1 Introduction to JSP and its key components: JSTL, custom tags,scriplets, and implicit objects.

Ans. JavaServer Pages (JSP) is a server-side technology that embeds Java code in HTML to generate dynamic web content. It extends Servlets by allowing a mix of static markup and dynamic logic, compiled into servlets at runtime.

Key components:

- **JSTL (JSP Standard Tag Library)**: A set of standard tags for common tasks like iteration (**<c:forEach>**), conditionals (**<c:if>**), formatting, and SQL/XML handling. It promotes clean code by avoiding scriptlets. Include via **<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>**.
- **Custom Tags**: User-defined tags (via Tag Files or Tag Classes implementing **SimpleTagSupport**) for reusable components, e.g., a custom "hello" tag. They encapsulate logic, making JSPs more modular.
- **Scriptlets**: Java code snippets embedded in JSP using **<% %>** (declarations: **<%! %>**, expressions: **<%= %>**). They allow direct scripting but are discouraged in modern JSP (use tags instead for maintainability).
- **Implicit Objects**: Pre-defined objects available in JSP scope without declaration, e.g., **request** (HttpServletRequest), **response** (HttpServletResponse), **session** (HttpSession), **application** (ServletContext), **out** (JspWriter), **pageContext** (PageContext), **page** (Object), **config** (ServletConfig), **exception** (Throwable, in error pages). They provide quick access to request data, sessions, etc.

Lab Exercise:

QUE.1 Create a JSP page that uses JSTL to iterate through a list, display scriptlets and access implicit objects.

Ans.

This JSP demonstrates all components: JSTL for looping a list, a scriptlet for computation, and implicit objects for user data.

Step 1: Create **demo.jsp**:

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

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


<!DOCTYPE html>

<html>

<head><title>JSP Demo</title></head>

<body>

  <h1>Welcome, <%= request.getParameter("name") %>!</h1>  <!-- Implicit: request; Expression -->
```

```
    <%-- Scriptlet: Compute and set a list --%>
    <%
        java.util.List<String> fruits = new java.util.ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        pageContext.setAttribute("fruitList", fruits);  // Use implicit pageContext
    %>

    <%-- JSTL: Iterate through list --%>
    <h2>Fruits List (using JSTL):</h2>
    <ul>
        <c:forEach var="fruit" items="${fruitList}">
            <li>${fruit}</li>
        </c:forEach>
    </ul>

    <%-- Custom Tag Example (simple tag file, assume created as /WEB-INF/tags/greet.tag) --%>
    <%@ taglib tagdir="/WEB-INF/tags" prefix="mytags" %>
    <mytags:greet name="User " />  <!-- Outputs: Hello, User! (implement in tag file) -->

    <p>Session ID: <%= session.getId() %></p>  <!-- Implicit: session -->
</body>
</html>
```

==========

Session Management and Cookies

Theory:

QUE.1 Overview of session management techniques: cookies, hidden form fields, URL rewriting and sessions.

Ans.

Session management tracks user state across HTTP requests (stateless by default). Techniques:

- **Cookies**: Small data stored on client browser (e.g., **Cookie** object with name/value). Sent with every request. Pros: Persistent. Cons: Disabled by users/privacy issues.
- **Hidden Form Fields**: Embed data in forms as **<input type="hidden" name="sessionId" value="abc">**. Pros: No client storage. Cons: Only works for form submissions, not direct navigation.
- **URL Rewriting**: Append session data to URLs (e.g., **;jsessionid=abc**). Pros: Works without cookies. Cons: Messy URLs, not bookmark-friendly.
- **Sessions (HttpSession)**: Server-side storage using a unique ID (often via cookies). Data stored in memory/database. Pros: Secure, handles complex state. Cons: Server overhead.
  Use **session.setAttribute()**/**getAttribute()**.

Choose based on app needs: Sessions for most cases; fall back to rewriting if cookies are disabled.

QUE.2 How to track user sessions in web applications.
Ans.
Track sessions using **HttpSession**:

1. On first request, create session: **HttpSession session = request.getSession(true);**.
2. Store data: **session.setAttribute("userId", 123);**.
3. Retrieve: **Integer id = (Integer) session.getAttribute("userId");**.
4. Invalidate: **session.invalidate();** (e.g., on logout).
   The container manages the session ID (via cookie or URL). Configure timeout in **web.xml**: **<session-config><session-timeout>30</session-timeout></session-config>**. For security, use HTTPS and regenerate IDs on login (**session.setAttribute()** after authentication).

Lab Exercise:

QUE.1 Implement a login system in JSP and servlet that uses cookies and session tracking to manage user authentication.
Ans.
This example: Login form → Servlet validates → Sets session attribute and cookie → Protected page checks session.

Step 1: Login form (**login.jsp**):

jsp

RunCopy code

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

<!DOCTYPE html>

<html>

<head><title>Login</title></head>

<body>

  <form method="POST" action="LoginServlet">
```

```
    Username: <input type="text" name="username"><br>
    Password: <input type="password" name="password"><br>
    <input type="submit" value="Login">
  </form>
</body>
</html>
```

Step 2: Login Servlet (**LoginServlet.java**):

java33 lines

Copy codeDownload code

Click to expand

```
package com.example;
...
```

Step 3: Protected page (**welcome.jsp**):

jsp

RunCopy code

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head><title>Welcome</title></head>
<body>
  <%
    HttpSession session = request.getSession(false);  // Don't create new
    if (session != null && session.getAttribute("loggedInUser ") != null) {
      String user = (String) session.getAttribute("loggedInUser ");
  %>
    <h1>Welcome, <%= user %>!</h1>
    <p>Cookie Value: <%
      Cookie[] cookies = request.getCookies();
      if (cookies != null) {
        for (Cookie c : cookies) {
          if ("userCookie".equals(c.getName())) {
```

```
                out.print(c.getValue());
                break;
            }
          }
        }
    %></p>
    <a href="LogoutServlet">Logout</a>
  <% } else { %>
    <p>Not logged in. <a href="login.jsp">Login</a></p>
  <% } %>
</body>
</html>
```

Step 4: Logout Servlet (**LogoutServlet.java**):

```
package com.example;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

@WebServlet("/LogoutServlet")
public class LogoutServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
    HttpSession session = req.getSession(false);
    if (session != null) {
```

```
            session.invalidate();  // End session
        }

        Cookie loginCookie = new Cookie("userCookie", "");
        loginCookie.setMaxAge(0);  // Delete cookie
        res.addCookie(loginCookie);

        res.sendRedirect("login.jsp");
    }
}
```

Step 5: Test: Login with "admin"/"pass" → Welcome page shows session/cookie data. Invalid → Error. Logout → Clears both.

If you need more details, expansions, or troubleshooting, let me know!