**1.) HTML Tags: Anchor, Form, Table, Image, List Tags, Paragraph, Break, Label**

**1.1) Introduction to HTML and its structure.**

HTML (HyperText Markup Language) is the standard language used to create webpages. It provides the structure of a webpage using **tags** enclosed in angle brackets (< >). A basic HTML document includes:

<!DOCTYPE html>: Defines the document type.

<html>: Root element.

<head>: Contains metadata like title, styles, scripts.

<body>: Contains the visible content of the page.

**1.2) Explanation of key tags:**

**<a> (Anchor tag):**  
Used to create hyperlinks to other pages, websites, or files. Example:  
<a href="https://example.com">Visit Example</a>

**<form> (Form tag):**  
Defines a form for user input. It can contain input fields, checkboxes, buttons, etc. Example:  
<form action="submit.php" method="post">...</form>

**<table> (Table tag):**  
Used to represent data in rows and columns. It is made up of <tr> (table row), <th> (header cell), and <td> (data cell).

**<img> (Image tag):**  
Embeds an image on a webpage. Example:  
<img src="image.jpg" alt="Description">

**List Tags (<ul>, <ol>, <li>):**

<ul>: Unordered list (bulleted).

<ol>: Ordered list (numbered).

<li>: List item inside a list.

**<p> (Paragraph tag):**  
Defines a block of text as a paragraph.

**<br> (Line break):**  
Inserts a single line break without starting a new paragraph.

**<label> (Label tag):**  
Used to label form inputs for better accessibility. Example:  
<label for="username">Username:</label><input type="text" id="username">

**2.) CSS: Inline CSS, Internal CSS, External CSS**

**2.1) Overview of CSS and its importance in web design.**

CSS (Cascading Style Sheets) is a language that **controls the look and layout of a web page**. While HTML gives structure, CSS adds **style** — colors, fonts, spacing, and positioning.

**Why CSS is Important**

**1. Makes websites look good** – adds colors, fonts, and layouts.

**2. Separates style from content** – easier to maintain.

**3. Responsive design** – works on desktops, tablets, and phones.

**4. Better user experience** – consistent and readable design.

**5. Supports animations and effects** – makes websites interactive.

**2.2) Types of CSS:**

**1. Inline CSS:**  
Applied directly within an HTML element using the style attribute.  
Example:

<p style="color:blue; font-size:18px;">This is inline styled text.</p>

**2. Internal CSS:**  
Defined inside a <style> tag within the <head> section of the HTML document. It affects only that specific page.  
Example:

<head>

<style>

p { color: green; font-size: 20px; }

</style>

</head>

**3. External CSS:**  
Stored in a separate .css file and linked to the HTML file using <link>. It is the most efficient way to apply styles across multiple pages.  
Example:

<head>

<link rel="stylesheet" href="styles.css">

</head>

**3.) CSS: Margin and Padding**

**3.1) Definition and difference between margin and padding.**

**Margin:**  
The margin is the space **outside an element’s border**, which separates the element from other elements. Margins are always transparent and do not take background color.

**Padding:**  
The padding is the space **inside an element’s border**, between the content and the border. Padding increases the spacing around the content but keeps it within the element’s background.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Margin** | **Padding** |
| **Position** | Outside the element’s border | Inside the element’s border |
| **Effect** | Creates space between elements | Creates space between content & border |
| **Background** | Always transparent | Inherits element’s background color |
| **Collapsing** | Margins of adjacent elements may collapse | Padding never collapses |

**3.2) How margins create space outside the element and padding creates space inside.**

**Margins**:  
Margins are used to create space **outside an element’s border**. They push the element away from other elements or the edge of the browser window. Margins are always transparent and do not take the background color of the element.

**Padding**:  
Padding is the space **inside an element’s border**, surrounding the content. It pushes the content inward and increases the clickable/visible area of the element. Padding inherits the element’s background color.

**In short:**

**Margin = Outside space** (separates elements from each other).

**Padding = Inside space** (separates content from its border).

**4.)CSS: Pseudo-Class**

**4.1) Introduction to CSS pseudo-classes like :hover, :focus, :active, etc.**

**Pseudo-classes** in CSS are used to define a special state of an element.

They are written with a colon (:) followed by the pseudo-class name.

Common pseudo-classes:

**:hover** → Applies a style when the mouse pointer is placed over an element.

**:focus** → Applies a style when an element (like an input box) is focused/active (clicked or tabbed into).

**:active** → Applies a style when an element (like a link or button) is being clicked.

**:visited** → Styles a link that has already been visited.

**:first-child, :last-child, :nth-child()** → Target specific elements in a list or structure.

**4.2) Use of pseudo-classes to style elements based on their state.**

Pseudo-classes are mainly used in **interactive designs** to improve user experience:

**:hover** → Highlight buttons/links when the user hovers over them.

**:focus** → Change input field appearance when the user types in it.

**:active** → Show button being pressed.

**:visited** → Indicate links already visited.

**Example**

<style>

a:hover {

color: red; /\* Changes link color when hovered \*/

}

a:visited {

color: purple; /\* Changes link color after visiting \*/

}

button:active {

background-color: green; /\* Changes button when clicked \*/

}

input:focus {

border: 2px solid blue; /\* Highlights input when focused \*/

}

</style>

**5.) CSS: ID and Class Selectors**

**5.1) Difference between id and class in CSS.**

**1. id Selector**

Defined with a # symbol (e.g., #header).

Used to apply styles to a **single unique element** on a page.

Each id must be **unique** within an HTML document.

**2. class Selector**

Defined with a . symbol (e.g., .button).

Used to apply styles to **multiple elements**.

A single class can be reused for many elements, and elements can have multiple classes.

|  |  |  |
| --- | --- | --- |
| **Feature** | **id** | **class** |
| **Selector** | #idName | .className |
| **Uniqueness** | Unique (used for one element only) | Reusable (can apply to many elements) |
| **Priority** | Higher specificity than class | Lower specificity |
| **Use Case** | Targeting a single, specific element | Styling multiple elements with same design |

**5.2) Usage scenarios for id (unique) and class (reusable).**

**When to use id:**

For elements that appear only **once** on a page.

Examples: header section (#header), footer (#footer), navigation bar container.

Also useful for linking directly with anchors (<a href="#section1">).

**When to use class:**

For elements that **repeat** or share the same styling.

Examples: buttons (.btn), product cards (.card), form inputs (.input-field).

Classes help maintain consistency across multiple elements.

**Example**

<style>

#main-title {

color: blue; /\* Unique styling \*/

}

.button {

background: green; /\* Reusable styling \*/

color: white;

}

</style>

<h1 id="main-title">Welcome Page</h1>

<button class="button">Login</button>

<button class="button">Register</button>

**6.) Introduction to Client-Server Architecture**

**6.1) Overview of client-server architecture.**

Client-server architecture is a **distributed computing model** where tasks are divided between two entities:

**Client:** The front-end system (like a web browser or app) that requests services.

**Server:** The back-end system that processes requests and sends responses.

Communication happens over a **network** (usually the Internet) using protocols like **HTTP/HTTPS**.

This model improves scalability, efficiency, and resource sharing.

**Example:**  
When you open a website:

Browser (**client**) → requests a page.

Web server (**server**) → processes request and sends back HTML, CSS, JS.

**6.2) Difference between client-side and server-side processing.**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Client-Side Processing** | **Server-Side Processing** |
| **Where it happens** | On the user’s device (browser). | On the web server. |
| **Examples** | HTML, CSS, JavaScript, form validation. | PHP, Python, Java, Node.js, database queries. |
| **Advantages** | Faster response, reduces server load. | More secure, handles sensitive data. |
| **Disadvantages** | Less secure, depends on browser speed. | Slower (network delays), increases server load. |

**6.3) Roles of a client, server, and communication protocols.**

**1. Client:**

Requests services (e.g., loads a webpage, submits a form).

Runs on user’s device (browser, mobile app).

**2. Server:**

Stores, processes, and delivers resources or services.

Runs applications, databases, APIs.

**3. Communication Protocols:**

Define rules for data exchange between client and server.

Common examples:

**HTTP/HTTPS** → For web communication.

**FTP** → For file transfer.

**SMTP/IMAP** → For email.

**7.) HTTP Protocol Overview with Request and Response Headers**

**7.1) Introduction to the HTTP protocol and its role in web communication.**

**HTTP (HyperText Transfer Protocol):**

A communication protocol used for transferring data on the World Wide Web.

It defines how **clients (browsers)** and **servers** exchange information.

**Role in Web Communication:**

When a user opens a webpage:

The **client** (browser) sends an HTTP request.

The **server** processes it and sends back an HTTP response.

HTTP is **stateless** (each request is independent).

Secure version is **HTTPS**, which uses encryption (SSL/TLS).

**7.2) Explanation of HTTP request and response headers.**

**HTTP Request Headers:**  
Sent from client → server, containing details about the request.  
Examples:

Host: → Specifies the domain (e.g., Host: [www.example.com](http://www.example.com)).

User-Agent: → Identifies the browser or app.

Accept: → Defines the content types the client can handle (e.g., text/html).

Cookie: → Sends stored cookies to the server.

**HTTP Response Headers:**  
Sent from server → client, containing details about the response.  
Examples:

Content-Type: → Specifies format (e.g., text/html, application/json).

Content-Length: → Size of the response body.

Set-Cookie: → Stores cookies on client side.

Cache-Control: → Defines caching rules.

Server: → Provides info about the server (e.g., Apache, Nginx).

**8.) J2EE Architecture Overview**

**8.1) Introduction to J2EE and its multi-tier architecture.**

**J2EE (Java 2 Platform, Enterprise Edition):**

A platform for building and deploying **enterprise-level applications** using Java.

Provides APIs and tools for web applications, distributed systems, and database-driven apps.

Supports scalability, portability, and security.

**Multi-Tier Architecture in J2EE:**  
J2EE applications are usually built in **three main tiers**:

**1. Client Tier:**

User interface (web browser, mobile app, desktop client).

Sends requests to the server.

**2. Middle Tier (Business Logic):**

Managed by a **web container** and **application server**.

Contains Servlets, JSP, EJB (Enterprise Java Beans).

Processes client requests and applies business logic.

**3. Data Tier (Database):**

Stores and manages application data.

Uses database servers (MySQL, Oracle, etc.).

**8.2) Role of web containers, application servers, and database servers.**

**Web Container:**

Part of the server that manages **Servlets and JSP**.

Provides runtime environment, request/response handling, and session management.

Example: Tomcat.

**Application Server:**

Hosts the **business logic** of applications (EJB, transactions, messaging).

Provides services like security, scalability, and load balancing.

Examples: JBoss, WebLogic, GlassFish.

**Database Server:**

Stores, retrieves, and manages data.

Communicates with the application server via JDBC (Java Database Connectivity).

Examples: MySQL, Oracle, PostgreSQL

**9.) Web Component Development in Java (CGI Programming)**

**9.1) Introduction to CGI (Common Gateway Interface).**

**CGI (Common Gateway Interface):**

A standard protocol that allows a **web server** to interact with external programs (scripts or executables).

Used to generate **dynamic web content** by processing user input (e.g., from forms).

CGI scripts can be written in many languages such as **Perl, C, Python, or Shell scripting**.

**Example:**

User fills a form → Web server calls a CGI script → Script processes data → Server returns output (like HTML).

**9.2) Process, advantages, and disadvantages of CGI programming.**

**Process of CGI**

User requests a webpage or submits a form.

Web server executes a CGI script.

Script processes input (e.g., queries database).

Script sends output back to the server in HTML format.

Server sends response to the client’s browser.

**Advantages of CGI**

Language independent (works with many languages).

Easy to implement for small-scale dynamic websites.

Supported by most web servers.

Simple to understand and set up.

**Disadvantages of CGI**

**Performance issue** → Each request starts a new process, which is slow.

**Scalability problem** → Not suitable for high-traffic websites.

Consumes more memory and CPU.

Security risks if not handled properly (since scripts directly execute on the server).

**10.) Servlet Programming: Introduction, Advantages, and Disadvantages.**

**10.1) Introduction to servlets and how they work.**

**Servlets:**

Servlets are **Java programs** that run on a web server to handle **requests and responses**.

They are part of **Java EE (Enterprise Edition)** and are used to create **dynamic web applications**.

Servlets run inside a **web container** (like Apache Tomcat) which provides an environment for execution.

**How Servlets Work (Servlet Lifecycle):**

**1. Client Request:** Browser sends a request to the server (usually HTTP).

**2. Servlet Loading:** Web container loads the servlet class (if not already loaded).

**3. Initialization (init()):** Called once when servlet is created.

**4. Request Handling (service()):** Handles requests using doGet(), doPost(), etc.

**5. Response Sent:** Servlet generates dynamic output (usually HTML) and sends it back to the client.

**6. Destroy (destroy()):** Called when servlet is removed from service.

**10.2) Advantages and disadvantages compared to other web technologies.**

**Advantages of Servlets**

**Platform-independent** (Java-based, runs anywhere with JVM).

**Efficient & Fast:** Runs inside a single process (unlike CGI which spawns a new process per request).

**Secure:** Uses Java’s security features.

**Reusable & Modular:** Can be reused across multiple applications.

**Scalable:** Suitable for large enterprise applications.

**Disadvantages of Servlets**

**Complexity:** Writing HTML inside Java code can be messy.

**Less Design Flexibility:** Not as designer-friendly compared to JSP or modern frameworks.

**More Development Effort:** Requires more coding compared to frameworks like Spring MVC.

**Comparison with CGI and JSP**

**Compared to CGI:**

Servlets are faster (no new process per request).

More scalable and memory-efficient.

**Compared to JSP:**

JSP is easier for presentation (HTML + Java), while Servlets are better for business logic.

In practice, both are often used together.

**11.) Servlet Versions, Types of Servlets.**

**11.1) History of servlet versions.**

**Servlet 1.0 (1997)** – Basic API, introduced by Sun.

**Servlet 2.0 (1998)** – Added request dispatching and filters.

**Servlet 2.2 (1999)** – Introduced web.xml deployment descriptor.

**Servlet 2.3 (2001)** – Added listener support.

**Servlet 2.4 (2003)** – Better integration with JSP, XML schema.

**Servlet 2.5 (2005)** – Annotation support (@WebServlet).

**Servlet 3.0 (2009)** – Asynchronous processing, pluggability.

**Servlet 3.1 (2013)** – Non-blocking I/O.

**Servlet 4.0 (2017)** – HTTP/2 support.

**Servlet 5.0 (2020)** – Part of Jakarta EE 9 (moved from javax to jakarta).

**Servlet 6.0 (2022)** – Jakarta EE 10, modern enhancements.

**11.2) Types of servlets: Generic and HTTP servlets.**

**1. Generic Servlet**

Protocol-independent (not tied to HTTP).

Extends GenericServlet class.

Defines service() method to handle requests.

**2. HTTP Servlet**

Specifically for **HTTP protocol**.

Extends HttpServlet class.

Provides methods like doGet(), doPost(), doPut(), etc.

**12.) Difference between HTTP Servlet and Generic Servlet.**

**12.1) Detailed comparison between HttpServlet and GenericServlet.**

|  |  |  |
| --- | --- | --- |
| **Feature** | **GenericServlet** | **HttpServlet** |
| **Definition** | An abstract class that defines a generic, protocol-independent servlet. | A subclass of GenericServlet designed specifically for handling HTTP requests. |
| **Package** | javax.servlet (or jakarta.servlet in newer versions) | javax.servlet.http (or jakarta.servlet.http) |
| **Protocol Support** | Protocol-independent (can be used for any protocol). | Works only with **HTTP protocol**. |
| **Base Class** | Extends GenericServlet directly from Servlet interface. | Extends GenericServlet. |
| **Methods to Override** | Must override service(ServletRequest req, ServletResponse res) method. | Usually override doGet(), doPost(), doPut(), doDelete(), etc. |
| **Ease of Use** | Less convenient since developer has to manually handle request types and protocol logic. | More convenient, as it provides built-in methods for each HTTP operation. |
| **Usage** | Rarely used in real applications, mainly for creating protocol-independent servlets. | Commonly used in web applications (since most apps use HTTP/HTTPS). |
| **Request/Response Objects** | Works with ServletRequest and ServletResponse (general). | Works with HttpServletRequest and HttpServletResponse (HTTP-specific). |
| **Typical Application** | Custom or experimental protocol handling. | Web applications (websites, REST APIs, etc.). |

**13.) Servlet Life Cycle.**

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

**1. init()**

Called **once** when the servlet is first loaded into memory.

Used to **initialize resources** like database connections, files, or configuration.

**Cannot** handle client requests and is **not called multiple times**.

**2. service()**

Called **every time** a client sends a request.

Handles all request types. In HttpServlet, it automatically calls doGet(), doPost(), etc., based on the request.

**Cannot** perform one-time initialization or cleanup; it’s only for request processing.

**3. destroy()**

Called **once** when the servlet is about to be removed from memory (e.g., server shutdown or undeployment).

Used to **release resources** like closing database connections or files.

**Cannot** handle client requests and is **not called multiple times**.

**14.) Creating Servlets and Servlet Entry in web.xml.**

**14.1) How to create servlets and configure them using web.xml.**

**1. Create a Servlet**

Make a Java class that **extends HttpServlet**.

Override doGet() or doPost() to handle requests.

public class HelloServlet extends HttpServlet {

protected void doGet(HttpServletRequest req, HttpServletResponse res)

throws IOException {

res.getWriter().println("Hello, Servlet!");

}

}

**2. Configure in web.xml**

<web-app>

<servlet>

<servlet-name>HelloServlet</servlet-name>

<servlet-class>HelloServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>HelloServlet</servlet-name>

<url-pattern>/hello</url-pattern>

</servlet-mapping>

</web-app>

**3. Access**

* URL: http://localhost:8080/YourAppName/hello

**15.) Logical URL and ServletConfig Interface.**

**15.1) Explanation of logical URLs and their use in servlets.**

**Logical URL**: The URL pattern you define in web.xml (or using annotations) to access a servlet.

Example: If you map /hello to HelloServlet, then the logical URL is:

<http://localhost:8080/YourAppName/hello>

**Use**:

Allows clients to access the servlet without knowing the actual class name.

Makes URL management flexible; you can change the servlet class without changing the URL.

Helps in separating user-friendly URLs from internal class structure.

**15.2) Overview of ServletConfig and its methods.**

**ServletConfig**: An object provided by the servlet container to pass **initialization parameters** to a servlet.

**Purpose**:

To get servlet-specific configuration from web.xml.

To access initialization parameters (<init-param>).

**Key Methods of ServletConfig**:

1. getInitParameter(String name) → Returns the value of a specific initialization parameter.

2. getInitParameterNames() → Returns all parameter names.

3. getServletName() → Returns the name of the servlet.

4. getServletContext() → Returns the ServletContext object (for context-wide parameters or resources).

**16.) RequestDispatcher Interface: Forward and Include Methods.**

**16.1) Explanation of RequestDispatcher and the forward() and include() methods.**

**RequestDispatcher**

**Purpose**: Used to **forward a request** from one servlet or JSP to another resource (servlet, JSP, or HTML).

Obtained via:

RequestDispatcher rd = request.getRequestDispatcher("otherResource");

**Methods**

**1. forward(request, response)**

Forwards the request to another resource **on the server**.

The client **does not know** about the change (URL in browser stays the same).

Example:

rd.forward(request, response);

**2. include(request, response)**

Includes content of another resource **in the current response**.

The response from both resources is combined.

Example:

rd.include(request, response);

**17.) ServletContext Interface and Web Application Listener.**

**17.1) Introduction to ServletContext and its scope.**

**ServletContext**

**Definition**: An object provided by the servlet container that represents the **entire web application**.

**Scope**:

**Application-wide** → shared by all servlets and JSPs in the same web application.

**Uses**:

Share data between servlets.

Access application-level parameters (defined in web.xml).

Access resources like files, logs, or URLs.

**Example**:

ServletContext context = getServletContext();

context.setAttribute("appName", "MyApp"); // store data

String name = (String) context.getAttribute("appName"); // retrieve data

**17.2) How to use web application listeners for lifecycle events.**

**Definition**: Special classes that listen for **lifecycle events** in a web application (like servlet creation, session creation, context initialization).

**Uses**:

Perform tasks when the application or session starts or ends.

Logging, resource initialization, cleanup tasks.

**Types of Listeners**:

1. **ServletContextListener** → reacts to context (application) lifecycle events.

Methods: contextInitialized() and contextDestroyed().

2. **HttpSessionListener** → reacts to session creation and destruction.

3. **ServletRequestListener** → reacts to request events.

**Example**:

public class MyAppListener implements ServletContextListener {

public void contextInitialized(ServletContextEvent sce) {

System.out.println("Application Started");

}

public void contextDestroyed(ServletContextEvent sce) {

System.out.println("Application Stopped");

}

}

Register in web.xml:

<listener>

<listener-class>MyAppListener</listener-class>

</listener>

**18.) Java Filters: Introduction and Filter Life Cycle.**

**18.1) What are filters in Java and when are they needed?.**

**Definition**: Filters are Java classes that **intercept requests and responses** in a web application.

**Purpose**:

Modify or examine request/response objects.

Perform tasks like **logging, authentication, encryption, input validation, compression**, etc.

**Key Point**: Filters do **not generate a response**; they pass control to servlets or JSPs.

**18.2) Filter lifecycle and how to configure them in web.xml.**

**1. init(FilterConfig config)**

Called once when the filter is first loaded.

Used for initialization (like reading parameters from web.xml).

**2. doFilter(ServletRequest request, ServletResponse response, FilterChain chain)**

Called **for every request** that matches the filter mapping.

Can modify request or response, then pass control using:

chain.doFilter(request, response);

**3. destroy()**

Called once when the filter is removed.

Used to release resources.

**19.) JSP Basics: JSTL, Custom Tags, Scriplets, and Implicit Objects.**

**19.1) Introduction to JSP and its key components: JSTL, custom tags,scriplets, and implicit objects.**

**1. Introduction to JSP**

**JSP (JavaServer Pages)**: A server-side technology that allows you to create **dynamic web pages** using HTML, Java code, and special JSP tags.

Converts JSP into a **servlet** internally and executes it on the server.

**2. Key Components of JSP**

**1. JSTL (JSP Standard Tag Library)**

Predefined tags to simplify common tasks: loops, conditions, database access, formatting, etc.

Example: <c:forEach> for looping.

**2. Custom Tags**

User-defined tags that encapsulate complex functionality into simple tags.

Helps in **reusability and cleaner code**.

Example: <mytags:showDate />.

**1. Scriplets**

Java code embedded directly in JSP using <% ... %>.

Example:

<% int a = 10; %>

<p>Value: <%= a %></p>

**Note**: Scriplets are discouraged in modern JSP development.

**2. Implicit Objects**

Predefined objects available in JSP without declaration.

Common ones:

request → HttpServletRequest

response → HttpServletResponse

session → HttpSession

application → ServletContext

out → JspWriter for sending output

**20.) Session Management and Cookies.**

**20.1) Overview of session management techniques: cookies, hidden form fields, URL rewriting, and sessions.**

**1. Cookies**

Small data stored on the **client browser**.

Sent with every request to the server.

Can store user-specific info like login status.

**Limitation**: Users can disable cookies.

**2. Hidden Form Fields**

Data stored in **HTML form fields** that are not visible to users.

Sent with form submission.

**Limitation**: Works only with form submissions.

**3. URL Rewriting**

Appends session information to URLs:

<http://example.com/page?jsessionid=12345>

Useful when cookies are disabled.

**Limitation**: URLs can get messy and may expose session IDs.

**4. HttpSession (Sessions)**

Server-side object to track **user state** across multiple requests.

Each user gets a **unique session ID**.

Can store objects like login info, shopping cart, preferences.

Example:

HttpSession session = request.getSession();

session.setAttribute("username", "Alice");

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

**20.2) How to track user sessions in web applications.**

**Step 1:** Create or get session using request.getSession().

**Step 2:** Store user-specific data using setAttribute().

**Step 3:** Retrieve data using getAttribute().

**Step 4:** Invalidate session when done using session.invalidate() (e.g., logout).