A Course File

On

FULL STACK WEB DEVELOPMENT

(IV- B. Tech. ––VII Semester)

Submitted to

DEPARTMENT OF COMPUTER SCIENCE& ENGINEERING

By

Mr. Trinadh(B.TECH, M.TECH)

(2025-26)

CONTENTS

|  |  |  |
| --- | --- | --- |
| Sl. No. | Particulars | Page No. |
| 1 | Syllabus |  |
| 2 | Student Entry Behavior or Pre-requisites |  |
| 3 | Course Objectives |  |
| 4 | Course Outcomes |  |
| 5 | Mapping Of Course Outcomes with PEOs |  |
| 6 | Mapping Of Course Outcomes with PSOs |  |
| 7 | Mapping Of Course Outcomes with POs |  |
| 8 | Mapping of Course with PEOs-PSOs-POs |  |
| 9 | Direct Course Assessment |  |
| 10 | Indirect Course Assessment |  |
| 11 | Overall Course Assessment and Attainment level |  |
| 12 | Pi diagrams, Bar charts, Histograms for representing results |  |
| 13 | Lesson/Course Plan |  |
| 14 | Actions and Assessment Plan |  |
| 15 | Lecture Notes |  |
| 16 | Question Bank |  |
| 17 | Internal Examination Question Papers with key |  |
| 18 | External/End Examination Question Papers |  |

**SYLLABUS FOR - FSWD**

# MODULE- I

HTML Common tags- List, Tables, images, forms, Frames; Cascading Style sheets. Introduction to Java Scripts, Objects and Functions in java script, Manipulating DOM,HTML DOM Events, Learning Ajax jQuery Programming: Selectors & Mouse Events, Form Events, DOM Manipulation, Effects & Animation, Traversing & Filtering

# MODULE - II

XML: Introduction to XML, Defining XML tags, their attributes and values, Document Type Definition, XML Schemes, Document Object Model, XHTML Parsing XML Data – DOM and SAX Parsers, CRUD operation.

Angular JS: Introduction, Expressions, Modules, directives, AngularJS HTML DOM, Events, Forms

# MODULE -III

Introduction to Servlets: Common Gateway Interface (CGI), Life cycle of a Servlet, deploying a servlet, The Servlet API, Reading Servlet parameters, Reading Initialization parameters, Handling Http Request & Responses, Using Cookies and Sessions, connecting to a database using JDBC.

# MODULE – IV

Introduction to JSP: The Anatomy of a JSP Page, JSP Processing, Declarations, Directives, Expressions, Code Snippets, implicit objects, Using Beans in JSP Pages, Using Cookies, and session for session tracking, connecting to database in JSP

# MODULE – V

Database Design using MySQL: An Overview of SQL, XAMPP and MySQL Setup, Create Tables, Columns and Insert Data, Selecting Data, Distinct, Aliases & Concat, Update, Delete & Alter, Foreign Keys, Table Joins. PHP Programming Fundamentals, PHP Data Types & Dates, Cookies, Sessions, Objects and Classes, Intro to PHP & MySQL, creating a MySQL Database, Connect & Fetch data from MySQL, Insert and Delete MySQL data from PHP

1. STUDENT ENTRY BEHAVIOR((List of Pre-Requisite Course if Any from I/II/III/IV Year)

|  |  |
| --- | --- |
| Course Objectives | Course Objective Statements |
| Objective - 1 |  |
| Objective – 2 |  |
| Objective – 3 |  |
| Objective – 4 |  |

1. COURSE OBJECTIVES

4. LEARNING/COURSE OUTCOMES (Specific Course Outcomes)

The Department of Computer Science at HITAM has the following expected outcomes upon completion of this course.

|  |  |
| --- | --- |
| Course Outcome | Course Outcome Statements |
| CO - 1 |  |
| CO – 2 |  |
| CO – 3 |  |
| CO – 4 |  |

Course Objective and Course Outcome Mapping

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Outcome  Objective | CO1 | CO2 | CO3 | CO4 |
| Objective - 1 |  |  |  |  |
| Objective – 2 |  |  |  |  |
| Objective – 3 |  |  |  |  |
| Objective – 4 |  |  |  |  |

5. Mapping of Course with PEOs-PSOs-POs

|  |  |  |
| --- | --- | --- |
| Sl. No. | PEOs Name | Program Education Objective Statements |
| 1 | PEO - 1 |  |
| 2 | PEO – 2 |  |
| 3 | PEO – 3 |  |

Program Specific Objectives (PSOs)

|  |  |  |
| --- | --- | --- |
| Sl.No. | PSOs Name | Program Specific Objective Statements |
| 1 | PSO - 1 |  |
| 2 | PSO – 2 |  |
| 3 | PSO – 3 |  |

Program Outcomes (POs)

|  |  |  |
| --- | --- | --- |
| PO Name | Graduate Attributes | PO Statements |
| PO1 | Engineering knowledge | Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. [PEO’s: 1,2 and 3] [PSO’s: 1,2 and 3] |
| PO 2 | Problem analysis | Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences. [PEO’s: 1,2 and 3] [PSO’s: 1,2 and 3] |
| PO 3 | Design/ development of solutions | Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations. [PEO’s: 1,2 and 3] [PSO’s: 1,2 and 3] |
| PO 4 | Conduct investigations of complex problems | Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions. [PEO’s: 1,2 and 3] [PSO’s: 1,2 and 3] |
| PO 5 | Modern tool usage | Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations. [PEO’s: 1,2 and 3] [PSO’s: 1,2 and 3] |
| PO 6 | The engineer and society | Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice. [PEO’s: 2 and 3] |
| PO 7 | Environment and sustainability | Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development. [PEO’s: 1,2 and 3] |
| PO 8 | Ethics | Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice. [PEO’s: 1,2 and 3] [PSO’s: 2 and 3] |
| PO 9 | Individual and team work | Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings. [PEO’s: 1,2 and 3] [PSO’s: 3] |
| PO 10 | Communication | Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions. [PEO’s: 1,2 and 3] [PSO’s: 2 and 3] |
| PO 11 | Project management and finance | Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments. [PEO’s: 1 and 3] [PSO’s: 2 and 3] |
| PO 12 | Life-long learning | Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change. [PEO’s: 1,2 and 3] [PSO’s: 1,2 and 3] |

Course Mapping

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Course Name | PEO1 | PEO2 | PEO3 | PSO1 | PSO2 | PSO3 |
| Design and Analysis of Algorithms | √ | √ |  | √ | √ | √ |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Course Name | Po1 | Po2 | Po3 | Po4 | Po5 | Po6 | Po7 | Po8 | Po9 | Po10 | Po11 | Po12 |
| Design and Analysis of Algorithms | √ | √ | √ | √ | √ | - | √ | √ | √ | √ | √ | √ |

6.Mapping Of Course Outcomes With PEOs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No | Course Outcomes | PEO1 | PEO2 | PEO3 |
| 1 | CO - 1 | √ | √ | √ |
| 2 | CO – 2 | √ | √ | √ |
| 3 | CO – 3 | √ | √ | √ |
| 4 | CO – 4 | √ | √ | √ |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No | Course Outcomes | PEO1 | PEO2 | PEO3 | Average |
| 1 | CO - 1 | 2 | 2 | 2 | 2 |
| 2 | CO – 2 | 3 | 3 | 2 | 2.7 |
| 3 | CO – 3 | 3 | 3 | 3 | 3 |
| 4 | CO – 4 | 3 | 2 | 2 | 2.3 |
|  | Average | 2.6 | 2.3 | 2.5 | 2.5 |

7.Mapping Of Course Outcomes With PSOs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No | Course Outcomes | PSO1 | PSO2 | PSO3 |
| 1 | CO - 1 | √ | √ |  |
| 2 | CO – 2 | √ | √ | √ |
| 3 | CO – 3 | √ | √ | √ |
| 4 | CO – 4 | √ | √ | √ |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No | Course Outcomes | PSO1 | PSO2 | PSO3 | Average |
| 1 | CO - 1 | 3 | 3 | 0 | 2 |
| 2 | CO – 2 | 2 | 3 | 2 | 2.3 |
| 3 | CO – 3 | 3 | 3 | 2 | 2.7 |
| 4 | CO – 4 | 3 | 3 | 2 | 2.7 |
|  | Average | 2.7 | 3 | 1.7 | 2.4 |

1. Mapping Of Course Outcomes With POs

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| No | Course Outcomes | Po1 | Po2 | Po3 | Po4 | Po5 | Po6 | Po7 | Po8 | Po9 | Po10 | Po11 | Po12 |
| 1 | CO - 1 | ✓ | ✓ | ✓ | ✓ | ✓ |  | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 2 | CO – 2 | ✓ | ✓ | ✓ | ✓ | ✓ |  | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 3 | CO – 3 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |  | ✓ |
| 4 | CO – 4 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |  | ✓ |
| 5 | CO – 5 | ✓ | ✓ | ✓ | ✓ | ✓ |  | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 6 | CO - 6 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |  | ✓ |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| No | Course Outcomes | Po1 | Po2 | Po3 | Po4 | Po5 | Po6 | Po7 | Po8 | Po9 | Po10 | Po11 | Po12 | Average |
| 1 | CO - 1 | 3 | 2 | 2 | 2 | 3 | 0 | 1 | 2 | 1 | 2 | 1 | 1 | 1.67 |
| 2 | CO – 2 | 3 | 2 | 2 | 2 | 3 | 0 | 1 | 2 | 1 | 2 | 2 | 2 | 1.8 |
| 3 | CO – 3 | 3 | 3 | 2 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 3 | 2.17 |
| 4 | CO – 4 | 3 | 3 | 2 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 3 | 2.17 |
|  | Average | 3 | 2.5 | 2 | 2.3 | 2.3 | 1.2 | 1.8 | 2.2 | 1.5 | 2 | 1.3 | 2.2 | 2.02 |

9. Direct Course Assessment

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No | Description | Targeted Performance | Actual Performance | Remarks | Course Attainment |
| 1 | Internal Marks(25) |  |  |  |  |
| 2 | External Marks(75) |  |  |  |  |
| 3 | Clearing of Subject(75) |  |  |  |  |
| 4 | Getting First Class |  |  |  |  |
| 5 | Distinction |  |  |  |  |
| 6 | Outstanding Performance |  |  |  |  |
| 7 | Assignments |  |  |  |  |
| 8 | Quiz Test |  |  |  |  |
| 9 | Subjective Test |  |  |  |  |
| 10 | Tutorials |  |  |  |  |
|  |  | Average |  | | |

HITAM

Department of Computer Science & Engineering

Direct Methods: Internal Marks Assessment

10. Indirect Course Assessment

(As mentioned-strong (3), moderate (2), weak (1) & no comment (0))

Mission Statement of CSE

* 1
* 2

Correlation of Mission Elements with Mission Statement of CSE Department related to the Course (only Ticking given by faculty)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No | Mission Elements | Strong | Moderate | Weak | No Comment |
| M-1 |  | √ |  |  |  |
| M-2 |  |  | √ |  |  |
| M-3 |  | √ |  |  |  |
| M-4 |  |  |  | √ |  |
| M-5 |  |  | √ |  |  |
| M-6 |  |  | √ |  |  |
| M-7 |  | √ |  |  |  |
| M-8 |  | √ |  |  |  |
| M-9 |  |  |  | √ |  |
| M-10 |  |  | √ |  |  |

Indirect Course Assessment through Student Satisfaction Survey

(Note for \*:Parameters used for course teaching like

a: Classroom teaching b: Simulations c:labs d: Mini\_Projects

e: Major Projects f: Conferences g: professional activities

h: Technical Clubs i: Guest Lectures j: Workshops k: Technical Fests l:Tutorials m:NPTLs n:Digital Library o: Industrial Visits

p: software Tools q: Internship/training r:Technical Seminars

s: NSS t: NSS u: sports etc.

Further assume other parameters if any)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Question Based on PEO/ PO/PSO/CO | Parameters  (a /b /c…/)\* | Strong (3) | Moderate (2) | Weak (1) | No comment (0) |
| 1 | Did the course impart fundamentals through interactive learning and contribute to core competence? | a,c,l,m,n,r |  |  |  |  |
| 2 | Did the course provide the required knowledge to foster continuous learning? | a,c,m,n,r |  |  |  |  |
| 3 | Whether the syllabus content anticipates & satisfies the industry and societal needs? | a,c,h,l,m,r |  |  |  |  |
| 4 | Whether the course focuses on value based education to be a socially committed professional? | a,c,h,l,m,r |  |  |  |  |
| 5 | Rate the role of the facilitator in mentoring and promoting the self learning abilities to excel academically and professionally? | a,c,h,l,m,r |  |  |  |  |
| 6 | Rate the methodology adopted and techniques used in teaching learning processes? | a,c,,l,m,n,r |  |  |  |  |
| 7 | Rate the course in applying sciences & engineering fundamentals in providing research based conclusions with the help of modern tools? | a,c,h,l,m,n,r |  |  |  |  |
| 8 | Did the course have any scope to design, develop and test a system or component? | a,c,h,l,m,r |  |  |  |  |
| 9 | Rate the scope of this course in addressing cultural, legal, health, environment and safety issues? | - |  |  |  |  |
| 10 | Scope of applying management fundamentals to demonstrate effective technical project presentations & report writing? | a,c,n,h |  |  |  |  |
| Total | |  |  |  |  |  |
| Average | |  |  |  |  |  |
| Total Average | |  | | | | |

11. Overall Course Assessment

|  |  |  |  |
| --- | --- | --- | --- |
| SNo | Assessment Type | Weightage | Attainment Level |
| 1 | Direct-Assignment, Quiz, Subjective, University Exams, Results, Bench Marks | 0.80 | 0.80(2.94)=2.35 |
| 2 | Indirect-Surveys-Questionnaire | 0.20 | 0.20(2.87)=0.57 |
|  | Overall | 1.00 | 2.92 |

Course Attainment level: Compiler Design Course=2.92

12. Pi diagrams, Bar charts, Histograms

|  |  |  |  |
| --- | --- | --- | --- |
| Pass % for last 2 Academic Years | Appeared | Passed | Pass% |
| 2024-25 |  |  |  |
| 2023-24 |  |  |  |

13. Lesson/Course Plan

Subject : FULL STACK WEB DEVELOPMENT

14. Actions and Assessment Plan (to improve the course to meet the objectives)

Students have to attend the lectures regularly and Scribe the Lecture notes on the day of the lecture.

There are the requirements other than attending lectures.

Actions

Based on these requirements certain actions are employed to improve the course plan to meet the.objectives.

Assignments

Lightweight homework assignments are posted which will be able to cover some related research and implementation works.

Mini project

Instructions on the Mini-Projects

This is a group project, with two or three students in each group, and of course, all students in a group will receive the same mark for the project.

Requirements

Students are required to submit

* + Project Report: The report must describe the system model: all the major modules (classes/methods) used in the system, and the relationships among them. Based on the System Model design the UML nine diagrams. For each diagram, the report must explain how the solutions were implemented, and list all the components and relationships. The explanation should be a concise high level description such that the implementation can be easily understood by its readers. The report should be written in font size 12 and at most 30 pages.
  + User Documentation: students must submit an electronic copy of the user documentation. The user documentation must contain
    - The installation guideline that tells the user how to install the software; and
    - A user manual for each module.

The document must also be accessible in the web page in the form of help menu.

* Each group will be asked to demo the system. The demo will last at most 30 minutes, and please make sure the project compiles and runs correctly on the lab machine and the web-pages display correctly using the browser available in the lab machines.

Students must make sure their system is ready before the scheduled demo. No later submission will be accepted.

15.Lecturer Notes:

**MODULE -I**

HTML Common tags- List, Tables, images, forms, Frames; Cascading Style sheets. Introduction to Java Scripts, Objects and Functions in java script, Manipulating DOM,HTML DOM Events, Learning Ajax jQuery Programming: Selectors & Mouse Events, Form Events, DOM Manipulation, Effects & Animation, Traversing & Filtering

**HTML HTML** => **H**yper**T**ext **M**arkup **L**anguage.

HTML is the one which decides the content to be displayed in the web pages.

<**tagname** attribute\_name="attribute value">no

→ Tag name / element name

**Tags / Elements**

<**h1**><**h2**><**h3**><**h4**><**h5**><**h6**> (Heading Tags).

<**div**></**div**>(division tag).

<**p**></**p**> (paragraph tag). <**br** /> (break new line).

<**hr** /> (horizontal line)

<**span**></**span**>

<**strong**></**strong**>

<**b**></**b**> (bold)

<**u**></**u**> (underline)

<**i**></**i**> (Italic)

<**label**></**label**> (for the input field)

**Tags**

Closing Tags - <div> <span> <p> <label> <ul> <li>

<b> <i> <u> <table> <select> <option>

<button> <textarea>

<h1>..<h6>

Self Closing Tags - <br /> <hr /> <input /> <img />

**Attributes**:

Common Attributes: name id

CSS Attributes: style class

Event attributes: onclick onchange onmouseover onmouseout

Custom attribute: give your own attribute and value

**Form Elements/Form Fields**

<input> <select> <option> <button>

# Display Tags

<div> <span> <p> <label> <ul> <li> <b> <i> <u>

**<div>** division tag - used to divide or layout the page.

eg:

<div>fdfdfd</div>

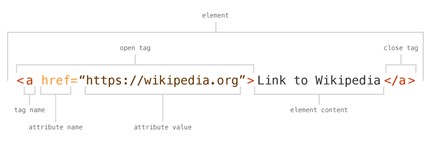
<div class="class-name1 class-name2" id="divid-12"></div>

<div><p>paragraph text here....</p></div>

<div><span>....</span><p>text here....</p><p>......</p></div> **Syntax**

Eg: <**img** src="path.png">

<a href=”#home”></a> (hyperlink/link are defined with the <a> tag and linked page are defined in href attribute).

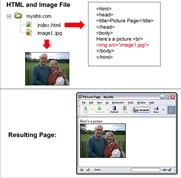


<**Img**> - Used to display image in website.

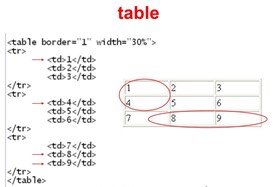
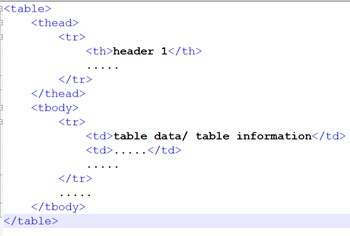
<**img** src=”source of image” alt=”if image not displayed” width=”” height=”” /> Eg: <img src="image.jpg" width="200" height="150" />

<img src="https://google.com/image.jpg" width="200" height="150" /> src – we have to specify the url of the image path in the server. Check

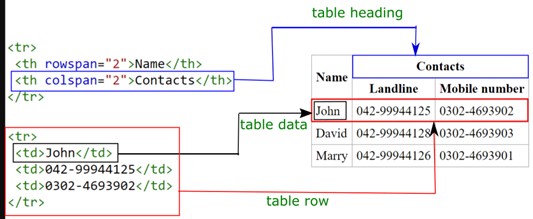
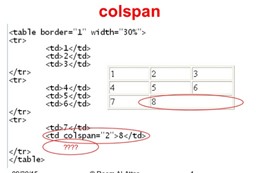
the below screenshot.



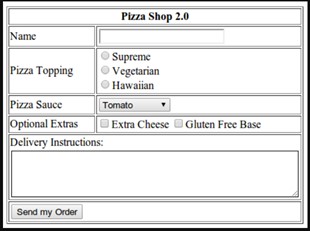
**Table** – This tag is used to display information in tabular format.



**rowspan**: Which is used to merge rows in the table. **colspan**: Which is used to merge columns of the table.



Forms



<**form action**="the page to be submitted" **method**=”POST/GET”>

**…forms elements ……**

</form>

<form action="postdata.java" method="post">

**Forms Element:**

[In the **input** tag, **field name** attribute must be submitted & used for grouping] [id & type & name => attributes].

The text input element allows people to type in one line of text. The input type is text.

Whatever typed into this text box will be send to the server with the variable name. Default input type is text. **Text Box**

<input type="text" id="firstname" name="firstname"> (Or)

<input id="firstname" name="firstname"> **Password**

<input type="password" id="pwd" name="pwd"> **Checkbox:**

Used to get multiple input from the user.

<input type="checkbox" id="vehicle1" name="vehicle1">

[properties => checked] **Radio Button**

<input type="radio" id="male" name="gender" >

[properties => checked]

**Dropdown / Dropdown list/Select Box:**

<**select** id="select\_gender" name="gender">

<option value="1"> Male </option>

<option value="2"> Female </option>

<option value="3"> Others </option>

</select>

[property=> **size** and **multiple** (to select ctrl + value )] **Option** tag properties=> value and selected

<option selected="selected" value="secret">Secret Sauce</option> Notes:

All the attributes used in this fields are optional **Text area:**

[textarea is used for a displaying address ,message ..etc]

<textarea name="delivery" cols="50" rows="4"></textarea>

Rows attribute defines the height of the text area

Cols attribute defines the width of the text area **Buttons:**

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

<input type="button" value=”click me”>

<input type="reset" value=”Clear All Form data/ Restore Form data”>

<button>Button Text Here</button>

**HTML input filed Attributes** placeholder => It gives the some information about the field required => the field must be filled value => It gives value to the input field(preselect/default value) readonly => It is only for reading not to be edited(Not editable) disabled => this will disable the particular field **HTML Select box Attributes:**

|  |  |
| --- | --- |
| readonly | => It is only for reading not to be edited(Not editable) |
| disabled | => this will disable the particular field |
| multiple | => this will allow to select multiple options |

**Iframe:**

<iframe src=”url” height=”” width=””></iframe>

By using the <iframe> we can open the multiple frames (screen/website/webpage) in your current screen.

**Lists**

<**ol**> order list

<li>….</li> -> List Items …..

</ol>

<**ul**> unordered list

<li>….</li> -> List Items

….

</ul>

**Html comments:** <!--comments -->

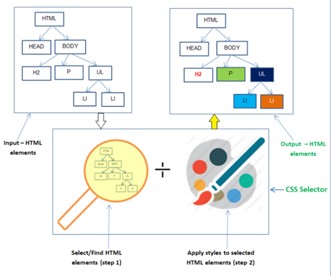
**HTML Attributes or Properties:**

Behaviour of the HTML element. [attribute or property name=”value”]

Common properties: name, id, class. We can add custom attributes in any html element.

# CSS: (Cascading Style Sheet)

- Used to design/styling the website.



Three ways of using CSS(Styles): **Internal**, **External** and **Inline** CSS Example:

**External CSS** – we can include/import external CSS file by using <link> tag.

<link rel="stylesheet" type="text/css" href="mystyle.css" /> href - location of the CSS file

**Internal CSS** – used to style the page by using style tag.

<style>

p {

color: red;

}

.class-name { }

</style>

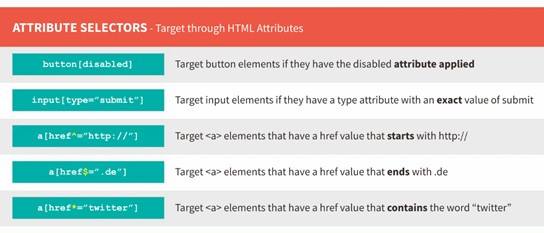
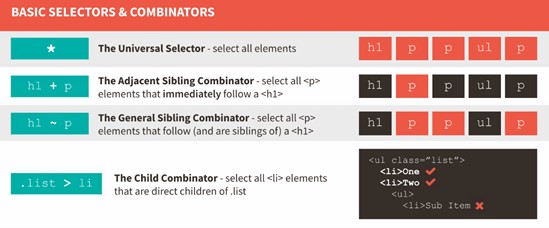
<tagname class=”class-name class-name1”>

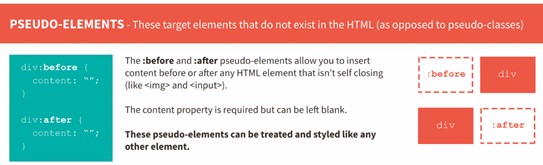
**Inline CSS** - styling the tag/section/part directly by using style attribute.

<body style="background-color: lightcyan;color:FFF;font-weight:

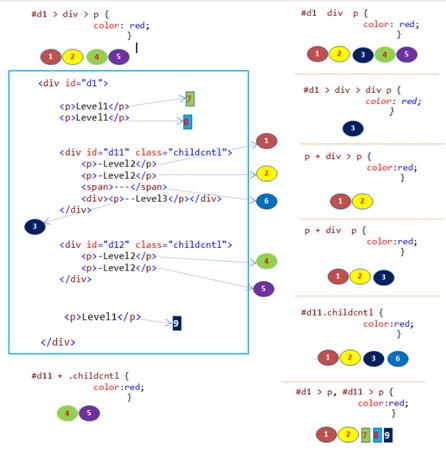
bold;marin-top:12px;">

CSS selectors:

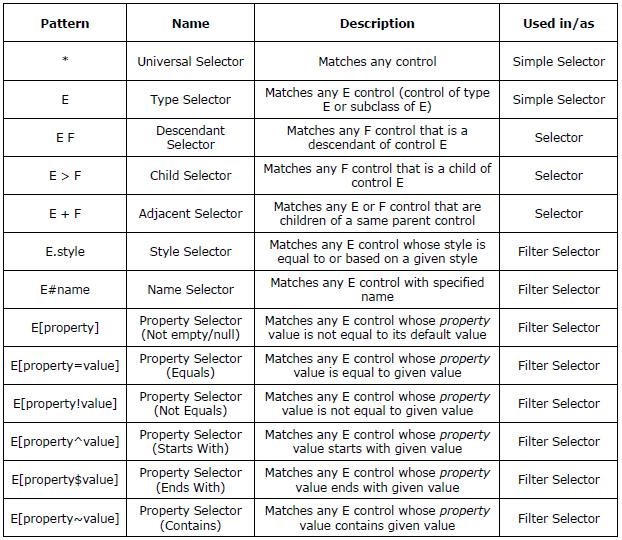








CSS selectors is one which used to the access the element, there are many selectors:



**Pseudo class [:] a:link**

=> used to give the special focus for the links a:visited

=> used to give the special focus for the visited links **a:hover**

=> used to give the special focus when the mouse is hovered **a:active**

=> used to give the special focus for the active links **p:first-child**

=> used to select first child and give the special focus for that one **p:last-child**

=> used to select last child and give the special focus for that one **p:nth-child(n)**

=> used to select nth(1,2,3..) child and give the special focus for that one **a:focus**

=> used to give the special focus for the selected elements

=> eg.. input:focus { background-color: yellow;}

**:target**

=> used to to highlight the content which is belonging to that link

=>eg: <a href=”#new”></a> <p id =”new”></p>

**CSS Comments:**

Single line => //

Multline => /\*…\*/ **Background:**

background-color:red; background-image:url(“”); background-repeat: repeat-x; (or) no-repeat; shorthand property:

=> background: red url("") no-repeat right top; **Margin:**

Used for adding outside the space around the element margin: 25px 1em 75rem 100px; (top right bottom left ) margin: 25px 50px 75px; (top right & left bottom) margin: 25px 50px; (top&bottom right & left) margi:auto ;(equal for all side) margin: 25px; (top & right & bottom & left ) margin-top: 30px; margin-right: -20px; margin-bottom: 0; margin-left: 1em;

**padding: [It’s same like margin]**

Adding space inside the element, space around the content padding: , padding-top: , padding-right: , padding-bottom: , padding-left:

**border**: [it is used to set the border around the content or element] **shorthand:**

=> border: 1px solid red; border-radius: ;(used for circle or ) border-top, border-right, border-bottom, border-left

**width:** width, max-width, min-width **height:** height, min-height, max-height

**CSS text and fonts:**

· text-align: right || left || center;

· font-weight: 600 || 700 || 800 || 900 || bold;

· font-family: Arial;

· font-size: 12px || 2rem || 1em;

· line-height:;

· word-spacing:;

· text-decoration:;

· list-style-type:;

· text-transform:; · text-indent:;

**display: used to hide or show elements**

display: block || inline || inline-block || grid || flex || table || none; visibility: similar to display, but it will just hide the element(space will be there) visibility: hidden; (it take up the space)

visibility: visible;

white-space: wrap || normal || no-wrap;

**position:**

· position: static; (default);

· position: sticky; (like scroll bar )

· position: fixed; (remain or retain the same position when scrolled)

· position: relative; (just move the element form its position ) position:absolute;(move the whole body ) top:; left:;

**overflow**: hidden || scroll || auto || visible; property specifies whether to clip the content or to add scrollbars when the content of an element is too big to fit in the specified area

· **visible** - Default. The overflow is not clipped.

· **hidden** - The overflow is clipped, and the rest of the content will be invisible · **scroll** - The overflow is clipped, and a scrollbar is added to see the rest of the content

· **auto** - Similar to scroll, but it adds scrollbars only when necessary **float:**

· float:right;(display the element in right)=>after =>clear :right;(to allocate the space for element to fit )

· float:left; (display the element in left) => after =>clear :left;

**Flex (display: flex):**

flex-direction: row || column;, flex-wrap: warp || no-wrap;, flex-flow(contain

both wrap and direction ), justify-content, align-items, align-content, flex-basis(width) **media quries:** (based on the screen resolution/width, CSS styles will be applied.) min-width – screen width should be minimum this px. max-width – screen width should be maximum this px.

For 480px, above 480px(minimum)

@media screen and (min-width: 480px) { body { }

}

For 892px or less than 892,

@media screen and (max-width: 892px) { body {}

p {}

}

· It’s used for making the screen responsive when we are resizing browser for the different pixel

**Javascript Variable [var]:**

● It stores the value act as a container and it can store any type of data type.

Two different kind of datatypes. (Primitive and Complex)

**Data types: Primitive or primary**

**String**

“” or ‘’ eg: “bala” or ‘bala’

String enclosed with **single** or **double** quotes

**Number =>** -999, 00, 11, 9999, 9.9, .99.999

**Boolean =>** true or false **Composite or Complex DataType Array Or List:**

* [] must enclosed with square brackets && separated by ,
* Start from 0 index or 0 position • It can hold any type of data type **Array example:**
* var Array = [“bala”,1,true,undefined,null,{raj:“ram”}]; • var array = [“bala”,”ram ”,‟‟sam”]; **Array accessing:**
* Array[0] = “bala”; Array[1] = 1;

**Object Or map:**

* {key name : value Pairs} must enclosed with curly braces

**Key name criteria**

* Name or “name” or key\_name or “keyName” or “key name”

**Value criteria**

* Value can hold any type of data type **Ways of Object accessing:**
* objectName.propertyName or objectName[“propertyName”]

Eg: const hero = { name: 'Bala' };

* or hero.name;//Bala
* e.g hero[“name”];// Bala
* const {property} = object • const { name } = hero;// Bala

**Delete Object property:**

* delete objectName[“propertyName”] or delete objectName.propertyName

**object example :**

* var map = {“name”:”bala”, key1:{name:”bala”,age:18 }}; **hasOwnProperty:**
* it is used to find whether the property is present inside the given object
* e.g var map={a:1,b:2} => map.hasOwnProperty(“a”) • it return true when present or return false **Ways of combining object:**

**First way:**

* var obj1={},obj2={}, obj3={…obj1,… obj2}; **Second way**:
* var obj1={},obj2={},obj3=object.assign(obj1,obj2); **Third way**:
* var obj1 = { name: "bala" };
* var obj2 = { age: "12" };
* function combiningObject(obj1, obj2) { var obj3 = {}; for (var Object in obj1) { obj3[Object] = obj1[Object];

}

for (var Object in obj2) { obj3[Object] = obj2[Object];

}

return obj3

}

var output = combiningObject(obj1, obj2); **Output** { name: "bala" , age: 12 }

**function:**

A JavaScript function is defined with the **function** keyword, followed by a **function name.**

followed by parentheses (para1,para2,para3) followed by { block of code }

**Different ways of declaring function:**

**Type1:**

function method1 () { alert("method1 on is called");

}

**method1**(); // calling the method var method2 = function () { alert("method2 is called");

}

**method2();**// calling the method function functionName(p) { return p;

}

var x = **functionName("bala");** var functionName = function (p) { return p;

}

var x = **functionName(2); Type 2:**

function greeting() { greet1(); greet2();

}

function greet1() { alert("hi");

}

function greet2() { alert("how are you !");

} greeting(); **Type 3: Inner Function(function within a function)** function process() { function process2() { var process3 = function () { alert("Hare");

} alert("Krishna"); process3();

}

process2(); alert("Hare");

}

**process(); Type 4:**

var a = function (a, b, c) { return a + b + c;

}

function b() { var c = a(1, 2, 3); return c;

}

var c = b();

alert(c);

**Type 6: Pass method by argument (callback method)** function method1(x) {

alert(x);

}

function method2(**callback**) { **callback**("bala");

}

**method2 (method1);**

function m(x) { alert(x);

}

function m1(var1, **callback**) { **callback**(var1);

}

**m1(2, m);**

**Example 1:**

function f1(par, **callback**) { **var a = callback**(par);

alert(a);

}

f1("balamurali", function (x) { return x;

});

**Example 2:**

function tryMe(param1, param2) { alert(param1 + " and " + param2);

}

function callbackTester(callback, para, para1) {

callback("hello", "goodbye");

}

callbackTester(tryMe);

**Example 3:**

function f1(param1, param2, param3) { alert(param1 + " and " + param2 + " " + param3);

}

function f2(callback) { var extraParam = "this data was missing"; callback(extraParam); .

}

var c= function (para) { f1("hello", "goodbye", para); }; f2(c); **(or)**

f2(function (para) { f1("hello", "goodbye", para); });

**Function in objects type1.1 : Object with function** var map = {

name: "bala", method: function () { alert("the name is bala");

}

};

**=> map.method(); example 2** var map = {

name: "bala", age: 18, method: function () { alert("the name is bala");

},

method1: function () { alert("the age is 18");

}

};

**=>map.method() ;=>map.method1();**

**example 3** var map = {

name: "bala", age: 18, location: "salem", method: function (val) { alert("the name is " + " " + val);

},

moreFunction: { method1: function (val2) { alert("the age is" + " " + val2);

},

method2: function (val3) { alert("the location is" + " " + val3);

}

}

}

**=> map.method("bala");**

**=> map.moreFunction.method1("18"); => map.moreFunction.method1("location"); example 4** var person = {

firstName: "bala", lastName: "murali", age: 15, id: 154, fullName: function () { return this.firstName + "" + this.lastName;

}

}

person.fullName() **example 4.1**

var person = { firstName: "bala", lastName: "murali", age: 15, id: 154, fullName: function () {

alert(this.id);

}

}

**person.fullName()** example 4.3 var person = {

firstName: "bala", lastName: "murali", age: 15, id: 154, fullName: function (para) { alert(para + "" + this.id);

}

}

**person.fullName("bala")**

**Special data types**

**Null && Undefined**

* Both have no value
* When we compare null == undefined is true
* When we compare null === undefined is false

**Immutable**

|  |  |
| --- | --- |
| • | Numbers |
| • | String |
| • | Boolean |
| • | Null |
| •  **Mutable** | Undefined |
| • | Array |
| • | Object |
| • | Functions |

**Javascript operational behaviour**

* Javascript operation always start from **left to right** and **top to bottom String methods** length(count from 1)
* Finds the **total length** of element present
* Var txt= “bala”;txt.length= 4

**indexOf(count from 0 && it add the counts for the special character)**

* It returns the first position of given string • Var txt= “bala”;txt.indexOf(“a”)= 3
* It **returns -1** if the text not found.

**lastIndexOf(count from 0 && it add the counts for the special character)**

* It returns the last position of given string • Var txt= “bala”;txt.lastIndexOf(“a”)= 3
* It **returns -1** if the text not found.

**trim()**

* It **removes** the **white space** present in string • Var txt= “ bala ”;txt.trim()=“bala”;

**concat()**

* It **appends** the two string
* var a1= “a”,a2= “b”;
* var txt = a1+ “ ” + a2; **=>”a b”**

a1.concat(a2) **=>ab**

**charAt()**

* It **find the character** of the **specified index** • var name= “bala”;name.charAt(0) =>b

**slice ()[not affect the original array]**

* It is used to cut out the particular part of string
* Slice(startindex , endindex[ **if we give 6 it will take upto 5th values]** ) **substring()**
* It is used to cut out the particular part of string same like slice method • subString(startindex,endindex) **split ()**
* It is used to convert **string to array**
* Var a= “a”;
* a.split();=>[“a”]

**Number methods**

**Number()** => Converting to Number **parseInt()**

* it allow only the integer value parseInt(“10.35”)=>10 **parseFloat()**
* it allow float && integer value parseInt(“10.35”)=>10.35

**Array methods**

**slice()**[**not affect** the original array] arrayName.slice(**startingIdx**, **endingIdx**)

* It is used to cut out the particular array element
* arrayName.slice(startindex , endindex[ **if we give 6 it will take upto 5th values]**).

**endindex** is optional, if not specified it will take last index position.

**splice()**[it **affect** the original array] arrayName.splice(**startingIdx**, **count**)

* It is used to cut out the particular part of element
* arrayName.splice(startindex ,count [ **if we give 6 it will take upto 5th values]**,items to add) **count** is optional, if not specified it will remove all the elements from startIdx.

**push()**

* It appends the **new element/Item in last position.**
* var array=[“a”,“a”,“a”,“a”]; array.push(“b”);

**concat()**

* It appends the two array
* var a1 = [“a”],a2= [“a1”];
* var a3= a1.concat(a2);=>[“a”,”a”];

**length()** -count from 1

* Finds the total length of element present
* var arrayList= [“bala”]; arrayList.length; 1

**sort ()[a-b ascending order][b-a descending order]**

* It is used to arrange array element in both ascending and descending order
* var num = [1,5,1,4,2];
* function myFunction() { num.sort(function(a, b){return a – b or return b - a }); return num;

}

**join ()**

* it convert the **array to the string**
* var txt=["a","a"];
* txt.join()=> “a,a”

**Array iteration methods**

**Functionality:**

In this array iteration method function passed as the parameter and call back itself until condition met..

**Notes:**

* 3 argument can be passed 1 is necessary and other two are optional
* First one check for the items and second one check for index (0,1,2) and third one for the whole array..

**forEach()[used for making iteration]**

* forEach(function (element,index,array)){ });

**map()[create new array]**

* map(function (element, index, array)){ }); **filter()[in return statement except 0 , false ,undefined]**
* filter(function (element, index, array)){ });

**find()[find the first element which pass the test ]** • find(function (element, index, array)){ });

**findIndex()[find the first element index which pass the test ]**

* findIndex(function (element, index, array)){ });

**JSON[ javascript object notation ]**

**Stringify()**

* **It convert the object into the string**
* Var myObj = { name: 'bala',age: 25,}
* Var myObjStr = JSON.stringify(myObj);
* Output:"{"name":"bala","age":26}"

**Parse()**

* **It convert the string into the object**
* Var myObj = “{ name: 'bala',age: 25,}”
* Var myObj1= JSON.parse (myObj);
* Output: { name: 'bala',age: 25,}

**DOM(document object model) :**

* document.getElementById(“”).innerHTML **javascript output :**
* document.getElementById(“”).innerHTML
* document.write();
* console.log();
* alert();

**key events onclick**

* used in button

**keyup**

* after key is pressed(after calling) **keypress**
* before key is pressed(before calling) **onchange**
* it is used in the selectbox,radiobutton,and checkbox

**variable scope global scope**

* variable declared outside the function **local scope**
* variable declared inside the function **example**
* var global=10;

function functionName(){

var local=5;

}

**Js dates[objects]**

* It allow us to set and get the date
* **var d = new Date**(year, month, day, hours, minutes, seconds,

milliseconds)=>it will display the current date that is today date

* **Tue Mar 31 2020 15:23:34**

**setDate**

* **var d = new Date()**
* **d.setFullYear(2019);**
* Sun Mar 31 2019 15:30:00(it will change the year from current date )
* **var d = new Date();**
* **d.setDate(d.getDate() + 50);**
* Wed May 20 2020

**getDate**

* **var d = new Date();**
* **d.getTime();=>1585650751531**
* **d.getFullYear();=>2020**
* d.getDay();=>(0(sunday)-6(saturday))
* d.getMonth ();=>(0(jan)-11(dec))

**Js math**

* math.ceil(3.4)=4;
* math.floor(3.4)=3;
* math.abs(-3)=3;
* math.pow(x,y)[x to power of y];
* math.sqrt(64)=8;

**Regex[regular expression]**

* i=>**ignore case sensitive;** ( it negelct the small and captial difference A=a ) • g=>**select globally**;(select all the similar value )

**test() method**

* **return true or false**
* /s/.test("s")=> true • /c/.test("s")=> false

**replace () method** • **find and replace the word**

* var str1 = "Visit";
* var str2 = str.replace("t","k");=>visik

**search() method**

* **find and return the position of the element**
* var str1 = "Visit";
* var str2 = str.search("t");=>4

**exec() method**

* **return the match values position or null**
* /l/.exec("bala")=>2 • /3/.exec("bala")=>null **match() method**
* **return the matched value or null**
* var str = "corano";
* var res = str.match(/o/g);=>o,o • var res = str.match(/K/g);=>null **conditional operators**
* var name= (condition)?val1:val2(like if and else) **conditional statement** • if(condition){}
* if(condition){}else{}
* if(condition) {}elseif{}else{}

**Loops[used for the iteration purpose until the given condition met ] Syntax for loop**

* for (initialization;condition; increment or decrement){ console.log();

}

**for in loop[for objects and array]**

**syntax**

* for (var name in object){

}

**example**

* var personDetails={name:"krishna",surname:"moorthy",age:2} for(var person in personDetails) { console.log (person + ":"+ personDetails[person]);

}

**Output: name:Krishna, surname:moorthy, age:25 for of loop[for array and string]**

**Syntax** for (var of iterabale){ }

**Example 1**

var fruits =["apple","mango","orange","strawberry"]; for(var fname of fruits) { console.log(fname);

} **output:**

* apple mango
* orange
* strawberry

**Example 2**

* var fruits ="apple";

for(var fname of fruits) { console.log(fname);

}

**output: a,p,p,l,e**

**Break =>** it is used to stop the loop at particular condition

Syntax: **if(condition){break};**

**Continue =>** It is used to skip the loop at particular condition

Syntax: **if(condition){continue}; while loop[executed when condition is true ] syntax**

* initialization
* while (condition) {

increment or decrement

}

do while loop[executed once even if condition is failed]

**syntax** • initialization

* do (condition)

{

increment or decrement;

}

while (condition) **Operator :**

* == - equal to (equal type )
* === - strict equal to(equal value and equal type)
* != - not equal to
* > - greater than
* >= - greater than equal to
* <- greater than
* <=- greater than equal to
* &&- AND operator
* ||- OR operator
* ! –not operator **Form data:**

**·** FormData is simply a data structure that can be used to store key-value pairs.

* you can create a FormData object **syntax const formData = new formData()**

**From data methods();**

**Append()**

* it is used to add a value to the object, with the specified key. If the key already exists, the value is added to that key, without eliminating the first one

**delete () =>** deletes a key value pair

**entries () =>** returns an Iterator object which you can use to list the available keys in the object **get () =>** to get the value associated with a key. If more than one value was appended, it returns the first one

**getAll() =>** get all the values associated with a key **has() =>** returns true if there’s a key

**keys() =>** returns an Iterator object which you can use to list the available keys in the object

**set() =>** to add a value to the object, with the specified key. If the key already exists, the value is replaced

**values() =>** returns an Iterator object for the values of the FormData object.

**Prototype:**

• the JavaScript **prototype** property allows you to add new properties to object

constructors:

**Example:**

function Person(first, last, age) { this.firstName = first; this.lastName = last; this.age = age;

}

Person.prototype.nationality = "English";

var personDetails = new Person("ram", "kumar", 50); console.log( personDetails.nationality); **Output:**

**It will automatically add the nationality property to the the**

**person….**

**JAVASCRIPT, NODEJS And TYPESCRIPT TYPESCRIPT**

* Typescript is the superset of JavaScript i.e that is the advanced version of JavaScript
* Typescript == JavaScript + some advanced features
* Typescript does not directly supported by the browser i.e it is not supported by browser
* To run the typescript we need to convert the typescript file to JavaScript file (.ts to .js)
* To convert the .ts to .js we need node.js it is language which is purely made on the javascript syntax with the help of the npm module we can convert the .ts file to .js file
* Then we can run the typescript
* Javascript does not support typescript but in typescript we can write

javascript

**Arrow function: (=>)**

* Uses of arrow function makes our code shorter and it make our code in compact way
* By using arrow functions, we can avoid having the function keyword, return keyword (optional) and curly brackets (optional).

**Ways of declaring arrow function Normal function:** var m1 = function(x, y) { return x \* y;}; m1(5,6) // output => 30

**Arrow function:**

**Example 1:** [without return and curly braces] var m1= (x,y) => x\*y; m1(5,6) // output:30

(or)

**[with return function and curly braces]** const m1= (x,y) => { return x\*y } m1(2,1) // output: 2

**Example 2: [with parentheses]** const a = (p) => "hello" +" "+ p; a(“raj”) // output: “hello raj”

**Example 3: [without parentheses (only single parameter)]** const c = d => "hi" + " " + d;

c(“ram”); //output:”hi ram”

**Example 4: forEach – callback method** const e= [1,2,3];

e.forEach(x => console.log(x));

//output:1,2,3

**Example 5:**

cost e= [1,2,3];

e.forEach(x => { let a = 2 + x; console.log(a); });

//output:3,4,5

**Example 5:**

var animals = ['cat', 'dog', 'tiger']; animals.forEach((element) => {

console.log(element);

});

//output: 'cat', 'dog', 'tiger'

**Example 6: map**

let n =[1,2,3]; let dn = n.map( i =>( i\*2));

//output:[2,4,6]

**Example 7:**

var obj= (id, name) => ({ id: id, name: name }); console.log(obj(101,”ram”));

**Example 8:**

const one= (s1) => (s2)=> console.log(s1,s2); one(“s1”)(“s2”); // output: s1 s2

**Spread operator** spread operator is used to spread an array or object elements

you can create new object from existing object I,e used to add object and array example const one={x:1,y:2}; const two={...one,z:5};

**// output : {x: 1, y: 2, z: 5}**

// console.log(one.x.two.x)// **output: 1 1**

**Type :** It is used to define the data type.

string, number, boolean, any **Syntax:** var identifier(varname):type(eg:string) = value (related to type); **Example:**

* var age:number = 32; => number variable
* var name:string = "John";=> string variable
* var isUpdated:boolean = true; => Boolean variable • var name:any=23; => assign any value

**const:**

The const makes a variable a constant where its value cannot be changed. Instead of let we can use const const declarations are block scoped

const cannot be updated or re-declared for same name **Example**:

const num:number = 100;//correct way const num:number = 150;//(re-declare different value for same name is wrong

)

**let:** let are used for the variable declaration as like the var.

let are block scoped let can be updated and cannot be re-declared.

**Example:**

let num:number = 100;//correct way let num:number = 150;//correct way

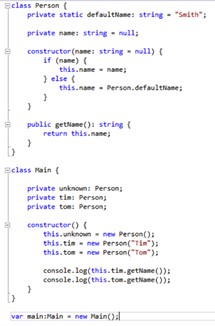
**Class**

* classes are used to create reusable components. Functionality are passed down to classes and objects are created from classes.
* The **super keyword** is used to call the parent constructor and passes the property values.
* classes can be extended to create new classes with inheritance(from parent),using the keyword extend
* void keyword it should return nothing.

**Syntax** class ClassName { field(it used for defining the data belongs to which data type ) constructor( Responsible for allocating memory for the objects ) function (action to done )

}

**Example**



**interface**

It defines the syntax for classes to follow. Classes that are derived must follow the structure provided by their interface.

It contain methods, property and function or arrow function

Implement keyword is used to implement the interface

**Syntax**

Interface IEmployee{ name:string;

empCode: number;

sayHi()=>string

}

var customer:IEmployee = { firstName:"Tom", lastName:"Hanks", sayHi: ():string =>{return "Hi there"}

}

**Module:**

A module can be created using the keyword export and a module can be used in another module using the keyword import **Import:** import keyword is used to import the one module to used in another module **Syntax:** import {class/interface name } from 'module\_name'; **example** import { Employee } from "./Employee";

**export:**

A module is a one which can contain functions, variables, interfaces and classes. Use the prefix export keyword with all the definitions you want to include in a module and want to access from other modules.

**Syntax** export interface Employee {

}

**Example:**

export class Addition {

constructor( x: number, y: number){

}

Sum() { console.log("SUM: " +(this.x + this.y));

} }

class Substraction{

constructor( a: number, b: number){

}

Substract(){ console.log("SUBSTRACTION: " +(this.a - this.b));

}

}

export Substraction;

import {Addition} from './addition'; import { Substraction } from './subraction';

let addObject = new Addition(10, 20); let subObject = new Substraction(20, 10); addObject.Sum(); subObject.Substract();

**1. Selectors and mouse events**

Selectors  
jQuery selectors use CSS syntax to find and select HTML elements, which are then used for manipulation or event handling. All jQuery selectors start with the '$()' factory function.

|  |  |  |
| --- | --- | --- |
| Syntax | Description | Example |
| $(“#id”) | Selects a single, unique element by its ID attribute. | $('#myButton') |
| $(“.class”) | Selects all elements with a given CSS class. | $('.myClass') |
| $(“element”) | Selects all elements with a given tag name. | $('p') |
| $(“element.class”) | Selects all elements with both the given tag and class. | $('p.intro') |
| $(“element, element”) | Selects multiple elements by combining selectors. | $('h1, h2, h3') |
| $(this) | Selects the current element within an event handler. | $(this) |

Mouse events  
These events are triggered by user interaction with a mouse.

|  |  |  |
| --- | --- | --- |
| Method | Description | Example |
| .click() | Executes a function when an element is clicked. | $('button').click(function() { // do something }); |
| .dblclick() | Executes a function when an element is double-clicked. | $('img').dblclick(function() { // do something }); |
| .hover() | Executes a function when the mouse enters and leaves an element. | $('div').hover( function() { // on mouse enter }, function() { // on mouse leave }); |
| .mouseenter() | Executes a function when the mouse pointer enters the selected element. | $('#box').mouseenter(function() { // do something }); |
| .mouseleave() | Executes a function when the mouse pointer leaves the selected element. | $('#box').mouseleave(function() { // do something }); |

**2. Form events**

Form events fire based on user interactions with form elements like <input>, <select>, and <textarea>.

|  |  |  |
| --- | --- | --- |
| Method | Description | Example |
| .submit() | Executes a function when a form is submitted. This can be used for validation. | $('form').submit(function(event) { event.preventDefault(); }); |
| .focus() | Executes a function when a form element gains focus. | $('input[type="text"]').focus(function() { // do something }); |
| .blur() | Executes a function when a form element loses focus. | $('input[type="text"]').blur(function() { // do something }); |
| .change() | Executes a function when the value of a form element is changed. | $('select').change(function() { // do something }); |

**3. DOM manipulation**

The Document Object Model (DOM) is a tree-like structure representing all HTML elements on a page. jQuery simplifies altering the content, structure, and styling of these elements.

Adding and removing elements

* .append() and .prepend(): Add content to the beginning or end of selected elements.
* .after() and .before(): Add content after or before selected elements.
* .remove(): Removes the selected element(s) and its children.
* .empty(): Removes only the child elements of the selected element(s).

Manipulating content and attributes

* .text(): Sets or returns the text content of selected elements.
* .html(): Sets or returns the HTML content of selected elements.
* .val(): Sets or returns the value of form elements.
* .attr(): Gets or sets the value of an attribute for the selected element.

**4. Effects and animation**

jQuery has pre-built effects for common animations and provides the powerful .animate() method for custom effects.

Built-in effects

* .hide() / .show() / .toggle(): Hides, shows, or toggles the visibility of elements.
* .fadeIn() / .fadeOut() / .fadeToggle(): Changes the opacity of elements to show or hide them.
* .slideUp() / .slideDown() / .slideToggle(): Hides or shows elements with a sliding motion.

Custom animations

* .animate({styles}, speed, callback): Creates custom animations by changing CSS properties over a specified duration.
* Chaining animations: Multiple .animate() calls are automatically queued and run sequentially on an element.

5. Traversing and filtering

Traversing and filtering methods are used to navigate up, down, and across the DOM tree to find and select specific elements based on their position or relationship.

**Traversing the DOM**

* Moving up:
  + .parent(): Gets the immediate parent of each selected element.
  + .parents(): Gets all ancestor elements of each selected element.
  + .closest(): Gets the closest ancestor that matches a selector.
* Moving down:
  + .children(): Gets all direct children of each selected element.
  + .find(): Gets all descendant elements that match a selector.
* Moving sideways:
  + .siblings(): Gets all sibling elements of each selected element.
  + .next() / .prev(): Gets the next or previous sibling of each selected element.

**Filtering elements**

* .first() / .last(): Selects the first or last element from the selection.
* .eq(index): Selects an element at a specific index from the selection.
* .filter(selector): Filters the selected elements to only keep those that match the selector.
* .not(selector): Removes elements from the selection that match the selector.

Ajax with jQuery

jQuery simplifies Ajax (Asynchronous JavaScript and XML) by handling asynchronous HTTP requests without requiring a full page refresh.

Core Ajax methods

* $.ajax({ url, method, data, success, error }): A powerful, highly configurable method for making Ajax requests.
* $.get(url, callback): A simplified method for an HTTP GET request.
* $.post(url, data, callback): A simplified method for an HTTP POST request.
* .load(url, data, callback): Loads data from the server and inserts the returned HTML into a selected element.

**MODULE -II**

XML (Extensible Markup Language) is a markup language designed for storing and transporting data, making it both human-readable and machine-readable. It's used for defining and storing data in a shareable manner, facilitating information exchange between systems like websites, databases, and applications. Unlike [HTML](https://www.google.com/search?safe=active&sca_esv=a1754dd2adcb8a92&rlz=1C1GCGC_enIN1127IN1127&sxsrf=AE3TifMLtTDUXdJ89nsebj4Bob4DbDR6hw%3A1755100651969&q=HTML&sa=X&ved=2ahUKEwjqkp-3k4iPAxV6UWwGHYfMGKcQxccNegQIHhAB&mstk=AUtExfAB3T1ERYrd5xBD5dBsFHit4ulcu4uqy2CfyXWf7-ik7J2laodIFEjOmbZ5YUx4u3fgfA7Pv3-x8Z6Z-xXKH95kWHUGexzxZhloODrPrw4JwI3-_09QkVFnW9zO5nOBiDN4cbprKOLhG4-Jp08N3lLXGpOWV_ZSDCtONDSgFxKCt01xXIh2TFKMYFrjgMSIbkzI8aNcwi9ViVbFDOXLdzsPxrtn5Cc6IFiO5dsdv9peFA7-7lA3u_X5Lf75-Glu5CRazxD3t57MuhlZl-n1iZUQ&csui=3), XML doesn't have predefined tags; instead, you define your own tags to suit your specific needs.

Key Characteristics:

* **Markup Language:** XML is a set of rules for encoding documents, not a programming language.
* **Data Storage and Transport:** It's designed to store data in a structured format and facilitate its transmission between systems.
* **Self-Descriptive:** XML tags describe the data they contain, making it easy to understand and process.
* **Extensible:** You define your own tags, making it flexible for various data structures.
* **Software and Hardware Independent:** XML can be used on different platforms and with different software.

XML Importance

* **Data Interchange:** XML is a standard for exchanging data between different systems, enabling interoperability.
* **Configuration Files:** It's commonly used for storing configuration settings in applications.
* **Web Services:** XML is a key component in many web service technologies.
* **Data Storage:** XML can be used as a format for storing data in databases and files.

**Comparing XML with other formats**

* XML vs. HTML: While both are markup languages, HTML focuses on displaying data with a fixed set of tags, while XML is used to store and transport data, allowing for custom tags.
* XML vs. JSON: JSON (JavaScript Object Notation) is a lighter-weight format often preferred for web APIs due to its simplicity, says Amazon Web Services. XML, however, excels in situations requiring complex document structures, schema validation, namespaces, and richer data typing capabilities.

**XML today:**

Despite the rise of JSON, XML continues to be relevant and widely used, particularly in enterprise systems, financial institutions, document management, and areas where its strengths like schema validation, namespaces, and complex data type support are crucial.

**Basic XML structure example**

xml\_introduction = """

<?xml version="1.0" encoding="UTF-8"?>

<root>

    <element>Content</element>

</root>

"""

* The XML above is quite self-descriptive:
* But still, the XML above does not DO anything. XML is just information wrapped in tags.
* Someone must write a piece of software to send, receive, store, or display it

**The Difference Between XML and HTML**

1. XML and HTML were designed with different goals:
2. XML was designed to carry data - with focus on what data is
3. HTML was designed to display data - with focus on how data looks
4. XML tags are not predefined like HTML tags are
5. XML Does Not Use Predefined Tags

The XML language has no predefined tags.

The tags in the example above (like <to> and <from>) are not defined in any XML standard. These tags are "invented" by the author of the XML document.HTML works with predefined tags like <p>, <h1>, <table>, etc.With XML, the author must define both the tags and the document structure.

XML is Extensible:Most XML applications will work as expected even if new data is added (or removed).Imagine an application designed to display the original version of note.xml (<to> <from> <heading> <body>).

Then imagine a newer version of note.xml with added <date> and <hour> elements, and a removed <heading>.The way XML is constructed, older version of the application can still work

**Defining XML tags, their attributes and values**

XML (Extensible Markup Language) provides a flexible framework for structuring data, allowing users to define their own tags (elements), attributes, and values to represent information in a meaningful way.

1. XML tags (elements)

* Customization: Unlike HTML's predefined tags, XML allows you to create your own tags to describe your data precisely.
* Structure: Tags act as containers for data and define the structure of your XML document. They establish a hierarchy, with a single root tag enclosing all other elements.
* Syntax:
  + Start tags begin with < and end with >, for example: <book>.
  + End tags begin with </ and end with >, for example: </book>.
  + Empty elements can be represented with a self-closing tag, like <empty\_element/>.
* Case Sensitivity: XML tags are case-sensitive. <book> is different from <Book>.
* Nesting: Elements must be properly nested. If an element starts inside another element, it must also end inside that element. For example, <parent><child></child></parent> is correct, while <parent><child></parent></child> is not.
* Root Element: Every XML document must have a single root element that encloses all other elements.

2. Attributes

Attributes are name-value pairs that provide additional information about an element and are placed within the element's start tag. They are written as attribute\_name="attribute\_value" using either single or double quotes for the value. An element cannot have duplicate attribute names.

3. Values

Values in XML can refer to attribute values or the text content within an element's tags. Attribute values are typically simple strings or numbers, while element content is the data held between the start and end tags. Elements can also contain both text and other elements (mixed content).

Example

xml

<?xml version="1.0" encoding="UTF-8"?>

<library type="public">

<book category="fiction" isbn="978-0140449179">

<title>1984</title>

<author>George Orwell</author>

<year>1949</year>

</book>

<book category="science" isbn="978-0140449193">

<title>Cosmos</title>

<author>Carl Sagan</author>

<year>1980</year>

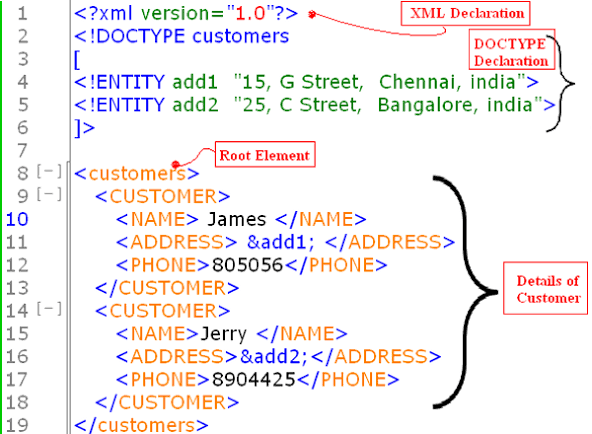
</book>

</library>

In this example, <library> is the root element with a type attribute. <book> elements have category and isbn attributes. Elements like <title>, <author>, and <year> contain text content as their values. This structure allows for flexible data representation using custom tags, attributes, and values.

**Document Type Definition**

A Document Type Definition (DTD) is a set of rules that defines the structure and the legal elements and attributes of an XML document. Think of it as a blueprint for an XML document.



**Here's a breakdown of DTDs:**

1. What it defines

* Elements: The tags you can use, like <note>, <to>, <from>, etc.
* Attributes: The properties of those elements, such as id or type.
* Relationships: How elements are nested and how they relate to each other, forming a tree structure.
* Data Types (limited): While XML Schema (XSD) offers robust data type definitions, DTDs primarily use #PCDATA (parsed character data) to signify text content.

2. Why use DTDs?

* Validation: DTDs enable XML parsers to validate whether an XML document conforms to the defined structure and rules. This is crucial for maintaining data integrity and consistency, especially when exchanging data between different systems or applications.
* Standardization: DTDs allow different groups or applications to agree on a common structure for data exchange, promoting interoperability.
* Documentation: DTDs act as a form of documentation, providing a clear understanding of the expected structure of an XML document.
* Entity Declarations: DTDs allow you to define reusable strings or special characters as entities, which can be referenced within the XML document.

3. How DTDs work

* DOCTYPE Declaration: An XML document associates itself with a DTD through a DOCTYPE declaration at the beginning of the document.
* Validation Process: A validating XML parser uses the DTD to check if the XML document adheres to the specified rules. If the document violates any of the rules, the parser reports an error, [Link: according to IBM https://www.ibm.com/docs/en/dmrt/9.5?topic=dtds-document-type-definitions-overview].

4. Types of DTD declarations

* Internal DTD: The DTD is declared within the XML file.

Example:

xml

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE note [

<!ELEMENT note (to,from,heading,body)>

<!ELEMENT to (#PCDATA)>

<!ELEMENT from (#PCDATA)>

<!ELEMENT heading (#PCDATA)>

<!ELEMENT body (#PCDATA)>

]>

<note>

<to>Tove</to>

<from>Jani</from>

<heading>Reminder</heading>

<body>Don't forget me this weekend!</body>

</note>

```

* External DTD: The DTD is defined in a separate .dtd file and referenced from the XML document.

**Example XML with external DTD reference:**

xml

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE note SYSTEM "note.dtd">

<note>

<to>Tove</to>

<from>Jani</from>

<heading>Reminder</heading>

<body>Don't forget me this weekend!</body>

</note>

```

Example note.dtd file:

xml

<!ELEMENT note (to,from,heading,body)>

<!ELEMENT to (#PCDATA)>

<!ELEMENT from (#PCDATA)>

<!ELEMENT heading (#PCDATA)>

<!ELEMENT body (#PCDATA)>

```## 5. DTD vs. XML Schema (XSD)

While DTDs are still in use, particularly for compatibility with older systems or simpler documents, XML Schema Definition (XSD) has largely replaced them. XSDs offer several advantages:

* Data Types: XSDs support various data types, such as integers and dates.
* Namespaces: XSDs allow namespaces, which help avoid naming conflicts.
* Complexity: XSDs can define more complex structures and validation rules.
* Extensibility: XSDs are more extensible and modular than DTDs.

DTDs provide a fundamental way to define the structure of XML documents and ensure their validity. However, for more complex applications requiring sophisticated data type handling, namespaces, and complex validation rules, XML Schema (XSD) is generally preferred.

**XML Schemes**

XML Schemas (XSD) are a powerful mechanism for defining the structure, content, and data types of XML documents. They act as a blueprint or rulebook, ensuring that XML data adheres to predefined formats and constraints, which is critical for data integrity and interoperability.

Key characteristics and benefits

* Structure Definition: XSDs precisely define the elements and attributes that can appear in an XML document, their order, and their nesting relationships.
* Data Type Support: XSD supports a rich set of built-in data types (like string, integer, date) and enables the creation of custom data types derived from these, allowing for precise control over data values. This is a significant advantage over DTDs, which lack data type support.
* Validation: XSDs enable validation of XML documents, ensuring that they conform to the defined schema. This helps to prevent errors and ensure data quality.
* Namespaces: XSDs support XML namespaces, which are crucial for preventing naming conflicts when combining XML documents or components from different sources.
* Extensibility: XSDs are extensible, allowing for the reuse of existing schema components and the derivation of new elements from existing ones.
* XML-based Syntax: XSDs are written in XML itself, making them easily processable by XML tools and parsers.

XML Schema vs. DTD

While DTDs (Document Type Definitions) were an earlier mechanism for defining XML document structure, XSDs offer several advantages that make them a preferred choice for many applications.



|  |  |  |
| --- | --- | --- |
| Feature | DTD | XSD |
| Syntax | Uses its own syntax, different from XML. | Uses XML syntax, allowing for the use of XML processing tools. |
| Data Types | Does not support data types. | Supports a rich set of data types and allows for custom data type creation. |
| Namespaces | Does not support namespaces. | Supports namespaces, enabling better organization and avoiding name conflicts. |
| Extensibility | Not extensible. | Extensible, supporting the reuse of schema components and the derivation of new elements. |
| Validation | Provides basic structural validation. | Offers a deeper level of validation, including patterns, lengths, and value constraints. |

**Uses and applications**

* Data Exchange: XSDs facilitate reliable and consistent data exchange between various systems and applications by defining a common language for data organization and content verification.
* Web Services: XSDs are widely used in web services for defining the structure of messages exchanged between clients and servers.
* Configuration Files: Many software applications use XSDs to define the structure of their configuration files, ensuring proper format and data validity.
* Document Processing: XSDs aid in document processing by providing a blueprint for creating, storing, and exchanging structured documents.
* Database Integration: XSDs help in linking XML data with database layouts, easing data transfer between XML-based systems and databases.

**XSD tools**

Several tools are available for creating, editing, validating, and visualizing XML Schemas, making it easier to work with these complex definitions. These tools often offer features like:

* Graphical editors: Visual representations of the schema structure, allowing for drag-and-drop editing.
* Validation and error checking: Identifying and reporting errors in the schema or in XML documents validated against it.
* Code generation: Generating code in various programming languages from the XSD, enabling easier data binding.
* Documentation generation: Creating human-readable documentation from the XSD, making it easier for developers to understand the schema.

**Document Object Model**

The Document Object Model (DOM) is a programming interface for web documents, specifically HTML and XML. It provides a structured, tree-like representation of the document, allowing programs (primarily JavaScript) to access, modify, and interact with the content, structure, and style of a web page dynamically**.**

1. What the DOM represents

When a web page is loaded, the browser creates the DOM, which is an in-memory representation of the HTML document. It acts as a bridge, or interface, between the static HTML document and the dynamic capabilities of scripting languages like JavaScript.

The DOM represents a document as a logical tree where:

* The document itself is the root node.
* HTML elements (like <html>, <head>, <body>, <div>, <p>, <img>) are represented as element nodes.
* Text content within elements are text nodes.
* Attributes of HTML elements are attribute nodes.
* Comments within the HTML are comment nodes.

2. Key functions of the DOM

The DOM empowers JavaScript to control and manipulate web pages in powerful ways, including:

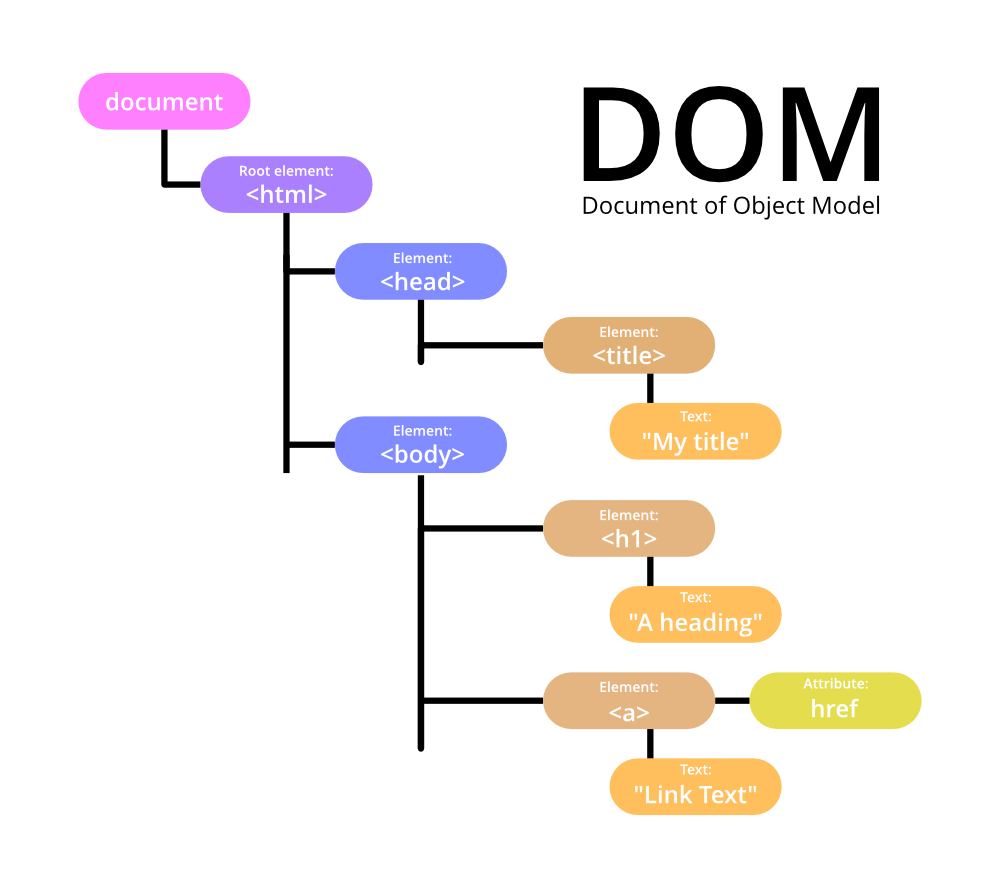
* Accessing and selecting elements: Finding specific HTML elements within the document based on their ID, class name, tag name, or CSS selectors.
* Modifying content: Changing the text content of elements (using textContent or innerHTML), updating HTML structure within elements, or altering attribute values.
* Modifying style: Dynamically applying CSS styles to elements to change their appearance.
* Adding and removing elements: Inserting new HTML elements into the document or deleting existing ones.
* Handling events: Attaching event handlers to elements to respond to user interactions like clicks, form submissions, or key presses.

3. How the DOM works

1. Parsing: The browser receives and parses the HTML document.
2. DOM Tree Construction: Based on the parsed HTML, the browser constructs the DOM tree in memory, representing all elements, attributes, and text as nodes.
3. CSS Parsing: The browser also parses CSS stylesheets, creating a CSSOM (CSS Object Model).
4. Render Tree Creation: The DOM tree is combined with the CSSOM to form a render tree that contains visual information about each element.
5. Layout and Painting: The browser calculates the layout of elements and paints the page visually.
6. JavaScript Interaction: JavaScript can then interact with the DOM using its API to access and modify elements. Changes to the DOM are immediately reflected in the render tree, triggering the browser to re-render the page4. Importance of the DOM

* Dynamic Content: The DOM enables web pages to update content without needing full page reloads, like dynamic forms or data displays.
* Interactivity: It allows developers to build interactive elements such as animations, dynamic menus, form validations, and other features that respond to user actions.
* Foundation for Web Development: The DOM is the bedrock for many modern web development libraries and frameworks like React, Angular, and Vue.js.
* Event Handling: It provides the mechanism for capturing and responding to various user and system events on a web page.

In essence, the DOM is what makes web pages come alive and interactive. It provides a structured, programmatic way for scripts to interact with the web document, creating engaging and responsive web experiences.



**Simple API for XML:**

SAX (Simple API for XML) parsers are a type of XML parser that operate on an event-driven model. Unlike DOM (Document Object Model) parsers, which build an in-memory tree representation of the entire XML document, SAX parsers read through the XML document sequentially and trigger events as they encounter different parts of the document.

**Key characteristics of SAX parsers:**

* Event-driven:

SAX parsers trigger events (e.g., startElement, endElement, characters, startDocument, endDocument) as they process the XML stream.

* Sequential processing:

The parser reads the XML document from beginning to end and does not allow random access to elements.

* Low memory consumption:

Because SAX parsers do not load the entire document into memory, they are well-suited for parsing large XML files.

* Read-only:

SAX parsers are primarily used for reading and extracting data from XML documents; they do not provide mechanisms for modifying the XML structure.

* Handler-based:

Users implement a handler class (often by extending DefaultHandler in Java) that defines the logic for processing the events triggered by the parser.

How SAX parsing works:

* A SAX parser instance is created, typically through a SAXParserFactory.
* A custom handler class is created, extending a default handler (e.g., DefaultHandler in Java) and overriding specific callback methods to handle desired events.
* The parse() method of the SAX parser is invoked, providing the XML input source and the custom handler.
* As the parser reads the XML, it triggers the corresponding callback methods in the handler (e.g., startElement() when an opening tag is encountered, characters() for text content, endElement() for a closing tag).
* The logic within these callback methods processes the extracted data or performs other actions based on the events.

**CRUD operations :**

CRUD operations (Create, Read, Update, Delete) can be performed on XML data using various programming languages and libraries. The core idea is to manipulate the XML structure to reflect the desired changes.

1. Create:

* Adding new elements/nodes:

New XML elements or attributes can be added to an existing XML document. This typically involves creating a new node object and appending it to a parent node within the document's structure.

* Example:

In Python, using xml.etree.ElementTree, you could create a new Element and use append() to add it to another element.

2. Read:

* Parsing and accessing data:

XML documents are parsed into a tree-like structure, allowing navigation and retrieval of data from specific elements or attributes. This often involves using XPath expressions or similar querying mechanisms to locate desired information.

* Example:

In C#, LINQ to XML allows querying and extracting data from XML documents using familiar LINQ syntax.

3. Update:

* Modifying existing data:

The content or attributes of existing XML elements can be changed. This involves locating the target element and then updating its text content or attribute values.

* Example:

In Java, using DOM parsers, you would get a reference to the element and then call methods like setTextContent() or setAttribute().

4. Delete:

* Removing elements/nodes:

Specific elements or attributes can be removed from the XML document. This typically involves locating the node to be deleted and then calling a removal method on its parent node.

* Example:

In PHP, using SimpleXML, you can use unset() on a SimpleXMLElement object to remove an element.

Common Approaches and Tools:

* DOM (Document Object Model):

Provides a tree-based representation of the XML document, allowing programmatic access and manipulation of nodes. Available in many languages (e.g., Java, JavaScript, Python).

* SAX (Simple API for XML):

An event-driven parser, more efficient for large XML files as it processes the document sequentially without loading the entire structure into memory. Primarily used for reading.

* LINQ to XML (C#):

A powerful and intuitive API for querying and manipulating XML data using Language Integrated Query (LINQ).

* XPath:

A language for navigating and selecting nodes in an XML document, often used in conjunction with other XML processing libraries.

* XML-RPC:

A protocol for remote procedure calls using XML to encode calls and responses, enabling CRUD operations on data stored remotely.

**Angular**

Pre-Requisites:

HTML 5

CSS 3

JavaScript (ES6)

TypeScript

Lab Setup:

NodeJS 20 or above

VSCode as IDE

NodeJS:

NodeJS is a runtime environment for java script.

It uses NPM - node package manager as a build tool

To Create a node application

md app-folder

cd app-folder

npm init -y

'npm init' will initialize 'package.josn' file which holds the project meta data and list of dependencies and list of applicateion life cycle scripts.

**Install a dependency**

npm i thrid-party-package-name

Install a dev-time dependency

npm i thrid-party-package-name --save-dev

**Uninstall a dependency**

npm uninstall thrid-party-package-name

**Installing a dependency globally**

npm i --global thrid-party-package-name

'node\_modules' is the folder that holds the downloaded dependencies in our application.

**npm-scripts**

npm start is a customizable script to launch our application

npm test is a customizable script to invoke test cases of our application

npm build is a customizable script to invoke build of our appliation

npm run script-name will allow us to trigger scripts of our own

**TypeScript**

is a microsoft product and is a suepr set of javascript with typesafty.

typescript = javascript + typesafty

**Data Types**

number

string

boolean

bigint

void

null

undefined

any

unknown

**User Defined Data Types**

class

interface

enum

**Angular Introduction**

is a single-page-app framework.

An app that works inside a browser and does not require page reload during use. Loads a single HTML page and dynamically update the page.

Need:

1. SPA’s are fast as resources are loaded only once, when the data is being transmitted back and forth.

2. Simplified and streamlined development

3. Cache local storage effectively.

**Its features are**

(1) HTML extendibility - we can create our own html-elements and attributes.

(2) Modularity

(3) Server-Side Rendering and Static Site Generation

(4) In-built modules for Api-communication, authentication and authorization, routing ..etc.,

**Angular Architecture:**

An angular application is built with resources like

Component

Directive

Service

Pipe

Module

Each and every artifact is a typescript class

Each of these are marked with a decorator to indicate the role of the artifact.

The configuration of each of these artifacts is passed as a json--obj to the decorators and is called meta-data.

Example:

**Component**

@Component({

selector:"",

templateUrl:"",

standalone:true

})

class DashboardComponent {

//body containing function , events

}

**Directive**

@Directive({

selector:"",

standalone:true

})

class StockStatusDirective {

}

**Service**

@Injectable({

providedIn:'root'

})

class StockService {

}

**Pipe**

@Pipe({

name:"".

standalone:true

})

class IntoWordsPipe {

}

**Module**

@NgModule({

declarations:[],

imports:[],

exports:[],

providers:[]

})

class SalesModule {}

Angular CLI

is a command-line-interface that works like a frontier of commands used to manage the life cycle of an angular application.

tools like angular-cli, testing tools (karma and jasmine), minification and build tools ..etc.,

are executed on nodejs and after building the app, the app runs on a browser.

modern angular also offers SSG and SSR to reduce intial loading time, and the SSR is executed

as well on nodejs.

ng new app-name => to create a new angular app

ng add feature-name => to add a new module or a feature

ng build => to compile ts into js and builds the app into 'dist' folder

ng serve => to compile ts into js and luanch the app on a dev-server at 4200

ng serve --port 9999 => to compile ts into js and luanch the app on a dev-server at 9999

ng serve --port 9999 -o => to compile ts into js and luanch the app on a dev-server at 9999, opens the browser

ng test => to invoke test cases

ng g c ComponentName => generate a new stand-alone component

--skip-tests will avoid generateing test cases

--no-standalone to create a component inside a module

--module will carry the module name

ng g directive DirectiveName

generate a new stand-alone directive

--skip-tests will avoid generateing test cases

--no-standalone to create a component inside a module

--module will carry the module name

ng g pipe PipeName

generate a new stand-alone pipe

--skip-tests will avoid generateing test cases

--no-standalone to create a component inside a module

--module will carry the module name

ng g service ServiceName

generate a new service

--skip-tests will avoid generateing test cases

ng g module ModuleName

generate a new module

**Angular Components**

a component in angular is a angular built html-element.

each component is made up of three parts

the component-class dashboard.component.ts holds the state and behaviour of the component

the template dashboard.component.html holds the html-dom to be rendered for this component

the styleSheet dashboard.component.css holds the style local to this component

dashboard.component.ts

@Component({

selector:"app-dashboard",

templateUrl:"dashboard.component.html",

styleSheets:["dashboard.component.css"],

standalone:true

})

export class DashboardComponent{

//state as fields and behaviour as methods

String userName;

}

<app-dashboard></app-dashboard>

**Data Binding**

is to access the fields and methods of a component-class in the component-template.

**Interpolation or Expressions**

is to render the value of an angular-expression in the content of an html-element.

<tag-name> {{angular-expression}} </tag-name>

<p> The current user is {{userName}} </p>

**Two-Way Data Binding**

is to bind the value of a field to an input-element and vice-versa.

'ngModel' is a built-in directive from 'FormModule' that is used to execute two-way dta binding.

<input [(ngModel)]="field" />

**One-Way Data Binding**

is to bind a field or method on to non-editable attributes of dom.

Attribute Binding

is to bind a field with a attribute of an element.

<tagName [attribute]="angularExpression"> content </tagName>

<p title="this is a para"> This is a para </p> <!- this is not binding -->

<p [title]="paraTitle"> This is a para </p> <!- this is binding the value of 'paratitle' -->

**Event Bidning**

is to bind a method to an event directives

Event-Driectives are built-in Angular defiend attributes to handle events.

* click
* dblClick
* focus
* change
* blur
* ngSubmit
* mouseover
* mouseup
* mousedown

<tagName (eventDirective)="method()"> content </tagName>

<button type="button" (click)="doSomething()"> clicke me </button>

**Style Binding**

is to bind a field with a css-property or 'ngStyle' directive.

<tagNAme [style.cssProperty]="angularExpression"> content </tagName>

<p [style.textAlign]="myTextalignField"> content </p>

<tagNAme [ngStyle]="aJsonObject"> content </tagName>

@Component({ .. })

class MyComponent {

myParaStyle:any;

construcotr(){

this.myParaStyle = {border:"1px solid black",textAlign:"right"};

}

}

<p [ngStyle]="myParaStyle"> content </p>

**Class Binding**

is to bind a field to eh 'class' attribute of an element.

this allows the dev to add or remove css-class dynamically.

<tagNAme [class.className]="boolenaAngularExpression"> content </tagName>

@Component({ .. })

class MyComponent {

isImportant:boolean

construcotr(){

this.isImportant = true;

}

}

<p [class.important]="isImportant"> This is a para </p>

<tagNAme [ngClass]="anArrayOfClassesOraJsonObj"> content </tagName>

@Component({ .. })

class MyComponent {

myParaClasses:string[];

construcotr(){

this.myParaClasses=["importnat","highlight"];

}

}

<p [ngClass]="myParaClasses"> This is a para </p>

@Component({ .. })

class MyComponent {

myParaClasses:any;

construcotr(){

this.myParaClasses={importnat:true,highlight:false};

}

}

<p [ngClass]="myParaClasses"> This is a para </p>

**Integrating Bootstrap**

bootstrap is a css-js library that offers responsive web design.

bootstrap-icons is a css library that offers icons.

npm i bootstrap bootstrap-icons

these are installed in the node\_modules folder.

the .css files of this library msut be added to the 'styles' section of angular.json file

the .js files of this library msut be added to the 'scripts' section of angular.json file

**Angular Routing**

Routing is to map a component to a url, and render the mapped component only

when its url is requested.

Angular provides RouterModule for this priupose.

**RouterModule**

Route object {

path:'urlToBeMapped',

pathMatch:'startsWith|full'

component:Component,

redirectTo:''

children:][],

loadChildren : lazyLoadingFunction,

canActive: routerGuardArray,

canLoad: routerGuardArray,

canDeactive: routerGuardArray,

}

Routes Route[]

Router buitl-in service used to navigate progrmatically

navigate("url");

navigateTo(["segment1","segment2"]);

ActivatedRoute built-in service used to read url-paramter, or url related

data like path, querystring ..etc.,

RouterLink built-in directive to be used on 'a' element instead of its href

RouterLinkActive built-in directive to be used on 'a' element to apply a css-class

only when a link is visited

RouterOutlet built-in component that reserve place on the layout, to be

replaced by the mapped component of the current url.

**Angular Flow Controls**

Legacy Directives from CommonsModule

NgIf

NgFor

NgSwitch NgSwtichCase NgDefault

Modern Flow Controls

are built-in angular native controls that need to additonal improts to use

@if(cond) {

//html dom

} @else {

//alternate html dom

}

@switch(exp){

@case (case1) {

//html dom if exp===case1

}

@case (case2) {

//html dom if exp===case2

}

@default {

//html dom for default senario

}

}

@for(loopingVar of array; track $index){

//html dom we wnat to repeat one for each value in the array

}@empty{

//html dome that shall render incase the array is empty

}

variables injectable by for

$index the index of the current element

$even is the current element index is even

$odd is the current element index is odd

$first is the current element index is the first

$last is the current element index is the last

$count the number of elements that are iterated over .

**Inter Component Communication via @Input decorator**

When a parent component has to share some object with a child component, it does it

through attributes, An attribute of a component is a field of the component class marked with

@Input decorator.

navbar.component.ts

@Component({

selector:"nav-bar",

....

})

class NavBar {

@Input()

title!:string;

}

app.component.html

<nav-bar title="title can be passed here"></nav-bar>

<nav-bar [title]="aVariableFromParentComponent"></nav-bar>

**Angular LifeCycle Hooks**

a lifecyle hook is a method that get invoked automatically at a spacific stage of

a component or directives's lifecycle.

constructor()

↓

ngOnChanges() from OnChanges /\* is to detect any chagnes that may occur on @input \*/

↓

ngOnInit() from OnInit /\* is used to excute a task after the component is

| loaded initially \*/

↓

ngOnChanges() from OnChanges /\* invokes everytime when a change occur on @input \*/

↓

ngDoCheck() /\* is sued to detect any chagnews that angular couldn't \*/

↓

ngAfterContentInit()

↓

ngAfterContentChecked()

↓

ngAfterViewInit()

↓

ngAfterViewChecked()

|

...... /\*once the component is closed or removred \*/

↓

ngOnDestroy()

@Component({

selector:"dashboard",.....

})

class Dashboard {

/\*....\*/

}

dashboard template

<section>

<h3>Some heading</h3>

</section>

app component template

<dashboard>

<nav>

</nav>

</dashboard>

View is any dom declared in the template of the component

the section and the h3 are said to be the view

we can access these in the dashboard component class using

@ViewChild decorator

Content is any dom passed to the body of a component

the nav is called the content.

we can access these in the dashboard component class using

@ContentChild decorator

**Angular Directives**

A directive is any angular defined element or attribute.

Types Of Directives

(a) Component Directives are otherwise called Components - angular defiend elements

(b) Structural Directives are used to control the appearence of an element

NgIf, NgFor, NgSwitch

(c) Attribute Directives are angular defiend attributes

builtin attribute directives like NgModel, NgStyle, NgClass ...etc.,

we can create a custom attribute directive as well

ng g directive DirectiveName --skip-tests

@Directive({

selector:"[attribute-name]"

})

class DirectiveName {

}

**Angular Forms**

Angular supports two types of forms:

* Template Driven Forms

are constructred in html and are bound to fields using ngModel directiove from FormsModule.

FormsModule

ngForm

ngModel

And a few validation related properties These forms cannot accommodate complicated object structures like arrays or nested objects.

These forms are not easy to tests as well.

These forms are recommended to handle a case that has not more than two fields.

* **Reactive Forms (Model Driven Forms)**

ReactiveFormsModule

FormControl

FormGroup

FormBuilder

These forms are built for any complicated object structure.

These forms are built on component class and are bound to the html dom. These forms are built on component class and are bound

to the html dom.

Testing these form easy. 99% we use these forms in angular. Validation

related properties

FormGroup valid,invalid

FormControl valid,invalid,touched,untouched,pristine,dirty

**MODULE - III**

**Introduction to servlets:**

Servlets are Java programs that enhance server functionalities by processing client requests and generating dynamic responses, primarily in web applications. They are a fundamental component of Java web development and operate within the server's Java Virtual Machine (JVM). This makes them platform-independent and highly scalable. Servlets essentially act as a middle layer between the web browser and the server, handling the communication and processing the business logic behind web applications.

Key functions of servlets:

* Handling Client Requests: Servlets receive requests from clients (typically web browsers) and process the data sent with the request, such as information submitted through HTML forms.
* Generating Dynamic Content: Based on client input or backend logic, servlets dynamically generate and format responses, often as HTML, XML, or other data formats, and send them back to the client.
* Server-side Logic: Servlets are well-suited for implementing business logic, connecting with databases using JDBC, performing authentication and authorization, and managing sessions.
* Middleware in MVC: In the Model-View-Controller (MVC) architecture, servlets typically act as controllers, handling incoming requests and managing the communication between the view (JSP) and the model (Java classes/databases).

Advantages of servlets:

Servlets offer several advantages, including portability due to being written in Java. They are efficient because they run within the server's memory and use a multithreaded model for handling requests, which is more resource-friendly than creating new processes for each request. Java's features contribute to servlet robustness, and their thread-based processing allows for efficient handling of multiple concurrent requests, making them scalable. Servlets also integrate well with other Java technologies like JDBC and JavaBeans.

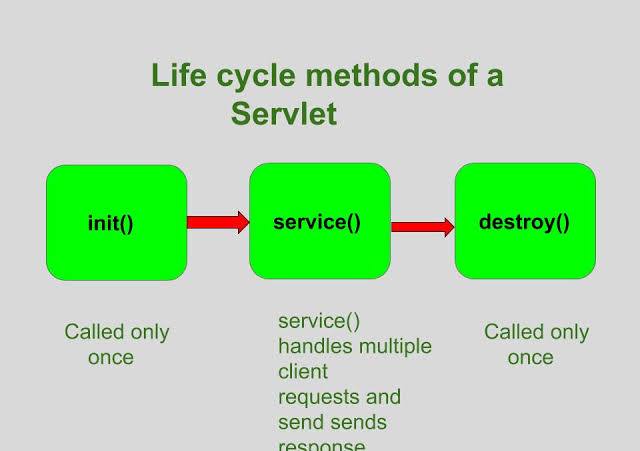
**Servlet lifecycle:**

A servlet's lifecycle is controlled by a servlet container. This lifecycle involves initialization when loaded (init() method), handling requests through the service() method (which dispatches to methods like doGet() or doPost()), and destruction (destroy() method) when no longer needed or when the server shuts down.

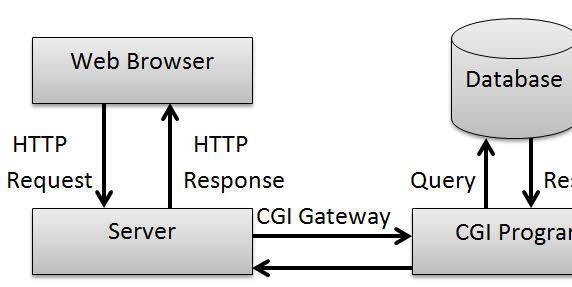
In essence Servlets are a fundamental Java technology for building dynamic web applications. They facilitate server-side processing, manage client-server communication, and enable the creation of robust and scalable web services.

The life cycle of a servlet is managed by the servlet container (e.g., Apache Tomcat) and consists of distinct stages:

* **Loading and Instantiation:**
  + When the web server starts or the first request for a servlet is received, the servlet container loads the servlet class.
  + Subsequently, a single instance of the servlet class is created. This instance is typically a singleton, handling all subsequent requests.
* **Initialization (init() method):**
  + After instantiation, the container calls the init(ServletConfig config) method on the servlet instance.
  + This method is called only once throughout the servlet's life cycle and is used for one-time initialization tasks, such as establishing database connections or loading configuration parameters.



**Common Gateway Interface (CGI):**



The Common Gateway Interface (CGI) is a standard protocol that enables web servers to execute external programs or scripts to generate dynamic content.

How CGI works

1. Client Request: A user's browser makes a request for a web page, typically by clicking a link or submitting a form.
2. Server Processing: The web server receives the request and identifies it as a CGI request.
3. Execution: The server launches the associated CGI script in a separate process.
4. Data Handling: The CGI script processes the request, potentially interacting with databases or other applications to retrieve or manipulate data. It receives information via environment variables for GET requests or standard input for POST requests.
5. Response Generation: The script generates dynamic content, typically in the form of HTML, based on the processed data.
6. Server Response: The server receives the generated content from the script and sends it back to the client's browser as its response.

Key features and characteristics

* Language Flexibility: CGI scripts can be written in various languages, including Perl, Python, C, and more.
* Dynamic Content: It enables the creation of web pages that change based on user input or other variables, facilitating interactive web applications.
* Form Handling: Commonly used for processing HTML forms submitted by users and generating customized responses.
* Middleware Role: It acts as middleware, facilitating communication between web servers and external databases or information sources.

Advantages of CGI

* Ease of Implementation: Relatively simple to set up for basic tasks.
* Reusability: Allows leveraging existing code or scripts written in various languages.
* Wide Compatibility: The CGI standard is widely supported across different systems and platforms.
* Reliability: Can be dependable for small, low-traffic websites or applications.

Disadvantages of CGI

* Performance Overhead: Launching a new process for every request can lead to slower response times, especially for high-traffic websites.
* Limited Caching: Difficult to cache data in memory between page loads, potentially impacting performance.
* Security Concerns: Requires careful coding practices and server configurations to avoid vulnerabilities like command injection or unauthorized access.
* Scalability Issues: Not ideal for handling large volumes of concurrent requests due to process creation overhead.

Modern alternatives

While CGI was crucial in the early days of dynamic web development, more efficient alternatives have emerged:

* Server-Side Scripting: Technologies like PHP, ASP.NET, and Ruby on Rails offer frameworks for handling web requests with better performance.
* Web Frameworks: Frameworks like Node.js and Django provide streamlined development, improved performance, and enhanced security.
* Java Servlets: Java-based alternatives that leverage threads for better scalability and efficiency.
* FastCGI, SCGI, AJP: Allow long-running application processes outside the web server, reducing process creation overhead.

Despite the emergence of newer technologies, understanding CGI remains valuable for comprehending the historical evolution of web development and the underlying principles that continue to influence modern practices.

**Deploying Servlet**

Deploying a servlet involves making it accessible and runnable within a web server or application server environment, typically a Servlet container like Apache Tomcat. The general process is as follows:

* **Develop the Servlet:**

Create your Java servlet class, extending HttpServlet and overriding methods like doGet() or doPost() to handle HTTP requests.

* **Compile the Servlet:**

Compile your Java servlet source code into a .class file. Ensure the servlet API JAR (e.g., servlet-api.jar or jakarta.servlet-api.jar) is included in your classpath during compilation.

* **Create Web Application Structure:**

Organize your web application files into a standard directory structure. This typically includes:

* + **Root Directory:** For static content (HTML, JSP, CSS, JavaScript, images).
  + **WEB-INF Directory:** A secure directory not directly accessible by clients.
    - **WEB-INF/classes:** Contains compiled servlet .class files and other Java classes, preserving package structure.
    - **WEB-INF/lib:** Contains any external JAR libraries required by your servlet.
    - **WEB-INF/web.xml:** The deployment descriptor, crucial for configuring your servlet.
* Configure web.xml (Deployment Descriptor):
  + Define your servlet using the <servlet> tag, specifying a unique name and the fully qualified class name.
  + Map a URL pattern to your servlet using the <servlet-mapping> tag, associating the servlet name with a specific URL pattern that clients will use to access it.
  + Alternatively, for Servlet 3.0+ containers, you can use the @WebServlet annotation directly on your servlet class to achieve the same mapping without web.xml entries.
* **Package the Web Application (Optional but Recommended):**

Create a Web Application Archive (WAR) file. This is a standard way to package web applications for easy deployment. Use build tools like Maven or Gradle, or manually create a ZIP file with a .war extension containing your web application structure.

* **Deploy to Servlet Container:**
  + **Manual Deployment:** Copy the web application directory or the WAR file to the appropriate deployment directory of your Servlet container (e.g., webapps directory in Apache Tomcat).
  + **IDE Deployment:** Most Integrated Development Environments (IDEs) provide built-in features to deploy web applications directly to a configured server.
* **Start the Server:**

Ensure your Servlet container (e.g., Tomcat) is running.

* **Test the Servlet:**

Access your servlet through a web browser using the configured URL pattern (e.g., <http://localhost:8080/your_app_name/your_servlet_url>).

**The Servlet API:**

The Servlet API provides the interfaces and classes for developing Java servlets, which are Java programs that extend the capabilities of a server. Servlets are primarily used to handle requests and generate dynamic responses in web applications. Key interfaces and classes within the API include:

* Servlet:

The core interface that all servlets must implement, defining lifecycle methods like init(), service(), and destroy().

* GenericServlet:

A convenience class that implements the Servlet and ServletConfig interfaces, providing a generic, protocol-independent servlet.

* HttpServlet:

An abstract class extending GenericServlet that provides methods specifically for handling HTTP requests (e.g., doGet(), doPost()).

* ServletConfig:

An object created by the container to pass initialization information to a servlet during its instantiation.

* ServletContext:

An object that defines a set of methods a servlet uses to communicate with its container, providing access to web application-wide information.

* ServletRequest and ServletResponse:

Interfaces representing the client request and the server's response, respectively.

**Reading Servlet Parameters (Request Parameters)**

Servlet parameters, also known as request parameters, are data sent from a client to a servlet, typically as part of an HTTP GET or POST request (e.g., form data, query string parameters). These parameters are accessed via the ServletRequest object.

* String getParameter(String name): Retrieves the value of a single parameter as a String. If the parameter does not exist, it returns null.
* String[] getParameterValues(String name): Retrieves an array of String values for a parameter that may have multiple values (e.g., a multi-select list or multiple checkboxes with the same name).
* Enumeration<String> getParameterNames(): Returns an Enumeration of all parameter names in the current request.

Servlet parameters are key-value pairs that a client sends with an HTTP request, typically from an HTML form or a URL's query string. They are temporary and vary with each request. You can access these parameters using methods from the HttpServletRequest object, which is passed to a servlet's doGet() or doPost() method.

Key methods

* request.getParameter("paramName"): Retrieves the value of a specific parameter as a String. Returns null if the parameter does not exist.
* request.getParameterValues("paramName"): Returns an array of String objects containing all values for a given parameter name. Use this for checkboxes or other inputs that can have multiple values.
* request.getParameterNames(): Returns an Enumeration<String> of all parameter names in the current request. This is useful for iterating through all available parameters.
* request.getParameterMap(): Returns a Map<String, String[]> of all parameters in the request, with parameter names as keys and an array of their values.

Example:

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class FormDataServlet extends HttpServlet {

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

response.setContentType("text/html");

PrintWriter out = response.getWriter();

*// Read a single-value parameter*

String username = request.getParameter("username");

out.println("<h3>Username: " + username + "</h3>");

*// Read a multi-value parameter (e.g., from checkboxes)*

String[] languages = request.getParameterValues("language");

if (languages != null) {

out.println("<h3>Favorite Languages:</h3>");

out.println("<ul>");

for (String lang : languages) {

out.println("<li>" + lang + "</li>");

}

out.println("</ul>");

}

}

**Reading initialization parameters :**

Initialization parameters, or "init parameters," are configuration settings defined for a specific servlet or the entire web application. They are set when the servlet is initialized and remain constant for its entire lifecycle.

* Servlet-specific parameters: Defined in the web.xml deployment descriptor within the <servlet> tag or by using the @WebInitParam annotation. They are accessed via the ServletConfig object, which is passed to the init() method.
* Context-wide parameters: Global to the entire web application and accessible by all servlets. They are defined within the <web-app> tag in web.xml. They are accessed via the ServletContext object.

Key methods

* config.getInitParameter("paramName"): Retrieves a servlet-specific init parameter value via the ServletConfig object.
* config.getInitParameterNames(): Returns an Enumeration<String> of all init parameter names for that specific servlet.
* context.getInitParameter("paramName"): Retrieves a context-wide init parameter value via the ServletContext object.

Example with web.xml:

First, define the parameters in web.xml.

xml

<web-app>

<servlet>

<servlet-name>ConfigServlet</servlet-name>

<servlet-class>com.example.ConfigServlet</servlet-class>

<init-param>

<param-name>emailSupport</param-name>

<param-value>support@example.com</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>ConfigServlet</servlet-name>

<url-pattern>/config</url-pattern>

</servlet-mapping>

<context-param>

<param-name>appName</param-name>

<param-value>MyWebApp</param-value>

</context-param>

</web-app>

Then, access the parameters in your servlet code.

Java

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletConfig;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class ConfigServlet extends HttpServlet {

private String supportEmail;

private String applicationName;

@Override

public void init(ServletConfig config) throws ServletException {

super.init(config);

*// Read servlet-specific init parameter*

this.supportEmail = config.getInitParameter("emailSupport");

*// Read context-wide init parameter*

this.applicationName = config.getServletContext().getInitParameter("appName");

}

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

response.setContentType("text/html");

PrintWriter out = response.getWriter();

out.println("<h3>Application Name: " + this.applicationName + "</h3>");

out.println("<h3>Support Email: " + this.supportEmail + "</h3>");

}

}

**Handling Http Request & Responses:**

In servlets, handling HTTP requests and responses is done through the

HttpServletRequest and HttpServletResponse objects, which are passed as arguments to a servlet's doGet(), doPost(), and other do methods. The HttpServletRequest object provides access to the client's request details, while the HttpServletResponse object is used to construct and send the server's response back to the client.

Handling HTTP requests

Key HttpServletRequest methods

* Reading request parameters: Retrieve data sent from an HTML form or a URL's query string.
  + request.getParameter("paramName"): Gets the value of a parameter as a String.
  + request.getParameterValues("paramName"): Returns a String array for parameters with multiple values (e.g., from checkboxes).
* Accessing request headers: Retrieve metadata sent by the client, such as the user-agent or content type.
  + request.getHeader("headerName"): Gets the value of a specific header.
  + request.getHeaderNames(): Returns an enumeration of all header names.
* Handling the request body: For POST requests, you can read the body directly as a stream.
  + request.getReader(): Gets a BufferedReader for character data.
  + request.getInputStream(): Gets a ServletInputStream for binary data.
  + Note: You can use either getReader() or getInputStream() on a request, but not both. Calling getParameter() on a POST request also consumes the request body, so it should not be combined with stream-reading methods.
* Getting request metadata:
  + request.getMethod(): Returns the HTTP method (e.g., "GET" or "POST").
  + request.getRequestURI(): Gets the URI of the requested resource.
  + request.getCookies(): Returns an array of Cookie objects sent with the request.

Handling HTTP responses

Key HttpServletResponse methods

* Writing the response body:
  + response.setContentType("text/html"): Sets the MIME type of the content, which should be called before writing the response.
  + response.getWriter(): Returns a PrintWriter to send character-based text (like HTML) to the client.
  + response.getOutputStream(): Returns a ServletOutputStream for sending binary data.
  + Note: You can only use either getWriter() or getOutputStream() for a response, not both.
* Setting response headers and status codes:
  + response.setStatus(int statusCode): Sets the HTTP status code, such as 200 for OK or 404 for Not Found.
  + response.addHeader(String name, String value): Adds a header to the response.
* Redirecting the client:
  + response.sendRedirect(String location): Sends a temporary redirect response, telling the client's browser to navigate to a new URL.
* Managing cookies:
  + response.addCookie(Cookie cookie): Sends a Cookie from the server to the client, which the client's browser may store.
* Sending error messages:
  + response.sendError(int sc, String msg): Sends an error response with a specified status code and message.

Example: A servlet handling GET and POST requests

This example demonstrates a servlet that handles both GET and POST requests, retrieving form data and sending a response.

The HTML form (index.html)

html

<!DOCTYPE html>

<html>

<head>

<title>Form Example</title>

</head>

<body>

<h2>GET Request Form</h2>

<form action="MyServlet" method="GET">

First Name: <input type="text" name="firstName"><br>

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

</form>

<h2>POST Request Form</h2>

<form action="MyServlet" method="POST">

Last Name: <input type="text" name="lastName"><br>

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

</form>

</body>

</html>

----------------- ------------------

The servlet (MyServlet.java)

java

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class MyServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

*// Retrieve parameter from the GET request*

String firstName = request.getParameter("firstName");

*// Set response content type*

response.setContentType("text/html");

*// Get a PrintWriter to write the response*

PrintWriter out = response.getWriter();

*// Write the HTML response*

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

out.println("<h2>Hello, " + (firstName != null ? firstName : "Guest") + "!</h2>");

out.println("<p>This is a response to your GET request.</p>");

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

}

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

*// Retrieve parameter from the POST request*

String lastName = request.getParameter("lastName");

*// Set response content type*

response.setContentType("text/html");

*// Get a PrintWriter to write the response*

PrintWriter out = response.getWriter();

*// Write the HTML response*

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

out.println("<h2>Thank you, " + (lastName != null ? lastName : "User") + ".</h2>");

out.println("<p>This is a response to your POST request.</p>");

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

}

}

**Connecting to a database using JDBC:**

Connecting to a database in a servlet involves using the Java Database Connectivity (JDBC) API to establish a connection, execute SQL queries, and process the results. The most reliable method for a web application is to use a connection pool provided by the server, rather than opening and closing a new connection for every request.

Prerequisites

Before you start, you'll need the following:

* A JDBC driver JAR file for your specific database (e.g., MySQL, PostgreSQL, or Oracle).
* The driver file added to your project's classpath. If you're using a modern web container like Tomcat, you can simply place the JAR file in the WEB-INF/lib directory.

For optimal performance and resource management, especially in web applications, you should use a DataSource for connection pooling. This is configured in your application server (e.g., Tomcat) and looked up in your servlet using JNDI (Java Naming and Directory Interface).

1. Configure the DataSource

In your web server's configuration (e.g., context.xml for Tomcat), define the database connection details in a Resource element.

Example context.xml for MySQL:

xml

<Context>

<Resource name="jdbc/YourDB"

auth="Container"

type="javax.sql.DataSource"

maxActive="100"

maxIdle="30"

maxWait="10000"

username="db\_user"

password="db\_password"

driverClassName="com.mysql.cj.jdbc.Driver"

url="jdbc:mysql://localhost:3306/your\_database"

/>

</Context>

2. Define the resource in web.xml

In your web.xml deployment descriptor, add a resource-ref to reference the DataSource defined in the server.

Example web.xml snippet:

xml

<resource-ref>

<res-ref-name>jdbc/YourDB</res-ref-name>

<res-type>javax.sql.DataSource</res-type>

<res-auth>Container</res-auth>

</resource-ref>

3. Look up the DataSource in your servlet

Use the init() method to perform the JNDI lookup once during servlet initialization. This prevents repeated lookups for every request.

java

import javax.naming.Context;

import javax.naming.InitialContext;

import javax.sql.DataSource;

import java.sql.Connection;

import java.sql.SQLException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

public class MyDataServlet extends HttpServlet {

private DataSource dataSource;

public void init() throws ServletException {

try {

Context initContext = new InitialContext();

Context envContext = (Context) initContext.lookup("java:/comp/env");

this.dataSource = (DataSource) envContext.lookup("jdbc/YourDB");

} catch (Exception e) {

throw new ServletException("Failed to get DataSource", e);

}

}

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

try (Connection conn = dataSource.getConnection()) {

*// Your JDBC code goes here*

} catch (SQLException e) {

*// Handle exceptions appropriately*

}

}

}

**Basic steps for using JDBC**

Regardless of whether you are using a pooled DataSource or a direct DriverManager connection, the core JDBC steps remain the same:

1. Get a connection: Get a Connection object from the DataSource or DriverManager.
2. Create a statement: Create a Statement or PreparedStatement object from the Connection. PreparedStatement is recommended to prevent SQL injection.
3. Execute a query: Use executeQuery() for SELECT statements or executeUpdate() for INSERT, UPDATE, or DELETE statements.
4. Process results: If a ResultSet is returned, iterate through it to process the data.
5. Close resources: Ensure that Connection, Statement, and ResultSet objects are closed to release database resources.

Example using try-with-resources

The try-with-resources statement, available since Java 7, automatically closes JDBC resources, making your code cleaner and more reliable.

java

import java.io.IOException;

import java.io.PrintWriter;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import javax.annotation.Resource;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.sql.DataSource;

public class UserServlet extends HttpServlet {

*// Inject the DataSource using annotation*

@Resource(name = "jdbc/YourDB")

private DataSource dataSource;

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

response.setContentType("text/html");

PrintWriter out = response.getWriter();

String query = "SELECT id, username FROM users";

try (Connection conn = dataSource.getConnection();

PreparedStatement ps = conn.prepareStatement(query);

ResultSet rs = ps.executeQuery()) {

out.println("<html><body><h2>User List</h2><ul>");

while (rs.next()) {

out.println("<li>ID: " + rs.getInt("id") + ", Name: " + rs.getString("username") + "</li>");

}

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

} catch (SQLException e) {

out.println("Error accessing the database: " + e.getMessage());

e.printStackTrace();

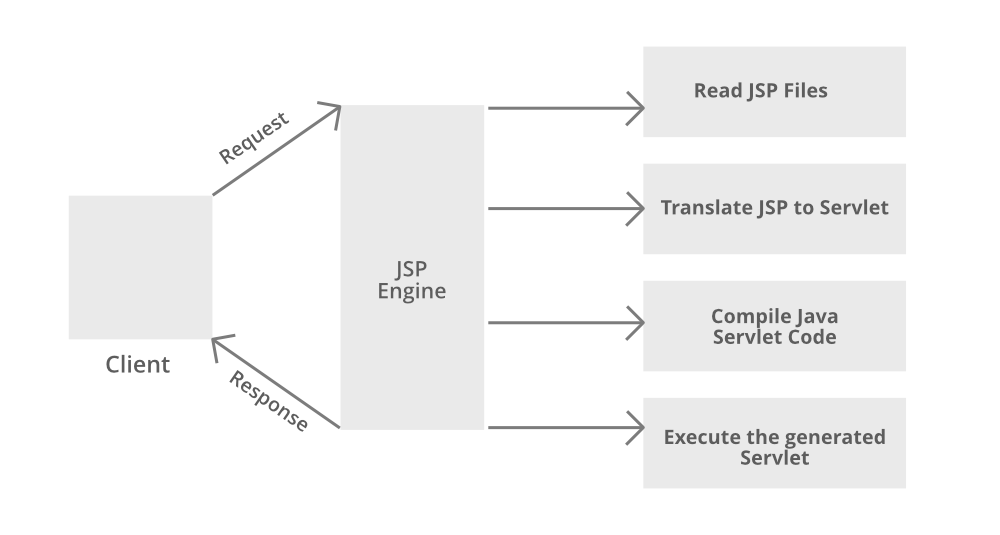
}

}

}

# **MODULE – IV**

**Introduction to JSP: The Anatomy of a JSP Page, JSP Processing:**



JSP (JavaServer Pages) is a server-side technology that allows developers to embed Java code within HTML pages to create dynamic web applications. A JSP page is processed by a JSP engine within a web container (like Apache Tomcat), which converts it into a servlet that is then executed to generate the final HTML response sent to the client.

The anatomy of a JSP page

A JSP page is a text-based document that combines static markup (like HTML) with dynamic JSP elements.

Static template text

This is the standard, static part of the web page, such as HTML tags, plain text, and CSS. The JSP container passes this content directly to the client's browser without modification.

Dynamic JSP elements

These are special tags that the JSP engine processes to generate dynamic content. The main types of elements include:

* Directives <%@ ... %>: Provide global instructions to the JSP container for the entire page.
  + page: Sets page-dependent attributes like the scripting language, error page, and imported Java classes (<%@ page import="java.util.Date" %>).
  + include: Inserts the content of another file during the translation phase (<%@ include file="header.jsp" %>).
  + taglib: Declares a tag library for custom actions (<%@ taglib uri="..." prefix="..." %>).
* Declarations <%! ... %>: Define class-level variables and methods that are added to the servlet class and are available throughout the JSP page.
* Scriptlets <% ... %>: Embed any valid Java code within the JSP page, typically for business logic or flow control.
  + Example: <% for (int i=0; i<5; i++) { out.println("Hello<br>"); } %>
* Expressions <%= ... %>: Evaluate a Java expression and insert the result directly into the HTML output. It is a shortcut for out.print().
  + Example: Current time is: <%= new java.util.Date() %>
* Actions <jsp: ... />: Perform predefined actions using an XML-like syntax. They control the flow between pages and interact with JavaBeans.
  + Example: <jsp:include page="footer.jsp" /> includes another page dynamically at request time.

**JSP processing**

When a client requests a .jsp page, the web container's JSP engine processes it through a series of phases.

1. Translation: The JSP engine translates the JSP page into a Java servlet source file. It converts static HTML into println() statements and JSP elements into Java code within the servlet.
2. Compilation: The generated servlet source file (.java) is compiled into a servlet class file (.class) that the Java Virtual Machine (JVM) can execute.
3. Class Loading: The servlet class is loaded into memory by the class loader.
4. Instantiation: An instance of the servlet class is created. The container manages a single instance of the servlet to handle multiple requests.
5. Initialization: The jspInit() method is called once to initialize the servlet instance. This is typically used for one-time setup tasks like creating a database connection pool.
6. Request Processing (Execution): For each client request, the \_jspService() method is invoked. This method handles the request and generates the dynamic content to be sent back to the client.
7. Destruction: When the servlet is removed from service (e.g., when the server shuts down), the jspDestroy() method is called once for cleanup tasks, such as releasing resources.

In JavaServer Pages (JSP), special tags and built-in objects are used to embed dynamic content and handle page logic.

**Directives**

Directives provide global information to the JSP container during the translation phase, influencing how the JSP page is compiled into a servlet.

* Syntax: <%@ directive attribute="value" %>
* page directive: Sets page-specific attributes.
  + <%@ page import="java.util.Date" %>: Imports Java classes.
  + <%@ page errorPage="error.jsp" %>: Specifies an error page for unhandled exceptions.
* include directive: Statically includes the content of another file during translation. It's best for reusable, static content like headers and footers.
  + <%@ include file="header.html" %>
* taglib directive: Declares a custom tag library for use in the JSP page.
  + <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

**Declarations**

Declarations are used to define class-level variables and methods that will be available to all parts of the JSP page.

* Syntax: <%! declaration %>
* The code within a declaration is placed outside the \_jspService() method in the generated servlet, making it a member of the class.
* Best for: Defining helper methods or instance variables that need to be shared across requests.
* Example:

jsp

<%!

private int hitCount = 0;

public String getPageTitle() {

return "My Dynamic Page";

}

%>

**Expressions**

Expressions are used to embed a Java expression's value directly into the HTML output.

* Syntax: <%= expression %>
* The expression is automatically converted into a string and inserted into the page's output stream. No semicolon is needed.
* Best for: Displaying dynamic data inline, such as the value of a variable or a method's return value.
* Example:

jsp

<p>The page title is: <%= getPageTitle() %></p>

<p>The current time is: <%= new java.util.Date() %></p>

Scriptlets (code snippets)

Scriptlets allow any valid Java code, including control-flow structures, to be embedded into the JSP page.

* Syntax: <% code %>
* The code within a scriptlet is placed inside the \_jspService() method of the generated servlet.
* Best for: Implementing control-flow logic, such as loops and conditional statements.
* Example:

jsp

<% for (int i = 1; i <= 5; i++) { %>

<p>This is paragraph #<%= i %></p>

<% } %>

**Implicit objects:**

Implicit objects are built-in Java objects that the JSP container automatically makes available on every JSP page. They can be used directly within scriptlets and expressions without explicit declaration.

--------------------------------------------------------------------------------------------------------------

|  |  |  |
| --- | --- | --- |
| Implicit Object | Type | Description |
| request | HttpServletRequest | Represents the client's HTTP request. Used to get parameters, headers, and cookies. |
| response | HttpServletResponse | Represents the HTTP response sent back to the client. Used to set headers, cookies, or redirect. |
| session | HttpSession | Represents the user's session across multiple requests. Used to store user-specific data. |
| application | ServletContext | Represents the web application context. Used to store application-wide data. |
| out | JspWriter | Used to write content to the client's output stream. |
| config | ServletConfig | Provides configuration information for the servlet. |
| pageContext | PageContext | Provides access to all namespaces (scopes) of the JSP page. |
| page | java.lang.Object | A synonym for this, referring to the current servlet instance. |
| exception | java.lang.Throwable | Represents an uncaught exception in an error page (isErrorPage="true"). |

Example using implicit objects

jsp

<p>User-Agent: <%= request.getHeader("User-Agent") %></p>

<%

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

if (username != null) {

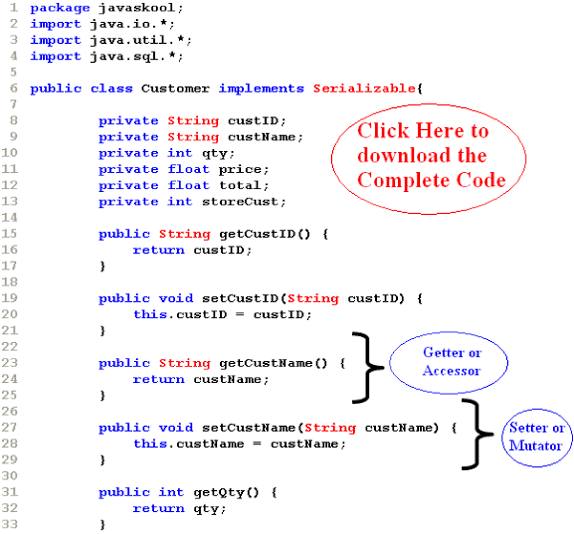
out.println("<p>Welcome back, " + username + "!</p>");

}

%>

**BEANS:**

Using Java Beans in JSP pages is a way to separate business logic from presentation, promoting a clean, reusable, and maintainable design. A JavaBean is a standard Java class that follows specific design conventions, which makes it easy to manipulate within JSP using special action tags.



Key JavaBean conventions

A Java class qualifies as a JavaBean if it adheres to these rules:

* It must have a public no-argument constructor, allowing the JSP container to instantiate it easily.
* Its properties (instance variables) should be private.
* It must provide public getter (getXxx) and setter (setXxx) methods to access and modify its private properties. For example, for a name property, there should be getName() and setName() methods.
* It should be serializable, enabling its state to be saved and restored, especially in distributed applications.

JSP action tags for using beans

JSP provides three main action tags to interact with JavaBeans.

1. <jsp:useBean>

This tag is used to find or create an instance of a JavaBean and assign it an ID. If a bean with the specified ID and scope already exists, it is used; otherwise, a new instance is created.

Syntax:

xml

<jsp:useBean id="beanName" class="package.ClassName" scope="..."/>

Use code with caution.

* id: A variable name used to reference the bean object in the JSP.
* class: The fully qualified name of the JavaBean class.
* scope: Defines the bean's lifecycle and visibility. The possible values are:
  + page (default): The bean is available only on the current page.
  + request: The bean is available for the duration of the current HTTP request.
  + session: The bean is available throughout the user's session.
  + application: The bean is shared across the entire web application and all users.

2. <jsp:setProperty>

This tag sets the values of a bean's properties by calling its setter methods.

Syntax:

xml

<jsp:setProperty name="beanName" property="propertyName" value="value"/>

Use code with caution.

or, to automatically match request parameters to bean properties:

xml

<jsp:setProperty name="beanName" property="\*"/>

Use code with caution.

* name: The ID of the bean set with <jsp:useBean>.
* property: The name of the property to set.
* value: The value to assign to the property.
* param: Sets a property value from a request parameter.

3. <jsp:getProperty>

This tag retrieves the value of a bean's property by calling its getter method and inserts the result into the output.

Syntax:

xml

<jsp:getProperty name="beanName" property="propertyName"/>

Use code with caution.

* name: The ID of the bean.
* property: The name of the property to retrieve.

Example: Using a user bean

This example demonstrates a typical use case for beans in a web application, where a bean holds data submitted from a form.

Step 1: Create the JavaBean class (User.java)  
This class encapsulates user data with private fields and public getter and setter methods.

java

package com.example;

import java.io.Serializable;

public class User implements Serializable {

private String firstName;

private String lastName;

*// No-argument constructor*

public User() {}

*// Getters and Setters*

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

}

Use code with caution.

Step 2: Create the JSP page (process.jsp)  
This page processes the form data, sets the bean's properties, and displays the results.

jsp

<%@ page import="com.example.User" %>

<jsp:useBean id="userBean" class="com.example.User" scope="request"/>

<%-- Automatically maps form parameters to bean properties --%>

<jsp:setProperty name="userBean" property="\*"/>

<!DOCTYPE html>

<html>

<head>

<title>JSP Bean Example</title>

</head>

<body>

<h1>User Information</h1>

<p>First Name: <jsp:getProperty name="userBean" property="firstName"/></p>

<p>Last Name: <jsp:getProperty name="userBean" property="lastName"/></p>

</body>

</html>

Use code with caution.

Step 3: Create the HTML form (index.html)  
This form sends data to the process.jsp page.

html

<!DOCTYPE html>

<html>

<head>

<title>User Form</title>

</head>

<body>

<form action="process.jsp" method="post">

<label>First Name:</label>

<input type="text" name="firstName"><br><br>

<label>Last Name:</label>

<input type="text" name="lastName"><br><br>

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

</form>

</body>

</html>

**Sessions:**

A web session is the conversation between a user and a server over a period of time. Session tracking is the process of maintaining a user's state across multiple requests within that session. This is essential because the HTTP protocol is stateless, meaning each request from a client is independent and not inherently connected to previous ones. The Java Servlet API provides the HttpSession interface to facilitate session tracking, with each JSP page having implicit access to the session object.

How JSP session tracking works

1. Session creation: When a user's browser makes its first request, the servlet container creates a unique HttpSession object for that user. A unique session ID is generated and sent back to the client, typically stored as a cookie in the browser. On subsequent requests, the browser sends this ID back, allowing the server to identify the user's session.
2. State management: The HttpSession object acts as a server-side storage for user-specific data, such as a username after login or a shopping cart for an e-commerce site. This state information is securely stored on the server and is not visible to the client, unlike cookies.
3. Session access: In JSP pages, the session object is one of the implicit objects, meaning it's readily available to use without manual instantiation. You can use its methods within scriptlets or leverage the Expression Language (EL) for easier access.

Key HttpSession methods

The session object provides several methods for managing session data:

* session.setAttribute(String name, Object value): Binds an object to the session with a specific name.
* session.getAttribute(String name): Retrieves the object bound to the session with the specified name. The return type is Object, so you must cast it to the correct type.
* session.removeAttribute(String name): Removes the object bound with the specified name from the session.
* session.invalidate(): Destroys the entire session and unbinds all objects. This is typically used for logging out a user.
* session.isNew(): Returns true if the client doesn't know about the session yet or has chosen not to join it.
* session.getId(): Returns the unique session ID.

**connecting to database in JSP**

To connect to a database in a JSP page, you typically

use JDBC (Java Database Connectivity), which provides a standard API for Java applications to interact with relational databases. While you can embed JDBC code directly into a JSP, using separate Java classes (like JavaBeans) is a more maintainable and secure approach.

Prerequisites

1. Install a database server, such as MySQL, and create the database and table you need.
2. Download the JDBC driver for your database (e.g., mysql-connector-java.jar) and place the .jar file in your web application's WEB-INF/lib directory.
3. Ensure your web server (e.g., Apache Tomcat) has access to the driver by restarting it if necessary.

Method 1: Connecting directly with a scriptlet (for demonstration)

This method involves embedding all the JDBC code directly into your JSP page using a scriptlet. While simple for learning, this is not recommended for production applications due to security risks and code maintainability issues.

jsp

<%@ page import="java.sql.\*" %>

<%!

// Database credentials and driver setup

private final String dbUrl = "jdbc:mysql://localhost:3306/your\_database\_name";

private final String dbUser = "your\_username";

private final String dbPassword = "your\_password";

private final String driverClass = "com.mysql.cj.jdbc.Driver";

// Static initialization block to load the driver once

static {

try {

Class.forName("com.mysql.cj.jdbc.Driver");

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

%>

<html>

<head>

<title>Database Connection Example</title>

</head>

<body>

<%

Connection conn = null;

Statement stmt = null;

ResultSet rs = null;

try {

// Step 1: Establish the connection

conn = DriverManager.getConnection(dbUrl, dbUser, dbPassword);

// Step 2: Create a statement and execute a query

stmt = conn.createStatement();

String sql = "SELECT id, name, email FROM users";

rs = stmt.executeQuery(sql);

%>

<h2>User List</h2>

<table border="1">

<tr>

<th>ID</th>

<th>Name</th>

<th>Email</th>

</tr>

<%

// Step 3: Process the result set

while (rs.next()) {

%>

<tr>

<td><%= rs.getInt("id") %></td>

<td><%= rs.getString("name") %></td>

<td><%= rs.getString("email") %></td>

</tr>

<%

}

%>

</table>

<%

} catch (SQLException se) {

// Log and handle SQL errors

out.println("<h3>Error connecting to database:</h3><pre>" + se.getMessage() + "</pre>");

se.printStackTrace();

} finally {

// Step 4: Close resources in a finally block

try {

if (rs != null) rs.close();

if (stmt != null) stmt.close();

if (conn != null) conn.close();

} catch (SQLException se) {

se.printStackTrace();

}

}

%>

</body>

</html>

**MODULE -V**

**Database Design using MySQL: An Overview of SQL:**

Database design with MySQL involves using Structured Query Language (SQL) to create an efficient and organized structure for storing and managing data. A robust database design is essential for ensuring data integrity, minimizing redundancy, and optimizing performance.

Phases of database design

The process of designing a database is systematic and typically broken down into the following stages:

1. Requirements gathering: Understand the purpose of the database and the information that needs to be stored, such as details for products or user orders.
2. Conceptual design: Create a high-level, visual model of the database, often using an Entity-Relationship Diagram (ERD). This helps identify the key entities (tables), their attributes (columns), and the relationships between them.
3. Logical design: Translate the conceptual model into a logical structure, defining specific tables, columns, and keys. This is when normalization rules are applied to minimize data redundancy.
4. Physical design: Implement the logical design on a specific Database Management System (DBMS), such as MySQL. This involves selecting appropriate data types, defining indexes, and considering storage and performance.

**Core concepts in database design**

Effective database design utilizes several key concepts. Data is organized into tables with rows and columns. Keys, such as primary keys which uniquely identify rows, and foreign keys which link tables, are fundamental. Relationships between tables, like one-to-many and many-to-many, help manage connected data and reduce redundancy; many-to-many relationships require an intermediate table. Normalization is a technique using "normal forms" to minimize redundancy and enhance data integrity. Constraints, such as NOT NULL, UNIQUE, and CHECK, are rules applied to columns to maintain data quality. Columns are assigned data types like INT, VARCHAR, and DATE to ensure they store appropriate data.

**Overview of SQL**

SQL is the standard language for interacting with relational databases like MySQL. SQL commands are categorized by function:

**Data Definition Language (DDL)**

DDL commands manage database structures. They include CREATE DATABASE, CREATE TABLE, ALTER TABLE to modify structures, and DROP TABLE to delete tables.

**Data Manipulation Language (DML)**

DML commands are used to work with data within tables. Key commands are INSERT INTO to add data, UPDATE to modify records, and DELETE FROM to remove records.

**Data Query Language (DQL)**

DQL commands are for retrieving data. The main command is SELECT to retrieve data. Other important commands include WHERE to filter records, JOIN to combine data from multiple tables, ORDER BY to sort results, and GROUP BY to summarize data.

**XAMPP and MySQL Setup:**

**XAMPP:**

To set up XAMPP, first, download the installer from [apachefriends.org](https://www.apachefriends.org/), then run it as an administrator, following the on-screen prompts to select components and installation location. After the installation completes, launch the XAMPP Control Panel, click "Start" for the Apache and MySQL modules, and then open your web browser to http://localhost/ to view the XAMPP dashboard and confirm the installation.

1. Download XAMPP

* Go to the official Apache Friends website and download the installer for your operating system (e.g., Windows).

2. Run the Installer

* Locate the downloaded .exe file and run it.
* You may see a warning about User Account Control (UAC); click "OK" or "Yes" to proceed.
* If you have antivirus software, it's recommended to temporarily deactivate it during installation to prevent interference.

3. Follow the Setup Wizard

* In the setup window, click "Next".
* Choose the default components, which include Apache, MariaDB (for MySQL), PHP, and Perl, or select the ones you need.
* Accept the default installation folder (e.g., C:\xampp), as installing in C:\Program Files can cause issues with User Account Control.
* Uncheck the "Learn more about Bitnami" option if you don't want to install additional features.
* Complete the installation by clicking "Next" and then "Finish".

4. Start the XAMPP Control Panel

* After installation, open the XAMPP Control Panel. You can find it in the Start menu.
* Click the "Start" button for the Apache and MySQL modules.
* You may need to allow Apache access to your network via the Windows firewall.

5. Access the Localhost Dashboard

* Open your web browser.
* Type http://localhost/ into the address bar.
* If you see the XAMPP dashboard, the installation was successful. From here, you can access modules like PHPMyAdmin for managing databases.

MY SQL:

Setting up MySQL typically involves downloading the appropriate installer for your operating system and then following the installation and configuration steps.

1. Download the MySQL Installer:

* Navigate to the official MySQL website.
* Locate the "Downloads" section and choose "MySQL Community Downloads."
* Select the "MySQL Installer for Windows" (or the appropriate version for macOS/Linux).
* Choose the larger "Community" MSI file for a comprehensive installation.

2. Run the Installer and Choose Setup Type:

* Execute the downloaded installer file.
* When prompted, select a setup type. "Developer Default" or "Full" are common choices that include the MySQL server and development tools like MySQL Workbench.
* Proceed through any requirement checks, allowing the installer to resolve dependencies if necessary.

3. Installation and Configuration:

* Click "Execute" to begin the installation of the selected components.
* Once installed, the installer will guide you through the configuration process.
* **Server Configuration Type:** Choose a configuration type (e.g., Development Machine).
* **Authentication Method:** Select a strong password encryption method.
* **Root Account Password:** Set a secure password for the 'root' user. This is crucial for administrative access.
* **Windows Service (Windows only):** Choose whether to run MySQL as a Windows service for automatic startup.
* **Apply Configuration:** Execute the configuration steps to finalize the setup.

4. Verification:

* After installation and configuration, you can verify the setup.
* **MySQL Command Line Client:** Open the MySQL Command Line Client (or terminal/command prompt) and log in with the 'root' user and the password you set.
* **MySQL Workbench:** If installed, launch MySQL Workbench and establish a connection to your local MySQL server using the 'root' user and password.

This process establishes a functional MySQL server on your system, ready for database creation and management.

After your MySQL environment is set up, you can write your SQL program. Below is the example to display " Hello World" using SQL.

**1. Create a database named test\_db**

CREATE DATABASE test\_db;

**2. Use the test\_db database**

USE test\_db;

**3. Create a table named greetings**

CREATE TABLE greetings (  
 id INT AUTO\_INCREMENT PRIMARY KEY,  
 message VARCHAR(255)  
);

**3. Insert the message 'Hello, World!' into the greetings table**

INSERT INTO greetings (message)  
VALUES ('Hello, World!');

**4. Retrieve the message from the greetings table**

SELECT message FROM greetings;

Aliases & CONCAT()

Aliases are temporary names given to columns or tables to make queries more readable and concise. They exist only for the duration of the query. The CONCAT() function is used to join or concatenate multiple strings together into a single string. You can use an alias to give a new, meaningful name to a column created with CONCAT().

Syntax

sql

*-- Column alias with CONCAT()*

SELECT CONCAT(column1, ' ', column2) AS alias\_name

FROM table\_name;

***-- Table alias***

SELECT alias\_name.column1, alias\_name2.column2

FROM table\_name AS alias\_name, table\_name2 AS alias\_name2;

Example

To display a full name by concatenating first\_name and last\_name columns from an employees table:

sql

SELECT CONCAT(first\_name, ' ', last\_name) AS full\_name

FROM employees;

UPDATE, DELETE, & ALTER

These are key Data Manipulation Language (DML) and Data Definition Language (DDL) commands used to modify data and table structures in a database.

* UPDATE: Modifies existing records in a table.
  + Syntax: UPDATE table\_name SET column1 = value1, column2 = value2 WHERE condition;
  + Example: UPDATE employees SET email = 'john.doe@example.com' WHERE employee\_id = 1;
* DELETE: Removes rows from a table.
  + Syntax: DELETE FROM table\_name WHERE condition;
  + Example: DELETE FROM employees WHERE employee\_id = 5;
* ALTER TABLE: Modifies the structure of an existing table.
  + Syntax for adding a column: ALTER TABLE table\_name ADD COLUMN new\_column\_name datatype;
  + Syntax for modifying a column: ALTER TABLE table\_name MODIFY COLUMN column\_name new\_datatype;
  + Syntax for dropping a column: ALTER TABLE table\_name DROP COLUMN column\_name;
  + Example: ALTER TABLE employees ADD COLUMN start\_date DATE;

**Foreign keys**

A foreign key is a field in one table that references the primary key in another table. This establishes a parent-child relationship between tables and maintains referential integrity, preventing actions that would break the link between them. An example is an orders table referencing the customer\_id from a customers table. You can find the SQL code for this example in the referenced documents.

**Table joins**

Table joins combine rows from two or more tables based on a related column. Common types of joins include:

* INNER JOIN: Returns only rows with matching values in both tables.
* LEFT JOIN: Returns all rows from the left table and matched rows from the right.
* RIGHT JOIN: Returns all rows from the right table and matched rows from the left.
* FULL OUTER JOIN: Returns all rows when there is a match in either table. While MySQL doesn't have a direct FULL JOIN, you can achieve a similar result using a UNION of LEFT JOIN and RIGHT JOIN.
* SELF JOIN: Joins a table to itself using aliases.

**PHP Programming:**

PHP programming fundamentals

PHP is a widely used server-side scripting language for web development. PHP code can be embedded directly into HTML files and is executed on a web server, producing dynamic content.

Basic syntax and variables

* Tags: PHP code is enclosed within <?php and ?> tags.
* Case sensitivity: Variable names are case-sensitive. $name and $Name are different variables.
* Statements: Each statement in PHP must end with a semicolon (;).
* Variables: Variable names must start with a dollar sign ($), followed by a letter or underscore, and are assigned values using the equals sign (=).
* Comments: Single-line comments start with //, while multi-line comments are enclosed between /\* and \*/.

Control structures

* if...else...elseif: Executes different blocks of code based on a condition.
* switch: Compares a single expression against different possible values.
* for loop: Repeats a block of code a fixed number of times.
* while loop: Executes a block of code as long as a condition is true.
* foreach loop: Iterates over elements in an array.

Functions

* User-defined functions: Declared with the function keyword, they encapsulate reusable blocks of code.
* Parameters: Functions can accept inputs, either by value or by reference.
* Built-in functions: PHP includes a vast library of built-in functions for tasks like string manipulation, file handling, and date/time operations.

PHP data types & dates

PHP is a loosely typed language, meaning you do not have to specify the data type of a variable. PHP automatically determines the type based on the value assigned.

Data types

PHP supports several data types, including String (character sequences in quotes), Integer (whole numbers), Float (numbers with decimals), and Boolean (true or false). It also includes Array (stores multiple values), Object (instances of classes), NULL (single NULL value), and Resource (reference to external resources like files or database connections).

Dates and time

PHP offers functions for handling dates and times:

* date(format, timestamp): Formats a local date/time. Common format characters are available in the source documents.
* time(): Returns the current Unix timestamp.
* strtotime(string): Converts a human-readable date string to a timestamp.
* date\_default\_timezone\_set(timezone): Sets the default timezone.

Cookies:

Cookies are small data pieces a web server stores on a user's browser to identify users and save information like preferences.