Html,css,js,bootstrap
Tags - end sem mai 10 marks
Mid term 5 marks tags ke upar questions.Per tags 3line ki definition aur syntax likhne honge.
Short Answer Questions:Readonly vs disabled fields

Answer Presentation :

1. **Heading, pointers**
2. **Diagram, example ,code, flowchart with clear explanation and depiction**
3. **Neat handwriting**

IMP

1. filters
   EVENT angular se - mouse over
   validation - form
   form
   intro
   Expression
2. Jquery aur ajax ka introduction hi krna h bs unit 2 se]
3. basic php
   form
   file handling
4. front - boot
   a. back - angular
   b. ANGULAR JS
5. Ek to control flow statements
   a. Ek form validation ka
6. difference between container and container fluid class

PYQ

Here is the JavaScript code for all three parts:

Part (a) - Function to check if input is a string and convert to uppercase

```
function toUpperCaseIfString(input) {

   if (typeof input === 'string') {
        return input.toUpperCase();
    } else {
        return "Input is not a string";
    }
}

// Example usage:
console.log(toUpperCaseIfString("hello")); // Output: "HELLO"
console.log(toUpperCaseIfString(123)); // Output: "Input is not a string"
```

Part (b) - Callback function example to handle data fetched from a server

```
function fetchData(callback) {
    // Simulating data fetch with setTimeout
    setTimeout(() => {
        const data = { name: "John Doe", age: 25 }; // Mock data
        callback(data);
    }, 1000);
}

function handleData(data) {
    console.log("Fetched data:", data);
}
```

```
// Example usage:

fetchData(handleData);
```

Part (c) - HTML DOM example for displaying fetched data on button click

```
<!DOCTYPE html>

<html lang="en">

<head>

  <title>Fetch and Display Data</title>

  <script>

    function fetchDataAndDisplay() {

      // Simulating data fetch

      setTimeout(() => {

        const data = "Data fetched from server";

        document.getElementById("output").innerText = data;

      }, 1000);

    }

  </script>

</head>

<body>

  <button onclick="fetchDataAndDisplay()">Fetch Data</button>

  <p id="output"></p>

</body>

</html>
```

The image contains two questions related to Bootstrap:

1. **Write a program to create a webpage by using Bootstrap classes.**
2. **Using the Bootstrap class, write a program to create a webpage for grid representation.**

Here are code examples for both tasks:

Q8: Webpage Using Bootstrap Classes

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Bootstrap Webpage</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body>
  <nav class="navbar navbar-expand-lg navbar-light bg-light">
    <div class="container-fluid">
      <a class="navbar-brand" href="#">My Website</a>
      <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
data-bs-target="#navbarNav">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav">
          <li class="nav-item"><a class="nav-link" href="#">Home</a></li>
          <li class="nav-item"><a class="nav-link" href="#">About</a></li>
          <li class="nav-item"><a class="nav-link" href="#">Contact</a></li>
        </ul>
      </div>
    </div>
  </nav>
  <div class="container mt-4">
    <h1 class="text-center">Welcome to My Bootstrap Page</h1>
    <p class="text-muted">This is a simple webpage using Bootstrap classes for styling.</p>
    <button class="btn btn-primary">Click Me</button>
  </div>
```

```
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/js/bootstrap.bundle.min.js"></scrip
t>
</body>
</html>
```

Q9: Webpage for Grid Representation

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Bootstrap Grid</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body>
    <div class="container mt-4">
        <h2 class="text-center">Grid Layout Example</h2>
        <div class="row">
            <div class="col-md-4 bg-primary text-white p-3">Column 1</div>
            <div class="col-md-4 bg-secondary text-white p-3">Column 2</div>
            <div class="col-md-4 bg-success text-white p-3">Column 3</div>
        </div>
        <div class="row mt-3">
            <div class="col-md-6 bg-warning text-dark p-3">Column 1 (Half Width)</div>
            <div class="col-md-6 bg-danger text-white p-3">Column 2 (Half Width)</div>
        </div>
    </div>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/js/bootstrap.bundle.min.js"></scrip
t>
</body>
</html>
```

Theory:

HTML,CSS,JS

Web Server

- A **web server** is a software or hardware system that hosts websites, delivering content to users over the Internet.
- It listens to client requests (usually from browsers) and responds by serving requested web pages or resources.

- It uses the **HTTP** (HyperText Transfer Protocol) or **HTTPS** (secure version) to communicate.

Working:

1. **Request**:
    - A user enters a URL in the browser (e.g., `https://example.com`).
    - The browser sends a **request** to the web server, usually via HTTP/HTTPS.
2. **Processing**:
    - The web server receives the request.
    - If the requested page is **static** (like an HTML file or image), the server retrieves the file from its storage.
    - If it's **dynamic** (like a PHP page or API response), the web server passes the request to appropriate application software (like PHP or Node.js) to generate the page or data.
3. **Response**:
    - The web server sends back the requested content (e.g., an HTML page, JSON data) to the client.
    - The browser renders the response for the user.
4. **Logging and Maintenance**:
    - Web servers log details of each request (IP address, time, requested URL) for monitoring and debugging.

Search Engines

A search engine is a **software system designed** to help **users find information** on the internet by **indexing websites** and **retrieving results based on user queries**.

- Indexing: It uses algorithms to index or organize vast amounts of data available on the internet.
- Retrieval: When a user inputs a query, the search engine retrieves and ranks the most relevant results based on its algorithms.

#Working (in steps):
1. Crawling: The search engine **uses bots (crawlers) to visit and scan the content** of web pages across the internet.
2. Indexing: The content collected by **crawlers** is organized in an index, which is a **massive database storing information about the pages.**
3. Ranking: The search engine uses **algorithms to determine the relevance** of each web page to the user's search query. Factors like keyword presence, page authority, and user experience impact ranking.

4. Retrieving: Based on the query entered by the user, the search engine **retrieves the most relevant web pages** from the index and displays them in ranked order (e.g., the most relevant appears at the top).

Flex property

The flex property is a **shorthand property** in CSS used within a flex container to define how a flex item will **grow or shrink to fit the space available.** It **specifies the flexibility of the item** along the main axis (the direction of the flex container). The flex property is a part of the CSS **Flexible Box Layout (Flexbox) module,** which provides a more **efficient way to lay out, align**, and distribute space among items in a container.

### Syntax
The flex property is defined using three components:
```css
flex: <flex-grow> <flex-shrink> <flex-basis>;
```

1. **flex-grow:** ability of a flex item to **grow relative to the other** flex items. Default is `0`, meaning it won't grow.
2. flex-shrink: A number that defines the ability of a **flex item to shrink relative to** the other flex items. Default is `1`, meaning it will shrink if necessary.
3. flex-basis: Specifies the **initial size of the flex** item before the **remaining space is distributed.** It can be a length (e.g., `200px`) or a percentage (e.g., `50%`). Default is `auto`.

### Usage
The flex property is used in flex items (children of a flex container) to control their size and layout based on the available space in the flex container.

### Example
Here's an example demonstrating the use of the flex property in a simple flexbox layout:
html
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Flexbox Example</title>
  <style>
    .container {
      display: flex; /* Enable flexbox */
      justify-content: space-between; /* Space items evenly */
      align-items: center; /* Align items vertically in the center */
      height: 100px; /* Set container height */
      background-color: #f0f0f0; /* Light gray background */
    }

    .item {
            flex: 1; /* Grow equally */
            padding: 20px; /* Add padding */
            text-align: center; /* Center text */
            background-color: #4CAF50; /* Green background */
            color: white; /* White text */
            margin: 5px; /* Add margin between items */
        }

        .item-2 {
            flex: 2; /* This item will take twice the space of others */
        }
    </style>
</head>
<body>

    <div class="container">
        <div class="item">Item 1</div>
        <div class="item item-2">Item 2</div>
        <div class="item">Item 3</div>
    </div>

</body>
</html>
```

CSS media queries

CSS Media Queries are a powerful CSS feature that allows you to apply different styles to a webpage **based on the characteristics of the device or viewport displaying the content**. This enables developers to **create responsive designs** that adapt to various screen sizes, resolutions, and orientations.

### How Media Queries Work
Media queries consist of a media type and one or more expressions that check for specific conditions (like width, height, resolution, etc.). When the conditions of the media query are met, the specified CSS rules are applied.

Syntax:
```css
@media media-type and (condition) {
    /* CSS rules go here */
}
```

### Common Media Query Conditions:

1. Width and Height:
- `min-width`: The minimum width of the viewport.
- `max-width`: The maximum width of the viewport.
- `min-height`: The minimum height of the viewport.
- `max-height`: The maximum height of the viewport.

2. Orientation:
- `orientation: portrait` (height greater than width).
- `orientation: landscape` (width greater than height).

3. Resolution:
- `resolution`: The pixel density of the device.

### Example of CSS Media Queries:

```css
/* Styles for mobile devices */
@media (max-width: 600px) {
    body {
        background-color: lightblue;
    }
    h1 {
        font-size: 20px;
    }
}

/* Styles for tablets */
@media (min-width: 601px) and (max-width: 900px) {
    body {
        background-color: lightgreen;
    }
    h1 {
        font-size: 24px;
    }
}

/* Styles for desktop screens */
@media (min-width: 901px) {
    body {
        background-color: lightcoral;
    }
    h1 {
        font-size: 28px;
    }
}
```

### Why Are Media Queries Useful for Different Screens?

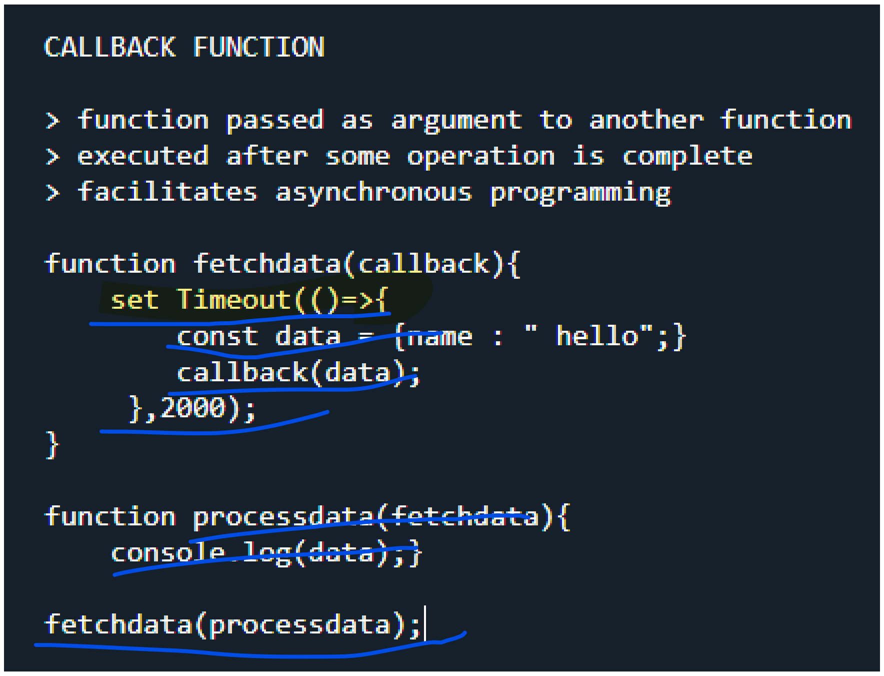
1. Responsive Design:
   - Media queries are crucial for creating responsive designs that adapt to different screen sizes. This ensures that users have an optimal viewing experience, regardless of the device they are using (mobile, tablet, desktop).

2. Improved User Experience:
   - By applying specific styles for different devices, you can enhance usability and readability. For example, increasing font sizes and adjusting layouts for smaller screens helps users read content without zooming.

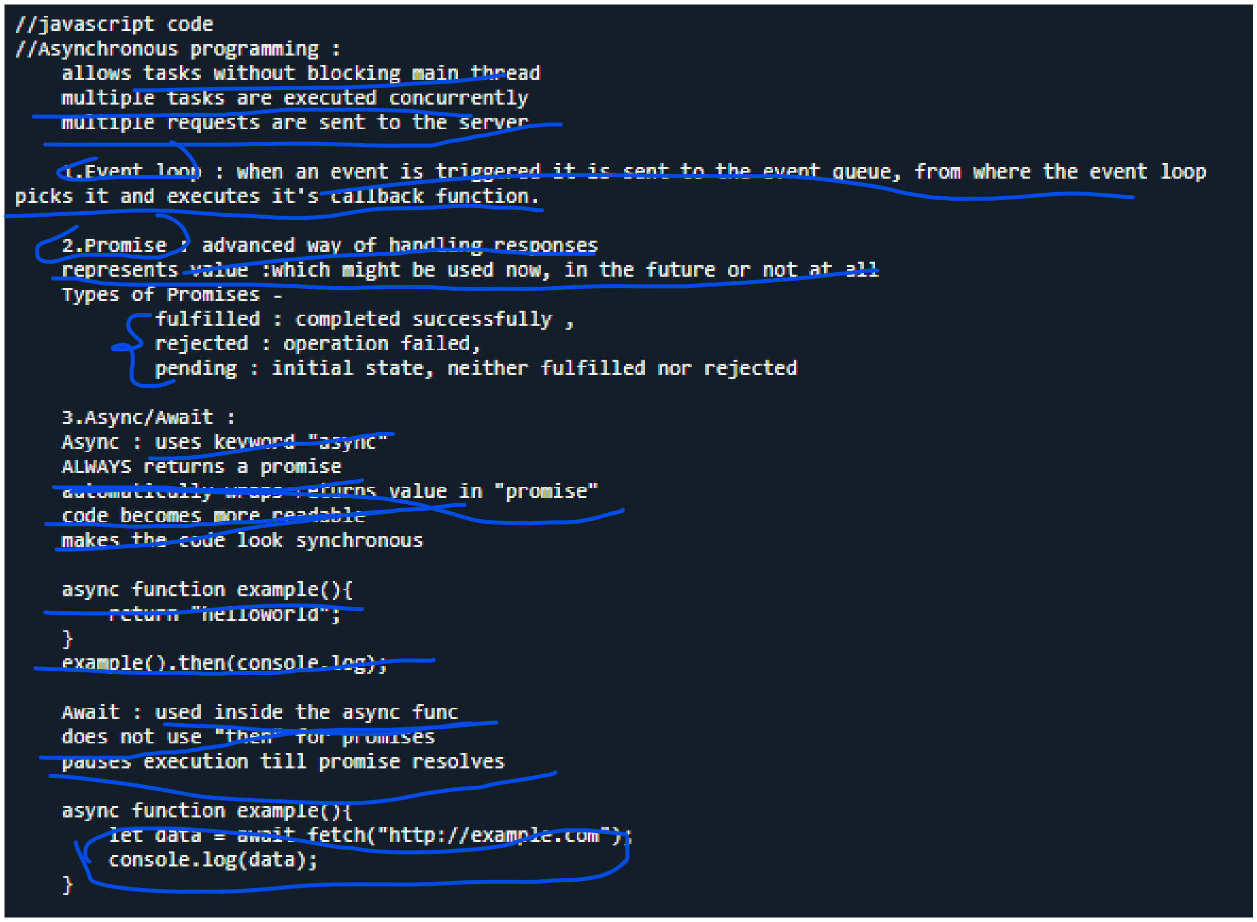
3. Optimized Performance:
   - Loading different styles for different devices can help optimize performance. For instance, you can hide certain images or styles that are not necessary on smaller devices, reducing loading time.

4. Future-Proofing:
   - With the increasing variety of devices and screen sizes (smartphones, tablets, laptops, TVs), using media queries helps ensure your website remains accessible and visually appealing on future devices.

JS callbacks and Asynchronous



```
CALLBACK FUNCTION

> function passed as argument to another function
> executed after some operation is complete
> facilitates asynchronous programming

function fetchdata(callback){
    set Timeout(()=>{
        const data = {name : " hello";}
        callback(data);
      },2000);
}

function processdata(fetchdata){
    console.log(data);}

fetchdata(processdata);
```



```
//javascript code
//Asynchronous programming :
    allows tasks without blocking main thread
    multiple tasks are executed concurrently
    multiple requests are sent to the server

    1.Event loop : when an event is triggered it is sent to the event queue, from where the event loop
picks it and executes it's callback function.

    2.Promise : advanced way of handling responses
    represents value :which might be used now, in the future or not at all
    Types of Promises -
            fulfilled : completed successfully ,
            rejected : operation failed,
            pending : initial state, neither fulfilled nor rejected

    3.Async/Await :
    Async : uses keyword "async"
    ALWAYS returns a promise
    automatically wraps returns value in "promise"
    code becomes more readable
    makes the code look synchronous

    async function example(){
        return "helloworld";
    }
    example().then(console.log);

    Await : used inside the async func
    does not use "then" for promises
    pauses execution till promise resolves

    async function example(){
        let data = await fetch("http://example.com");
        console.log(data);
    }
```

JavaScript is a single-threaded language, which means it can only execute one task at a time. However, it provides mechanisms to handle asynchronous operations, allowing certain tasks to run in the background while the main thread continues executing. Callbacks play a crucial role in this asynchronous behavior.

### Callbacks
A callback is a function that is passed as an argument to another function and is executed after some operation is completed. Callbacks allow you to define code that will run once a particular task finishes, facilitating asynchronous programming.

Example of Callbacks:
~~~
```javascript
function fetchData(callback) {
   setTimeout(() => {
      // Simulating data fetching
      const data = { id: 1, name: 'John Doe' };
      callback(data); // Call the callback function with the fetched data
   }, 2000); // Simulate a 2-second delay
}

function processData(data) {
   console.log('Data fetched:', data);
}

// Calling fetchData and passing processData as a callback
fetchData(processData);
~~~

Explanation:
- The `fetchData` function simulates fetching data with a 2-second delay using `setTimeout`.
- Once the data is "fetched," it calls the `callback` function (in this case, `processData`) and passes the data to it.
- `processData` then logs the fetched data to the console.

### Asynchronous Programming
Asynchronous programming allows tasks to be executed without blocking the main thread, enabling JavaScript to handle multiple operations concurrently, such as fetching data from a server or waiting for a timer.

#### Key Concepts:

1. Event Loop:
  - The event loop is a mechanism that allows JavaScript to perform non-blocking operations despite being single-threaded. It manages the execution of code, collects and processes events, and executes queued sub-tasks.

2. Promises:

- Promises are a more advanced way to handle asynchronous operations. They represent a value that may be available now, or in the future, or never.
- A promise can be in one of three states:
  - Pending: Initial state, neither fulfilled nor rejected.
  - Fulfilled: The operation completed successfully.
  - Rejected: The operation failed.

Example of Promises:

```
javascript
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const data = { id: 1, name: 'John Doe' };
            resolve(data); // Resolve the promise with the fetched data
        }, 2000);
    });
}

fetchData()
    .then(data => {
        console.log('Data fetched:', data);
    })
    .catch(error => {
        console.error('Error fetching data:', error);
    });
```

3. Async/Await:

- Async/Await is a syntactic sugar built on top of promises, making asynchronous code look more like synchronous code. It allows you to write cleaner and more readable asynchronous code.

Example of Async/Await:

```
javascript
async function fetchData() {
    return new Promise((resolve) => {
        setTimeout(() => {
            const data = { id: 1, name: 'John Doe' };
            resolve(data);
        }, 2000);
    });
}
```

```
async function getData() {
    try {
        const data = await fetchData(); // Wait for the promise to resolve
        console.log('Data fetched:', data);
    } catch (error) {
        console.error('Error fetching data:', error);
    }
}
getData();
```

`async` and `await` in JavaScript

### 1. What is `async`?

- `async` is a keyword used to define a function that always returns a `Promise`.
- It allows you to write asynchronous code in a **more readable, synchronous-looking** manner.
- The function marked with `async` **automatically wraps its return value i**n a `Promise`. If a non-Promise value is returned, it is wrapped in a resolved `Promise`.

```
async function example() {
  return "Hello";  // Returns a resolved Promise with value "Hello"
}
example().then(console.log);  // Outputs: Hello
```

### 2. What is `await`?

- `await` is a keyword used **inside `async` functions to pause execution until a `Promise`** resolves.
- It works only inside `async` functions.
- `await` allows you to work with the resolved value of the `Promise` without using `.then()`.

~~~
```js
async function fetchData() {
  let data = await fetch("https://api.example.com");
  console.log(data);  // This line waits for fetch to complete
}
~~~

### 4. Error Handling with `async` and `await`

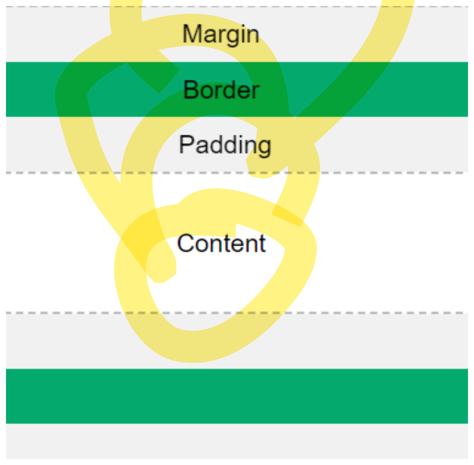
- You can use `try...catch` to handle errors that occur in `async` functions, just like you would in synchronous code.
- If an awaited `Promise` is rejected, the control moves to the `catch` block.

```js
```

```
async function fetchWithErrorHandling() {
  try {
    let response = await fetch("invalid-url");
    let data = await response.json();
  } catch (error) {
    console.log("Error:", error);  // Handle error here
  }
}
```

box model

The CSS Box Model is a **fundamental** concept in web design that describes the structure and layout of elements in a web page. It represents each HTML element as a **rectangular box, consisting of four areas:**



### Breakdown of the CSS Box Model:

1. **Content**:
   - This is the **innermost** part of the box and **holds the actual content** (text, images, etc.).
   - Its size can be set using properties like `**width` and `height`.**

2. **Padding**:
   - Padding **surrounds the content, providing space inside the element.**
   - You can control the padding with properties like `padding-top`, `padding-right`, `padding-bottom`, and `padding-left`.
   - Example: `padding: 10px;` will add 10px of padding around all sides of the content.

3. **Border**:

- The border **surrounds the padding** (if any) and content. It can be styled using properties like `border-width`, `border-style`, and `border-color`.
- Example: `border: 2px solid black;` creates a solid black border of 2px around the element.

4. Margin:

- Margin is the **outermost part** of the box and provides **space between the element and neighboring elements.**
- It can be controlled with properties like `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`.
- Example: `margin: 20px;` will add 20px of space outside the border, separating the element from others.

### Example Code of the Box Model in Use:

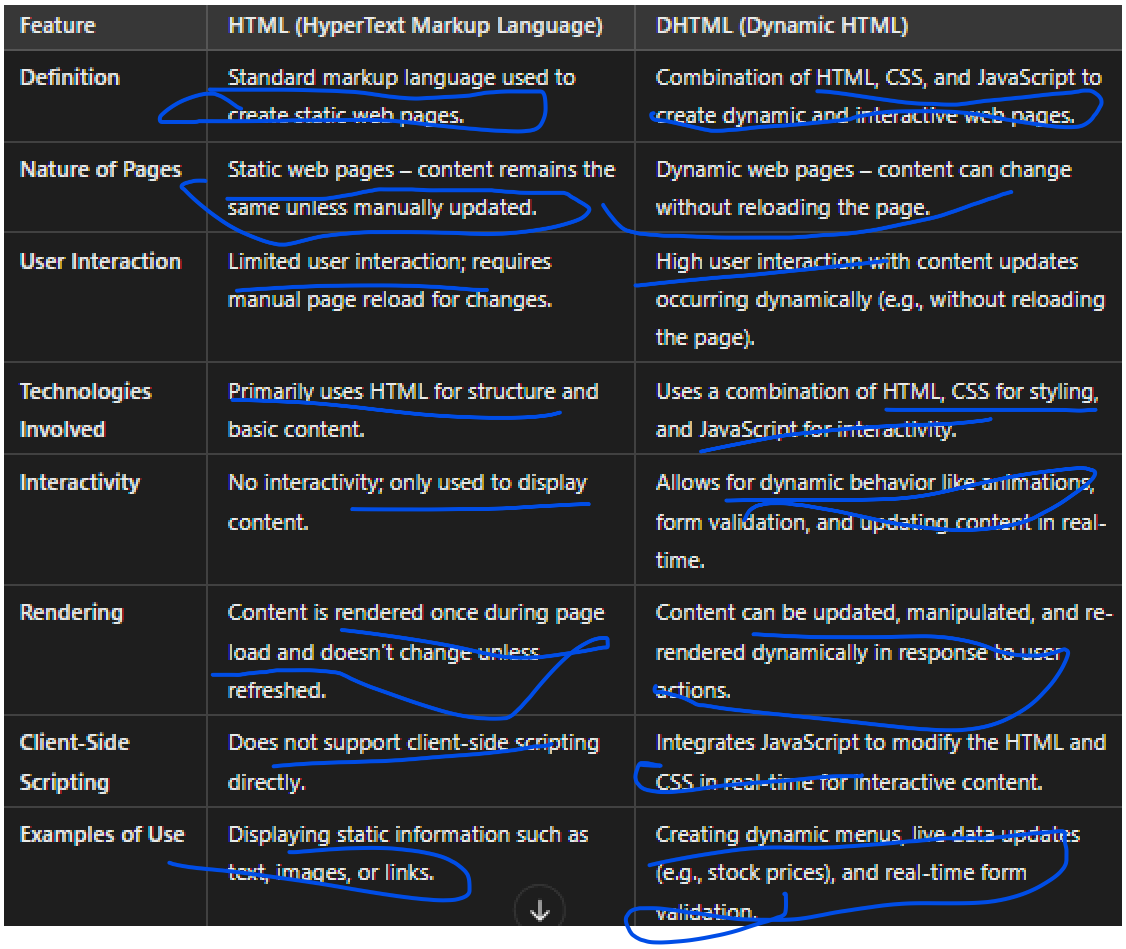
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Box Model Example</title>
    <style>
        .box {
            width: 200px;
            height: 100px;
            padding: 20px;
            border: 5px solid black;
            margin: 30px;
            background-color: lightgray;
        }
    </style>
</head>
<body>

    <div class="box">
        This is a box model example.
    </div>

</body>
</html>
```

### Explanation of the Code:
- The `.box` element has:

- `width: 200px;` – The content area is 200px wide.
- `height: 100px;` – The content area is 100px high.
- `padding: 20px;` – There's 20px of space between the content and the border.
- `border: 5px solid black;` – A 5px solid black border around the padding.
- `margin: 30px;` – The space outside the border, separating the element from other elements, is 30px.

This code will display a rectangular box with content and demonstrate the CSS box model's layers in action.

DHTML vs HTML



Feature	HTML (HyperText Markup Language)	DHTML (Dynamic HTML)
Definition	Standard markup language used to create static web pages.	Combination of HTML, CSS, and JavaScript to create dynamic and interactive web pages.
Nature of Pages	Static web pages – content remains the same unless manually updated.	Dynamic web pages – content can change without reloading the page.
User Interaction	Limited user interaction; requires manual page reload for changes.	High user interaction with content updates occurring dynamically (e.g., without reloading the page).
Technologies Involved	Primarily uses HTML for structure and basic content.	Uses a combination of HTML, CSS for styling, and JavaScript for interactivity.
Interactivity	No interactivity; only used to display content.	Allows for dynamic behavior like animations, form validation, and updating content in real-time.
Rendering	Content is rendered once during page load and doesn't change unless refreshed.	Content can be updated, manipulated, and re-rendered dynamically in response to user actions.
Client-Side Scripting	Does not support client-side scripting directly.	Integrates JavaScript to modify the HTML and CSS in real-time for interactive content.
Examples of Use	Displaying static information such as text, images, or links.	Creating dynamic menus, live data updates (e.g., stock prices), and real-time form validation.

event driven programming

Event-driven programming is a programming paradigm in which the flow of the program is determined by events. Events can be user actions (like clicks, mouse movements, keyboard input), system-generated signals (like timers), or messages from other programs or threads. In this model, the program reacts to these events rather than executing a predetermined sequence of instructions.

### Why JavaScript is Event-Driven
1. Asynchronous Nature:
- JavaScript is inherently asynchronous, allowing it to perform tasks without blocking the main thread. This means it can handle multiple events at the same time, which is crucial for web applications that require interactivity and responsiveness.

2. Event Loop:
- JavaScript operates within an event loop. This mechanism continuously checks for events and executes the corresponding callback functions. When an event occurs, such as a user clicking a button, the event is pushed to the event queue. The event loop picks up this event and executes the associated callback function.

3. Event Handlers:
- In JavaScript, developers attach event handlers (functions) to elements that define what should happen when specific events occur. For instance, when a user clicks a button, the associated event handler is triggered, allowing the program to respond dynamically.

Example:
```javascript
document.getElementById('myButton').addEventListener('click', function() {
    alert('Button clicked!');
});
```

4. User Interactivity:
- Most web applications are interactive, meaning they need to respond to user actions. JavaScript's event-driven model allows developers to create rich user interfaces that react instantly to user input, such as clicks, keystrokes, or mouse movements.

5. Handling Multiple Events:
- JavaScript can listen for and handle various types of events on the same element or multiple elements. This flexibility allows developers to create complex user interactions without tightly coupling the code to the sequence of operations.

### Example of Event-Driven Behavior
html
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Event Driven Example</title>
    <script>
        window.onload = function() {
            document.getElementById('myButton').addEventListener('click', function() {
                alert('Button was clicked!');
            });
        };
    </script>
</head>
<body>
    <button id="myButton">Click Me!</button>
</body>
</html>
```

### Explanation of the Example:
1. Event Listener: When the page loads, an event listener is attached to the button with the ID `myButton`. This listener waits for the `click` event.
2. Callback Function: When the button is clicked, the callback function is executed, displaying an alert.
3. Non-blocking: The browser remains responsive, allowing the user to interact with other elements even while waiting for the button to be clicked.

Nested HTML elements

Definition:
- Nested elements refer to HTML **elements placed inside other** HTML elements. This helps create a **hierarchical structure** on the webpage. The outer element is called the parent element, and the inner one is the **child element.** Nesting allows for better organization of content and enables more complex layouts or designs.

Key Points:
- Parent Element: The container or outer element.
- Child Element: The element that is nested inside the parent.
- HTML allows multiple levels of nesting, meaning child elements can also have their own child elements.
- Proper indentation is crucial in nested elements for readability.

### Example of Nested Elements:

1. Paragraph Inside a Div: `<p>` nested inside a `<div>`.
2. List Items Inside an Unordered List: `<li>` nested inside `<ul>`.
3. Form Elements Inside a Form: `<input>`, `<button>`, etc., inside a `<form>`.

### Sample Code:

```html
<!DOCTYPE html>
<html>
<head>
    <title>Nested Elements Example</title>
</head>
<body>

    <div class="container">
        <h1>Welcome to Nested Elements</h1>
        <p>This paragraph is nested inside a div element.</p>

        <ul>
            <li>List Item 1</li>
            <li>List Item 2
                <ul>
                    <li>Sub Item 1</li>
                    <li>Sub Item 2</li>
                </ul>
            </li>
            <li>List Item 3</li>
        </ul>

        <form action="/submit" method="post">
            <label for="name">Name:</label>
            <input type="text" id="name" name="name" />
            <button type="submit">Submit</button>
        </form>
    </div>

</body>
</html>
```

Tags in HTML

HTML tags are the fundamental building blocks used to create the structure of a webpage. Tags are enclosed in angle brackets (< >), and they define the different elements on a page, such as headings, paragraphs, links, images, tables, and more.

Key Concepts of HTML Tags:

- **Opening Tag**: Indicates the start of an HTML element (e.g., `<p>` for a paragraph).
- **Closing Tag**: Marks the end of an HTML element (e.g., `</p>` for a paragraph).
- **Self-Closing Tag**: Some elements don't require a closing tag, such as images (`<img />`).
- **Attributes**: Tags can have attributes that provide additional information about the element (e.g., `<img src="image.jpg" alt="description" />`).

Elements in HTML

- Definition:HTML elements are the building blocks of a web page. They define the structure and content of a webpage. HTML elements can be classified into different types based on their behavior, functionality, and structure.

#### 1. Block-level Elements
- Definition: Block-level elements start on a new line and take up the full width available, stretching out to the left and right as far as they can. They are mainly used for larger content sections.
- Examples: `<div>`, `<h1>`, `<p>`, `<section>`, `<article>`, `<header>`, `<footer>`
- Use: Structuring and dividing content on a webpage.

Example Code:
```html
<div>
  <h1>This is a heading</h1>
  <p>This is a paragraph of text within a block-level element.</p>
</div>
```

#### 2. Inline Elements
- Definition: Inline elements do not start on a new line and only take up as much width as necessary. These are typically used for small pieces of content within block-level elements.
- Examples: `<span>`, `<a>`, `<strong>`, `<em>`, `<img>`, `<input>`
- Use: Formatting small sections of text or embedding media inside larger elements.

Example Code:
```html
<p>This is a paragraph with <strong>bold text</strong> and a <a href="#">link</a>.</p>
```

#### 3. Empty (Void) Elements

- Definition: Empty elements do not have any content or closing tags. They typically perform some self-contained task.
- Examples: `<img>`, `<br>`, `<hr>`, `<input>`
- Use: Inserting self-closing items like images, line breaks, or input fields.

Example Code:
```html
<img src="image.jpg" alt="Example Image">
<br>
<input type="text" placeholder="Enter your name">
```

#### 4. Form Elements
- Definition: Form elements are used to create interactive forms that can be used to collect user input.
- Examples: `<form>`, `<input>`, `<textarea>`, `<button>`, `<select>`, `<option>`
- Use: Creating web forms for user data input, such as text fields, checkboxes, and submit buttons.

Example Code:
```html
<form action="/submit" method="post">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name">
  <button type="submit">Submit</button>
</form>
```

#### 5. Semantic Elements
- Definition: Semantic elements clearly define their meaning in a human- and machine-readable way, helping both developers and search engines understand the structure of the content.
- Examples: `<article>`, `<section>`, `<nav>`, `<aside>`, `<header>`, `<footer>`
- Use: Improving readability and accessibility by marking up specific content sections.

Example Code:
```html
<header>
  <h1>Welcome to My Website</h1>
</header>
<nav>
  <ul>
    <li><a href="#home">Home</a></li>
    <li><a href="#about">About</a></li>
  </ul>
```

</nav>

#### 6. Interactive Elements
- Definition: These elements are specifically designed for user interaction, typically triggering events or actions when interacted with.
- Examples: `<button>`, `<a>`, `<details>`, `<input type="checkbox">`, `<input type="submit">`
- Use: Facilitating user actions, like submitting forms, navigating, or triggering pop-ups.

Example Code:
```html
<button onclick="alert('Button clicked!')">Click Me</button>
```

#### 7. Heading Elements
- Definition: Heading elements define titles or subtitles on a webpage, ranging from `<h1>` (the most important) to `<h6>` (the least important).
- Examples: `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`
- Use: Structuring headings and subheadings to create a hierarchy in the content.

Example Code:
```html
<h1>Main Title</h1>
<h2>Subheading</h2>
<h3>Section Title</h3>
```

#### 8. List Elements
- Definition: List elements are used to group items in an ordered or unordered format.
- Examples: `<ul>`, `<ol>`, `<li>`, `<dl>`, `<dt>`, `<dd>`
- Use: Displaying items in a list format, either numbered or bulleted.

Example Code:
```html
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>

<ol>
  <li>First Item</li>
  <li>Second Item</li>
</ol>
```

#### 9. Table Elements
- Definition: These elements are used to create tables to organize data in rows and columns.
- Examples: `<table>`, `<tr>`, `<td>`, `<th>`, `<thead>`, `<tbody>`, `<tfoot>`
- Use: Structuring data in a grid-like format for better presentation.

Example Code:
```html
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>John</td>
      <td>25</td>
    </tr>
  </tbody>
</table>
```

linking external CSS

There are three primary ways to link external CSS to an HTML document:

### 1. Linking External CSS File (Most common way)
You can use the `<link>` tag in the `<head>` section of your HTML to link an external CSS file.

#### Code Example:
```html
<!DOCTYPE html>
<html>
<head>
    <title>External CSS Example</title>
    <!-- Linking an external CSS file -->
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Hello, World!</h1>
</body>
```

```
</html>
```

#### Explanation:
- The `<link>` tag is used to reference an external CSS file.
- `rel="stylesheet"` specifies the relationship, indicating this is a style sheet.
- `href="styles.css"` is the path to the external CSS file. The `styles.css` file should be located in the same directory as the HTML file, or you can use a relative or absolute path.

### 2. Inline CSS (Within HTML Elements)
You can directly apply styles to individual HTML elements using the `style` attribute within the element itself.

#### Code Example:
```
html
<!DOCTYPE html>
<html>
<head>
    <title>Inline CSS Example</title>
</head>
<body>
    <h1 style="color: blue; font-size: 24px;">Hello, World!</h1>
</body>
</html>
```

Explanation:
- The `style` attribute is added directly to the HTML tag (in this case, `<h1>`).
- Inline CSS is useful for quick, single-use styles but should be avoided for large-scale styling.

### 3. Internal CSS (Within `<style>` Tag in the `<head>` Section)
You can embed CSS directly within the HTML file using the `<style>` tag inside the `<head>`.

#### Code Example:
```html
<!DOCTYPE html>
<html>
<head>
    <title>Internal CSS Example</title>
    <!-- Internal CSS defined within the <style> tag -->
    <style>
        body {
            background-color: lightgray;
        }
        h1 {
            color: blue;
            font-size: 24px;
        }
    </style>
</head>
<body>
    <h1>Hello, World!</h1>
</body>
</html>
```

#### Explanation:
- The `<style>` tag is placed inside the `<head>` section.
- This method is good for styling a single HTML document, but not ideal for projects with multiple pages that share the same styling.

### **4. Importing CSS with `@import` Rule**
You can import an external CSS file into another CSS file using the `@import` rule. This is done within the CSS file, not in the HTML file itself.

#### Code Example:

```css
/* styles.css */
@import url('more-styles.css');

body {
    background-color: lightblue;
}
```

```html
<!DOCTYPE html>
<html>
<head>
    <title>@import CSS Example</title>
    <!-- Linking the main CSS file which imports another CSS file -->
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
```

    <h1>Hello, World!</h1>
</body>
</html>
```

#### Explanation:
- The `@import` rule is written inside the CSS file (`styles.css`) to import another CSS file (`more-styles.css`).
- This method is less efficient because it adds an extra request to the server, which can slow down the loading of the page.

## The DOM Model

represents the structure of an HTML or XML document as a **tree of objects.** It provides a way for scripts (like **JavaScript) to interact with the document's content,** structure, and style **dynamically**.

**### Key Concepts of the DOM Model**

**1. Tree Structure:**
   - The DOM represents a **document as a tree structure**, where each node in the tree represents **a part of the document.** The top node is the document itself, and it branches out to represent elements, attributes, text, comments, etc.
   - **Each element in an HTML document is represented as a node**. For example, `<html>`, `<head>`, `<body>`, `<div>`, and `<p>` are all nodes in the DOM tree.

**2. Nodes:**
   - The DOM consists of different types of nodes:
     - **Element** Nodes: Represent HTML tags (e.g., `<div>`, `<p>`, `<ul>`).
     - **Text** Nodes: Represent the text content inside elements.
     - **Attribute** Nodes: Represent the attributes of elements (e.g., `class`, `id`).
     - **Comment** Nodes: Represent comments in the HTML.

3**. Access and Manipulation:**
   - JavaScript can access and manipulate the DOM, allowing developers to dynamically change the content, structure, and style of a webpage.
   - Common operations include:
     - **Selecting Elements:** Accessing specific elements in the DOM using methods like `getElementById`, `getElementsByClassName`, or `querySelector`.
     - **Modifying Content: Changing the text or HTML** content of elements.
     - **Changing Styles: Modifying CSS** styles directly via JavaScript.
     - **Adding or Removing Elements:** Creating new elements and appending them to the document or removing existing elements.

**### Example of the DOM**
```

Consider the following simple HTML document:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM Example</title>
</head>
<body>
    <h1 id="title">Hello, World!</h1>
    <p class="content">This is a paragraph.</p>
    <button id="changeTextButton">Change Text</button>

    <script>
        // Accessing the DOM
        const titleElement = document.getElementById('title');
        const buttonElement = document.getElementById('changeTextButton');

        // Modifying the DOM
        buttonElement.addEventListener('click', function() {
            titleElement.textContent = 'Text Changed!'; // Change the text of the title
        });
    </script>
</body>
</html>
```

### Explanation of the Example:

1. HTML Structure:
   - The HTML document consists of a heading (`<h1>`), a paragraph (`<p>`), and a button (`<button>`).

2. Accessing Elements:
   - In the JavaScript code, the `document.getElementById` method is used to access the `<h1>` element and the button.

3. Event Handling:
   - An event listener is added to the button, which listens for click events. When the button is clicked, the text of the `<h1>` element is changed using the `textContent` property.

### Benefits of the DOM
- Dynamic Content: The DOM allows developers to create interactive and dynamic web pages by modifying content in real-time without requiring a page refresh.
- Separation of Concerns: The DOM separates the structure (HTML) from the behavior (JavaScript), enabling cleaner and more maintainable code.
- Cross-Browser Compatibility: The DOM provides a standardized interface for interacting with documents, ensuring consistency across different web browsers.

CSS Selectors

CSS selectors are patterns used to select and style HTML elements based on their attributes, types, classes, IDs, or other criteria. They form the basis for applying CSS styles to specific parts of a webpage.

### Uses of CSS Selectors
- Target-Specific Elements: Selectors allow developers to **apply styles to specific elements** in a document.
- Control Presentation: By using different selectors, developers can **control the appearance of** elements **based on their attributes, position, or relation** to other elements.
- Improve Maintainability: Using selectors helps keep CSS **organized** and maintainable by clearly defining which styles apply to which elements.

### Types of CSS Selectors
1. Universal Selector (`*`): Selects all elements.
2. Type Selector (`element`): Selects all elements of a given type **(e.g., `div`, `p`).**
3. Class Selector (`.class-name`): Selects all elements with a specific class.
4. ID Selector (`#id-name`): Selects a **single element** with a specific ID.
5. Attribute Selector (`[attribute]`): Selects elements with a specified attribute.
6. Descendant Selector (`ancestor descendant`): Selects elements that are descendants of a specified ancestor.
7. Child Selector (`parent > child`): Selects direct children of a specified parent.
8. Pseudo-classes (`:pseudo-class`): Selects elements based on their **state (e.g., `:hover`, `:focus`).**

### Code Example
Here's a simple example demonstrating various CSS selectors:
html
```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>CSS Selectors Example</title>
   <style>
      /* Universal Selector */
```

```
        * {
          box-sizing: border-box;
        }

        /* Type Selector */
        p {
          color: blue;
        }

        /* Class Selector */
        .highlight {
          background-color: yellow;
        }

        /* ID Selector */
        #main-title {
          font-size: 24px;
          font-weight: bold;
        }

        /* Descendant Selector */
        div p {
          font-style: italic;
        }

        /* Child Selector */
        ul > li {
          list-style-type: square;
        }

        /* Pseudo-class Selector */
        a:hover {
          color: red;
        }
      </style>
    </head>
    <body>
      <h1 id="main-title">CSS Selectors Example</h1>
      <p>This is a regular paragraph.</p>
      <p class="highlight">This paragraph is highlighted.</p>
      <div>
        <p>This is a paragraph inside a div.</p>
        <ul>
          <li>List item 1</li>
```

```
            <li>List item 2</li>
        </ul>
    </div>
    <a href="#">Hover over me!</a>
</body>
</html>
```

Block Element?

An element that **starts on a new line and takes up the full width available**, stretching out to the **left and right as far** as it can. They typically contain other **block-level or inline elements** and push down content that comes after them to a new line.

### Key Characteristics of Block Elements:
- Always starts on a new line.
- Takes up the full width available (unless width is explicitly defined).
- Can contain both block and inline elements.
- Commonly **used to define sections of a webpage.**

### 10 Common Block Elements in HTML:

1. **`<div>`**: A generic container used to group elements.
   - Use: Useful for styling sections or parts of the webpage.
   - Example:
     ```html
     <div class="container">
       <h1>Welcome to ScholarSync</h1>
       <p>Collaborate with researchers worldwide.</p>
     </div>
     ```
     - Use Case: Grouping content to style multiple elements collectively.

2. **`<h1>` to `<h6>`**: Header elements used for headings of different levels.
   - Use: Represents different levels of headings.
   - Example:
     ```html
     <h1>ScholarSync</h1>
     <h2>Features of ScholarSync</h2>
     ```
     - Use Case: Structuring the hierarchy of content on a page, from most important (`h1`) to least important (`h6`).

3. **`<p>`:** Paragraph element, used to define blocks of text.

  - Use: **For displaying blocks of text content.**
  - Example:
```html
<p>ScholarSync connects students and professors for academic collaborations.</p>
```
    - Use Case: Presenting textual information in a webpage.

4. **`<section>`:** Used to define sections of a webpage.
  - Use: Useful for **dividing a page into logical sections**.
  - Example:
```html
<section>
  <h2>Our Mission</h2>
  <p>To streamline academic collaborations.</p>
</section>
```
    - Use Case: Structuring distinct sections of content, like "About Us," "Features," etc.

5. **`<article>`**: Represents **independent content** that could be **distributed** or **reused** (like a blog post).
  - Use: For self-contained content.
  - Example:
```html
<article>
  <h2>New Collaboration Features in ScholarSync</h2>
  <p>We're adding new ways for researchers to connect...</p>
</article>
```
    - Use Case: Articles, blog posts, or news sections.

6. **`<nav>`**: Used to **define a block of navigation links.**
  - Use: Represents a section for navigational links.
  - Example:
```html
<nav>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">Features</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
</nav>
```
    - Use Case: Creating navigation bars or menus.

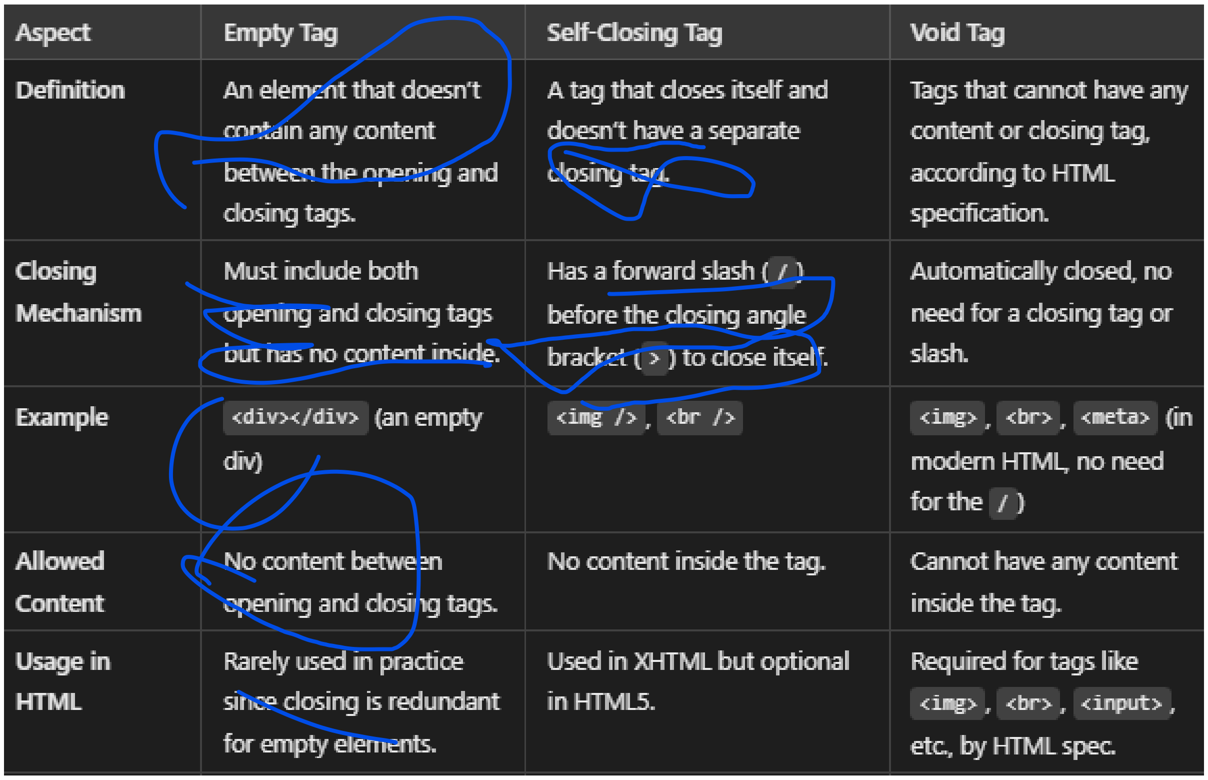
7. **`<header>`**: Represents the header section of a document or section.
   - Use: Contains introductory content or navigational links.
   - Example:
     ```html
     <header>
       <h1>Welcome to ScholarSync</h1>
       <nav>
         <a href="#features">Features</a>
         <a href="#about">About Us</a>
       </nav>
     </header>
     ```
     - Use Case: Adding a header for a page or section.

8. **`<footer>`**: Represents the footer of a document or section.
   - Use: Contains footer information such as copyright or links.
   - Example:
     ```html
     <footer>
       <p>&copy; 2024 ScholarSync</p>
       <a href="#privacy">Privacy Policy</a>
     </footer>
     ```
     - Use Case: Defining a footer section at the bottom of a webpage.

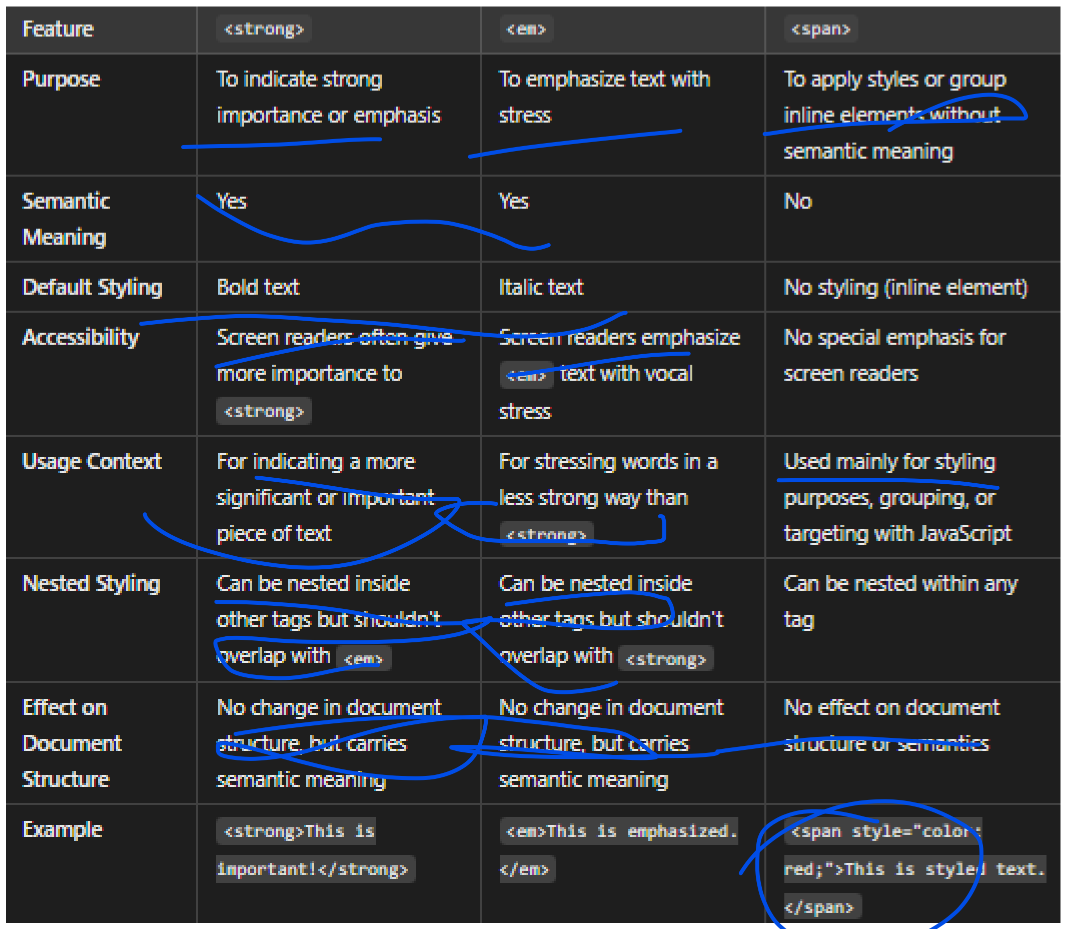
9. **`<aside>`:** Used for content related to the main content, like a sidebar.
   - Use: Typically contains tangential content.
   - Example:
     ```html
     <aside>
       <h2>Related Articles</h2>
       <ul>
         <li><a href="#">How to Collaborate on ScholarSync</a></li>
         <li><a href="#">Tips for Academic Networking</a></li>
       </ul>
     </aside>

     ```
     - Use Case: Displaying side content or advertisements.

10. **`<form>`**: Used to **collect user input through various controls** (input fields, buttons, etc.).
    - Use: For **submitting user data to a server.**
    - Example:
      ```html
      <form action="/submit" method="POST">
      ```

```
  <label for="name">Name:</label>
  <input type="text" id="name" name="name">
  <button type="submit">Submit</button>
</form>
```
- Use Case: Creating forms for user input, such as login, sign-up, or surveys.



Aspect	Empty Tag	Self-Closing Tag	Void Tag
Definition	An element that doesn't contain any content between the opening and closing tags.	A tag that closes itself and doesn't have a separate closing tag.	Tags that cannot have any content or closing tag, according to HTML specification.
Closing Mechanism	Must include both opening and closing tags but has no content inside.	Has a forward slash (`/`) before the closing angle bracket (`>`) to close itself.	Automatically closed, no need for a closing tag or slash.
Example	`<div></div>` (an empty div)	`<img />`, `<br />`	`<img>`, `<br>`, `<meta>` (in modern HTML, no need for the `/`)
Allowed Content	No content between opening and closing tags.	No content inside the tag.	Cannot have any content inside the tag.
Usage in HTML	Rarely used in practice since closing is redundant for empty elements.	Used in XHTML but optional in HTML5.	Required for tags like `<img>`, `<br>`, `<input>`, etc., by HTML spec.



Feature	`<strong>`	`<em>`	`<span>`
Purpose	To indicate strong importance or emphasis	To emphasize text with stress	To apply styles or group inline elements without semantic meaning
Semantic Meaning	Yes	Yes	No
Default Styling	Bold text	Italic text	No styling (inline element)
Accessibility	Screen readers often give more importance to `<strong>`	Screen readers emphasize `<em>` text with vocal stress	No special emphasis for screen readers
Usage Context	For indicating a more significant or important piece of text	For stressing words in a less strong way than `<strong>`	Used mainly for styling purposes, grouping, or targeting with JavaScript
Nested Styling	Can be nested inside other tags but shouldn't overlap with `<em>`	Can be nested inside other tags but shouldn't overlap with `<strong>`	Can be nested within any tag
Effect on Document Structure	No change in document structure, but carries semantic meaning	No change in document structure, but carries semantic meaning	No effect on document structure or semantics
Example	`<strong>This is important!</strong>`	`<em>This is emphasized.</em>`	`<span style="color: red;">This is styled text.</span>`

SAQ

### 1. Flex (CSS Flexbox)

- Definition: A layout model that allows elements to align and distribute space within a container.
- Example: Align items horizontally or vertically.
- Use: To create responsive layouts.
- Short Code:
```css
.container {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

### 2. Grid Class (CSS Grid)

- Definition: A layout system for **dividing a webpage into columns and rows.**
- Example: Creating complex layouts with rows and columns.
- Use: For designing **responsive and multi-dimensional** layouts.

- Short Code:
  ```css
  .grid-container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
  }

---

## Tables in HTML

Tables are used to organize and display data in rows and columns. HTML tables are commonly used for displaying tabular data, like lists, financial data, or schedules.

### HTML Table Structure

1. `<table>`: Defines the table.
2. `<tr>`: Defines a table row.
3. `<th>`: Defines a table header cell (bold and centered by default).
4. `<td>`: Defines a table data cell (contains the content of the table).

### Example Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>HTML Table Example</title>
  <style>
    table {
      width: 100%;
      border-collapse: collapse;
    }
    th, td {
      border: 1px solid black;
      padding: 8px;
      text-align: center;
    }
    th {
      background-color: #f2f2f2;
    }
  </style>
```
```

```
</head>
<body>

<h2>Student Information Table</h2>

<table>
  <tr>
    <th>Name</th>
    <th>Age</th>
    <th>Grade</th>
  </tr>
  <tr>
    <td>John Doe</td>
    <td>20</td>
    <td>A</td>
  </tr>
  <tr>
    <td>Jane Smith</td>
    <td>22</td>
    <td>B+</td>
  </tr>
  <tr>
    <td>Sam Brown</td>
    <td>19</td>
    <td>A-</td>
  </tr>
</table>

</body>
</html>
```

Use of Tables in HTML:

1. **Data Organization**:
    - Useful for displaying structured data like lists, schedules, financial data, or comparison tables.
2. **Layout Structure**:
    - Can be used for simple layout structures (though CSS grid/flexbox are more modern alternatives).
3. **Form Submissions**:

- Tables can be used for presenting form data or output (in combination with forms).

Unit 2 - Javascript

jQuery//

- Fast, small and feature rich js library
- Simplifies tasks
    - Document traversal
    - Event handling ,
    - animation
- Easier to write js
- Use
    - DOM Manipulation
    - Event handling
    - Animation
    - Cross browser compatibility

jQuery is a **fast, small, and feature-rich JavaScript library**. It simplifies things like **HTML document traversal and manipulation, event handling,** animation, and AJAX interactions for rapid web development. By providing a simpler API that works across a multitude of browsers, **jQuery makes it easier to write JavaScript**.

### Key Uses of jQuery:

1. DOM Manipulation:
 - jQuery allows you to easily select, modify, and manipulate HTML elements. It simplifies tasks like changing styles, attributes, or content of elements.
 Example:
 ~~~javascript
 $('#myElement').text('Hello, jQuery!'); // Change the text of an element with ID myElement

**2. Event Handling:**
 - jQuery simplifies event handling, allowing you to easily respond to user interactions like clicks, form submissions, and key presses.

 Example:
 ```javascript
 $('#myButton').click(function() {
 alert('Button clicked!');
 });
 ~~~

3. Animation Effects:
- jQuery includes built-in methods for creating animations and effects, like showing/hiding elements or fading them in and out.

Example:
$('#myElement').fadeOut(); // Fade out the element

4. Cross-Browser Compatibility:
- jQuery handles many of the inconsistencies across different browsers, allowing developers to write less code and worry less about compatibility issues.

### How jQuery Differs from JavaScript

1. Simplicity and Ease of Use:
   - jQuery provides a simplified API that makes it easier to perform common tasks. For example, selecting elements in jQuery is much simpler and more intuitive than using pure JavaScript.

   jQuery Example:
   ```javascript
   $('#myElement') // jQuery way to select an element
   ```

   JavaScript Example:
   ```javascript
   document.getElementById('myElement') // Vanilla JavaScript way
   ```

2. Chaining:
   - jQuery allows method chaining, meaning you can call multiple methods on the same jQuery object in a single line. This results in more concise code.

   Example:
   ```javascript
   $('#myElement').css('color', 'red').slideUp().fadeIn();
   ```

3. Cross-Browser Compatibility:
   - jQuery automatically handles many cross-browser issues, so developers do not need to write specific code for different browsers.

4. Built-in Animations and Effects:
   - jQuery has built-in methods for animations and effects (like `fadeIn()`, `fadeOut()`, and `slideToggle()`) that can be cumbersome to implement in plain JavaScript.

Event-Handling Functions in JavaScript

1. addEventListener()
- Definition: **Attaches an event handler** to an element **without overwriting existing event** handlers.
- Example: **Listening for a click event.**
- Code:
```
  document.getElementById('myButton').addEventListener('click', function() {
    alert('Button clicked!');
  });
```

- Uses:
  - Attaching multiple event handlers to a single element.
  - Supports various events like `**click**`, `**keydown**`, `**mouseover**`.

#### 2. onclick
- Definition: Handles click events directly by **assigning a function to the `onclick` property** of an element.
- Example: Assigning a click event.
- Code:
```js
  document.getElementById('myButton').onclick = function() {
    alert('Button clicked!');
  };
```
- Uses:
  - Quick and simple click event handler.
  - **Can be overwritten easily** with another `onclick` handler.

#### 3. onload
- Definition: **Triggers when an element (like a window or image) has fully loaded**.
- Example: Running code after the page loads.
- Code:
```
  window.onload = function() {
    alert('Page loaded!');
  };
```

- Uses:
  - Useful for executing scripts after resources like images and scripts are fully loaded.

#### 4. onmouseover
- Definition: **Triggers when the mouse pointer moves over an element.**

- Example: Displaying a tooltip on hover.
- Code:
  js
  document.getElementById('myElement').onmouseover = function() {
    alert('Mouse hovered!');
  };
  ~~~
- Uses:
  - Useful for interactive UI elements, showing pop-ups or tooltips.

#### 5. onkeydown
- Definition: Fires when a key is pressed down.
- Example: Capturing key presses in a form.
- Code:
  ```js
  document.addEventListener('keydown', function(event) {
        alert(`Key pressed: ${event.key}`);
  });
  ~~~
- Uses:
  - Handling keyboard inputs for forms, games, and other interactive applications.

#### 6. onblur
- Definition: Fires when an element loses focus.
- Example: Validating form input when the user leaves a text field.
- Code:
  ```js
  document.getElementById('inputField').onblur = function() {
    alert('Field lost focus!');
  };
  ```
- Uses:
  - Useful in forms to trigger validation or changes when the user leaves a field.

8. onsubmit
- Definition: Fires when a form is submitted.
- Example: Validating a form before submission.
- Code:
  ```js
  document.getElementById('myForm').onsubmit = function(event) {
    event.preventDefault(); // Prevent form submission
    alert('Form submitted!');
  };
  ```

- Uses:
  - Form validation and handling custom actions on submission.

Jquery

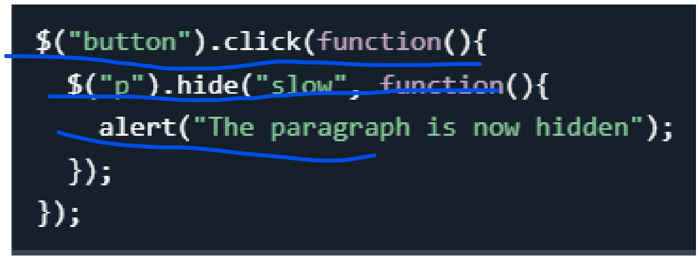
- Fast, feature rich javascript library
- Simplifies : document traversal, animatio effects, event handling
- Makes it easier to write javascript
- **Syntax : $ (selector - HTML elements). action() // action to perform**
- **Used in**
  - **Event handling**
  - **Dom manipulation**
  - Animation
  - Cross browser compatibility
- Selectors
- 1. Current element $(this).hide() - hides the.
- 2. Element selector $("p").hide() - hides all <p> elements.
- 3. Class selector $(".test").hide() - hides all elements with
- 4. ID selector $("#test").hide() - hides the element with id="test".
- 5. Universal selector $(*).hide()

- **Event handling**

```
$("p").click(function(){
        $(this).hide();
});
```

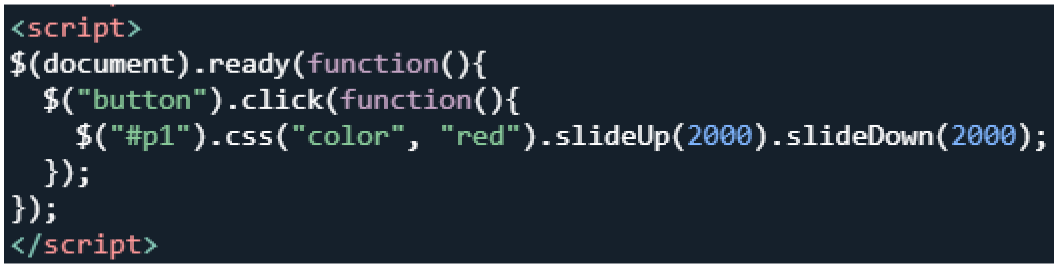
  - Hover : mouse enters and leaves
  - Focus : form field gets focus
  - Mouseup : mouse buttons released
  - Mousedown : mouse buttons pressed
  - Mouseleave : pointer leaves elt
  - Mouseenter : pointer enters elt
  - Click : clicks on elt

- **jQuery Callback**
    - Js statement executed line by line, causes errors
    - To prevent jQuery callback used
    - **Callback : exceuted after current effect finished**



```
$("button").click(function(){
  $("p").hide("slow", function(){
    alert("The paragraph is now hidden");
  });
});
```

- Jquery Chaining
    - Earlier, run multiple commands on 1 element one after other
    - Now, run multiple jQuery methods all together



```
<script>
$(document).ready(function(){
  $("button").click(function(){
    $("#p1").css("color", "red").slideUp(2000).slideDown(2000);
  });
});
</script>
```

    -
    - $("#newid").css("color", "red").slideUp(2000).slideDown(2000);

Ajax - **Asynchronous JavaScript and XML)**

AJAX is a technique used to make asynchronous requests to a web server without reloading the entire page. It allows for dynamic updates to parts of a web page, providing a smoother user experience. AJAX uses JavaScript, HTML, and XML (or JSON) to exchange data between the client and server in the background.

Key Benefits of AJAX:

1. **Improved User Experience**: Updates specific sections of a page without refreshing the entire page.
2. **Faster Web Applications**: Only necessary data is sent and received, reducing bandwidth usage and server load.
3. **Asynchronous**: Allows for background processing, meaning the user can continue interacting with the page.

Sample Code (AJAX with jQuery)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>AJAX Example</title>
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
  <script>
    function fetchData() {
      $.ajax({
        url: "data.php",  // Server-side script to fetch data
        method: "GET",     // HTTP method
        success: function(response) {
          $('#result').html(response);  // Display result in the <div> with id="result"
        },
        error: function() {
          $('#result').html("Error loading data");
        }
      });
    }
  </script>
</head>
<body>

<h2>AJAX Example</h2>
<button onclick="fetchData()">Fetch Data</button>

<div id="result"></div>

</body>
</html>
```

Server-Side Code (data.php)

```
<?php
  echo "Hello, this is the data returned from the server!";
?>
```

How It Works:

1. The user clicks the **Fetch Data** button.
2. The **fetchData()** JavaScript function is triggered, sending an AJAX request to the server (`data.php`).

3. The server responds with data (in this case, a simple text message).
4. The response is displayed in the `<div id="result"></div>` without refreshing the page.

Control flow statements

Control flow statements in JavaScript are used to **dictate the order in which statements** are **executed** in a program. They allow you to **control the flow of execution based on certain conditions or loops**. Here are the main types of control flow statements in JavaScript:

### 1. Conditional Statements
These statements allow you to **execute certain blocks of code** based on specific conditions.

- if Statement:
Executes a block of code if a specified condition is true.

```javascript
let age = 18;
if (age >= 18) {
    console.log("You are an adult.");
}
```

- if...else Statement:
Executes one block of code if the condition is true and another block if it's false.

```javascript
let age = 16;
if (age >= 18) {
    console.log("You are an adult.");
} else {
    console.log("You are a minor.");
}
```

- else if Statement:
Allows you to check multiple conditions.

```javascript
let score = 85;
if (score >= 90) {
    console.log("Grade: A");
} else if (score >= 80) {
    console.log("Grade: B");
} else {
    console.log("Grade: C");
}
```

- switch Statement:
A more concise way to execute different blocks of code based on the value of a variable.

```javascript
let fruit = "banana";
switch (fruit) {
    case "apple":
        console.log("You chose an apple.");
        break;
    case "banana":
        console.log("You chose a banana.");
        break;
    default:
        console.log("Unknown fruit.");
}
```

### 2. Looping Statements
These statements allow you to execute a block of code multiple times.

- for Loop:
Repeats a block of code a specified number of times.

```javascript
for (let i = 0; i < 5; i++) {
    console.log("Iteration: " + i);
}
```

- while Loop:
Repeats a block of code as long as a specified condition is true.

```javascript
let count = 0;
while (count < 5) {
    console.log("Count: " + count);
    count++;
}
```

- do...while Loop:
Similar to the `while` loop, but it executes the block of code at least once before checking the condition.

```javascript
let count = 0;
do {
    console.log("Count: " + count);
    count++;
} while (count < 5);
```

### 3. Break and Continue Statements
These statements are used to control the flow within loops.

- break: Exits the loop immediately.

```javascript
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        break; // Exits the loop when i is 5
    }
    console.log(i);
}
```

- continue: Skips the current iteration and continues with the next one.

```javascript
for (let i = 0; i < 10; i++) {
    if (i % 2 === 0) {
        continue; // Skips even numbers
    }
    console.log(i); // Logs odd numbers
}
```

Form Validation in JavaScript

Definition: Ensuring user inputs in forms meet specific criteria before submission.

Example:
```
<form id="myForm">
    <input type="email" id="email" required>
    <button type="submit">Submit</button>
</form>
<script>
    document.getElementById("myForm").onsubmit = function(event) {
        const email = document.getElementById("email").value;
        if (!email.includes("@")) {
            alert("Invalid email!");
            event.preventDefault();
        }
    }

</script>
```

Forms in HTML

Here’s an example of a simple HTML form with various input types and minimal CSS styling:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Simple Form</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 20px;
      padding: 20px;
      background-color: #f4f4f9;
    }
  </style>
</head>
<body>

<h2>Simple Form Example</h2>

<form action="/submit" method="POST">
  <label for="text-input">Text Input:</label>
  <input type="text" id="text-input" name="text-input" placeholder="Enter your name" required>
```

```
    <label for="email-input">Email:</label>
    <input type="email" id="email-input" name="email-input" placeholder="Enter your email"
required>

    <label for="password-input">Password:</label>
    <input type="password" id="password-input" name="password-input" placeholder="Enter
your password" required>

    <label for="number-input">Number:</label>
    <input type="number" id="number-input" name="number-input" min="0" max="100" required>

    <label for="date-input">Date:</label>
    <input type="date" id="date-input" name="date-input" required>

    <label for="file-input">File Upload:</label>
    <input type="file" id="file-input" name="file-input">

    <label for="select-option">Select:</label>
    <select id="select-option" name="select-option">
        <option value="option1">Option 1</option>
        <option value="option2">Option 2</option>
        <option value="option3">Option 3</option>
    </select>

    <label for="textarea">Textarea:</label>
    <textarea id="textarea" name="textarea" rows="4" placeholder="Enter your
message"></textarea>

    <label>Radio Buttons:</label>
    <input type="radio" id="radio1" name="radio-group" value="option1">
    <label for="radio1">Option 1</label>
    <input type="radio" id="radio2" name="radio-group" value="option2">
    <label for="radio2">Option 2</label>

    <label>Checkbox:</label>
    <input type="checkbox" id="checkbox1" name="checkbox1" value="agree">
    <label for="checkbox1">I agree to the terms and conditions</label>

    <button type="submit">Submit</button>
</form>

</body>
</html>
```

Unit 3 - PHP

Form Validation in PHP

- input provided by the user meets predefined rules
- prevents invalid, incomplete, or harmful data

Key Methods in PHP Form Validation

1. **Server-Side Validation**: Ensures validation happens on the server, regardless of the client.
2. **Input Sanitization:** Removes or escapes harmful characters.
3. **Validation Techniques:**
   - Required fields
   - Specific data types (e.g., email, numbers)
   - Minimum and maximum length
   - Matching patterns (e.g., regex for a phone number)

Explanation of Methods

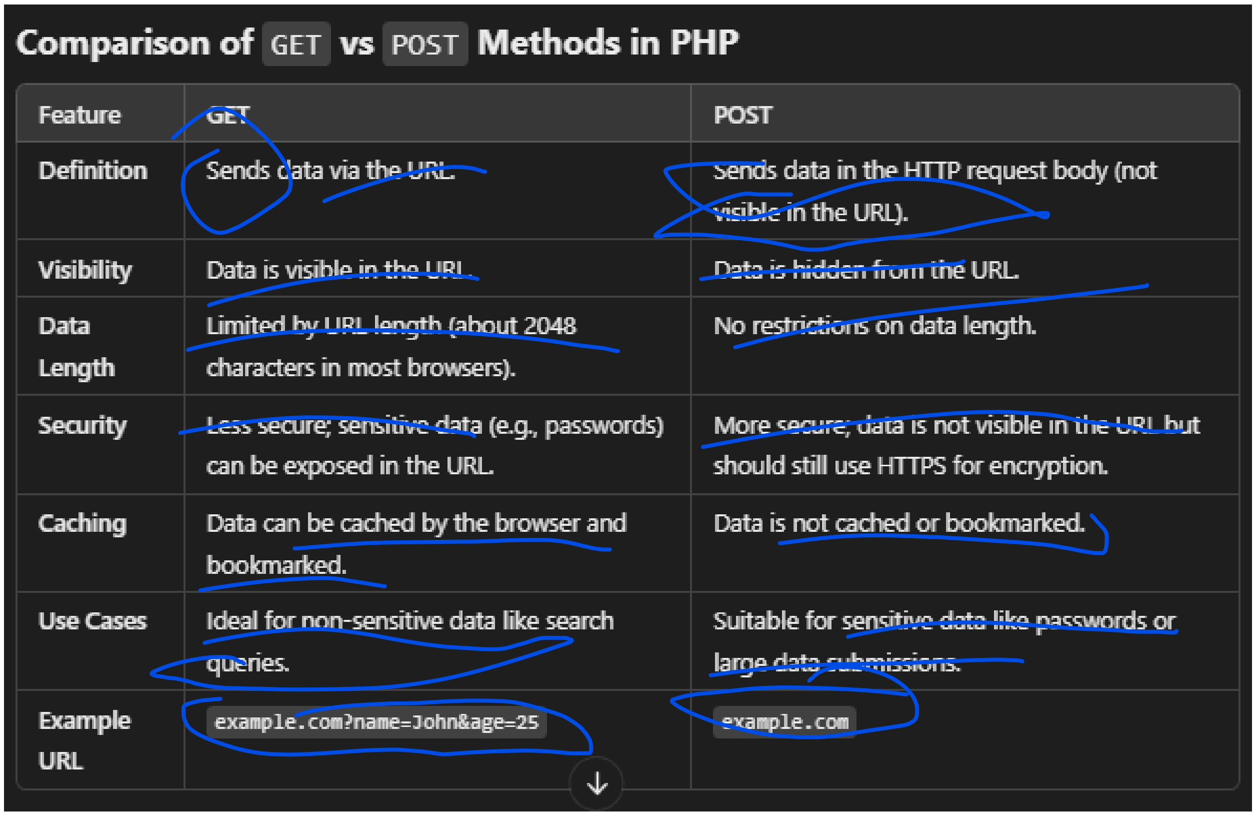
1. **$_SERVER["REQUEST_METHOD"]**:
   - Detects the request type (GET or POST).
   - Used to ensure validation only happens after the form is submitted.
2. **empty()**:Checks if a field is empty.
3. **preg_match()**:Validates input against a regex pattern.
4. **filter_var()**:Filters and validates variables (e.g., checks if an email is valid).
5. **htmlspecialchars()**:Prevents cross-site scripting (XSS) by converting special characters to HTML entities.
6. **trim()**:Removes unnecessary spaces.
7. **stripslashes()**:Removes escape slashes.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>PHP Form Validation</title>
</head>
<body>
    <?php
    // Initialize variables
    $name = $gender = $address = $phone = $email = "";
    $nameErr = $genderErr = $phoneErr = $addressErr = $emailErr = "";

    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        // Validate Name
        if (empty($_POST["name"])) {
            $nameErr = "Name is required.";
        } else {
            $name = sanitizeInput($_POST["name"]);
        }

        // Validate Gender
        if (empty($_POST["gender"])) {
            $genderErr = "Gender is required.";
        } else {
            $gender = sanitizeInput($_POST["gender"]);
        }

        // Validate Phone
        if (empty($_POST["phone"])) {
            $phoneErr = "Phone number is required.";
        } else {
            $phone = sanitizeInput($_POST["phone"]);
            if (!preg_match("/^[0-9]{10}$/", $phone)) {
                $phoneErr = "Phone number must be 10 digits.";
            }
        }

        // Validate Address (Optional but with minimum length)
        if (!empty($_POST["address"])) {
            $address = sanitizeInput($_POST["address"]);
            if (strlen($address) < 10) {
                $addressErr = "Address must be at least 10 characters.";
            }
```

("/^[0-9]{10}$/

```
        }

        // Validate Email (Optional)
        if (!empty($_POST["email"])) {
            $email = sanitizeInput($_POST["email"]);
            if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
                $emailErr = "Invalid email format.";
            }
        }
    }

    // Function to sanitize input
    function sanitizeInput($data) {
        return htmlspecialchars(stripslashes(trim($data)));
    }
    ?>

    <!-- HTML Form -->
    <h2>PHP Form Validation</h2>
    <form method="post" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]); ?>">
        <!-- Name -->
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" value="<?php echo $name; ?>">
        <span style="color: red;">* <?php echo $nameErr; ?></span>
        <br><br>

        <!-- Gender -->
        <label for="gender">Gender:</label>
        <input type="radio" id="male" name="gender" value="Male" <?php if ($gender == "Male")
echo "checked"; ?>> Male
        <input type="radio" id="female" name="gender" value="Female" <?php if ($gender ==
"Female") echo "checked"; ?>> Female
        <span style="color: red;">* <?php echo $genderErr; ?></span>
        <br><br>

        <!-- Address -->
        <label for="address">Address:</label>
        <textarea id="address" name="address" rows="4" cols="30"><?php echo $address;
?></textarea>
        <span style="color: red;"><?php echo $addressErr; ?></span>
        <br><br>

        <!-- Phone -->
        <label for="phone">Phone Number:</label>
        <input type="text" id="phone" name="phone" value="<?php echo $phone; ?>">
        <span style="color: red;">* <?php echo $phoneErr; ?></span>
        <br><br>

        <!-- Email -->
        <label for="email">Email:</label>
        <input type="text" id="email" name="email" value="<?php echo $email; ?>">
        <span style="color: red;"><?php echo $emailErr; ?></span>
        <br><br>

        <!-- Submit Button -->
        <input type="submit" value="Submit">
    </form>

    <!-- Display Input Data -->
    <?php
    if ($_SERVER["REQUEST_METHOD"] == "POST" && empty($nameErr) &&
empty($genderErr) && empty($phoneErr) && empty($addressErr) && empty($emailErr)) {
        echo "<h3>Your Input:</h3>";
        echo "Name: $name<br>";
        echo "Gender: $gender<br>";
        echo "Address: $address<br>";
        echo "Phone Number: $phone<br>";
        echo "Email: $email<br>";
    }
    ?>
</body>
</html>
```

Get vs Post in PHP



Comparison of `GET` vs `POST` Methods in PHP

Feature	GET	POST
Definition	Sends data via the URL.	Sends data in the HTTP request body (not visible in the URL).
Visibility	Data is visible in the URL.	Data is hidden from the URL.
Data Length	Limited by URL length (about 2048 characters in most browsers).	No restrictions on data length.
Security	Less secure; sensitive data (e.g., passwords) can be exposed in the URL.	More secure; data is not visible in the URL but should still use HTTPS for encryption.
Caching	Data can be cached by the browser and bookmarked.	Data is not cached or bookmarked.
Use Cases	Ideal for non-sensitive data like search queries.	Suitable for sensitive data like passwords or large data submissions.
Example URL	`example.com?name=John&age=25`	`example.com`

Regular Expressions in PHP - Summary

- **Definition**: Sequence of characters forming a search pattern for text search and replace operations.
- **Components**:
  - *Delimiter*: Marks the beginning and end of the pattern (e.g., /, #, ~).
  - *Pattern*: The text or sequence to match.
  - *Modifiers*: Optional flags, e.g., `i` for case-insensitivity.

Common Functions:

1. **preg_match()**: Checks if a pattern is found in a string.

Example:
 $str = "Visit W3Schools";
$pattern = "/w3schools/i";

echo preg_match($pattern, $str); // Output: 1

-

2. **preg_match_all()**: Counts all occurrences of a pattern in a string.

Example:
$str = "The rain in SPAIN falls mainly on the plains.";
$pattern = "/ain/i";
echo preg_match_all($pattern, $str); // Output: 4

-

3. **preg_replace()**: Replaces all matches of a pattern with another string.

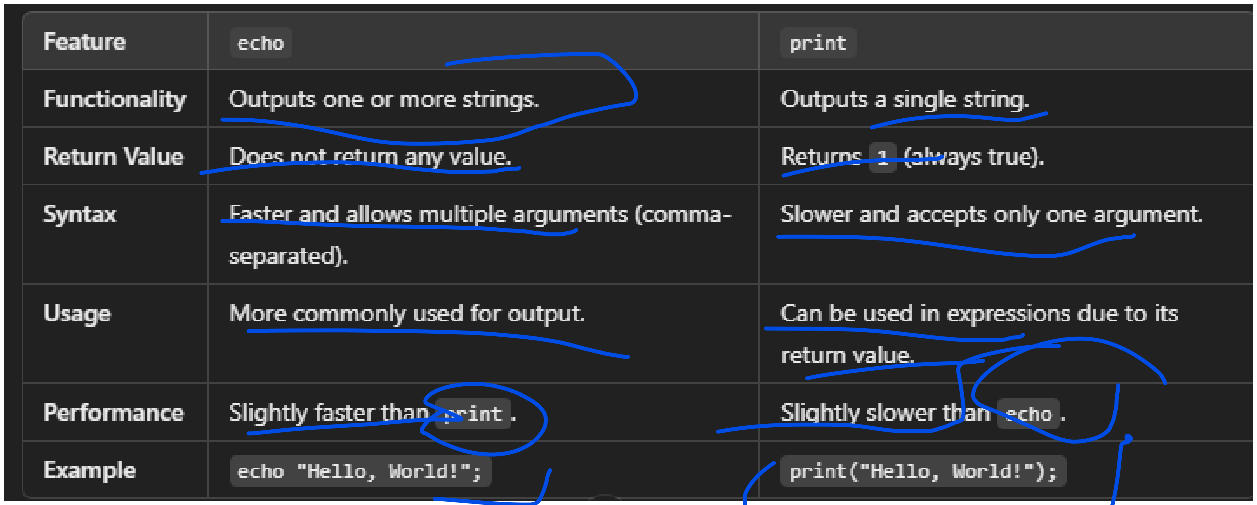
Example:
$str = "Visit Microsoft!";
$pattern = "/microsoft/i";
echo preg_replace($pattern, "W3Schools", $str); // Output: Visit W3Schools!

-

Key Notes:

- Regular expressions are versatile for searching and replacing text.
- Common modifier: `i` (case-insensitive).
- Alternate delimiters (#, ~) are useful if the pattern contains `/`.

Echo vs Print



Feature	`echo`	`print`
Functionality	Outputs one or more strings.	Outputs a single string.
Return Value	Does not return any value.	Returns `1` (always true).
Syntax	Faster and allows multiple arguments (comma-separated).	Slower and accepts only one argument.
Usage	More commonly used for output.	Can be used in expressions due to its return value.
Performance	Slightly faster than `print`.	Slightly slower than `echo`.
Example	`echo "Hello, World!";`	`print("Hello, World!");`

Code :

```
<?php
echo "Hello, ", "World!", " PHP!";
?>

<?php
print("Hello, World!");
?>
```

Both are used to display output, but **echo** is preferred for better performance and multiple arguments.

Variables in PHP

1. Naming Conventions

- Must start with a $ sign followed by a letter or underscore.
- Cannot start with a number.
- Only contain letters, numbers, and underscores (_).
- Case-sensitive: $var and $Var are different.

Examples:

```
$validVariable = "Hello";

$_anotherVariable = 10;
```

Invalid: $3variable, $my-variable.

---

2. Scope of Variables

1. **Local Scope**:
    - Declared inside a function.
    - Accessible only within that function.

```
function myFunc() {
    $localVar = "I am local";

    echo $localVar; // Accessible

}

// echo $localVar; // Error: Undefined variable
```

2. **Global Scope**:
   - Declared outside of all functions.
   - Use global keyword to access inside functions.

```
$globalVar = "I am global";

function myFunc() {

    global $globalVar;

    echo $globalVar; // Accessible using global keyword

}
```

3. **Static Scope**:
   - Retains its value between function calls.

```
function counter() {

    static $count = 0;

    $count++;

    echo $count;

}

counter(); // Output: 1

counter(); // Output: 2
```

4. **Superglobal**:
    - Built-in variables accessible from anywhere.
    - Examples: $_POST, $_GET, $_SESSION, $_SERVER.

---

3. Types of Variables

1. **String**:
    - Holds textual data.
    - Example: $name = "John";.
2. **Integer**:
    - Holds whole numbers.
    - Example: $age = 25;.
3. **Float/Double**:
    - Holds decimal numbers.
    - Example: $price = 19.99;.
4. **Boolean**:
    - Holds true or false.
    - Example: $isLoggedIn = true;.
5. **Array**:
    - Holds multiple values.
    - Example: $fruits = array("Apple", "Banana");.
6. **Object**:
    - Represents an instance of a class.

Example:

```
 class Car {

    public $color;

}
```

```
$myCar = new Car();
```

7. **NULL**:
   - Represents a variable with no value.
   - Example: `$var = NULL;`
8. **Resource**:
   - Holds a reference to an external resource (e.g., database connection).

Flowchart: Variable Scope

```
+---------------------------+

|       Declare Variable    |

+---------------------------+

         ↓

Is it inside a function? -> No -> Global Scope

         ↓ Yes

Is it marked "static"? -> Yes -> Static Scope

         ↓ No

  Local Scope
```

Quick Code Example

```
<?php

// Global Scope
```

```
$globalVar = "Global";

function myFunction() {
    global $globalVar;
    echo $globalVar; // Accessing global variable

    static $staticVar = 0; // Static Scope
    $staticVar++;
    echo $staticVar;

    $localVar = "Local"; // Local Scope
    echo $localVar;
}
myFunction();
```

Cookies, Sessions, and Filters in PHP

1. Cookies

- **Definition**: A small piece of data stored on the client's browser.
- **Purpose**: Used to remember user preferences or data across multiple pages.
- **Lifetime**: Can persist beyond the session if an expiration time is set.

Syntax:

```
setcookie(name, value, expire, path, domain, secure, httponly);
```

Examples:

Set a Cookie:

```
setcookie("user", "John", time() + (86400 * 30), "/"); // Expires in 30 days
```

1.

Access a Cookie:

```
if(isset($_COOKIE["user"])) {

   echo "User: " . $_COOKIE["user"];

}
```

2.

Delete a Cookie:

```
setcookie("user", "", time() - 3600, "/");
```

3.

---

2. Sessions

- **Definition**: A way to store information on the server for individual users.
- **Purpose**: Used to maintain user-specific data during a single session (e.g., login).
- **Lifetime**: Lasts until the browser is closed or manually destroyed.

Steps:

Start a session:

```
session_start();
```

1.

Store data:
```
$_SESSION["username"] = "John";
```

2.

Access data:
```
echo "Welcome, " . $_SESSION["username"];
```

3.

Destroy a session:
```
session_start();

session_unset(); // Removes session variables

session_destroy(); // Destroys the session
```

4.

Key Difference Between Cookies and Sessions:

- **Cookies** store data on the client-side.
- **Sessions** store data on the server-side.

---

3. Filters

- **Definition**: Used to validate and sanitize user inputs.
- **Purpose**: Prevent malicious inputs (e.g., SQL injection, XSS attacks).

Common Filter Functions:

Sanitize:
```
$email = filter_var($input, FILTER_SANITIZE_EMAIL);
```

1.

Validate:
```
if (filter_var($email, FILTER_VALIDATE_EMAIL)) {

   echo "Valid email";
} else {
    echo "Invalid email";
}
```

2.

Examples of Filters:

Filter an Integer:

```
 $int = "123abc";
$sanitizedInt = filter_var($int, FILTER_SANITIZE_NUMBER_INT);
echo $sanitizedInt; // Output: 123
```

1.

Validate a URL:

```
 $url = "http://example.com";
if (filter_var($url, FILTER_VALIDATE_URL)) {
    echo "Valid URL";
} else {
    echo "Invalid URL";
}
```

2.

Filter List:

- FILTER_SANITIZE_STRING (deprecated in PHP 8).
- FILTER_SANITIZE_EMAIL.
- FILTER_VALIDATE_INT, FILTER_VALIDATE_EMAIL, etc.

Comparison Table

Feature	Cookies	Sessions	Filters
Storage	Client-side (browser).	Server-side.	Not for storage, used for sanitizing/validating inputs.
Lifetime	Persistent (can set expiration).	Ends with the session or manual destroy.	N/A.
Purpose	Remember user preferences or data.	Maintain state across pages (e.g., login).	Prevent invalid/malicious inputs
Examples	`$_COOKIE` to access stored data.	`$_SESSION` to access session data.	`filter_var` for validation/sanitization.

PHP Callback Functions

Definition

- A **callback function** is a function that is passed as an argument to another function and is executed at a later time.
- Callback functions can be **user-defined** or **built-in**.

Types of Callback Functions

1. **Simple Function Callback**
2. **Object Method Callback**

3. **Static Class Method** Callback

Syntax and Examples

1. Simple Function Callback

Pass a function name as a string to another function.

Example:

```
<?php

function greet($name) {

    return "Hello, $name!";

}


function callCallback($callback, $param) {

    return $callback($param);

}


echo callCallback('greet', 'John'); // Output: Hello, John!

?>
```

---

2. Object Method Callback

Use an array to specify an object and a method.

Example:

```
<?php
```

```
class Greeter {

    public function greet($name) {

        return "Hi, $name!";

    }

}


$greeter = new Greeter();

function executeCallback($callback, $param) {

    return call_user_func($callback, $param);

}


echo executeCallback([$greeter, 'greet'], 'Alice'); // Output: Hi,
Alice!

?>
```

---

3. Static Class Method Callback

Use an array with the class name and the static method.

Example:

```
<?php

class MathOperations {

    public static function square($num) {

        return $num * $num;

    }
}


function performCallback($callback, $param) {
    return call_user_func($callback, $param);
}


echo performCallback(['MathOperations', 'square'], 5); // Output: 25
?>
```

Built-in Functions That Use Callbacks

array_map(): Applies a callback to each array element.

```
 <?php
$numbers = [1, 2, 3];
$squared = array_map('MathOperations::square', $numbers);
print_r($squared); // Output: [1, 4, 9]
?>
```

1.

array_filter(): Filters array elements using a callback.

```
 <?php
$numbers = [1, 2, 3, 4, 5];
$even = array_filter($numbers, function($num) {
    return $num % 2 === 0;
});
print_r($even); // Output: [2, 4]
?>
```

2.

usort(): Sorts an array using a callback for comparison.

```
 <?php
$fruits = ['banana', 'apple', 'cherry'];
usort($fruits, function($a, $b) {
    return strcmp($a, $b);
});
print_r($fruits); // Output: ['apple', 'banana', 'cherry']
?>
```

3.

Flowchart: Callback Function Execution

Start

↓

Function A is called

↓

Function A receives Function B as an argument

↓

Function B is executed inside Function A

↓

End

Key Points

- **Syntax for call_user_func**: call_user_func($callback, $param1, $param2, ...)
- Useful for **dynamic function calls** and **higher-order functions**.
- Can be **anonymous functions**, **closures**, or predefined methods.

PHP and JSON

1. What is JSON?

- **JSON** (**JavaScript Object Notation**) is a lightweight data format used for exchanging data between a client and server.
- **Features**:
    - Human-readable.
    - Language-independent.
    - Used for structured data representation (like arrays, objects).

2. Working with JSON in PHP

PHP provides built-in functions to handle JSON data.

3. Encoding PHP Data to JSON

Convert PHP data structures (arrays or objects) into JSON strings.

Function: json_encode()
Example:

```
<?php
```

```
$data = array("name" => "John", "age" => 25, "city" => "New York");

$jsonData = json_encode($data);

echo $jsonData; // Output: {"name":"John","age":25,"city":"New York"}

?>
```

---

4. Decoding JSON to PHP

Convert JSON strings into PHP arrays or objects.

Function: `json_decode()`
Example:

```
<?php

$jsonData = '{"name":"John","age":25,"city":"New York"}';

$data = json_decode($jsonData, true); // Decode as associative array

print_r($data);

/* Output:

Array

(

    [name] => John

    [age] => 25

    [city] => New York

)

*/

?>
```

---

5. Common PHP and JSON Functions

Function	Description	Example
`json_encode()`	Converts PHP data to JSON string.	`json_encode($data);`
`json_decode()`	Converts JSON string to PHP data (array/object).	`json_decode($json, true);`
`json_last_error()`	Returns the last error occurred during JSON operations.	`json_last_error();`
`json_last_error_msg()`	Returns a readable error message for the last JSON error.	`json_last_error_msg();`

Exception Handling in PHP

1. What is Exception Handling?

- **Definition**: Mechanism to handle errors and exceptional conditions in a program without crashing.
- **Purpose**: Allows graceful handling of unexpected situations by "throwing" and "catching" exceptions.

2. Key Terms

1. **Try Block**: Code that may throw an exception is placed here.
2. **Catch Block**: Code to handle the exception is placed here.
3. **Throw**: Used to trigger an exception manually.
4. **Finally Block** (optional): Code that will always execute, regardless of whether an exception occurred.

---

3. Example: Basic Exception Handling

```
<?php

function divide($a, $b) {

    if ($b == 0) {

        throw new Exception("Division by zero is not allowed.");

    }

    return $a / $b;

}


try {

    echo divide(10, 0);

} catch (Exception $e) {

    echo "Error: " . $e->getMessage();

}

?>
```

Output:

```
Error: Division by zero is not allowed.
```

---

5. Multiple catch Blocks

- Use multiple catch blocks for different exception types.

```
<?php

try {

    throw new InvalidArgumentException("Invalid argument provided.");

} catch (InvalidArgumentException $e) {

    echo "Caught InvalidArgumentException: " . $e->getMessage();

} catch (Exception $e) {

    echo "Caught general Exception: " . $e->getMessage();

}

?>
```

---

6. The finally Block

- Executes regardless of whether an exception was thrown.

```
<?php

try {

    throw new Exception("Something went wrong.");

} catch (Exception $e) {

    echo "Caught Exception: " . $e->getMessage();

} finally {

    echo "Execution complete.";
}

?>
```

Output:

```
Caught Exception: Something went wrong. Execution complete.
```

---

7. Custom Exceptions

- Create user-defined exception classes by extending the Exception class.

Example:

```
<?php

class CustomException extends Exception {

    public function errorMessage() {

        return "Custom Error: " . $this->getMessage();

    }

}


try {

    throw new CustomException("This is a custom exception.");

} catch (CustomException $e) {

    echo $e->errorMessage();

}

?>
```

8. Common Exception Methods

Method	Description
getMessage()	Gets the exception message.
getCode()	Gets the exception code.
getFile()	Gets the file where the exception occurred.
getLine()	Gets the line number where the exception occurred.
getTrace()	Gets the stack trace as an array.
getTraceAsString()	Gets the stack trace as a string.

Example:

```
<?php

try {

    throw new Exception("Error occurred", 500);

} catch (Exception $e) {
    echo "Message: " . $e->getMessage() . "<br>";

    echo "Code: " . $e->getCode() . "<br>";

    echo "File: " . $e->getFile() . "<br>";

    echo "Line: " . $e->getLine() . "<br>";

}

?>
```

9. Advantages of Exception Handling

- Separates error-handling code from regular code.
- Improves code readability and maintainability.
- Allows for custom error messages.

Flowchart: Exception Handling Process

```
Start

  ↓

Execute Try Block

  ↓

Exception Thrown?

  ↓             ↘

Yes             No

  ↓              ↘

Catch Block      Finally Block
```

File Handling in PHP

File handling allows you to read, write, and manipulate files in PHP. It is used for operations like creating, opening, reading, writing, and closing files.

1. Common Functions in File Handling

Function	Description
`fopen()`	Opens a file or creates it if it doesn't exist.
`fclose()`	Closes an open file.
`fread()`	Reads from an open file.
`fwrite()`	Writes data to an open file.
`file_get_contents()`	Reads the entire file into a string.
`file_put_contents()`	Writes a string to a file. Overwrites the file if it exists.

`unlink()`	Deletes a file.
`feof()`	Checks if the end of the file has been reached.
`filesize()`	Returns the size of the file in bytes.

---

2. File Modes for `fopen()`

Mode	Description
`'r'`	Read-only. Starts at the beginning of the file.
`'w'`	Write-only. Truncates the file to zero length or creates it if it doesn't exist.
`'a'`	Write-only. Appends to the file or creates it if it doesn't exist.
`'x'`	Write-only. Creates a new file. Returns `FALSE` if the file exists.
`'r+'`	Read/Write. Starts at the beginning of the file.

'w+'	Read/Write. Truncates the file to zero length or creates it if it doesn’t exist.
'a+'	Read/Write. Appends to the file or creates it if it doesn’t exist.

Program: File Handling in PHP

```
<?php

// Step 1: Create a new file and write to it

$fileName = "example.txt";

$file = fopen($fileName, "w"); // Open in write mode

if ($file) {

    fwrite($file, "This is the first line of the file.\n"); // Write to file

    fclose($file); // Close the file

    echo "File created and initial content written.\n";

} else {

    echo "Failed to create file.\n";

}


// Step 2: Read from the file

$file = fopen($fileName, "r"); // Open in read mode

if ($file) {
    echo "Reading file content:\n";
    while (!feof($file)) { // Check end of file
        echo fgets($file); // Read line by line
    }
    fclose($file); // Close the file
} else {
    echo "Failed to open file for reading.\n";
}

// Step 3: Append data to the file
$file = fopen($fileName, "a"); // Open in append mode
if ($file) {
    fwrite($file, "This is appended content.\n"); // Append data
    fclose($file); // Close the file
    echo "Content appended to file.\n";
} else {
    echo "Failed to open file for appending.\n";
}
```

```
// Step 4: Re-read the file to confirm appended content

$file = fopen($fileName, "r"); // Open in read mode

if ($file) {

    echo "Reading file content after appending:\n";

    while (!feof($file)) { // Check end of file

        echo fgets($file); // Read line by line

    }

    fclose($file); // Close the file

} else {

    echo "Failed to open file for reading.\n";

}


// Step 5: Delete the file

if (unlink($fileName)) { // Delete the file

    echo "File deleted successfully.\n";

} else {

    echo "Failed to delete the file.\n";

}

?>
```

Program Explanation

1. **Create a New File**:
    - Use `fopen()` with mode `"w"` to create a new file or overwrite an existing file.
    - Write initial content using `fwrite()`.
    - Close the file with `fclose()`.
2. **Read from the File**:
    - Use `fopen()` with mode `"r"` to open the file in read mode.
    - Use `fgets()` in a `while` loop to read line by line until `feof()` indicates the end of the file.
3. **Append Data to the File**:
    - Use `fopen()` with mode `"a"` to open the file in append mode.
    - Append data using `fwrite()`.
4. **Check End of File**:
    - Use `feof()` inside the `while` loop to determine when the end of the file is reached.
5. **Delete the File**:
    - Use `unlink()` to delete the file.

Output

1. File created and initial content written.
2. Displays the content of the file.
3. Appends new content.
4. Reads the file again to confirm the appended content.
5. Deletes the file and confirms its deletion.

4. Handling Errors

Check for errors during file operations using conditional statements.

Example:

```
<?php

$file = fopen("nonexistent.txt", "r");
```

```
if (!$file) {

    echo "Error: File not found!";

} else {

    fclose($file);

}

?>
```

5. File Upload Example

```
<!DOCTYPE html>

<html>

<body>

    <form action="upload.php" method="post"
enctype="multipart/form-data">

        Select file to upload:

        <input type="file" name="fileToUpload">

        <input type="submit" value="Upload File">

    </form>

</body>

</html>
```

upload.php:

```
<?php
```

```
$targetDir = "uploads/";

$targetFile = $targetDir . basename($_FILES["fileToUpload"]["name"]);


if (move_uploaded_file($_FILES["fileToUpload"]["tmp_name"],
$targetFile)) {

    echo "File uploaded successfully.";

} else {

    echo "Failed to upload file.";

}

?>
```

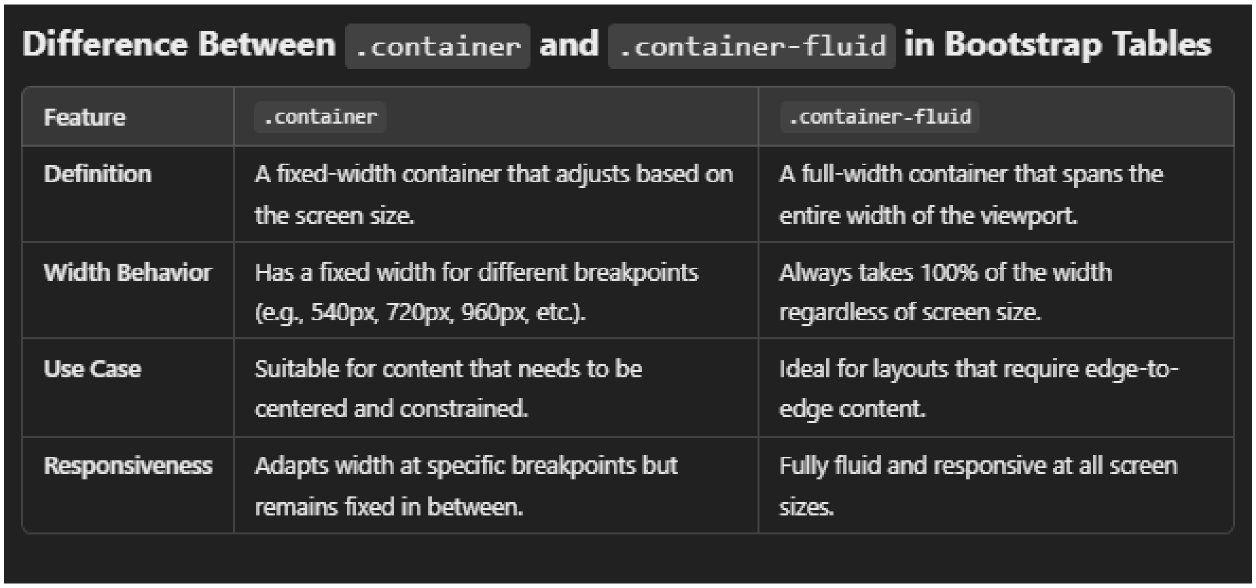
(NOT IN SYLLABUS-BUT IT SAYS ADVANCE PHP HENCE INCLUDED)include and require in PHP

- **include**:
  - Includes and evaluates a specified file.
  - If the file is not found, it will issue a **warning**, but the script will continue.
  - Use when the file is not crucial for the application to work.
- **require**:
  - Includes and evaluates a specified file.
  - If the file is not found, it will issue a **fatal error** and stop script execution.
  - Use when the file is essential for the application to function.
- **Error Handling**:

- `include`: Warning (script continues).
- `require`: Fatal error (script stops).

Unit 4 - Bootstrap

Container vs Container



Difference Between `.container` and `.container-fluid` in Bootstrap Tables

Feature	`.container`	`.container-fluid`
Definition	A fixed-width container that adjusts based on the screen size.	A full-width container that spans the entire width of the viewport.
Width Behavior	Has a fixed width for different breakpoints (e.g., 540px, 720px, 960px, etc.).	Always takes 100% of the width regardless of screen size.
Use Case	Suitable for content that needs to be centered and constrained.	Ideal for layouts that require edge-to-edge content.
Responsiveness	Adapts width at specific breakpoints but remains fixed in between.	Fully fluid and responsive at all screen sizes.

Example Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Bootstrap Containers</title>
  <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body>
```

```
<!-- Fixed-width container -->
<div class="container">
  <h2>Container Example</h2>
  <p>This container has a fixed width, depending on the screen size.</p>
</div>

<!-- Full-width container -->
<div class="container-fluid">
  <h2>Container Fluid Example</h2>
  <p>This container spans the entire width of the viewport.</p>
</div>

</body>
</html>
```

Bootstrap Grid System

Bootstrap’s grid system allows for responsive web design, dividing the page into 12 columns. It adapts to different screen sizes using a series of containers, rows, and columns.

Uses:

- **Responsive Design**: Automatically adjusts content layout based on screen size.
- **Layout Creation**: Allows the design of complex layouts with flexibility.
- **Alignment**: Aligns elements within columns and rows with ease.

Grid Structure:

1. **Container**: Holds the grid and controls the layout width.
2. **Row**: Defines a horizontal group of columns.
3. **Column**: The content area inside the row, which can be divided into up to 12 columns.

Grid Breakpoints:

- `.col-`: Mobile (extra small devices).
- `.col-sm-`: Small devices (≥576px).

- `.col-md-`: Medium devices (≥768px).
- `.col-lg-`: Large devices (≥992px).
- `.col-xl-`: Extra large devices (≥1200px).

Example Code:

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width,
initial-scale=1.0">

  <title>Bootstrap Grid Example</title>

  <link
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap
.min.css" rel="stylesheet">

</head>

<body>

  <div class="container">

    <div class="row">

      <div class="col-12 col-md-4">

        <div class="p-3 border bg-light">Column 1</div>

      </div>

      <div class="col-12 col-md-4">

        <div class="p-3 border bg-light">Column 2</div>

      </div>

      <div class="col-12 col-md-4">
```

```
        <div class="p-3 border bg-light">Column 3</div>

      </div>

    </div>

  </div>



  <script
src="https://code.jquery.com/jquery-3.3.1.slim.min.js"></script>

  <script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/poppe
r.min.js"></script>

  <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.m
in.js"></script>

</body>

</html>
```

- The `container` centers the content.
- The `row` holds the columns.
- The `col-12` specifies that each column will take up full width on smaller screens.
- The `col-md-4` ensures each column takes up one-third of the row on medium and larger screens.

Bootstrap Tags

Uses:

- **UI Elements**: Enhance user interfaces with pre-designed components.
- **Responsive**: Adapts to different screen sizes for mobile-friendly design.
- **Simplify Styling**: Provides ready-to-use styles, reducing custom CSS.

Common Bootstrap Tags:

1. **`<button>`**: For clickable buttons.
2. **`<nav>`**: Defines a navigation bar.
3. **`<form>`**: Used to create forms.
4. **`<input>`**: For form inputs like text fields, checkboxes, etc.
5. **`<a>`**: Used for links with predefined styles.
6. **`<div>`**: For creating flexible containers or sections.
7. **`<span>`**: Inline container for styling small content like labels.
8. **`<img>`**: Used for responsive images with `.img-fluid` class.
9. **`<table>`**: For creating responsive and styled tables.
10. **`<ul>` and `<ol>`**: For unordered and ordered lists, often styled with `.list-group`.

Example Code:

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width,
initial-scale=1.0">

  <title>Bootstrap Tags Example</title>
  <link
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap
.min.css" rel="stylesheet">

</head>

<body>


  <!-- Button Example -->

  <button class="btn btn-primary">Primary Button</button>


  <!-- Navigation Bar -->

  <nav class="navbar navbar-expand-lg navbar-light bg-light">

    <a class="navbar-brand" href="#">Navbar</a>

    <button class="navbar-toggler" type="button"
data-toggle="collapse" data-target="#navbarNav"
aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle
navigation">

      <span class="navbar-toggler-icon"></span>

    </button>

    <div class="collapse navbar-collapse" id="navbarNav">

      <ul class="navbar-nav">

        <li class="nav-item active">

          <a class="nav-link" href="#">Home</a>

        </li>

        <li class="nav-item">

          <a class="nav-link" href="#">Features</a>
        </li>

        <li class="nav-item">

          <a class="nav-link" href="#">Pricing</a>

        </li>

      </ul>

    </div>

  </nav>


  <!-- Image Example -->

  <img src="image.jpg" class="img-fluid" alt="Responsive Image">


  <!-- Table Example -->

  <table class="table">

    <thead>

      <tr>

        <th scope="col">#</th>

        <th scope="col">Name</th>

        <th scope="col">Age</th>

      </tr>

    </thead>

    <tbody>

      <tr>

        <th scope="row">1</th>
```

```
        <td>John Doe</td>

        <td>30</td>

      </tr>

      <tr>

        <th scope="row">2</th>

        <td>Jane Smith</td>

        <td>25</td>

      </tr>

    </tbody>

  </table>



  <script
src="https://code.jquery.com/jquery-3.3.1.slim.min.js"></script>

  <script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/poppe
r.min.js"></script>

  <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.m
in.js"></script>



</body>

</html>
```

Explanation:

- **Button**: `class="btn btn-primary"` applies the Bootstrap button style.
- **Navigation Bar**: `navbar`, `navbar-light`, `bg-light` classes style the navigation bar.
- **Image**: `class="img-fluid"` makes the image responsive.
- **Table**: `class="table"` adds Bootstrap's table styling.

Why Bootstrap is Used:

1. **Responsive Design**:
   - Automatically adjusts layouts to different screen sizes (mobile, tablet, desktop).
2. **Pre-built Components**:
   - Provides ready-to-use components like buttons, navbars, forms, modals, etc., for faster development.
3. **Consistency**:
   - Ensures uniform design across different devices and browsers.
4. **Customization**:
   - Easily customizable via CSS variables and themes for tailored designs.
5. **Cross-browser Compatibility**:
   - Works well across all major browsers (Chrome, Firefox, Safari, Edge).
6. **Grid System**:
   - Simplifies creating complex, responsive grid layouts with a 12-column structure.
7. **Faster Development**:
   - Saves time by providing a framework with pre-designed styles, reducing the need to code from scratch.

8. **Mobile-first Approach**:
   - Built with a mobile-first design philosophy, ensuring compatibility with mobile devices.
9. **Support and Documentation**:
   - Comprehensive documentation and large community support make it easy for developers to implement.
10. **Flexibility**:
    - Works seamlessly with both traditional websites and single-page applications (SPAs).
11. **Open-source**:
    - Free to use, with an active community continuously improving and updating the framework.
12. **JavaScript Plugins**:
    - Includes built-in JavaScript plugins for additional functionality (e.g., tooltips, carousels, modals).

What is AngularJS?

- JavaScript framework developed by Google
- build dynamic web application
- allows developers to create single-page applications (SPAs) by extending HTML with declarative syntax and two-way data binding.
- simplifies development of complex applications - providing tools for routing, templating, form validation, and more.

Features of AngularJS:

1. **Two-way Data Binding**: Automatically syncs data between the model and the view.
2. **Directives**: Extend HTML capabilities with custom attributes and tags.

3. **Dependency Injection**: Makes it easier to manage services and their dependencies.
4. **Templates**: Uses declarative syntax to render dynamic content.
5. **Routing**: Supports routing for navigation in single-page applications (SPA).
6. **MVC Architecture**: Follows the Model-View-Controller pattern for separation of concerns.

Filters in AngularJS

Filters are used to format data before displaying it to the user. Filters are applied in the view (HTML) and can modify the data (like formatting text, dates, or numbers) to suit the display requirements.

Common Filters in AngularJS:

- `currency`: Formats a number as a currency.
- `date`: Formats a date.
- `filter`: Filters an array based on a condition.
- `uppercase`: Converts text to uppercase.
- `lowercase`: Converts text to lowercase.
- `limitTo`: Limits the number of items to be displayed in an array.

Example Code to Convert String to Uppercase using AngularJS Filter

```
<!DOCTYPE html>

<html ng-app="myApp">

<head>

  <title>AngularJS Filters Example</title>

  <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min
.js"></script>
```

```
  <script>
    // Define an AngularJS module and controller
    var app = angular.module('myApp', []);


    app.controller('myCtrl', function($scope) {
      $scope.text = "hello world!";
    });
  </script>
</head>
<body>


<div ng-controller="myCtrl">
  <h1>Original Text: {{ text }}</h1>
  <h2>Uppercase Text: {{ text | uppercase }}</h2>
</div>


</body>
</html>
```

- **`ng-app="myApp"`**: Initializes the AngularJS application.
- **`ng-controller="myCtrl"`**: Links the controller to the view.
- **`{{ text | uppercase }}`**: The **uppercase** filter is applied to convert the text to uppercase before displaying it.
- The original text is `"hello world!"`, and it is converted to `"HELLO WORLD!"` using the filter.

How Filters Work:

- Filters can be applied directly in the view using the pipe `|` symbol.
- The `uppercase` filter takes the `text` model, converts it to uppercase, and displays it.

Events in angular

Here are key events in Angular with short explanations and example code:

1. Click Event

Triggered when a user clicks an element.

```
<button (click)="onClick()">Click Me</button>



onClick() {

  console.log('Button clicked!');

}
```

2. Change Event

Triggered when an input field value changes.

```
<input type="text" (change)="onChange($event)">

onChange(event: any) {

  console.log('Value changed:', event.target.value);

}
```

3. Input Event

Triggered when user types in an input field (key press).

```
<input type="text" (input)="onInput($event)">

onInput(event: any) {

  console.log('Input value:', event.target.value);

}
```

4. Mouseover Event

Triggered when the mouse is over an element.

```
<div (mouseover)="onMouseOver()">Hover over me</div>

onMouseOver() {

  console.log('Mouse over the element');

}
```

5. Keyup Event

Triggered when the user releases a key.

```
<input type="text" (keyup)="onKeyUp($event)">

onKeyUp(event: KeyboardEvent) {

  console.log('Key pressed:', event.key);

}
```

6. Keydown Event

Triggered when a key is pressed down.

```
<input type="text" (keydown)="onKeyDown($event)">

onKeyDown(event: KeyboardEvent) {

  console.log('Key down:', event.key);

}
```

7. Focus Event

Triggered when an input element gains focus.

```
<input type="text" (focus)="onFocus()">


onFocus() {

  console.log('Input focused');

}
```

8. Blur Event

Triggered when an input element loses focus.

```
<input type="text" (blur)="onBlur()">


onBlur() {

  console.log('Input blurred');
}
```

9. Submit Event

Triggered when a form is submitted.

```
<form (submit)="onSubmit($event)">

  <button type="submit">Submit</button>

</form>


onSubmit(event: Event) {

  event.preventDefault();

  console.log('Form submitted');

}
```