HTML Overview: The Fundamentals

HTML (**HyperText Markup Language**) is the standard language for creating web pages. It structures content on the web using elements and tags.

1. Basic Structure of an HTML Document

Every HTML page follows a general structure:

```
html
```

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-
scale=1.0">
<title>My First Web Page</title>
 </head>
<body>
 <h1>Welcome to My Web Page</h1>
 <p>This is a simple HTML document.</p>
</body>
</html>
```

Explanation:

- <!DOCTYPE html> → Declares the document as HTML5.
- <html> → Root element that wraps all content.
- <head> → Contains metadata like character encoding, page title, and responsive settings.
- <body> → Holds the visible content of the webpage.

2. Essential HTML Elements

Text Formatting Elements

```
html
```

```
<p>This is a paragraph.</p>
<b>Bold Text</b>
<i>Italic Text</i>
<u>Underlined Text</u>
<strong>Important Text</strong>
<em>Emphasized Text</em>
```

Headings (H1-H6)

```
html
```

```
<h1>Main Heading</h1> <h2>Subheading</h2> <h3>Smaller Heading</h3>
```

3. Lists in HTML

Unordered List (Bullet Points)

```html
<ul> <li>Apple</li> <li>Banana</li> <li>Cherry</li> </ul>
```

Ordered List (Numbered)

```html
<ol> <li>Step 1</li> <li>Step 2</li> <li>Step 3</li> </ol>
```

4. Links & Images

Hyperlinks

```html
<a href="https://www.example.com">Visit Example</a>
```

- href → Specifies the link URL.

Images

```html
<img src="image.jpg" alt="A beautiful scenery">
```

- src → Image path or URL.
- alt → Alternative text for accessibility.

5. Tables in HTML

```html
<table border="1">
<tr>
<th>Name</th>
<th>Age</th>
</tr>
<tr>
<td>John</td>
<td>25</td>
</tr>
</table>
```

- <table> → Defines the table.
- <tr> → Table row.
- <th> → Table header.

- <td> → Table data cell.

6. Forms & Input Fields

```html
<form action="/submit" method="POST">
<label for="name">Name:</label>
<input type="text" id="name" name="name">
<label for="email">Email:</label>
<input type="email" id="email" name="email">
<input type="submit" value="Submit">
</form>
```

- <form> → Collects user input.
- <input> → Creates different input fields.
- type="submit" → Creates a submit button.

7. Semantic HTML (For Better SEO & Accessibility)

```html
<header>
<h1>Site Title</h1>
</header>
<nav>
<ul>
<li><a href="#">Home</a></li>
<li><a href="#">About</a></li>
</ul>
</nav>
<main>
<article>
<h2>Article Title</h2> <p>Article content...</p>
</article>
</main>
<footer>
<p>&copy; 2025 My Website</p>
</footer>
```

- <header> → Represents the top section.
- <nav> → Navigation bar.
- <main> → Main content area.
- <article> → Independent content section.
- <footer> → Bottom section of the page.

8. Embedding Multimedia

Video

```html
<video controls width="500">
<source src="video.mp4" type="video/mp4">
Your browser does not support the video tag.
</video>
```

Audio

```html
<audio controls>
<source src="audio.mp3" type="audio/mpeg">
Your browser does not support the audio element.
</audio>
```

Conclusion

HTML is the backbone of web development. By combining it with **CSS** (for styling) and **JavaScript** (for interactivity), you can create fully functional websites.

HTML5 Fundamentals: The Modern Web Standard

HTML5 is the latest version of **HTML (HyperText Markup Language)**, designed to improve web structure, multimedia support, and accessibility. It introduces **new elements, attributes, APIs, and better support for modern web applications**.

1. Basic HTML5 Document Structure

HTML5 has a simpler and cleaner structure compared to older versions.

```html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-
scale=1.0">
<title>HTML5 Web Page</title>
</head>
<body>
<header>
<h1>Welcome to HTML5</h1>
</header>
```

```html
<section>
<p>This is a section in HTML5.</p>
</section>
<footer>
<p>&copy; 2025 My Website</p>
</footer>
</body>
</html>
```

Key Features of HTML5

- <!DOCTYPE html> → Declares the document as **HTML5**.
- <meta charset="UTF-8"> → Ensures proper character encoding.
- <meta name="viewport" content="width=device-width, initial-scale=1.0"> → Responsive design support.
- **New semantic elements** (e.g., <header>, <section>, <footer>).

2. New Semantic Elements in HTML5

HTML5 introduces **semantic tags** that improve readability and SEO.

```html
<header>Website Header</header>
<nav>Navigation Menu</nav>
<article>Blog Post</article>
<section>Page Section</section>
<aside>Sidebar Content</aside>
<footer>Website Footer</footer>
```

Benefits:

- ✅ **Better SEO**
- ✅ **Improved Accessibility**
- ✅ **More structured and meaningful code**

3. Forms & Input Types in HTML5

HTML5 introduces new input types for better form validation and user experience.

```html
<form action="/submit" method="POST">
<label for="name">Name:</label>
<input type="text" id="name" name="name" required>
<label for="email">Email:</label>
<input type="email" id="email" name="email" required>
```

```html
<label for="dob">Date of Birth:</label>
<input type="date" id="dob" name="dob">
<label for="range">Select Range:</label>
<input type="range" id="range" name="range" min="1" max="100">
<input type="submit" value="Submit">
</form>
```

New Input Types:

- type="email" → Validates email format.
- type="date" → Selects a date from a calendar.
- type="range" → Selects a value within a range.
- type="search", type="url", type="tel", etc.

4. Audio & Video Support in HTML5
Adding Videos

```html
<video controls width="500">
<source src="video.mp4" type="video/mp4"> Your browser does not
support the video tag. </video>
```

Adding Audio

```
html
<audio controls>
<source src="audio.mp3" type="audio/mpeg">
Your browser does not support the audio element.
</audio>
```

Why HTML5 Media?

✓ No need for external plugins (e.g., Flash).

✓ Native support in modern browsers.

5. Canvas for Graphics & Animations

The <canvas> element allows drawing graphics using JavaScript.

```
html
<canvas id="myCanvas" width="500" height="300"></canvas>
<script>
let canvas = document.getElementById("myCanvas");
let ctx = canvas.getContext("2d");
ctx.fillStyle = "blue";
ctx.fillRect(50, 50, 200, 100);
</script>
```

html

CopyEdit

✓ Used for **games, charts, and dynamic graphics**.

6. Local Storage & Session Storage (Replaces Cookies)

HTML5 introduces localStorage and sessionStorage to store data in the browser.

```
html
<script>
localStorage.setItem("username", "JohnDoe");
alert(localStorage.getItem("username"));
</script>
```

✓ **localStorage** → Persistent storage (even after closing the browser).

✓ **sessionStorage** → Data is removed when the session ends.

7. Geolocation API (Get User Location)
HTML5 allows websites to get a user's location (with permission).

```
html
<button onclick="getLocation()">Get Location</button>
<p id="location"></p>
<script>
function getLocation() {
if (navigator.geolocation) {
 navigator.geolocation.getCurrentPosition(showPosition); }
}
function showPosition(position) {
 document.getElementById("location").innerText = "Latitude: " +
position.coords.latitude + " Longitude: " +
position.coords.longitude;
}
</script>
```

✓ Used for **maps, weather apps, and location-based services**.

8. Responsive Web Design with HTML5
HTML5 promotes **responsive design** with the <meta viewport> tag.

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

Combine with **CSS3 media queries** for a fully responsive website.

Conclusion
HTML5 is **faster, cleaner, and more powerful** than older versions. It enables **better multimedia, semantic structure, and API support** for modern web development.
Next Steps

CSS Overview & Fundamentals

CSS (Cascading Style Sheets) is used to style HTML elements and control the layout of web pages. It enhances the appearance of websites by allowing developers to apply colors, fonts, spacing, animations, and responsiveness.

1. CSS Syntax

A CSS rule consists of a **selector** and a **declaration block**:

```
css
selector { property: value; }
```

Example:

```
css
p { color: blue; font-size: 16px; }
```

Here, p (paragraph) is the selector, and the properties color and font-size define its appearance.

2. Types of CSS

There are three main ways to apply CSS:

a) Inline CSS

Defined directly within an HTML element using the style attribute.

```
html
<p style="color: red;">Hello World</p>
```

b) Internal CSS

Written inside a <style> tag within the <head> section of an HTML file.

```
html
<style> p { color: green; } </style>
```

c) External CSS

Stored in a separate .css file and linked using <link>.

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

styles.css

```
css
p { color: purple; }
```

3. CSS Selectors

Selectors define which elements a style applies to. Common types include:

- **Universal Selector** (*) – Applies to all elements.

```
sss
```

```
* { margin: 0; padding: 0; }
```

- **Element Selector** – Targets specific HTML elements.

```
css
```

```
h1 { color: blue; }
```

- **Class Selector** (.) – Targets elements with a specific class.

```
css
```

```
.btn { background-color: black; }
```

- **ID Selector** (#) – Targets a unique element with an ID.

```
css
```

```
#header { font-size: 24px; }
```

- **Group Selector** (,) – Applies styles to multiple elements.

```
css
```

```
h1, p { font-family: Arial; }
```

- **Child Selector** (>) – Targets direct children of an element.

```
css
```

```
div > p { color: brown; }
```

- **Descendant Selector** (space) – Targets nested elements.

```
js
```

```
div p { color: orange; }
```

4. Box Model

The box model defines how elements are structured with:

- **Content** – The actual text or image.

- **Padding** – Space inside the border.
- **Border** – The outline around the element.
- **Margin** – Space outside the element.

Example:

```
css
.box { width: 200px; padding: 20px; border: 5px solid black; margin: 10px; }
```

5. Positioning & Layout

CSS provides different ways to position elements:

- **Static** (default) – Normal document flow.
- **Relative** – Positioned relative to itself.
- **Absolute** – Positioned relative to the nearest positioned ancestor.
- **Fixed** – Stays in place even when scrolling.
- **Sticky** – Sticks when scrolling past a certain point.

Example:

```
css
.fixed-box { position: fixed; top: 0; left: 0; width: 100%; background: red; }
```

6. Flexbox & Grid (Modern Layouts)

a) Flexbox (1D Layout – row or column)

```
css
.container { display: flex; justify-content: center; align-items: center; }
```

b) Grid (2D Layout – rows & columns)

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

7. Media Queries (Responsive Design)

Used to make websites adaptable to different screen sizes.

```
css
@media (max-width: 768px) { body { background-color: lightgray; } }
```

8. Animations & Transitions

a) Transitions

```js
button { background: blue; transition: background 0.3s ease; }
button:hover { background: red; }
```

b) Animations

```css
@keyframes bounce {
0% { transform: translateY(0); }
50% { transform: translateY(-20px); }
100% { transform: translateY(0); }
}
.box { animation: bounce 2s infinite; }
```

9. CSS Variables

Variables allow reusable values.

```js
:root { --primary-color: blue; } h1 { color: var(--primary-color); }
```

Conclusion

CSS is a powerful tool for styling web pages, controlling layouts, and creating interactive experiences. Mastering **selectors**, **positioning, flexbox/grid, and responsive design** is key to becoming a great UI/UX designer.

Introduction to JavaScript

1. What is JavaScript?

JavaScript is a high-level, dynamic programming language that is primarily used for creating interactive and dynamic content on web pages. Initially, it was used to add simple interactivity like form validation or animations. However, with the evolution of the language, it now powers entire web applications, from simple websites to complex, server-side applications.

JavaScript is client-side, meaning it runs in the user's browser, although it can also be used on the server-side (with platforms like Node.js). It is one of the core technologies for building web applications, alongside HTML and CSS.

Key features of JavaScript:

1. Event-driven: Reacts to user inputs such as clicks or keyboard presses.
2. Dynamic typing: Variables don't need explicit types.
3. Object-oriented and functional programming support.
4. Asynchronous programming with Promises and async/await.

2. How JavaScript Fits in Web Development

JavaScript plays a vital role in modern web development, and it complements HTML (HyperText Markup Language) and CSS (Cascading Style Sheets).

- HTML defines the structure/content of the webpage (headings, paragraphs, images, etc.).

```
html
```

```
<h1>Welcome to my website!</h1>
<p>This is a paragraph.</p>
```

- CSS styles the webpage and controls the layout (colors, fonts, spacing, positioning).

```
css
```

```
h1 {
  color: blue;
  font-size: 2em;
}
p {
  color: gray;
}
```

- JavaScript adds interactivity to the webpage, such as:
    - Responding to user input (clicks, typing, etc.)

- Manipulating HTML elements (adding/removing content)
- Making asynchronous requests (e.g., fetching data from a server)
- Animations or transitions

Example of how JS fits in:

- HTML provides the structure (e.g., a button).
- CSS styles the button.
- JavaScript adds functionality, such as showing an alert when the button is clicked.

```
html
```

```
<button id="clickMe">Click Me!</button>

<script>
  document.getElementById('clickMe').addEventListener('click',
function() {
    alert('Button was clicked!');
  });
</script>
```

3. Setting Up the Environment (Browser, VSCode)

To start writing and testing JavaScript, you need a text editor and a web browser.

a. Using the Browser

- Every modern web browser (Google Chrome, Mozilla Firefox, Microsoft Edge, etc.) has a JavaScript engine that executes your JavaScript code.
- To test your code, all you need to do is open the browser, open a web page (or create a simple HTML file with embedded JavaScript), and view it.

b. Setting Up VSCode (Visual Studio Code)

- VSCode is a free, open-source code editor that is very popular for web development. It has extensions and features that make writing JavaScript much easier (syntax highlighting, autocompletion, debugging tools, etc.).

Steps to set up:

- Download and Install VSCode: Go to the VSCode website and download the installer for your OS.
- Install Extensions: For JavaScript, you can install extensions like ESLint, Prettier, and Debugger for Chrome to make your development environment more efficient.
- Write JavaScript: Create a `.html` or `.jsl` file in VSCode and start coding.

Example: Create a simple index.html and app.js file.

- index.html:

```
html
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>JavaScript Demo</title>
  </head>
  <body>
    <h1>JavaScript Basics</h1>
    <button id="greetBtn">Greet Me!</button>
    <script src="app.js"></script>
  </body>
</html>
```

- app.js:

```
js
```

```
document.getElementById('greetBtn').addEventListener('click',
function() {
  alert('Hello, welcome to JavaScript!');
});
```

Now open index.html in the browser to test your JavaScript.

4. Using the Browser Console

The browser's developer tools have a built-in JavaScript console that allows you to write and test JavaScript interactively. This is a powerful tool for debugging and testing code quickly.

Steps to open the browser console:

Google Chrome: Press Ctrl + Shift + J (Windows/Linux) or Cmd + Option + J (Mac).
Mozilla Firefox: Press Ctrl + Shift + K (Windows/Linux) or Cmd + Option + K (Mac).
Microsoft Edge: Press F12 or Ctrl + Shift + I, then go to the Console tab.
Once the console is open, you can:

Directly type JavaScript commands and see the result immediately.

```
js
```

```
console.log('Hello from the console!');
```

Run any JavaScript code directly without having to create an HTML page.
Console Methods:

```
js
```

```
console.log(): Outputs information to the console.
console.error(): Outputs error messages.
console.warn(): Outputs warnings.
console.table(): Displays tabular data in a neat format.
```

Example:

```
js
```

```
let name = "John";
let age = 25;
console.log(`Name: ${name}, Age: ${age}`);
```

By using the console, you can test small snippets of code quickly before integrating them into your actual project.

This introduction provides a basic understanding of what JavaScript is, how it fits into web development with HTML and CSS, how to set up a development environment using VSCode, and how to use the browser console for testing and debugging. This foundation will help you move forward in learning JavaScript and creating interactive web applications!

Basics: Syntax and Operators in JavaScript

1. Variables (let, const, var)
Variables are used to store data values. JavaScript allows you to declare variables using let, const, or var.

let: Used for variables that can be reassigned.

```
js
```

```
let age = 25;
age = 26; // Reassigned successfully
```

const: Used for variables whose value cannot be reassigned.

```
js
```

```
const name = "John";
// name = "Doe"; // Error: Assignment to constant variable.
```

var: Older way of declaring variables. It has function scope and is not commonly used in modern JavaScript (prefer let and const).

```
js
```

```
var city = "London";
city = "Paris"; // Reassigned successfully
```

Best Practice: Prefer let and const over var due to better scoping behavior.

2. Data Types

JavaScript has several built-in data types, and they can be categorized into primitive and non-primitive types.

Primitive Data Types:

These are immutable and contain a single value.

String: Represents a sequence of characters (text).

```
js
```

```
let name = "Alice";
```

Number: Represents both integer and floating-point numbers.

```
js
```

```
let age = 30;  // Integer
let price = 19.99;  // Floating-point number
```

Boolean: Represents true or false values.

```
js
```

```
let isActive = true;
let isClosed = false;
```

Null: Represents a deliberate non-value (an empty or undefined value).

```
js
```

```
let user = null;
```

Undefined: A variable that has been declared but not assigned any value.

```
js
```

```
let location;
```

```
js
console.log(location);  // Output: undefined
```

Symbol (ES6): Represents a unique and immutable identifier.

```
js
const id = Symbol('id');
```

BigInt (ES11): Allows working with large integers beyond the range of Number.

```
js
const largeNumber = BigInt(123456789012345678901234567890);
```

Non-Primitive Data Types:

These are mutable and can hold multiple values or objects.

Object: Represents a collection of key-value pairs (properties).

```
js
let person = {
  name: "John",
  age: 30
};
```

Array: An ordered collection of values.

```
js
let numbers = [1, 2, 3, 4, 5];
```

3. Operators

Operators perform various operations on values and variables. Here are the different types of operators in JavaScript:

Arithmetic Operators

These operators are used to perform mathematical calculations.

+ (Addition): Adds two values.

```
js
let sum = 10 + 5;  // 15
```

- (Subtraction): Subtracts one value from another.

```js
let difference = 10 - 5;  // 5
```

*** (Multiplication): Multiplies two values.**

```js
let product = 10 * 5;  // 50
```

/ (Division): Divides one value by another.

```js
let quotient = 10 / 5;  // 2
```

% (Modulus): Returns the remainder when dividing one value by another.

```js
let remainder = 10 % 3;  // 1
```

++ (Increment): Increases a number by 1.

```js
let count = 5;
count++;  // 6
```

-- (Decrement): Decreases a number by 1.

```js
let count = 5;
count--;  // 4
```

Assignment Operators

These operators are used to assign values to variables.

= (Simple assignment): Assigns a value to a variable.

```js
let x = 10;
```

+= (Addition assignment): Adds a value to a variable.

```js
let x = 10;
x += 5;  // x = 15
```

-= (Subtraction assignment): Subtracts a value from a variable.

```js
let x = 10;
x -= 3;  // x = 7
```

***= (Multiplication assignment): Multiplies a variable by a value.**

```js
let x = 10;
x *= 2;  // x = 20
```

/= (Division assignment): Divides a variable by a value.

```js
let x = 10;
x /= 2;  // x = 5
```

%= (Modulus assignment): Divides a variable by a value and assigns the remainder.

```js
let x = 10;
x %= 3;  // x = 1
```

Comparison Operators

These operators are used to compare values.

== (Equality): Compares two values for equality, ignoring data types.

```js
5 == "5";  // true
```

=== (Strict Equality): Compares two values for equality, including data types.

```
js
5 === "5";  // false
```

!= (Inequality): Checks if two values are not equal.

```
js
5 != 6;  // true
```

!== (Strict Inequality): Checks if two values are not equal, considering data types.

```
js
5 !== "5";  // true
```

> (Greater than): Checks if the left value is greater than the right value.

```
js
10 > 5;  // true
```

< (Less than): Checks if the left value is less than the right value.

```
js
5 < 10;  // true
```

>= (Greater than or equal to): Checks if the left value is greater than or equal to the right value.

```
js
10 >= 10;  // true
```

<= (Less than or equal to): Checks if the left value is less than or equal to the right value.

```
js
5 <= 10;  // true
```

Logical Operators

These operators are used to combine multiple conditions.

&& (Logical AND): Returns true if both conditions are true.

```
js
```

```
true && false;  // false
```

|| (Logical OR): Returns true if at least one condition is true.

```
js
```

```
true || false;  // true
```

! (Logical NOT): Reverses the truth value of a condition.

```
js
```

```
!true;  // false
```

JavaScript Control Structures

Control structures help control the flow of execution in a JavaScript program. The two main types are **Conditional Statements and Loops**.

1. Conditional Statements

Conditional statements allow a program to make decisions based on conditions.

A. if Statement

Executes a block of code only if the condition is true.

```
js
```

```
let age = 18;

if (age >= 18) {
  console.log("You are eligible to vote.");
}
```

B. if...else Statement

Executes one block of code if the condition is true, otherwise executes another block.

```
js
```

```
let num = 10;
if (num > 0) {
  console.log("Positive number");
} else {
  console.log("Negative number or zero");
}
```

C. if...else if...else Statement

Checks multiple conditions and executes the corresponding block.

```
js
```

```
let score = 75;

if (score >= 90) {
  console.log("Grade: A");
} else if (score >= 80) {
  console.log("Grade: B");
} else if (score >= 70) {
  console.log("Grade: C");
} else {
  console.log("Grade: F");
}
```

D. Ternary Operator (? :)

A shorter way to write if...else.

```
js
```

```
let isLoggedIn = true;
console.log(isLoggedIn ? "Welcome back!" : "Please log in.");
```

E. switch Statement

Used when multiple conditions depend on a single value.

```
js
```

```
let day = 3;

switch (day) {
  case 1:
```

```
js
```

```
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  default:
    console.log("Invalid day");
}
```

Note: Always use break; to prevent fall-through.

2. Loops

Loops help execute a block of code multiple times.

A. for Loop

Used when the number of iterations is known.

```
js
```

```
for (let i = 1; i <= 5; i++) {
  console.log("Iteration:", i);
}
```

B. while Loop

Executes as long as a condition is true.

```
js
```

```
let count = 1;

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

C. do...while Loop

Similar to while, but executes at least once before checking the condition.

```
js
```

```
let num = 1;

do {
  console.log("Number:", num);
  num++;
} while (num <= 5);
```

3. Loop Control Statements

A. break Statement

Stops the loop immediately.

```
js
```

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) break; // Stops at 3
  console.log(i);
}
```

B. continue Statement

Skips the current iteration and continues to the next.

```
js
```

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) continue; // Skips 3
  console.log(i);
}
```

4. 1. Function Declaration

A function declared using the function keyword. It can be called before its definition due to hoisting.

```
js
```

```
function greet() {
  console.log("Hello, World!");
}

greet(); // Output: Hello, World!
```

Function with Parameters & Return Value

```
js
```

```
function add(a, b) {
  return a + b;
}

let sum = add(5, 3);
console.log(sum); // Output: 8
```

2. Function Expression

A function stored in a variable. It cannot be called before its definition.

```
Js
```

```
const multiply = function (x, y) {
  return x * y;
};

console.log(multiply(4, 2)); // Output: 8
```

3. Arrow Functions (ES6)

A shorter syntax for function expressions.

```
js
```

```
const subtract = (a, b) => a - b;
console.log(subtract(10, 3)); // Output: 7
```

Arrow Function with Multiple Statements

```
js
```

```
const greetUser = (name) => {
  console.log("Hello, " + name);
  console.log("Welcome!");
};

greetUser("Alice");
```

Note: Arrow functions do not have their own this context.

4. IIFE (Immediately Invoked Function Expression)

A function that runs immediately after its declaration.

```js
(function () {
  console.log("This runs immediately!");
})();
```

IIFE with Parameters

```js
(function (name) {
  console.log(`Hello, ${name}!`);
})("John");
```

Summary

Function Type	Syntax	Hoisting
Function Declaration	function name() {}	✅ Yes
Function Expression	const name = function() {}	❌ No
Arrow Function	const name = () => {}	❌ No
IIFE	(function(){})()	✅ Yes

JavaScript Data Structures

Data structures store and organize data efficiently. The main ones in JavaScript are Arrays, Objects, Sets, and Maps.

1. Arrays and Array Methods

An array is an ordered list of values.

Creating an Array

```js
let fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits[0]); // Output: Apple
```

Array Methods

Method	Description	Example
.push(value)	Adds value at the end	fruits.push("Mango");
.pop()	Removes the last element	fruits.pop();
.shift()	Removes the first element	fruits.shift();
.unshift(value)	Adds value at the beginning	fruits.unshift("Grapes");
.map(fn)	Creates a new array by applying fn to each element	nums.map(x => x * 2);
.filter(fn)	Creates a new array with elements that match fn	nums.filter(x => x > 10);
.forEach(fn)	Loops through array elements	fruits.forEach(fruit => console.log(fruit));

Example: Using map and filter

```js
let numbers = [1, 2, 3, 4, 5];

// Multiply each number by 2
let doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6, 8, 10]

// Get numbers greater than 2
let filtered = numbers.filter(num => num > 2);
console.log(filtered); // [3, 4, 5]
```

2. Objects and Object Manipulation

An object is a collection of key-value pairs.

Creating an Object

```js
let person = {
  name: "Alice",
  age: 25,
  isStudent: false
};
```

```
js
```

```
console.log(person.name); // Output: Alice
```

Modifying Objects

```
js
```

```
person.age = 26; // Update property
person.city = "New York"; // Add new property
delete person.isStudent; // Remove property

console.log(person);
```

Looping through an Object

```
js
```

```
for (let key in person) {
  console.log(`${key}: ${person[key]}`);
}
```

3. Sets and Maps

A. Sets (Store Unique Values)

```
js
```

```
let mySet = new Set([1, 2, 3, 3, 4]);

mySet.add(5); // Add value
mySet.delete(2); // Remove value
console.log(mySet.has(3)); // true
console.log(mySet.size); // 4
```

B. Maps (Key-Value Pairs with Any Key Type)

```
js
```

```
let myMap = new Map();
myMap.set("name", "Alice");
myMap.set(1, "Number Key");

console.log(myMap.get("name")); // Alice
```

```
js
```

```
console.log(myMap.has(1)); // true
console.log(myMap.size); // 2
```

JavaScript DOM Manipulation

The Document Object Model (DOM) allows JavaScript to interact with HTML elements dynamically.

1. Selecting Elements

To manipulate elements, we first need to select them.

A. getElementById (Selects a single element by ID)

```
js
```

```
let heading = document.getElementById("title");
console.log(heading.innerText);
```

B. querySelector & querySelectorAll

- querySelector("selector") → Selects the first matching element.
- querySelectorAll("selector") → Selects all matching elements as a NodeList.

```
js
```

```
let firstParagraph = document.querySelector("p");
let allParagraphs = document.querySelectorAll("p");
```

2. Changing Content and Attributes

A. Changing Text Content

```
js
```

```
let text = document.getElementById("message");
text.innerText = "Hello, World!"; // Changes text content
```

B. Changing HTML Content

```
js
```

```
document.getElementById("container").innerHTML = "<h2>New
Content</h2>";
```

C. Modifying Attributes

```
js
```

```
let img = document.querySelector("img");
img.src = "new-image.jpg"; // Change image source
img.alt = "New Image"; // Change alt text
```

D. Modifying CSS Styles

```
js
```

```
let box = document.getElementById("box");
box.style.backgroundColor = "blue";
box.style.fontSize = "20px";
```

3. Event Handling (addEventListener)

Events allow user interaction with the webpage.

A. Click Event

```
js
```

```
let button = document.getElementById("myButton");

button.addEventListener("click", function () {
  alert("Button Clicked!");
});
```

B. Mouse Events (mouseover, mouseout)

```
js
```

```
let div = document.getElementById("hoverBox");

div.addEventListener("mouseover", function () {
  div.style.backgroundColor = "yellow";
});

div.addEventListener("mouseout", function () {
  div.style.backgroundColor = "white";
});
```

C. Keyboard Events (keydown, keyup)

```
js
```

```
document.addEventListener("keydown", function (event) {
  console.log("Key pressed:", event.key);
});
```

4. **Event Handling (addEventListener**

Forms allow user input, and JavaScript can validate and process that input.

A. Getting Input Values

```
js
```

```
let form = document.getElementById("myForm");

form.addEventListener("submit", function (event) {
  event.preventDefault(); // Prevent page refresh

  let name = document.getElementById("name").value;
  console.log("Entered Name:", name);
});
```

B. Validating Form Input

```
js
```

```
let inputField = document.getElementById("email");
inputField.addEventListener("blur", function () {
  if (!inputField.value.includes("@")) {
    alert("Invalid email!");
  }
});
```

Asynchronous JavaScript

JavaScript is single-threaded, meaning it executes one task at a time. However, asynchronous operations allow non-blocking behavior using callbacks, promises, and async/await.

1. Call Stack and Event Loop

The Call Stack manages function execution, while the Event Loop handles asynchronous operations.

Example: Call Stack Execution

```
js
```

```
function first() {
  console.log("First");
}
function second() {
  console.log("Second");
}
first();
second();
console.log("Third");
```

Output:

```
sql
```

```
First
Second
Third
```

Asynchronous Example (Event Loop in Action)

```
js
```

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout Callback");
}, 2000);

console.log("End");
```

Output:

```
pgsql
```

```
Start
End
Timeout Callback (after 2 seconds)
```

The Event Loop moves the setTimeout callback to the stack only after the delay.

2. Callbacks

A callback function is passed as an argument and executed later.

Example: Using a Callback

```Js
function fetchData(callback) {
  setTimeout(() => {
    console.log("Data fetched");
    callback();
  }, 2000);
}

function processData() {
  console.log("Processing Data");
}

fetchData(processData);
```

Output:

```java
Data fetched (after 2 seconds)
Processing Data
Callbacks can lead to Callback Hell if nested too deeply.
```

3. Promises

A Promise represents a future value (either resolved or rejected).

Creating a Promise

```javascript
let fetchData = new Promise((resolve, reject) => {
  setTimeout(() => {
    let success = true;
    if (success) {
      resolve("Data loaded");
    } else {
      reject("Error loading data");
    }
  }, 2000);
});

fetchData
  .then((result) => console.log(result))
  .catch((error) => console.log(error));
Output (after 2 seconds):
Data loaded
```

```javascript
.then() handles success, .catch() handles errors.
```

4. async/await
async/await is a cleaner way to handle asynchronous operations.

Example: Using async/await

```js
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Data received"), 2000);
  });
}
async function getData() {
  console.log("Fetching...");
  let data = await fetchData();
  console.log(data);
}
getData();
```

```
Output
```

```
Fetching...
(2-second delay)
Data received
```

Await makes JavaScript pause execution until the Promise resolves.

Error Handling in JavaScript

Errors can occur due to invalid input, network failures, or unexpected conditions. JavaScript provides mechanisms to catch and handle these errors gracefully.

1. try...catch
The try block runs the code, and if an error occurs, the catch block handles it.
Basic Example

```
js
```

```
try {
  let result = 10 / 0; // No error, but may cause unexpected behavior
  console.log(result);

  let x = y + 5; // ReferenceError (y is not defined)
} catch (error) {
  console.log("An error occurred:", error.message);
}
```

```
Output
```

```
An error occurred: y is not defined
```

The script continues running instead of crashing.

Handling Specific Errors

```
js
```

```
try {
  JSON.parse("{invalidJson}");
} catch (error) {
  if (error instanceof SyntaxError) {
    console.log("Invalid JSON format:", error.message);
  } else {
    console.log("Unexpected error:", error.message);
  }
}
```

Output:

```
pgsql
```

```
Invalid JSON format: Unexpected token i in JSON at position 1
```

This approach ensures different errors are handled appropriately.

2. finally Block

The finally block runs always, whether an error occurs or not.

```
js
```

```
try {
```

```
js
  console.log("Trying...");
  throw new Error("Something went wrong!");
} catch (error) {
  console.log("Caught error:", error.message);
} finally {
  console.log("This always runs.");
}
```

Output:

```
go
Trying...
Caught error: Something went wrong!
```

This always runs.
Use finally for cleanup actions like closing database connections.

3. Throwing Custom Errors
You can create and throw your own errors.

A. Throwing an Error

```
js
function divide(a, b) {
  if (b === 0) {
    throw new Error("Cannot divide by zero");
  }
  return a / b;
}

try {
  console.log(divide(10, 0));
} catch (error) {
  console.log("Error:", error.message);
}
```

Output:

```
js
Error: Cannot divide by zero
```

B. Custom Error Class

```
js
```

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomError";
  }
}

try {
  throw new CustomError("This is a custom error!");
} catch (error) {
  console.log(`${error.name}: ${error.message}`);
}
```

Output:

```
js
```

```
CustomError: This is a custom error!
```

Modern JavaScript (ES6+) Features

ES6+ introduced powerful features that make JavaScript cleaner and more efficient.

1. Template Literals (``)

Template literals allow multi-line strings and string interpolation using backticks ``` `.

Example: String Interpolation

```
js
```

```
let name = "Alice";
let age = 25;

console.log(`Hello, my name is ${name} and I am ${age} years old.`);
```

Example: Multi-line Strings

```
js
```

```
let message = `
This is a multi-line string.
```

```
js
```

```
It spans multiple lines without using \n.
`;

console.log(message);
```

2. Destructuring

Destructuring makes extracting values from arrays and objects easier.

A. Array Destructuring

```
js
```

```
let colors = ["Red", "Green", "Blue"];
let [first, second, third] = colors;

console.log(first); // Red
console.log(second); // Green
console.log(third); // Blue
```

```
js
```

```
let person = { name: "Bob", age: 30, city: "New York" };
let { name, age } = person; // Extracts values
console.log(name); // Bob
console.log(age); // 30
```

C. Default Values

```
js
```

```
let { country = "USA" } = person; // If `country` is missing, use
"USA"
console.log(country); // USA
```

3. Spread and Rest Operators (...)

A. Spread Operator (...) – Expands elements

Example: Copying an Array

```
js
```

```
let arr1 = [1, 2, 3];
let arr2 = [...arr1, 4, 5]; // Creates a new array

console.log(arr2); // [1, 2, 3, 4, 5]
```

Example: Copying an Object

```
js
```

```
let person1 = { name: "Alice", age: 25 };
let person2 = { ...person1, city: "London" };

console.log(person2); // { name: 'Alice', age: 25, city: 'London' }
```

B. Rest Operator (...) – Collects values into an array

Example: Function Arguments

```
js
```

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3, 4)); // 10
```

Example: Extracting Remaining Values

```
js
```

```
let [firstColor, ...restColors] = ["Red", "Green", "Blue", "Yellow"];
console.log(firstColor); // Red
console.log(restColors); // ['Green', 'Blue', 'Yellow']
```

4. Modules (import, export)

Modules allow code to be split into multiple files for better organization.

A. Exporting a Function (export)

math.js

```
js
```

```
export function add(a, b) {
  return a + b;
}
```

B. Importing a Function (import)

app.js

```
js
```

```
import { add } from "./math.js";
console.log(add(5, 3)); // 8
```

C. Default Export

greet.js

```
js
```

```
export default function greet(name) {
  return `Hello, ${name}!`;
}
```

main.js

```
js
```

```
import greet from "./greet.js";

console.log(greet("Alice")); // Hello, Alice!
```


Useful JavaScript APIs

APIs (Application Programming Interfaces) allow JavaScript to interact with external resources and browser features

1. Fetch API (Making HTTP Requests)

The fetch() API allows us to retrieve data from external sources (APIs, databases, etc.).

A. Fetching Data from an API

```
js
```

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(response => response.json()) // Convert to JSON
  .then(data => console.log(data)) // Display data
  .catch(error => console.error("Error fetching data:", error));
```

Output:

```
json
```

```
{
  "userId": 1,
  "id": 1,
  "title": "sample title",
  "body": "sample content"
}
.then() handles the response, .catch() handles errors.
```

B. Using async/await for Fetch Requests

```
js
```

```
async function fetchData() {
  try {
    let response = await
fetch("https://jsonplaceholder.typicode.com/users/1");
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error:", error);
  }
}
fetchData();
```

async/await makes fetch requests cleaner and more readable.

C. Sending Data (POST Request)

```
js
```

```
fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({ title: "New Post", body: "This is a test
post." })
})
  .then(response => response.json())
  .then(data => console.log("Created Post:", data))
  .catch(error => console.error("Error:", error));
```

2. LocalStorage and SessionStorage

The LocalStorage and SessionStorage APIs store data in the browser.

Storage Type	Persists after Refresh	Persists after Closing Browswer
LocalStorage	✅ Yes	✅ Yes
SessionStorage	✅ Yes	❌ No

A. Storing and Retrieving Data in LocalStorage

```js
localStorage.setItem("username", "JohnDoe"); // Store data
console.log(localStorage.getItem("username")); // Retrieve data

localStorage.removeItem("username"); // Remove data
localStorage.clear(); // Clear all data
```

B. Using SessionStorage

```js
sessionStorage.setItem("sessionID", "12345");
console.log(sessionStorage.getItem("sessionID"));

sessionStorage.removeItem("sessionID");
sessionStorage.clear();
```

3. Geolocation API (Getting User Location)

The Geolocation API allows access to the user's location.

A. Getting the User's Current Location

```js
navigator.geolocation.getCurrentPosition(
  position => {
    console.log("Latitude:", position.coords.latitude);
    console.log("Longitude:", position.coords.longitude);
  },
  error => {
    console.error("Error getting location:", error.message);
  }
);
```

If the user denies permission, an error is returned.

B. Watching Position Updates (Live Tracking)

```js
navigator.geolocation.watchPosition(position => {
  console.log("Updated Latitude:", position.coords.latitude);
  console.log("Updated Longitude:", position.coords.longitude);
});
```

Best Practices for Writing Clean and Readable JavaScript Code

1. **Use Meaningful Variable and Function Names**
   - Avoid single-letter variables (except for loops like i, j, k).
   - Use camelCase for variables and functions (userProfile, fetchData).
   - Use PascalCase for classes (UserProfile).
2. **Write Small and Focused Functions**
   - Keep functions short (preferably under 20 lines).
   - Ensure a function does one thing only (Single Responsibility Principle).
3. **Use ES6+ Features**
   - Prefer const and let over var to avoid scope issues.
   - Use template literals instead of string concatenation.
   - Use arrow functions for concise syntax (const add = (a, b) => a + b;).
4. **Avoid Magic Numbers and Hardcoded Strings**
   - Use constants:

```js
const MAX_USERS = 100;
const ERROR_MESSAGE = "Invalid input!";
```

5. **Follow Consistent Formatting and Style Guides**
   - Use Prettier or ESLint to enforce a coding style.
   - Keep indentation consistent (2 or 4 spaces).
6. **Write Modular and Reusable Code**
   - Break down code into modules and import/export where necessary.

```js
import { fetchData } from './api.js';
export function processUser() { ... }
```

7. **Avoid Deeply Nested Code**
   - Use early returns to reduce nesting.

```
js
function process(data) { if (!data) return; // Process data }
```

Best Practices for Debugging JavaScript Code

1. **Use** console.log() **Wisely**
    - Log relevant information to track issues.

```
js
console.log(`User data:`, user);
```

2. **Use the Browser Developer Tools**
    - Chrome DevTools (F12 → Console, Sources, Network tabs).
3. **Use Breakpoints for Step-by-Step Debugging**
    - Open DevTools → Sources → Add breakpoints in JS files.
4. **Use** debugger; **Statement**

```
js
function test() {
debugger;
console.log('Check this point');
}
```

5. **Check for Errors in the Console**
    - Always check for syntax or runtime errors in the console.
6. **Write Unit Tests for Critical Functions**
    - Use Jest or Mocha for automated testing.

```
js
test('adds 1 + 2 to equal 3', () => { expect(add(1, 2)).toBe(3); });
```

Best Practices for Commenting and Documenting Code

1. **Use Comments to Explain "Why", Not "What"**
    - Avoid redundant comments.

```
js
// Bad:
let count = 10; // Assigning 10 to count
```

```
js
```

```
// Good:
// Set default count value for pagination
let count = 10;
```

2. **Use JSDoc for Function Documentation**

```
js
```

```
/**
* Calculates the total price after discount.
* @param {number} price - The original price.
* @param {number} discount - The discount percentage.
* @returns {number} - The final price after discount.
*/
function getDiscountedPrice(price, discount) {
return price - (price * discount) / 100;
}
```

3. **Document API Endpoints and Responses**
    - Use Swagger/OpenAPI for API documentation.
4. **Write a README for Projects**
    - Explain setup, usage, and key features.

Hands-On Projects (Ideas)

To-Do List App (DOM Manipulation)
Weather App (API Fetching)
Simple Calculator (Event Handling)
Quiz App (Control Structures)