Unit 2: JavaScript

- Ankita Kumari

Introduction

JavaScript

1. Is a Client-side scripting language:
    - Executes in the user's browser (not on the server).
    - Used to make web pages dynamic and interactive.
2. Enhances interactivity in web pages
    - Enables features like form validation, animations, real-time updates.
3. Runs in the browser (no compilation needed)
    - Interpreted language (executed line-by-line by the browser).
    - No separate compilation step required.

JavaScript powers modern web development through front-end frameworks like React and Angular and back-end runtime environments like Node.js. Beyond the web, it enables cross-platform mobile apps (React Native, Ionic), desktop applications (Electron.js), and even game development (Phaser, Three.js). Its versatility extends to emerging domains like IoT and embedded systems, where JavaScript runs on microcontrollers and edge devices.

History & Evolution

- Developed by Brendan Eich in 1995 for Netscape Navigator.
- Originally named "Mocha" → "LiveScript" → "JavaScript" (marketing decision due to Java's popularity).
- ECMAScript (ES): The standardized version of JavaScript.

Embedding JavaScript in HTML

A. Internal JavaScript

Using the <script> tag inside HTML:

```
<script>
  alert("Hello, World!"); // Simple pop-up
</script>
```

B. External JavaScript

Linking an external .js file: - Best practice for larger projects (separation of concerns)

```
<script src="script.js"></script>
```

C. Inline Event Handlers

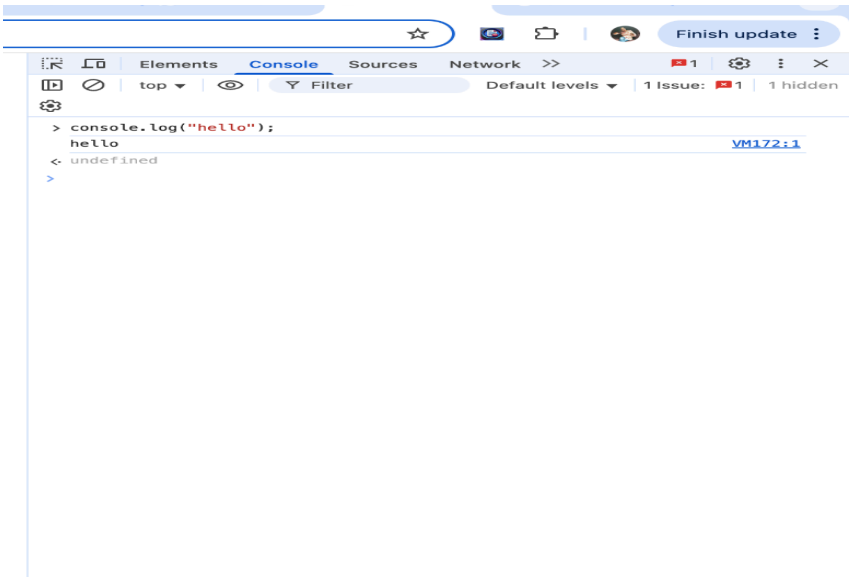
Directly in HTML elements (not recommended for complex logic):

<button onclick="alert('Clicked!')">Click Me</button>

Example

A. Browser Console

To access the JavaScript console in Chrome or Firefox, right-click on any webpage, select 'Inspect', then navigate to the 'Console' tab where you can directly execute commands like console.log('Hello World') for debugging and testing. Output will appear in the console.



B. HTML File

Create index.html file as given below and then open it in a browser to see the result.

```
<!DOCTYPE html>
<html>
<head>
  <title>JS Demo</title>
</head>
<body>
  <script>
    document.write("<h1>Hello, World!</h1>"); // Writes to the page
  </script>
</body>
</html>
```

Note:

Placing the <script> tag for external.js at the end of <body> ensures the JavaScript loads and executes after the HTML is fully parsed. This approach improves performance by allowing

page content to render first (preventing render-blocking) and guarantees DOM elements exist before scripts access them.

JavaScript Syntax & Data Types

Syntax Rules

- JavaScript is **case-sensitive**:

```
let name = "John";
let Name = "Doe";  // Different variable
```

- Statements end with ; (optional but recommended):

```
let x = 5;  // Good practice
let y = 10  // Works but may cause issues
```

- Comments
  - Single-line:

```
// This is a comment
```

  - Multi-line:

```
/*
  This is a
  multi-line comment
*/
```

Variables

Declaration Keywords

1. var
   - Function-scoped, reassignable, and hoisted
   - Legacy usage - avoid in modern code due to unpredictable scoping
2. let
   - Block-scoped and reassignable (not hoisted)
   - Preferred for variables that need reassignment
3. const
   - Block-scoped and immutable (not hoisted)
   - Preferred for constants and values that shouldn't change

Note: Hoisting means that variable and function declarations are moved to the top of their scope during compilation, allowing them to be used before they appear in the code.

Examples:

```
var age = 25;      // Avoid (function-scoped)
let score = 100;   // Block-scoped, can change
const PI = 3.14;   // Block-scoped, cannot change
```

Naming Conventions

- Use **camelCase**: userName, totalCount
- Start with **letter**, _, or $ (cannot start with a number)
- Avoid **reserved words** (class, function, etc.)

Data Types

Primitive Types

Type	Example	Description
number	42, 3.14	Integers & floats
string	"Hello", 'World'	Text (single/double quotes)
boolean	true, false	Logical values
null	null	Intentional empty value
undefined	undefined	Default value of uninitialized variables
symbol	Symbol('id')	Unique identifiers (ES6)
bigint	123n	Large integers (ES2020)

Dynamic Typing

In JavaScript, variables can change types:

```
let x = 5;   // number
x = "hello"; // Now string
```

Type Coercion

Automatic type conversion (can cause bugs):

```
"5" + 2 = "52"   // string concatenation
```

```
"5" - 2 = 3     // numeric subtraction
```

EXAMPLE

```
// Number
let age = 25;
console.log(age, typeof age); // 25 "number"

// String
let name = "Alice";
console.log(name, typeof name); // Alice "string"

// Boolean
let isStudent = true;
console.log(isStudent, typeof isStudent); // true "boolean"

// Null (bug: typeof null returns "object")
let empty = null;
console.log(empty, typeof empty); // null "object"

// Undefined
let x;
console.log(x, typeof x); // undefined "undefined"

// Symbol
let id = Symbol("id");
console.log(id, typeof id); // Symbol(id) "symbol"

// BigInt
let bigNum = 12345678901234567890n;
console.log(bigNum, typeof bigNum); // 123...n "bigint"
```

The typeof operator is used to determine the data type of a variable or expression. It returns a string indicating the type of the operand.

```
const name = "Alex";        // String
const age = 25;             // Number
const isStudent = true;     // Boolean

// Create sentence using template literals (backticks ``)
const bio = `My name is ${name}, I'm ${age} years old. Student: ${isStudent}.`;

console.log(bio);
// Output: "My name is Alex, I'm 25 years old. Student: true."
```

Operators & Expressions

Arithmetic Operators

Perform mathematical operations on numbers (or strings, with type coercion).

Operator	Name	Example	Result
+	Addition	5 + 2	7
-	Subtraction	5 - 2	3
*	Multiplication	5 * 2	10
/	Division	5 / 2	2.5
%	Modulus (Remainder)	5 % 2	1
**	Exponentiation	5 ** 2	25

Assignment Operators

Assign values to variables and modify them.

Operator	Example	Equivalent To	Result (if x = 5)
=	x = 5		x = 5
+=	x += 3	x = x + 3	x = 8
-=	x -= 2	x = x - 2	x = 3
*=	x *= 4	x = x * 4	x = 20
/=	x /= 5	x = x / 5	x = 1
%=	x %= 2	x = x % 2	x = 1

Comparison Operators

Return true or false based on comparisons.

Equality (Loose vs Strict)

Operator	Name	Example	Result (5 vs "5")
==	Loose equality	5 == "5"	true (type coercion)
===	Strict equality	5 === "5"	false (no coercion)
!=	Loose inequality	5 != "5"	false
!==	Strict inequality	5 !== "5"	true

Relational Operators

Operator	Name	Example	Result
>	Greater than	5 > 3	true
<	Less than	5 < 3	false
>=	Greater than or equal	5 >= 5	true
<=	Less than or equal	5 <= 3	false

Logical Operators

Combine or invert boolean values.

Operator	Name	Example	Result
&&	AND	true && false	false
||	OR	true || false	true
!	NOT	!true	false

Ternary Operator

A shorthand for if-else.

condition ? exprIfTrue : exprIfFalse;

Example:

```
let age = 20;
let canVote = (age >= 18) ? "Yes" : "No";
console.log(canVote); // "Yes"
```

Operator Precedence

Determines the order of operations.

1. Parentheses () override all precedence.
2. Logical NOT (!) has higher priority than arithmetic (++, --).
3. Exponentiation (**) > Multiplication/Division > Addition/Subtraction.
4. Comparison (>, ===, etc.) > Logical (&&, ||).
5. Assignment (=, +=) has the lowest priority.

Control Structures – Conditional Statements

if, else if, else Statements

Used to execute different code blocks based on conditions.

Syntax

```
if (condition1) {
  // Runs if condition1 is true
} else if (condition2) {
  // Runs if condition1 is false but condition2 is true
} else {
  // Runs if all conditions are false
}
```

Example

```
let score = 85;

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");
}
// Output: "Grade: B"
```

switch-case Statements

Best for comparing one value against multiple possible cases.

Syntax

```
switch (expression) {
  case value1:
    // Code to run if expression === value1
    break;
  case value2:
    // Code to run if expression === value2
    break;
  default:
    // Code to run if no cases match
}
```

Example

```
let day = 3;
let dayName;

switch (day) {
  case 1:
    dayName = "Monday";
    break;
  case 2:
    dayName = "Tuesday";
    break;
  case 3:
    dayName = "Wednesday";
    break;
  default:
    dayName = "Invalid day";
}
console.log(dayName); // "Wednesday"
```

Note:

JavaScript treats non-boolean values as true or false in conditions.

Falsy Values (Evaluate to false)

- false
- 0, -0, 0n
- "" (empty string)
- null
- undefined
- NaN

Everything else is truthy (e.g., "0", [], {}).

Nested Conditionals

Conditionals inside other conditionals (use sparingly to avoid complexity).

Example: Age & Membership Check

```
let age = 25;
let isMember = true;

if (age >= 18) {
  if (isMember) {
    console.log("Welcome, premium user!");
  } else {
    console.log("Welcome, guest!");
  }
} else {
  console.log("You must be 18+ to enter.");
}
// Output: "Welcome, premium user!"
```

Control Structures – Loops

for **Loop**

Best for known iterations (when you know how many times to loop).

Syntax

```
for (initialization; condition; update) {
  // Code to repeat
}
```

Example

```
for (let i = 1; i <= 5; i++) {
  console.log(i); // 1, 2, 3, 4, 5
}
```

while **Loop**

Best for unknown iterations (loop until a condition is met).

Syntax

```
while (condition) {
  // Code to repeat
}
```

Example

```
let dice = 0;
while (dice !== 6) {
  dice = Math.floor(Math.random() * 6) + 1;
  console.log(`Rolled: ${dice}`);
}
// Keeps rolling until dice === 6
```

do-while **Loop**

Runs at least once, then checks the condition.

Syntax

```
do {
  // Code to repeat
} while (condition);
```

Example

```
let input;
do {
  input = prompt("Enter 'yes' to continue:");
} while (input !== "yes");
// Runs once, then repeats until input is "yes"
```

break **&** continue

- break: Exits the loop immediately.
- continue: Skips to the next iteration.

Example 1

```
for (let i = 1; i <= 10; i++) {
  if (i % 2 === 0) continue; // Skip even nos.
```

```
  console.log(i); // 1, 3, 5, 7, 9
}
```

Example 2

```
let num = 0;
while (true) {
  num += 5;
  if (num > 50) break; // Exit when num > 50
  console.log(num); // 5, 10, ..., 50
}
```

Looping Through Arrays

for **Loop (Traditional)**

```
const fruits = ["Apple", "Banana", "Cherry"];
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]); // Apple, Banana, Cherry
}
```

for-of **Loop (Modern ES6)**

```
for (const fruit of fruits) {
  console.log(fruit); // Apple, Banana, Cherry
}
```

forEach() **Method**

```
fruits.forEach((fruit) => {
  console.log(fruit); // Apple, Banana, Cherry
});
```

Examples

A. Multiplication Table (5x5)

```
for (let i = 1; i <= 5; i++) {
  let row = "";
  for (let j = 1; j <= 5; j++) {
    row += `${i * j}\t`; // \t for tab spacing
  }
  console.log(row);
}
```

```
// Output:
// 1   2   3   4   5
// 2   4   6   8   10
// ...
// 5   10  15  20  25
```

B. Pyramid Pattern

```
let n = 5;
for (let i = 1; i <= n; i++) {
  let spaces = " ".repeat(n - i);
  let stars = "*".repeat(2 * i - 1);
  console.log(spaces + stars);
}
// Output:
//     *
//    ***
//   *****
//  *******
// *********
```

C. Sum Array Elements

```
const nums = [10, 20, 30];
let sum = 0;
for (const num of nums) {
  sum += num;
}
console.log(sum); // 60
```

D. Find First Negative Number

```
const numbers = [5, 3, -2, 8];
for (const num of numbers) {
  if (num < 0) {
    console.log(`First negative: ${num}`); // -2
    break;
  }
}
```

E. Reverse a String

```
let str = "hello";
let reversed = "";
for (let i = str.length - 1; i >= 0; i--) {
  reversed += str[i];
}
console.log(reversed); // "olleh"
```

Functions

Function Declaration

Function declaration are Hoisted (can be called before definition)

```
greet(); // Works even though called before declaration

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

Function Expression

Function Expression are not hoisted (must be defined before calling) and are often assigned to variables:

```
const greet = function() {
  console.log("Hello!");
};
greet(); // Works
```

Parameters & Arguments

- **Parameters**: Variables listed in function definition.
- **Arguments**: Actual values passed to the function.

Example

```
function add(a, b) {  // Parameters: a, b
  return a + b;
}
add(3, 5);  // Arguments: 3, 5 ⟶ Returns 8
```

Rest Parameters (ES6)

Rest parameters allow you to represent an indefinite number of arguments as an array, providing a cleaner and more flexible way to work with multiple function arguments.

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
} // 0 here is initial value of total
sum(1, 2, 3); // 6
```

Return Statement

The return statement specifies the value a function outputs and immediately exits the function, with functions returning *undefined* by default if no return is specified.
For arrow functions without braces, the return is implicit (e.g., const double = x => x * 2), while normal functions require an explicit return. You can use early returns for conditional exits, and returning objects/arrays allows simulating multiple return values.

Example

```
function isEven(num) {
  return num % 2 === 0;
}
console.log(isEven(4)); // true


function noReturn() {
  // No return statement
}
console.log(noReturn()); // undefined
```

Default Parameters (ES6)

Default parameters in ES6 allow you to set fallback values for function arguments when they're missing or undefined, ensuring the function still works with incomplete inputs.

```
function greet(name = "Guest") {
  console.log(`Hello, ${name}!`);
}
greet(); // "Hello, Guest!"
greet("Alice"); // "Hello, Alice!"
```

Anonymous Functions

Anonymous functions are unnamed functions that are commonly used as callbacks or in Immediately Invoked Function Expressions (IIFEs), where they are defined and executed in place.
For example:

```
// As a callback
setTimeout(function() {
  console.log("This runs after 1 second");
```

```
}, 1000);

// As an IIFE
(function() {
  console.log("This runs immediately");
})();
```

They are useful for one-time operations where naming the function isn't necessary.

Arrow Functions (ES6)

Arrow functions provide a shorter syntax for writing functions in ES6 and automatically inherit the this value from their surrounding (lexical) context, unlike regular functions that have their own this binding.
For example:

```
// Concise syntax
const add = (a, b) => a + b;

// Lexical 'this' (inherits from parent scope)
const obj = {
  value: 10,
  getValue: function() {
    setTimeout(() => {
      console.log(this.value); // Correctly logs 10
    }, 100);
  }
};
```

They're ideal for short callbacks and when you need to preserve the surrounding context's *this* value.

Syntax

Scenario	Arrow Function	Equivalent Regular Function
No parameters	() => {...}	function() {...}
One parameter	x => {...}	function(x) {...}
Multiple params	(x, y) => {...}	function(x, y) {...}
Single expression	x => x * 2	function(x) { return x*2 }

Example

```
// Before (regular function)
const numbers = [1, 2, 3];
const squares = numbers.map(function(n) {
  return n * n;
});

// After (arrow function)
const squares = numbers.map(n => n * n);
```

Callback Functions

Callback functions are functions passed as arguments to other functions to be executed later, commonly used in event handling (like addEventListener), array operations (such as map and filter), and asynchronous tasks (like setTimeout or fetch).

Example 1

```
// Event handler callback
button.addEventListener('click', () => {
  console.log('Button clicked!');
});

// Array method callback
const doubled = numbers.map(num => num * 2);

// Asynchronous callback
fetch(url)
  .then(response => response.json());
```

Example 2

```
const nums = [1, 2, 3, 4, 5];
const evens = nums.filter(function(num) {
  return num % 2 === 0;
});

// Arrow function version
const evens = nums.filter(num => num % 2 === 0);
```

Closures

Closures occur when an inner function retains access to variables from its outer (enclosing) function's scope, even after the outer function has finished executing.

Example

```
function outer() {
  const message = "Hello";
  function inner() {
    console.log(message); // Remembers 'message' from outer()
  }
  return inner;
}
const closureFunc = outer();
closureFunc(); // Logs "Hello" (still accesses outer's variable)
```

Closures let functions "remember" variables from where they were created, which helps:

- Keep variables private (like hidden data),
- Create functions with memory (like counters that update),
- Work smoothly with delayed tasks (like timers or API calls).

Example 1

```
function makeCounter() {
  let count = 0; // Hidden by the closure
  return () => count++; // Remembers 'count'
}
const counter = makeCounter();
console.log(counter()); // 0, then 1, 2... (count persists)
```

Example 2

```
function createBankAccount(initialBalance) {
  let balance = initialBalance; // "Private" variable

  return {
    deposit(amount) {
      balance += amount;
    },
    withdraw(amount) {
      if (amount <= balance) balance -= amount;
    },
    getBalance() {
      return balance;
    }
  };
}
const account = createBankAccount(1000);
account.deposit(500);
console.log(account.getBalance()); // 1500
console.log(account.balance); // undefined (can't access directly)
```

Arrays & Array Methods

JavaScript arrays and their built-in methods let you store, organize, and manipulate ordered collections of data efficiently.

Example:

```
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2); // [2, 4, 6]
```

Creating Arrays

```
// Using Array Literal (Most Common)
const fruits = ['Apple', 'Banana', 'Cherry'];

// Using Array Constructor (Rarely Used)
const numbers = new Array(1, 2, 3);
// Special Cases
const empty = [];            // Empty array
const mixed = [1, 'Hello', true]; // Mixed data types
```

Accessing Elements

Operation	Syntax	Example
Get by index	array[index]	fruits[0] ⟶ "Apple"
Get last element	array.at(-1)	fruits.at(-1) ⟶ "Cherry"
Get array length	array.length	fruits.length ⟶ 3

Array Methods

Adding/Removing Elements

Method	Description	Example
push()	Adds items to the **end** of an array.	[1, 2].push(3) // [1, 2, 3]
pop()	Removes the **last item** and returns it.	[1, 2].pop() // 2

shift()	Removes the **first item** and returns it.	[1, 2].shift() // 1
unshift()	Adds items to the **start** of an array.	[1, 2].unshift(0) // [0, 1, 2]
splice()	Adds/removes items at a **specific index**.	[1, 2, 3].splice(1, 1, 99) // [1, 99, 3]

Iteration

Method	Description	Example
forEach()	Executes a function for **each item**.	[1, 2].forEach(num => console.log(num))
map()	Creates a **new array** by transforming each item.	[1, 2].map(num => num * 2) // [2, 4]
filter()	Returns items that pass a **condition**.	[1, 2, 3].filter(num => num > 1) // [2, 3]
reduce()	Combines items into a **single value**.	[1, 2].reduce((sum, num) => sum + num, 0) // 3

Searching

Method	Description	Example
find()	Returns the **first match** (or undefined).	[1, 2, 3].find(num => num > 1) // 2
includes()	Checks if an item exists (returns true/false).	[1, 2].includes(2) // true
indexOf()	Returns the **index** of an item (or -1).	['a', 'b'].indexOf('b') // 1

Transformation

Method	Description	Example

sort()	Sorts items (default: as strings).	[3, 1].sort((a, b) => a - b) // [1, 3]
reverse()	Reverses the array order.	[1, 2].reverse() // [2, 1]
slice()	Copies a **portion** of an array.	[1, 2, 3].slice(1) // [2, 3]
concat()	Merges arrays into a **new array**.	[1].concat([2]) // [1, 2]

Utility

Method	Description	Example
join()	Combines items into a **string** (with a separator).	['a', 'b'].join('-') // "a-b"
every()	Checks if **all items** pass a test.	[1, 2].every(num => num > 0) // true
some()	Checks if **at least one** item passes a test.	[1, -2].some(num => num < 0) // true

Example

Q. Convert this array of strings into an array of objects:

```
const data = ['Alice,25', 'Bob,30', 'Carol,28'];
// Expected: [{ name: 'Alice', age: 25 }, ...]
```

Solution:

```
const users = data.map(str => {
  const [name, age] = str.split(',');
  return { name, age: Number(age) };
});
```

Objects & JSON

JavaScript objects and JSON (JavaScript Object Notation) allow you to store, access, and manipulate structured data as key-value pairs, making them essential for organizing complex information and exchanging data between servers and web applications.

Objects: Store properties/methods (e.g., { name: "Alice", age: 30 }).

JSON: Lightweight data format (e.g., APIs) that converts easily to/from objects using JSON.parse() and JSON.stringify().

Example

```
// Object
const user = { name: "Bob", isAdmin: false };

// Convert to JSON (for APIs)
const json = JSON.stringify(user); // '{"name":"Bob","isAdmin":false}'

// Convert back to object
const parsed = JSON.parse(json); // { name: "Bob", isAdmin: false }
```

Object Literals ({ key: value })

Objects store **key-value pairs** (properties/methods).

Syntax

```
const person = {
  name: "Alice",     // Property
  age: 25,           // Property
  greet() {          // Method
    console.log(`Hello, I'm ${this.name}`);
  }
};
```

Accessing Properties

Dot Notation (Static Keys)

```
console.log(person.name); // "Alice"
person.greet();        // Calls method
```

Bracket Notation (Dynamic Keys)

```
const key = "age";
console.log(person[key]); // 25

// Useful for special characters/spaces:
const obj = { "full name": "Alice Cooper" };
console.log(obj["full name"]);
```

Methods in Objects

Methods are functions defined as properties within objects, allowing them to perform actions or computations using the object's data.

- Belong to objects: Called like `object.method()`.
- Access object properties using `this`.

Example

```
const person = {
  name: "Alice",
  greet: function() {
    console.log(`Hello, ${this.name}!`);
  }
};

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

JSON (JavaScript Object Notation)

A **text format** for data interchange (derived from JS objects).

Valid JSON Example

```
{
  "name": "Alice",
  "age": 25,
  "courses": ["Math", "CS"]
}
```

JSON Methods

JSON.stringify(): Converts JS objects → JSON strings.

```
const student = {
  name: "Rahul",
  grades: [90, 85, 95]
};
const jsonStr = JSON.stringify(student);
console.log(jsonStr);
// '{"name":"Rahul","grades":[90,85,95]}'
```

JSON.parse(): Converts JSON strings → JS objects.

```
const json = '{"name":"Rahul","age":20}';
const obj = JSON.parse(json);
console.log(obj.name); // "Rahul"
```

Example: Student Object to JSON

Step 1: Create Object

```
const student = {
  name: "Priya",
  age: 22,
  courses: ["Physics", "Chemistry"],
  isActive: true,
  getInfo() {
    return `${this.name}, ${this.age}`;
  }
};
```

Step 2: Convert to JSON

```
const jsonData = JSON.stringify(student);
console.log(jsonData);
// {"name":"Priya","age":22,"courses":["Physics","Chemistry"],"isActive":true}
// Note: Methods are omitted!
```

Step 3: Parse Back to Object

```
const parsedStudent = JSON.parse(jsonData);
console.log(parsedStudent.name); // "Priya"
```

Document Object Model (DOM)

The DOM (Document Object Model) is a tree-like programming interface that represents an HTML/XML document as nodes and objects, allowing JavaScript to dynamically interact with webpage elements by modifying content, attributes, styles, adding/deleting elements, and handling events after the page loads. It powers interactive websites (e.g., form validation, dynamic content updates) and is the foundation for frameworks like React/Vue.

Example

```
// Change a paragraph's text
document.querySelector("p").textContent = "New content!";
```

DOM Tree Structure

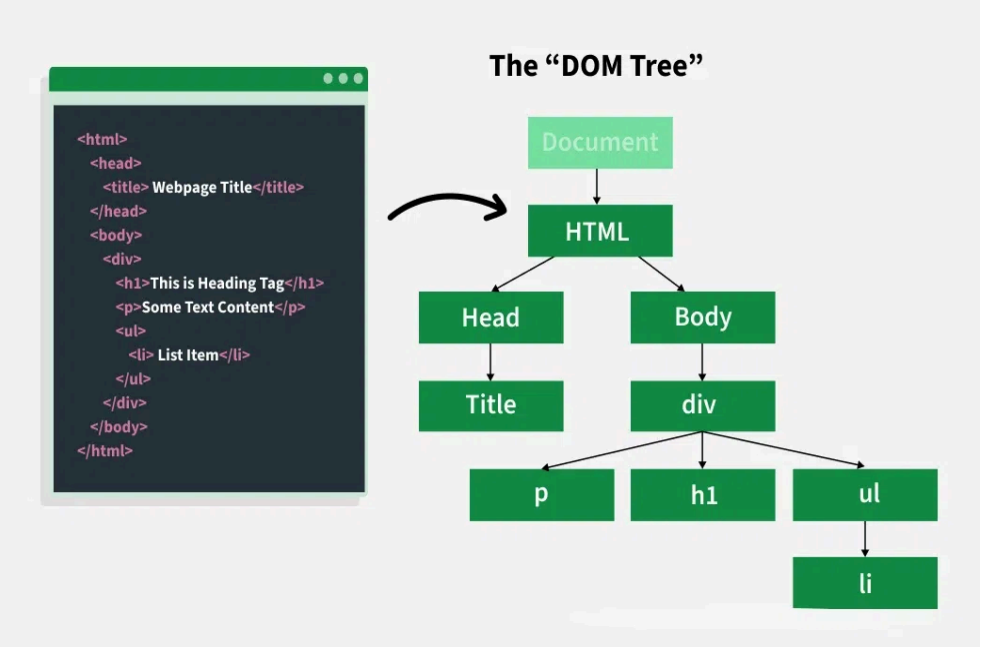
- **Document:** Root node (window.document).
- **Elements:** HTML tags (e.g., , <p>).
- **Nodes:** Everything in the DOM (elements, text, comments).

Example

```
<!DOCTYPE html>
<html>
 <head>
  <title>DOM Example</title>
 </head>
 <body>
  <h1 id="header">Hello</h1>
 </body>
</html>
```

DOM Tree

```
document
├── html
│   ├── head
│   │   └── title → "DOM Example"
│   └── body
│       └── h1 (id="header") → "Hello"
```



Selecting Elements

1. **getElementById:** Selects a single element by its id.

```
const header = document.getElementById("header");
console.log(header.textContent); // "Hello"
```

2. **querySelector:** Uses CSS selectors to return the first match.

```
const firstPara = document.querySelector("p"); // First <p>
const button = document.querySelector(".btn"); // First .btn
const link = document.querySelector("a[target]"); // First <a> with target
```

3. **getElementsByClassName:** Returns a live HTMLCollection of elements with the class.

```
const items = document.getElementsByClassName("item");
console.log(items[0]); // First .item
```

Modifying Content

1. **innerHTML:** Gets/sets HTML content (can include tags).

```
const div = document.querySelector("div");
div.innerHTML = "<strong>Warning!</strong> This is important.";
```

2. **textContent:** Gets/sets text content only (ignores HTML tags).

```
const heading = document.querySelector("h1");
heading.textContent = "New Heading"; // Safe
```

Example 1

```
<!DOCTYPE html>
<html>
<body>
 <h1 id="title">Original Title</h1>
 <div id="container"></div>
 <script>
  const title = document.getElementById("title");

  title.textContent = "Updated Title";

  const container = document.querySelector("#container");
  container.innerHTML = "<p style='color: red;'>Dynamic content!</p>";
 </script>
</body>
</html>
```

Example 2

```
<!--HTML File-->
<input type="text" id="nameInput">
<button id="submit">Submit</button>
<p id="output"></p>
```

```
//JS File
document.getElementById("submit").addEventListener("click", () => {
  const name = document.getElementById("nameInput").value;
  document.getElementById("output").textContent = `Hello, ${name}!`;
});
```

DOM – Manipulation & Styling

DOM manipulation and styling in JavaScript allow you to dynamically modify HTML elements by changing their styles, classes, and attributes—enabling interactive and responsive web pages. It helps in creating dynamic UIs (e.g., dark mode toggles, form validation).

Key Actions

- Styles: Adjust CSS (e.g., element.style.color = "red").
- Classes: Toggle/add/remove (e.g., element.classList.add("active")).
- Attributes: Update src, href, etc. (e.g., img.setAttribute("src", "new-image.jpg")).

Example

```
// Make a button red and disabled
const button = document.querySelector("button");
button.style.backgroundColor = "red";
button.setAttribute("disabled", true);
```

Changing Styles (element.style)

Directly manipulate inline styles of an element.

Syntax

```
element.style.property = "value";
// Property names are camelCased (not kebab-case)
// (e.g., fontSize instead of font-size)
```

Example

```
const heading = document.querySelector("h1");
heading.style.color = "blue";
heading.style.fontSize = "24px";
heading.style.backgroundColor = "#f0f0f0";
```

Notes

- Affects **inline styles** only (overrides external/internal CSS).
- Use style.cssText for multiple changes:

```
div.style.cssText = "color: red; font-size: 20px;";
```

- To remove a style:

```
heading.style.color = ""; // Reverts to CSS
```

Adding/Removing Classes (classList)

classList lets you safely add, remove, or toggle CSS classes on an element without overwriting existing classes, making it the preferred method over className for dynamic class management.

Methods

Method	Description	Example
add()	Adds a class	div.classList.add("active")
remove()	Removes a class	div.classList.remove("hidden")
toggle()	Toggles a class	btn.classList.toggle("dark")
contains()	Checks if a class exists	if (el.classList.contains("foo"))

Example

```
const button = document.querySelector("button");
button.classList.add("btn-primary");
button.classList.remove("btn-default");
button.classList.toggle("active"); // Adds/removes
```

Note: Avoid className as it replaces ALL existing classes, while classList.add() safely appends new classes without affecting others.

Modifying Attributes

setAttribute() / getAttribute()

setAttribute() updates an element's attribute (e.g., href, data-id), while getAttribute() retrieves its current value.

Example

```
const link = document.querySelector("a");

// Set attribute
link.setAttribute("href", "https://example.com");
link.setAttribute("target", "_blank");

// Get attribute
const href = link.getAttribute("href");
```

Direct Property Access

Some commonly used HTML attributes are mirrored as DOM object properties, allowing direct read/write:

Example

```
// Direct property access (faster & cleaner)
element.id = "new-id";  // Same as setAttribute("id", "new-id")
const href = link.href;  // Same as getAttribute("href")
input.checked = true;   // For checkboxes/radio buttons
img.src = "new-image.jpg"; // Updates the image source
```

Common Directly Accessible Properties

Attribute	DOM Property	Works With
id	element.id	All elements
href	link.href	<a>, <link>, <area>
src	img.src	<img>, <script>
value	input.value	Form elements (<input>, <textarea>)
checked	input.checked	Checkboxes/radios
disabled	input.disabled	Buttons, inputs

className	element.className	CSS classes (but prefer classList)
style	element.style	Inline CSS

Assignment: Implement a theme switcher that toggles between dark and light modes using DOM manipulation.

Events & Event Handling

Used to handle user interactions (clicks, mouse movements, keyboard input) using JavaScript event listeners.

Events

Events are actions or occurrences in the browser that JavaScript can respond to.

Common Event Types

Event Category	Examples
Mouse Events	click, mouseover, mouseout
Keyboard Events	keydown, keyup, keypress
Form Events	submit, focus, change
Window Events	load, resize, scroll

Event Handlers

Inline event handlers

Inline event handlers (like onclick="...") mix HTML with JavaScript, making code harder to maintain and limiting you to one handler per event—modern JavaScript avoids this approach. It binds logic directly in HTML, which becomes messy in larger projects.

Example

```
<button onclick="alert('Button clicked!')">Click Me (Inline Alert)</button>

<!-- OR with a function call -->
<button onclick="handleClick()">Click Me (Function Call)</button>
```

```
<script>
  function handleClick() {
    console.log("Inline handler executed");
    document.body.style.backgroundColor = "lightblue";
  }
</script>
```

addEventListener() method

addEventListener() method attaches event handlers in JavaScript (not HTML), enabling multiple listeners per event, better control over event phases, and cleaner separation of concerns.

Example

```
// Get DOM elements
const button = document.getElementById("my-btn");
const output = document.getElementById("output");

// Define event handler function
function handleClick() {
  output.textContent = "Status: Clicked!";
  console.log("Button clicked (traditional function)");
}

// Attach event listener (recommended)
button.addEventListener("click", handleClick);

// Add a SECOND listener (arrow function)
button.addEventListener("click", () => {
  console.log("Additional click handler (arrow function)");
});

// Remove listener after 5 seconds
setTimeout(() => {
  button.removeEventListener("click", handleClick);
  output.textContent = "Status: Listener Removed (try clicking again)";
}, 5000);
```

Key advantages:

- Supports multiple callbacks for the same event.
- Works with arrow functions or traditional functions.
- Allows removal with removeEventListener().

Event Object Properties

The callback function receives an event object (e) containing useful properties like e.target (the clicked element) and methods like e.preventDefault() to control default browser behavior.

Property/Method	Description	Example
e.target	Element that triggered the event	console.log(e.target.id)
e.preventDefault()	Stops default behavior (e.g., form submission)	e.preventDefault()
e.type	Event type ("click", "keydown")	if (e.type === "click")
e.clientX/e.clientY	Mouse pointer coordinates	console.log(e.clientX, e.clientY)

Example

Use of *preventDefault()* stops the browser's default action (page reload/redirect) when the form is submitted. The form data stays on-screen, and your JavaScript logic (like validation or AJAX) can handle the submission. It is often used in Single-page applications (SPAs).

```
document.querySelector("form").addEventListener("submit", (e) => {
  e.preventDefault(); // Stops page reload
  console.log("Form submitted!");
});
```

Event Propagation: Bubbling & Capturing

Capturing: The event travels from the topmost parent down to the target element (rarely used).
Bubbling: The event travels from the target element up to the parents (default behavior).

Example

```
// Capturing phase (rarely used)
document.querySelector("div").addEventListener("click", () => {
  console.log("Div clicked!");
}, true); // Set `true` for capturing

// Bubbling phase (default)
button.addEventListener("click", (e) => {
  console.log("Button clicked!");
  e.stopPropagation(); // Stops bubbling
});
```

Form Handling & Validation

JavaScript form validation ensures user input meets specific rules (like required fields or email formats) before submission, preventing invalid data from being sent to the server.

Form Submission

There are two methods of form submission:

Inline onsubmit

Avoid inline onsubmit in HTML (e.g., <form onsubmit="return validate()">) because it mixes JavaScript with markup, limits flexibility, and can't attach multiple handlers.

```
<form onsubmit="return handleSubmit()">
  <input type="text" id="name" placeholder="Name" required>
  <button type="submit">Submit</button>
</form>

<script>
  function handleSubmit() {
    const name = document.getElementById("name").value;
    if (!name) {
      alert("Name is required!"); // Basic validation
      return false; // Blocks form submit
    }
    return true; // Allows submit
  }
</script>
```

addEventListener('submit')

Always use addEventListener('submit') in JavaScript (e.g., form.addEventListener('submit', (e) => { ... })) for clean separation of concerns, support for multiple listeners, and full control over validation/submission.

```
<form id="myForm">
  <input type="text" id="name" placeholder="Name" required>
  <button type="submit">Submit</button>
</form>

<script>
  document.getElementById("myForm").addEventListener("submit", (e) => {
    e.preventDefault(); // Stops default reload

    const name = document.getElementById("name").value;
    if (!name) {
      alert("Name is required!"); // Validation
      return; // Exit early
    }
    // Proceed with submission
  });
</script>
```

Input Validation Techniques

JavaScript form validation ensures that user-submitted data meets specific criteria before being processed or sent to a server. It includes techniques like checking for required fields, validating email formats with regular expressions, and enforcing password strength rules (e.g., minimum length, uppercase letters, and numbers). Real-time validation provides immediate feedback, improving user experience by preventing errors early.

Example

```
form.addEventListener("submit", (e) => {
  if (!isValidEmail(email.value)) {
    e.preventDefault();
    showError("Invalid email!");
  }
})
```

Preventing Default Submission

Use e.preventDefault() to stop the form from submitting until validated. Then submit manually with form.submit() or send data via fetch() for AJAX.

Example

```
form.addEventListener("submit", async (e) => {
  e.preventDefault();
  if (validateForm()) {
    const response = await fetch("/api/register", {
      method: "POST",
      body: new FormData(form)
    });
    // Handle response...
  }
});
```

Example: Signup Form with Validation

HTML File

```
<form id="signup-form">
  <input type="text" id="username" placeholder="Username">
  <input type="email" id="email" placeholder="Email">
  <input type="password" id="password" placeholder="Password">
  <button type="submit">Sign Up</button>
  <p id="error" class="error"></p>
</form>
```

CSS File

```
.error { color: red; }
input:invalid { border: 1px solid red; }
```

JavaScript File

```
const form = document.getElementById("signup-form");
const errorEl = document.getElementById("error");

form.addEventListener("submit", (e) => {
  e.preventDefault();
  errorEl.textContent = "";

  const username = form.username.value.trim();
  const email = form.email.value.trim();
  const password = form.password.value.trim();

  // Validation
  if (!username) {
    showError("Username required");
```

```
    return;
  }
  if (!isValidEmail(email)) {
    showError("Invalid email");
    return;
  }
  if (!isStrongPassword(password)) {
    showError("Password must be 8+ chars with a number and uppercase");
    return;
  }

  // If valid, submit
  alert("Form submitted successfully!");
  form.reset();
});

function showError(message) {
  errorEl.textContent = message;
}
```

AJAX (Asynchronous JavaScript and XML)

AJAX (Asynchronous JavaScript and XML) is a web development technique that allows web pages to communicate with a server asynchronously, without requiring a page reload. By using the XMLHttpRequest object or modern fetch() API, AJAX enables dynamic content updates, form submissions, and data retrieval in the background, creating a smoother and more interactive user experience. Unlike traditional form submissions, AJAX processes responses programmatically in JavaScript, allowing developers to handle success or error messages without disrupting the user's workflow. For example, submitting a form via fetch() can update the UI instantly upon success, while gracefully displaying errors if validation fails. AJAX is a cornerstone of modern Single-Page Applications (SPAs), where seamless data exchange between client and server is essential.

Example

```
fetch("/api/submit", {
  method: "POST",
  body: JSON.stringify({ userInput: "value" })
})
.then(response => response.json())
.then(data => updateUI(data))
.catch(error => showError(error));
```

Error Handling & Debugging

JavaScript provides structured error handling using *try-catch-finally* blocks. The try block contains code that might throw an error, while the catch block handles the error gracefully by logging or displaying user-friendly messages. The optional finally block executes code regardless of whether an error occurred, making it useful for cleanup tasks. We can Prevent crashes with proper error handling

Syntax

```
try {
  // Code that might fail
  riskyOperation();
} catch (error) {
  // Handle the error
  console.error("Something went wrong:", error.message);
} finally {
  // Cleanup code (always runs)
  resetForm();
}
```

We can also use custom error handling

```
if (!userInput) {
  throw new Error("Input cannot be empty!"); // Custom error
}
```

For debugging, developers rely on tools like *console.log()* to trace values, browser DevTools to set breakpoints and inspect variables, and the *debugger* keyword to pause execution for step-by-step analysis.

```
function buggyFunction() {
  debugger; // Execution stops here
  // Inspect variables in DevTools
}
```

Common JavaScript Errors

Error Type	Example	Solution
Syntax Error	Missing) or }	Check console for line numbers
Type Error	null.length	Add null checks
Reference Error	Using undeclared variables	Check variable names

Network Error	Failed API request	Use .catch() with fetch()

Example

This function fetches data from an API endpoint (*/api/data*) and handles both success and error scenarios gracefully.

```
async function fetchData() {
  try {
    const response = await fetch("/api/data");
    if (!response.ok) throw new Error("Network error");
    return await response.json();
  } catch (error) {
    showToast("Failed to load data. Please retry.");
    console.error("API Error:", error);
  }
}
```

Assignment

1. Create a responsive web application that dynamically displays data from an XML file using HTML, CSS, JavaScript, and AJAX. Ensure the project has a clean UI, proper error handling, and demonstrates modern asynchronous data fetching techniques.
2. Create event driven program for following:
    a. To find a given number is even or odd.
    b. To find a given year is a leap year or not.
    c. To find greater no. between two entered numbers.
    d. Print a table of numbers from 5 to 15 and their squares and cubes using alert.
    e. Print the largest of three numbers.
    f. Find the factorial of a number n.
    g. Enter a list of positive numbers terminated by Zero. Find the sum and average of these numbers.
    h. A person deposits Rs 1000 in a fixed account yielding 5% interest. Compute the amount in the account at the end of each year for n years.
    i. Read n numbers. Count the number of negative numbers, positive numbers and zeros in the list.