What is JavaScript?

- **Definition**:
  JavaScript (JS) is a **high-level, interpreted programming language** used to make web pages interactive, dynamic, and functional.
- **Key Characteristics**:
  1. **Client-Side Scripting** – Runs in the browser, enabling real-time interaction without reloading pages.
  2. **Interpreted Language** – Code is executed line by line by the browser's JavaScript engine (like Chrome's V8).
  3. **Lightweight & Flexible** – Works well with HTML & CSS.
  4. **Event-Driven** – Can respond to user actions like clicks, hovers, or keyboard input.
  5. **Cross-Platform** – Runs on all major browsers and also outside browsers (with Node.js).
- **Use Cases**:
  - Adding interactivity (e.g., forms validation, dropdown menus).
  - Creating animations & effects.
  - Building full-stack web apps (Frontend with React, Backend with Node.js).
  - Game development, mobile apps (React Native), and even AI tools.
- **Example**:
- <button onclick="alert('Hello, JavaScript!')">Click Me</button>
- 👉 When the button is clicked, JavaScript shows an alert box.

✍️ Short Notes (for revision):

- **JavaScript** = High-level, interpreted scripting language.
- Runs **inside browser** (and outside with Node.js).
- Used to make websites **dynamic & interactive**.
- Works with **HTML & CSS**.
- Features: **event-driven, lightweight, cross-platform**.
- Example: alert("Hello, JavaScript!").

History of JavaScript -

- **Birth of JavaScript**:
  - Created in **1995** by **Brendan Eich** at **Netscape Communications**.
  - Originally developed in just **10 days**.
  - First called **Mocha**, then **LiveScript**, and finally named **JavaScript**.
- **Why the name JavaScript?**
  - At that time, **Java** was very popular.
  - Netscape renamed it JavaScript (marketing strategy) to ride on Java's popularity.

    - But **Java ≠ JavaScript** (totally different languages).
- **Standardization**:
    - In **1997**, JavaScript was submitted to **ECMA International** for standardization.
    - The standardized version was called **ECMAScript (ES)**.
    - **ECMAScript 1 (ES1)** was released in **1997**.
- **Important Versions**:
    - **ES3 (1999)** → First major widely used version.
    - **ES5 (2009)** → Added strict mode, JSON support, etc.
    - **ES6 / ECMAScript 2015 (2015)** → Biggest update with let, const, arrow functions, classes, promises, etc.
    - **ES7+ (2016 onwards)** → Yearly updates with new features like async/await, optional chaining, etc.
- **Today**:
    - JavaScript powers **web, mobile, backend, games, and AI apps**.
    - Supported by all modern browsers with engines like **V8 (Chrome)**, **SpiderMonkey (Firefox)**, **JavaScriptCore (Safari)**.

✍️ Short Notes (for revision)

- **1995** → Created by **Brendan Eich** at Netscape in 10 days.
- First names: **Mocha → LiveScript → JavaScript**.
- Name was **marketing**; Java ≠ JavaScript.
- **1997** → Standardized as **ECMAScript (ES)** by ECMA.
- Key versions:
    - **ES3 (1999)** – First major version.
    - **ES5 (2009)** – Strict mode, JSON.
    - **ES6 (2015)** – Biggest update (let, const, classes, promises).
    - **ES7+** – Yearly updates (async/await, etc.).
- Today → Runs on browsers & servers, most used language.

Ways to Write JavaScript -

1. Inline JavaScript

- **Written directly inside an HTML tag's attribute (like onclick, onmouseover, etc.).**
- **Used for small, quick actions.**
- **Example:**
- **<button onclick="alert('Hello!')">Click Me</button>**

⚠️ **Not recommended for large projects (bad for readability & maintainability).**

2. Internal JavaScript

- Written inside the <script> tag within the HTML file.
- Usually placed at the end of <body> for faster page loading.
- Example:
- <!DOCTYPE html>
- <html>
- <body>
- <h2>Internal JS Example</h2>
- <script>
- alert("Hello from Internal JS!");
- </script>
- </body>
- </html>

3. External JavaScript

- Written in a separate .js file and linked with HTML using the <script src="file.js"></script> tag.
- Best practice for real projects because:
  - Code is reusable across pages.
  - HTML & JS stay separate (cleaner code).
- Example:
- index.html
- <script src="app.js"></script>
- app.js
- alert("Hello from External JS!");

✅ Best Practice: Always prefer external JS for scalability, and use internal for small demos/testing. Avoid inline JS in professional projects.

✍️ Short Notes (for revision)

- Inline JS → Written inside HTML tags (e.g., onclick="..."). ❌ Not recommended.
- Internal JS → Written inside <script> tag in HTML file. ✅ Good for small projects.
- External JS → Written in .js file, linked via <script src="..."></script>. ✅ Best practice.

JavaScript Values & Variables -

1. Values in JavaScript

Values are the data that variables hold.

- Types of Values:
  - Fixed values (Literals) → Hard-coded values.

        - **Examples:**
            - **Numbers: 10, 3.14**
            - **Strings: "Hello", 'World'**
            - **Booleans: true, false**
            - **Null: null**
            - **Undefined: undefined**
    - **Variable values → Stored in variables for reuse.**

2. Variables in JavaScript

- **Definition: Variables are containers for storing values (data).**
- **Declaration Keywords:**
    1. **var (old, function-scoped, avoid in modern code).**
    2. **let (introduced in ES6, block-scoped, re-assignable).**
    3. **const (introduced in ES6, block-scoped, cannot be re-assigned).**

3. Rules for Naming Variables

- **Must start with letter, $, or _.**
- **Cannot start with a number.**
- **Case-sensitive (age and Age are different).**
- **Cannot use reserved keywords (let, var, function, etc.).**
- **Use camelCase for best practice (e.g., userName).**

4. Examples

```
// Numbers

let age = 20;



// Strings

const name = "John";



// Boolean

let isStudent = true;



// Null & Undefined

let data = null;

let value; // undefined
```

✍️ Short Notes (for revision)

- Values = data (literals like 10, "hello", true, null, undefined).
- Variables = containers to store values.
- Declared with:
  - var → old, function-scoped.
  - let → block-scoped, re-assignable.
  - const → block-scoped, constant.
- Naming rules: Start with letter/$/_, no keywords, case-sensitive.
- Example:
- let age = 25;
- const pi = 3.14;
- var name = "Alice";

JavaScript Data Types -

1. Primitive Data Types

👉 Stored directly in memory (immutable, simple).

1. Number → Represents both integers & floating-point.
2. let age = 25;
3. let pi = 3.14;
4. String → Text enclosed in " " or ' ' or ` ` (template literals).
5. let name = "John";
6. let message = `Hello, ${name}`;
7. Boolean → Logical values: true or false.
8. let isStudent = true;
9. Undefined → Variable declared but not assigned a value.
10. let x;
11. console.log(x); // undefined
12. Null → Represents intentional empty value.
13. let data = null;
14. Symbol (ES6) → Unique & immutable identifier.
15. let id = Symbol("id");
16. BigInt (ES11, 2020) → For very large numbers beyond Number limit.
17. let big = 1234567890123456789012345678901234567890n;

2. Non-Primitive (Reference) Data Types

👉 Stored by reference (complex, mutable).

1. **Object → Collection of key-value pairs.**
2. **let person = { name: "Alice", age: 22 };**
3. **Array → Ordered list of values.**
4. **let numbers = [1, 2, 3, 4];**
5. **Function → Block of reusable code.**
6. **function greet() {**
7. **return "Hello!";**
8. **}**

3. Special Note: typeof Operator

- **Used to check data type.**
- **Example:**
- **typeof 123; // "number"**
- **typeof "hello"; // "string"**
- **typeof null; // "object" (special case/bug in JS)**
- **typeof undefined;// "undefined"**

✍️ Short Notes (for revision)

- **Data Types = Primitive + Non-Primitive**
- **Primitive (7 types) → Number, String, Boolean, Undefined, Null, Symbol, BigInt.**
- **Non-Primitive (3 main types) → Object, Array, Function.**
- **typeof → used to check type.**
- **Example:**
- **let age = 25; // Number**
- **let name = "John"; // String**
- **let isOk = true; // Boolean**
- **let user = {id:1}; // Object**
- **let arr = [1,2,3]; // Array**

Truthy & Falsy -

1. Definition

- **In JavaScript, every value is either truthy or falsy when evaluated in a Boolean context (like if, while, ||, &&).**
- **Truthy → Values that are treated as true.**
- **Falsy → Values that are treated as false.**

2. Falsy Values (Only 8 in JS)

👉 **These evaluate to false in a Boolean context:**

1. **false**
2. **0 (zero)**
3. **-0 (negative zero)**
4. **0n (BigInt zero)**
5. **"" (empty string)**
6. **null**
7. **undefined**
8. **NaN (Not-a-Number)**

Example:

```
if (0) {

  console.log("This will NOT run");

}
```

3. Truthy Values

👉 **Everything else apart from the falsy list is truthy.**
Examples:

- **true**
- **Any non-zero number: 1, -5, 3.14**
- **Non-empty strings: "hello", "0"**
- **Objects & Arrays (even empty ones): {}, []**
- **Functions**
- **" " (string with a space)**

Example:

```
if ("hello") {

  console.log("This WILL run");

}
```

4. Use Cases

- **Checking if a variable has a value:**
- **let name = "";**
- **if (name) {**
- ** console.log("Name exists");**
- **} else {**
- ** console.log("Name is empty");**
- **}**
- **Using logical OR (||) for default values:**
- **let user = "" || "Guest";**
- **console.log(user); // Guest**

✍️ Short Notes (for revision)

- **Falsy Values (8 total):**
  false, 0, -0, 0n, "", null, undefined, NaN.
- **Truthy Values: Everything else (non-empty strings, numbers ≠0, objects, arrays, functions, etc.).**
- **Examples:**
- **if (0) console.log("No"); // falsy**
- **if ("hello") console.log("Yes"); // truthy**

parseInt() & parseFloat() -

1. parseInt()

- **Converts a string into an integer (whole number).**
- **Syntax:**
- **parseInt(string, radix);**
  - **string → The value to parse.**
  - **radix → (Optional) Base of number system (2–36).**
- **Examples:**
- **parseInt("42"); // 42**
- **parseInt("42px"); // 42 (stops at non-digit)**
- **parseInt("3.14"); // 3 (ignores decimal part)**
- **parseInt("101", 2); // 5 (binary to decimal)**
- **parseInt("FF", 16); // 255 (hexadecimal to decimal)**

⚠️ **If the first character cannot be converted → returns NaN.**

2. parseFloat()

- **Converts a string into a floating-point (decimal) number.**
- **Syntax:**
- **parseFloat(string);**
- **Examples:**
- **parseFloat("42"); // 42**
- **parseFloat("3.14"); // 3.14**
- **parseFloat("42.5px"); // 42.5 (stops at non-digit)**
- **parseFloat("abc"); // NaN**

3. Difference Between parseInt() & parseFloat()

Feature	parseInt()	parseFloat()
Output Type	Integer	Floating-point
Handles Decimals	Truncates	Keeps decimals
Radix Support	✅ Yes	❌ No
Example ("3.14")	3	3.14

4. Special Notes

- Both functions ignore leading spaces:
- parseInt(" 99"); // 99
- Both stop parsing at first invalid character.
- Number() function can also convert strings → but stricter.
- Number("42px"); // NaN

✍️ Short Notes (for revision)

- parseInt(str, radix) → Converts string → integer. Ignores decimals.
  Example: parseInt("42.7") → 42
- parseFloat(str) → Converts string → decimal number. Keeps decimals.
  Example: parseFloat("42.7") → 42.7
- Difference:
  - parseInt("3.14") → 3
  - parseFloat("3.14") → 3.14
- Returns NaN if not valid number.

Expressions & Operators -

1. Expressions

- An expression is any valid piece of code that produces a value.
- Can be a single value, variable, or a combination of values and operators.
- Examples:
- 5 + 3 // 8
- "Hello " + "World" // "Hello World"

- let x = 10;
- x * 2 // 20

2. Operators in JavaScript

Operators are special symbols used to perform operations on operands (values/variables).

(A) Arithmetic Operators

Perform mathematical operations.

+ // Addition

- // Subtraction

*** // Multiplication**

/ // Division

% // Modulus (remainder)

**** // Exponentiation (ES6)**

++ // Increment

-- // Decrement

Example:

```
let a = 5, b = 2;

console.log(a + b); // 7

console.log(a % b); // 1
```

(B) Assignment Operators

Assign values to variables.

= // Assignment

+= // Add & assign

-= // Subtract & assign

***= // Multiply & assign**

/= // Divide & assign

%= // Modulus & assign

Example:

```
let x = 10;

x += 5; // x = 15
```

(C) Comparison Operators

Compare two values → returns true or false.

```
==   // Equal (loose, type conversion allowed)

===  // Strict equal (no type conversion)

!=   // Not equal

!==  // Strict not equal

>    // Greater than

<    // Less than

>=   // Greater than or equal

<=   // Less than or equal
```

Example:

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

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

(D) Logical Operators

Used for logical conditions.

```
&&   // AND

||   // OR

!    // NOT
```

Example:

```
true && false; // false

true || false; // true

!true;       // false
```

(E) String Operators

- + → Concatenation
- += → Concatenate & assign

let s1 = "Hello";

let s2 = "World";

console.log(s1 + " " + s2); // "Hello World"

(F) Conditional (Ternary) Operator

Shorthand if-else.

condition ? valueIfTrue : valueIfFalse;

Example:

let age = 18;

let result = (age >= 18) ? "Adult" : "Minor";

(G) Type Operators

- typeof → Returns type of variable.
- instanceof → Checks if an object is instance of a class.

typeof 42; // "number"

[] instanceof Array; // true

(H) Bitwise Operators (work at binary level)

& // AND

| // OR

^ // XOR

~ // NOT

<< // Left shift

>> // Right shift

>>> // Unsigned right shift

✍️ Short Notes (for revision)

- Expression = Code that produces a value (5+3, "Hi"+"JS").
- Operators Types:
  - Arithmetic: + - * / % ** ++ --

- **Assignment: =, +=, -=, *=, /=**
- **Comparison: ==, ===, !=, !==, >, <, >=, <=**
- **Logical: &&, ||, !**
- **String: + (concat), +=**
- **Ternary: condition ? val1 : val2**
- **Type: typeof, instanceof**
- **Bitwise: &, |, ^, ~, <<, >>, >>>**

Ternary Operator in JavaScript -

1. Definition

- **The ternary operator (? :) is a shorthand way of writing an if...else statement.**
- **Called ternary because it works with three operands:**
  condition ? valueIfTrue : valueIfFalse

2. Syntax

condition ? expression1 : expression2;

- **If condition is true → expression1 runs.**
- **If condition is false → expression2 runs.**

3. Examples

Basic Example

```
let age = 20;

let status = (age >= 18) ? "Adult" : "Minor";

console.log(status); // "Adult"
```

Nested Ternary (avoid if too complex)

```
let score = 85;

let grade = (score >= 90) ? "A" :

        (score >= 75) ? "B" :

        (score >= 50) ? "C" : "F";

console.log(grade); // "B"
```

With Function

```
function isEven(num) {

 return (num % 2 === 0) ? "Even" : "Odd";

}

console.log(isEven(7)); // "Odd"
```

4. When to Use

✅ Use ternary for short, simple conditions.
❌ Avoid for long nested logic (use if...else instead for readability).

✍️ Short Notes (for revision)

- Ternary Operator = Shorthand if-else.
- Syntax:
- condition ? valueIfTrue : valueIfFalse;
- Example:
- let age = 18;
- let type = (age >= 18) ? "Adult" : "Minor"; // "Adult"
- ✅ Good for short conditions.
- ❌ Avoid multiple nesting (hurts readability).

Control Statements & Loops -

1. Control Statements

Control statements are used to control the flow of program execution.

(A) Conditional Statements

1. if Statement – Executes block if condition is true.
2. if (age >= 18) {
3. console.log("You are an adult");
4. }
5. if...else Statement – Executes one block if condition is true, another if false.
6. if (marks >= 40) {
7. console.log("Pass");
8. } else {
9. console.log("Fail");

10. }
11. if...else if...else Statement – Multiple conditions.
12. if (score >= 90) {
13. console.log("A");
14. } else if (score >= 75) {
15. console.log("B");
16. } else {
17. console.log("C");
18. }
19. switch Statement – Used for multiple choices (better than many if...else).
20. let day = 3;
21. switch(day) {
22. case 1: console.log("Monday"); break;
23. case 2: console.log("Tuesday"); break;
24. case 3: console.log("Wednesday"); break;
25. default: console.log("Invalid day");
26. }

2. Loops

Loops allow us to repeat a block of code multiple times.

(A) for Loop

Used when number of iterations is known.

```
for (let i = 1; i <= 5; i++) {

  console.log(i);

}
```

(B) while Loop

Repeats while condition is true.

```
let i = 1;

while (i <= 5) {

  console.log(i);

  i++;

}
```

(C) do...while Loop

Runs at least once, then checks condition.

```
let i = 1;

do {

 console.log(i);

 i++;

} while (i <= 5);
```

(D) for...of Loop (ES6)

Iterates over iterable objects (arrays, strings, etc.).

```
let arr = [10, 20, 30];

for (let val of arr) {

 console.log(val);

}
```

(E) for...in Loop

Iterates over object properties (keys).

```
let person = { name: "John", age: 25 };

for (let key in person) {

 console.log(key, person[key]);

}
```

3. Jump Statements

1. **break – Exits loop immediately.**
2. **for (let i = 1; i <= 5; i++) {**
3. **if (i === 3) break;**
4. **console.log(i); // 1, 2**
5. **}**
6. **continue – Skips current iteration, moves to next.**
7. **for (let i = 1; i <= 5; i++) {**
8. **if (i === 3) continue;**
9. **console.log(i); // 1, 2, 4, 5**
10. **}**

✍️ Short Notes (for revision)

- **Control Statements:**

    - if, if...else, if...else if, switch.
- Loops:
    - for → fixed iterations.
    - while → repeats while condition true.
    - do...while → runs at least once.
    - for...of → iterates over array/string values.
    - for...in → iterates over object keys.
- Jump Statements:
    - break → exit loop.
    - continue → skip iteration.

Functions in JavaScript -

1. Definition

- A function is a block of reusable code that performs a specific task.
- Helps in code reusability, readability, modularity.

2. Function Syntax

```
function functionName(parameters) {

  // code to be executed

  return value;

}
```

3. Types of Functions

(A) Function Declaration / Named Function

```
function greet() {

  return "Hello!";

}

console.log(greet()); // Hello!
```

(B) Function Expression

- Function stored inside a variable.

```
const greet = function() {
  return "Hi!";
};
console.log(greet());
```

(C) Anonymous Function

- **Function without a name (often used in expressions or callbacks).**

```
setTimeout(function() {
  console.log("Anonymous function");
}, 1000);
```

(D) Arrow Function (ES6)

- **Shorter syntax for functions.**

```
const add = (a, b) => a + b;
console.log(add(5, 3)); // 8
```

(E) Immediately Invoked Function Expression (IIFE)

- **Runs immediately after definition.**

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

(F) Function with Parameters & Default Parameters

```
function multiply(a, b = 1) {
  return a * b;
}
console.log(multiply(5)); // 5
```

4. Return Statement

- **Functions can return values using return.**

```
function square(x) {
```

```
  return x * x;

}

console.log(square(4)); // 16
```

5. Special Types

- Recursive Function: Function calling itself.
- Callback Function: Function passed as an argument.
- Higher-Order Function: Function that takes/returns another function.

✍️ Short Notes (for revision)

- Function = Block of reusable code.
- Syntax:
- function name(params) { return value; }
- Types:
  - Function Declaration
  - Function Expression
  - Anonymous Function
  - Arrow Function (ES6)
  - IIFE (runs immediately)
- Features:
  - Parameters + Default values
  - return keyword
  - Can be recursive / callback / higher-order.
- Example:
- const add = (a, b) => a + b;
- console.log(add(2, 3)); // 5

var, let & const -

1. var

- Introduced in ES1 (old).
- Function-scoped → Accessible within the function where declared.
- Can be re-declared and updated.
- Hoisted (moved to top of scope) but initialized with undefined.

Example:

```
var x = 10;
```

```
var x = 20; // ✅ allowed

console.log(x); // 20


function test() {

 var y = 5;

 console.log(y);

}

// console.log(y); ❌ Error (function-scoped)
```

2. let

- Introduced in ES6 (2015).
- Block-scoped → Accessible only inside {} block.
- Can be updated, but not re-declared in the same scope.
- Hoisted but not initialized (Temporal Dead Zone).

Example:

```
let a = 10;

a = 15;  // ✅ allowed

// let a = 20; ❌ Error (cannot re-declare in same scope)


if (true) {

 let b = 5;

 console.log(b); // ✅ 5

}

// console.log(b); ❌ Error (block-scoped)
```

3. const

- Introduced in ES6 (2015).
- Block-scoped like let.
- Must be initialized at declaration.
- Cannot be updated or re-declared.
- However, if assigned to an object/array, the contents *can* be modified (not reassigned).

Example:

const pi = 3.14;

// pi = 3.15; ❌ Error (cannot reassign)

const arr = [1, 2, 3];

arr.push(4); // ✅ allowed (modifying contents)

console.log(arr); // [1,2,3,4]

4. Key Differences

Feature	var	let	const
Scope	Function-scoped	Block-scoped	Block-scoped
Re-declaration	✅ Allowed	❌ Not allowed	❌ Not allowed
Update	✅ Allowed	✅ Allowed	❌ Not allowed
Hoisting	✅ Hoisted (init = undefined)	✅ Hoisted (TDZ)	✅ Hoisted (TDZ)
Initialization req	❌ Not required	❌ Not required	✅ Required

✍️ Short Notes (for revision)

- var → Function-scoped, can re-declare & update, hoisted with undefined.
- let → Block-scoped, can update but not re-declare, hoisted (TDZ).
- const → Block-scoped, cannot re-declare or update, must initialize, but objects/arrays can be mutated.

Example:

var x = 1;

let y = 2;

const z = 3;

Template Strings -

1. Definition

- **Introduced in ES6 (2015).**
- **Template Strings (also called Template Literals) allow easier string creation and interpolation.**
- **Declared using backticks (`), instead of quotes (' ' or " ").**

2. Features

(A) String Interpolation

- **Allows embedding variables/expressions inside ${}.**

```
let name = "John";

let age = 25;

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

// My name is John and I am 25 years old.
```

(B) Multi-line Strings

- **Can write strings across multiple lines without \n.**

```
let msg = `This is line 1

This is line 2

This is line 3`;

console.log(msg);
```

(C) Expression Evaluation

- **Can directly put expressions inside ${}.**

```
let a = 5, b = 10;

console.log(`Sum = ${a + b}`); // Sum = 15
```

(D) Function Calls Inside

```
function greet() { return "Hello"; }

console.log(`${greet()}, World!`); // Hello, World!
```

(E) Tagged Templates (Advanced)

- **Special kind of template string where a function processes the template.**

```
function tag(strings, ...values) {

 return strings[0] + values[0].toUpperCase();

}

let user = "john";

console.log(tag`Hello ${user}`); // Hello JOHN
```

3. Why Use Template Strings?

- **Cleaner syntax for string concatenation.**
- **Easier to work with dynamic values.**
- **Better for creating HTML templates in JS.**

✍️ Short Notes (for revision)

- **Template Strings = Strings with backticks (`).**
- **Features:**
  - **Interpolation: `Hello ${name}`**
  - **Multi-line support**
  - **Expressions inside ${}**
  - **Can use functions inside**
- **Example:**
- **let name = "Alice", age = 20;**
- **console.log(`Name: ${name}, Age: ${age}`);**
- **// Name: Alice, Age: 20**

Default Arguments -

1. Definition

- **Default arguments (introduced in ES6, 2015) allow you to give a default value to function parameters.**

- If the function is called without that parameter or with undefined, the default value will be used.

2. Syntax

```
function functionName(param1 = defaultValue1, param2 = defaultValue2) {
  // function body
}
```

3. Examples

Basic Example

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

greet("John");  // Hello, John!
greet();       // Hello, Guest!
```

Multiple Parameters

```
function add(a = 0, b = 0) {
  return a + b;
}

console.log(add(5, 10)); // 15
console.log(add(5));     // 5
console.log(add());      // 0
```

Expression as Default Value

```
function calcPrice(price, tax = price * 0.1) {
  return price + tax;
}
```

console.log(calcPrice(100)); // 110

Function as Default Value

```
function randomNum(val = Math.random()) {

 return val;

}

console.log(randomNum()); // e.g. 0.54321
```

4. Key Points

- **Default values only apply when:**
  - **Argument is not passed, OR**
  - **Argument is undefined.**
- **If argument is null, the default is NOT used.**

```
function test(x = 10) {

 console.log(x);

}

test(undefined); // 10 ✅

test(null);     // null ❌ (default not applied)
```

✍️ Short Notes (for revision)

- **Default Arguments = Assign default values to function parameters.**
- **Syntax:**
- **function fn(param = value) { ... }**
- **Used when argument is missing or undefined.**
- **Example:**
- **function greet(name = "Guest") {**
- ** console.log(`Hello, ${name}`);**
- **}**
- **greet(); // Hello, Guest**

Arrow Functions -

1. Definition

- **Arrow functions were introduced in ES6 (2015).**
- **They provide a shorter syntax for writing functions.**
- **They don’t have their own this, arguments, super, or new.target (important difference).**

2. Syntax

Basic Form

```
// Traditional Function

function add(a, b) {

 return a + b;

}



// Arrow Function

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

With One Parameter (parentheses optional)

```
const square = x => x * x;
```

No Parameters

```
const greet = () => "Hello!";
```

With Multiple Statements (need {} and return)

```
const multiply = (a, b) => {

 let result = a * b;

 return result;

};
```

3. Key Features

1. **Concise Syntax**
   - **Short and clean.**
   - **Great for one-liner functions.**
2. **Implicit Return**
   - **If body has a single expression → no need for return.**
3. **const double = n => n * 2;**
4. **Lexical this Binding**
   - **Arrow functions don’t create their own this.**

- They use this from their surrounding context.

```
Example:
function Person() {
  this.age = 0;

  setInterval(() => {
    this.age++; // `this` refers to Person, not setInterval
    console.log(this.age);
  }, 1000);
}

new Person();
```

No arguments Object

- Arrow functions don't have their own arguments.
- Must use rest parameters (...args).

```
const sum = (...nums) => nums.reduce((a, b) => a + b, 0);
console.log(sum(1, 2, 3)); // 6
```

4. Where Arrow Functions Are Useful

- Callbacks (e.g., map, filter, reduce).
- Event handlers (when this should come from parent).
- One-liner utility functions.

✍️ Short Notes (for revision)

- Arrow Function = Short function syntax (ES6).
- Syntax:
- const fn = (param) => expression;
- Features:
  - Shorter than function.
  - Implicit return (if one-liner).
  - Lexical this (inherits this from parent).
  - No arguments object.
- Example:
- const add = (a, b) => a + b;
- console.log(add(2, 3)); // 5

Destructuring -

1. Definition

- **Destructuring is a feature introduced in ES6 (2015).**
- **It allows you to unpack values from arrays or properties from objects into variables in a clean, concise way.**

2. Array Destructuring

Basic Example

```
const arr = [10, 20, 30];

const [a, b, c] = arr;

console.log(a, b, c); // 10 20 30
```

Skip Values

```
const [x, , z] = [1, 2, 3];

console.log(x, z); // 1 3
```

Default Values

```
const [p = 5, q = 10] = [7];

console.log(p, q); // 7 10
```

Swap Variables

```
let m = 1, n = 2;

[m, n] = [n, m];

console.log(m, n); // 2 1
```

3. Object Destructuring

Basic Example

```
const person = { name: "Alice", age: 22 };

const { name, age } = person;

console.log(name, age); // Alice 22
```

Rename Variables

```
const { name: fullName, age: years } = person;

console.log(fullName, years); // Alice 22
```

Default Values

```
const { city = "Unknown" } = person;

console.log(city); // Unknown
```

4. Nested Destructuring

Array Example

```
const nums = [1, [2, 3]];

const [one, [two, three]] = nums;

console.log(one, two, three); // 1 2 3
```

Object Example

```
const user = {
  id: 1,
  profile: { username: "doxt", email: "test@mail.com" }
};

const { profile: { username, email } } = user;

console.log(username, email); // doxt test@mail.com
```

5. Function Parameters Destructuring

Object Parameters

```
function greet({ name, age }) {
  console.log(`Hello ${name}, age ${age}`);
}

greet({ name: "John", age: 30 });
```

Array Parameters

```
function sum([a, b]) {
  return a + b;
}

console.log(sum([5, 10])); // 15
```

6. Spread & Rest with Destructuring

const [first, ...rest] = [1, 2, 3, 4];

console.log(first, rest); // 1 [2,3,4]

const { a, ...others } = { a: 1, b: 2, c: 3 };

console.log(a, others); // 1 { b:2, c:3 }

✍️ Short Notes (for revision)

- **Destructuring = Extracting values from arrays/objects into variables.**
- **Array Destructuring:**
- **const [a, b] = [1, 2]; // a=1, b=2**
- **Object Destructuring:**
- **const {name, age} = {name:"Tom", age:20};**
- **Supports default values, renaming, nested objects, rest/spread.**
- **Useful in function parameters and cleaner code.**

Array & Array Properties -

1. What is an Array?

- **An array is a special type of object in JavaScript used to store ordered collection of values.**
- **Values can be of any data type (numbers, strings, objects, functions, etc.).**
- **Arrays are zero-indexed (first element at index 0).**

Example:

let arr = [10, "hello", true, {a:1}, [2,3]];

console.log(arr[0]); // 10

console.log(arr[3]); // {a:1}

2. Ways to Create Arrays

// Literal notation (most common)

let fruits = ["apple", "banana", "mango"];

```
// Using new Array()

let numbers = new Array(1, 2, 3);


// Empty Array

let empty = [];
```

3. Important Array Properties

🔹 length

- Returns the number of elements in the array.

```
let nums = [1, 2, 3];

console.log(nums.length); // 3
```

🔹 constructor

- Shows the function that created the array's prototype.

```
console.log(nums.constructor === Array); // true
```

🔹 prototype (for methods)

- Defines methods/properties available to all arrays.

```
console.log(Array.prototype); // shows all default methods
```

🔹 index

- Arrays are zero-based indexed.

```
let colors = ["red", "green", "blue"];

console.log(colors[0]); // red
```

🔹 Array.isArray()

- Checks if a value is an array.

```
console.log(Array.isArray([1, 2])); // true

console.log(Array.isArray("hello")); // false
```

🔹 toString()

- Converts array to a comma-separated string.

```
console.log([1,2,3].toString()); // "1,2,3"
```

🔹 valueOf()

- Returns the array itself.

```
let a = [1,2,3];

console.log(a.valueOf()); // [1,2,3]
```

🔹 push() and pop()

- push() → Add element at end.
- pop() → Remove last element.

```
let arr = [1,2];

arr.push(3); // [1,2,3]

arr.pop();   // [1,2]
```

🔹 shift() and unshift()

- shift() → Remove first element.
- unshift() → Add element at beginning.

🔹 concat()

- Joins arrays into a new one.

🔹 slice() and splice()

- slice(start, end) → Returns a portion (does not modify original).
- splice(start, deleteCount, items...) → Modifies array.

🔹 includes()

- Checks if a value exists in array.

```
console.log([1,2,3].includes(2)); // true
```

✍️ Short Notes (for revision)

- Array = Ordered collection, zero-indexed, can store mixed data.
- Create:
- let arr = [1,2,3];
- let arr2 = new Array(1,2,3);
- Properties:

- length → number of elements.
- constructor → Array function reference.
- prototype → shared methods.
- Array.isArray(val) → check array.

- Common Methods:
  - push()/pop() → end (add/remove).
  - shift()/unshift() → start (remove/add).
  - concat() → merge arrays.
  - slice() → copy part (no change).
  - splice() → add/remove (changes array).
  - includes() → check value exists.

Objects & Object Properties -

1. What is an Object?

- An object is a collection of key–value pairs.
- Keys (also called properties) are strings or symbols, and values can be any data type (number, string, array, function, another object, etc.).
- Objects are used to store and organize data.

Example:

```
let person = {
  name: "Rahul",
  age: 25,
  isStudent: true,
  greet: function() {
    return "Hello, " + this.name;
  }
};
console.log(person.name); // Rahul
console.log(person.greet()); // Hello, Rahul
```

2. Ways to Create Objects

🔹 Object Literal (most common)

```
let obj = { key: "value" };
```

🔹 Using new Object()

```
let obj = new Object();

obj.name = "Aman";
```

🔹 Using Constructor Function

```
function Person(name, age) {
 this.name = name;
 this.age = age;
}

let p1 = new Person("Ravi", 30);
```

🔹 Using class (ES6)

```
class Car {
 constructor(brand) {
  this.brand = brand;
 }
}

let c1 = new Car("BMW");
```

🔹 Using Object.create()

```
let proto = { greet() { return "Hi!"; } };

let obj = Object.create(proto);
```

3. Object Properties

- Property = Key + Value

```
let obj = { name: "John", age: 22 };
```

Accessing Properties

1. **Dot notation → obj.key**
2. **Bracket notation → obj["key"]**

Common Object Property Methods

🔹 **Object.keys(obj) → returns array of keys.**

```
console.log(Object.keys({a:1,b:2})); // ["a","b"]
```

🔹 **Object.values(obj) → returns array of values.**

```
console.log(Object.values({a:1,b:2})); // [1,2]
```

🔹 **Object.entries(obj) → returns array of key-value pairs.**

```
console.log(Object.entries({a:1,b:2})); // [["a",1],["b",2]]
```

🔹 **Object.assign(target, source) → copies properties.**

```
let target = {a:1};

let source = {b:2};

Object.assign(target, source);

console.log(target); // {a:1, b:2}
```

🔹 **Object.freeze(obj) → makes object immutable (cannot add/update/delete).**

```
let obj = {a:1};

Object.freeze(obj);

obj.a = 10; // ignored
```

🔹 **Object.seal(obj) → allows update, but not add/remove.**

🔹 **Object.hasOwn(obj, prop) (ES2022) → check if property exists.**

```
let obj = {x:10};

console.log(Object.hasOwn(obj, "x")); // true
```

4. Property Descriptors

Every property has attributes:

- **value → stored value.**
- **writable → can value be changed?**

- enumerable → will it show in loops?
- configurable → can property be deleted/modified?

```
let obj = {};

Object.defineProperty(obj, "x", {

 value: 42,

 writable: false,

 enumerable: true,

 configurable: false

});

console.log(obj.x); // 42
```

✍️ Short Notes (for revision)

- Object = Collection of key–value pairs.
- Create:
  - Literal → {key: value}
  - new Object()
  - Constructor function
  - class
  - Object.create(proto)
- Access:
  - obj.key
  - obj["key"]
- Useful Methods:
  - Object.keys(obj) → array of keys
  - Object.values(obj) → array of values
  - Object.entries(obj) → array of [key, value]
  - Object.assign(target, source) → copy properties
  - Object.freeze(obj) → no change allowed
  - Object.seal(obj) → only update allowed
  - Object.hasOwn(obj, prop) → check property
- Property Attributes: value, writable, enumerable, configurable

Rest & Spread Operators -

Both operators use the same syntax: ... (three dots), but their purpose is different.

1. Spread Operator (...)

- **Expands/Unpacks elements of an array or object.**
- **Used to copy, merge, or pass elements individually.**

🔹 Examples

Array Copying

```
let arr1 = [1, 2, 3];

let arr2 = [...arr1];

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

Merging Arrays

```
let a = [1, 2];

let b = [3, 4];

let c = [...a, ...b];

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

Function Arguments

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

let nums = [1, 2, 3];

console.log(add(...nums)); // 6
```

Object Copying & Merging

```
let obj1 = {x:1, y:2};

let obj2 = {z:3};

let obj3 = {...obj1, ...obj2};

console.log(obj3); // {x:1, y:2, z:3}
```

2. Rest Operator (...)

- Collects/Packs multiple elements into a single variable (array or object).
- Used in function parameters and destructuring.

🔹 Examples

Function Parameters

```
function sum(...numbers) {
 return numbers.reduce((a, b) => a + b, 0);
}

console.log(sum(1, 2, 3, 4)); // 10
```

Array Destructuring

```
let [first, ...rest] = [10, 20, 30, 40];

console.log(first); // 10

console.log(rest);  // [20, 30, 40]
```

Object Destructuring

```
let obj = {a:1, b:2, c:3};

let {a, ...others} = obj;

console.log(a);     // 1

console.log(others); // {b:2, c:3}
```

3. Key Differences

Feature	Spread (...) ➡️ Expand	Rest (...) ⬅️ Collect
Purpose	Expands elements	Collects elements
Usage in Functions	Pass arguments	Gather arguments
Usage in Destructuring	Unpacks values	Stores remaining values
Example	add(...arr)	function sum(...args)

✍️ Short Notes (for Revision)

- **Spread (...) → Expands elements.**
  - **Copy / merge arrays & objects.**
  - **Pass array elements as function args.**
- **Rest (...) → Collects elements.**
  - **Gather extra function arguments.**
  - **Collect remaining array/object parts.**

Examples:

```
// Spread

let arr = [1, 2, 3];

console.log([...arr, 4]); // [1, 2, 3, 4]


// Rest

function sum(...nums) { return nums.reduce((a, b) => a + b, 0); }

console.log(sum(1,2,3)); // 6
```

Strings in JavaScript -

1. Definition

- **A string is a sequence of characters enclosed in single quotes ', double quotes ", or backticks `.**
- **Strings are immutable (cannot be changed once created).**

```
let str1 = "Hello";

let str2 = 'World';

let str3 = `Hello World`; // template string
```

2. Creating Strings

- **Using quotes (' ', " ")**
- **Using template literals (` `) → allows multi-line & interpolation.**

```
let name = "John";

console.log(`Hello, ${name}!`); // Hello, John!
```

3. String Properties

Property	Description	Example
length	Returns string length	"Hello".length → 5

4. Common String Methods

🔹 Accessing Characters

```
let str = "JavaScript";

console.log(str[0]);   // J

console.log(str.charAt(2)); // v
```

🔹 Case Conversion

```
"hello".toUpperCase(); // "HELLO"

"WORLD".toLowerCase(); // "world"
```

🔹 Searching

```
"JavaScript".indexOf("Script"); // 4

"JavaScript".includes("Java");  // true

"JavaScript".startsWith("Java"); // true

"JavaScript".endsWith("Script"); // true
```

🔹 Extracting Substrings

```
"Hello World".slice(0, 5);   // "Hello"

"Hello World".substring(6);  // "World"

"Hello World".substr(6, 3);  // "Wor" (deprecated but still works)
```

🔹 Replacing

"JS is great".replace("great", "awesome"); // "JS is awesome"

"JS JS".replaceAll("JS", "JavaScript"); // "JavaScript JavaScript"

🔹 Splitting & Joining

"red,green,blue".split(","); // ["red", "green", "blue"]

["a", "b", "c"].join("-"); // "a-b-c"

🔹 Trimming

" hello ".trim(); // "hello"

" hello ".trimStart(); // "hello "

" hello ".trimEnd(); // " hello"

🔹 Repeat & Pad

"Hi".repeat(3);   // "HiHiHi"

"5".padStart(3, "0"); // "005"

"5".padEnd(3, "0"); // "500"

5. Template Literals

- Use backticks `.
- Supports multi-line and interpolation.

let name = "Alice";

let msg = `Hello, ${name}!

Welcome to JavaScript.`;

console.log(msg);

6. Important Points

- Strings are immutable → cannot change characters directly.

let s = "Hello";

s[0] = "Y"; // ❌ Not allowed

console.log(s); // "Hello"

- But you can create a new string instead.

```
s = "Y" + s.slice(1);

console.log(s); // "Yello"
```

✍️ Short Notes (for Revision)

- String = sequence of characters (' ', " ", ` `).
- Immutable → cannot modify characters directly.
- Properties: length.
- Methods:
  - Access: charAt(), [index]
  - Case: toUpperCase(), toLowerCase()
  - Search: indexOf(), includes(), startsWith(), endsWith()
  - Extract: slice(), substring(), substr()
  - Replace: replace(), replaceAll()
  - Split/Join: split(), join()
  - Trim: trim(), trimStart(), trimEnd()
  - Repeat/Pad: repeat(), padStart(), padEnd()
- Template literals → multi-line + interpolation.

Math in JavaScript -

1. Definition

- Math is a built-in object in JavaScript.
- Provides mathematical constants & functions.
- Not a constructor → cannot do new Math().
- All properties & methods are static → use as Math.method().

2. Math Properties (Constants)

Property	Description	Example
Math.PI	Ratio of circle's circumference to diameter	3.14159...
Math.E	Euler's number	2.718...
Math.SQRT2	√2	1.414...
Math.SQRT1_2	√½	0.707...
Math.LN2	Natural log of 2	0.693...
Math.LN10	Natural log of 10	2.302...
Math.LOG2E	$\log_2 e$	1.442...
Math.LOG10E	$\log_{10} e$	0.434...

3. Math Methods

🔹 Rounding & Absolute

```
Math.round(4.6);   // 5 (nearest integer)

Math.floor(4.9);   // 4 (down)

Math.ceil(4.1);    // 5 (up)

Math.trunc(4.9);   // 4 (remove decimal)

Math.abs(-10);     // 10 (absolute value)
```

🔹 Power & Roots

```
Math.pow(2, 3);  // 8

Math.sqrt(16);   // 4

Math.cbrt(27);   // 3
```

🔹 Min & Max

```
Math.min(2, 5, -1, 9); // -1

Math.max(2, 5, -1, 9); // 9
```

🔹 Random Number

```
Math.random();        // 0 ≤ value < 1

Math.floor(Math.random() * 10); // 0 - 9

Math.floor(Math.random() * (max - min + 1)) + min; // range
```

🔹 Trigonometry

```
Math.sin(Math.PI / 2); // 1

Math.cos(0);          // 1

Math.tan(Math.PI / 4); // 1
```

🔹 Logarithmic & Exponential

```
Math.log(1);   // 0 (natural log)

Math.log10(100); // 2

Math.log2(8);    // 3

Math.exp(1);     // e^1 = 2.718
```

🔹 Other Useful

```
Math.sign(-5);   // -1

Math.sign(0);    // 0

Math.sign(7);    // 1
```

4. Common Use Cases

- Random number generation (games, OTPs).
- Rounding values (prices, scores).
- Trigonometry in graphics/animations.
- Scientific calculations.

✍️ Short Notes (for Revision)

- Math = built-in object for mathematical operations.
- Properties: PI, E, SQRT2, LN2, LN10 ...

- **Methods:**
  - **Rounding: round(), floor(), ceil(), trunc(), abs()**
  - **Power/Roots: pow(), sqrt(), cbrt()**
  - **Min/Max: min(), max()**
  - **Random: random() (+ formula for range)**
  - **Trigonometry: sin(), cos(), tan()**
  - **Logs/Exp: log(), log10(), log2(), exp()**
  - **Others: sign()**
- **Note: Math is not a constructor → use Math.method() directly.**

Window Object -

1. Definition

- **window is the global object in browsers.**
- **Represents the browser window/tab.**
- **All global variables, functions, and objects become properties of window.**
- **In Node.js, the equivalent global object is global.**

2. Key Features of window

- **Acts as the root object for browser APIs.**
- **Provides access to:**
  - **DOM (document, location, history)**
  - **BOM (Browser Object Model: alert, prompt, confirm, etc.)**
  - **Global JS Functions (setTimeout, setInterval)**

3. Common Window Properties

Property	Description	Example
window.document	Access the HTML document (DOM).	window.document.title
window.innerHeight	Height of the viewport.	console.log(window.innerHeight)
window.innerWidth	Width of the viewport.	console.log(window.innerWidth)
window.location	URL info & navigation.	window.location.href
window.history	Browsing history object.	window.history.back()
window.navigator	Info about browser/user.	navigator.userAgent
window.screen	Screen resolution info.	screen.width

4. Common Window Methods

🔹 Dialog Boxes

```
window.alert("Hello!");      // Alert box
window.confirm("Are you sure?"); // OK/Cancel
window.prompt("Enter name:"); // Input dialog
```

🔹 Timing Functions

```
setTimeout(() => console.log("Runs once"), 2000);
setInterval(() => console.log("Runs repeatedly"), 1000);
```

🔹 Navigation

```
window.open("https://google.com"); // Opens new tab/window
window.close();                    // Closes window
```

5. Important Notes

- In JavaScript, you can often omit window since it’s the global scope.
- alert("Hi"); // same as window.alert("Hi")
- In strict mode, global variables may not always attach to window.
- In Node.js, window does not exist → use global.

✍️ Short Notes (for Revision)

- Window = Global Object in browser (represents tab/window).
- Provides DOM, BOM, and Global functions.
- Properties:
  - document, location, history, navigator, screen, innerWidth, innerHeight
- Methods:
  - Dialogs → alert(), confirm(), prompt()
  - Timing → setTimeout(), setInterval()
  - Navigation → open(), close()
- Note: Can omit window. when calling.

DOM in JavaScript -

1. Definition

- DOM = Document Object Model
- It is a programming interface for HTML/XML documents.
- Represents a webpage as a tree structure (nodes).
- Allows JavaScript to access, modify, add, or delete elements dynamically.

2. DOM Structure

- The entire webpage = Document Object.
- Document is made of nodes:
  - Element nodes → <p>, , <h1>
  - Text nodes → text inside elements
  - Attribute nodes → class, id, etc.
  - Comment nodes → <!-- comment -->

📌 Example Tree:

```
<html>

 <body>

  <h1>Hello</h1>
```

```
 </body>
</html>
```

Becomes:

```
Document
 └── html
     └── body
         └── h1
             └── "Hello"
```

3. Accessing DOM Elements

```
document.getElementById("idName");        // by ID
document.getElementsByClassName("class"); // by class
document.getElementsByTagName("p");       // by tag
document.querySelector(".class");         // first match
document.querySelectorAll("div");         // all matches
```

4. Manipulating DOM Elements

🔹 Changing Content & Style

```
document.getElementById("demo").innerHTML = "New Text";
document.getElementById("demo").style.color = "red";
```

🔹 Changing Attributes

```
document.getElementById("img").src = "pic.jpg";
```

🔹 Creating & Appending Elements

```
let newEl = document.createElement("p");
newEl.innerText = "I am new!";
document.body.appendChild(newEl);
```

🔹 Removing Elements

```
let el = document.getElementById("demo");

el.remove();
```

5. DOM Events

- DOM allows adding interactivity using events.

```
document.getElementById("btn").onclick = function() {

 alert("Button Clicked!");

};
```

Or modern way:

```
document.getElementById("btn").addEventListener("click", () => {

 alert("Clicked!");

});
```

6. Why DOM is Important?

- Makes webpages dynamic & interactive.
- Allows JS to:
  - Update content
  - Modify structure
  - Handle events
  - Create animations

✍️ Short Notes (for Revision)

- DOM = Document Object Model → tree structure of webpage.
- Nodes: element, text, attribute, comment.
- Access: getElementById, getElementsByClassName, querySelector, etc.
- Modify: innerHTML, style, setAttribute, appendChild, remove().
- Events: onclick, addEventListener.
- Importance: makes websites dynamic & interactive.

BOM in JavaScript -

1. Definition

- BOM = Browser Object Model
- Represents browser-specific objects that are not part of the HTML page.
- It allows JavaScript to interact with the browser window itself.
- Example: opening/closing new windows, navigating, screen size, history, etc.

📌 Think:

- DOM → Page (Document)
- BOM → Browser (Window, Navigator, History, etc.)

2. Main BOM Objects

🔹 window

- Global object (root of BOM).
- Example:
- alert("Hello"); // window.alert("Hello")

🔹 navigator

- Gives info about browser & system.
- console.log(navigator.userAgent);
- console.log(navigator.language);
- console.log(navigator.onLine); // true/false

🔹 location

- Represents current page URL.
- console.log(location.href); // full URL
- location.reload(); // reload page
- location.href = "https://google.com"; // redirect

🔹 history

- Browsing history of current session.
- history.back(); // go back
- history.forward(); // go forward

🔹 screen

- Information about the user's screen.
- console.log(screen.width);
- console.log(screen.height);

3. Common BOM Methods

🔹 Dialogs

```
alert("Hello!");

confirm("Are you sure?");

prompt("Enter your name:");
```

🔹 Timers

```
setTimeout(() => console.log("Runs after 2s"), 2000);

setInterval(() => console.log("Runs every 1s"), 1000);
```

🔹 Window Navigation

```
window.open("https://example.com");

window.close();
```

4. Difference between DOM and BOM

Feature	DOM	BOM
Definition	Represents HTML document	Represents browser environment
Root Object	document	window
Examples	document.getElementById()	navigator, location, history
Purpose	Manipulate webpage content	Interact with browser

✍️ Short Notes (for Revision)

- BOM = Browser Object Model → controls browser, not page.
- Main Objects:
  - window (root)
  - navigator → browser info
  - location → URL, reload, redirect
  - history → back/forward

    - screen → display info
- Methods:
    - Dialogs → alert(), confirm(), prompt()
    - Timers → setTimeout(), setInterval()
    - Window → open(), close()
- DOM vs BOM: DOM = document/page, BOM = browser environment.

Event Listener -

1. Definition

- An event listener is a function that waits for an event (like click, hover, keypress) on an element and then executes code.
- Modern way: addEventListener() method.
- Unlike inline onclick, it separates HTML & JS → better practice.

2. Syntax

element.addEventListener(event, callback, useCapture);

- event → type of event (e.g., "click", "keydown")
- callback → function to run when event occurs
- useCapture (optional, default = false) → controls event bubbling/capturing

3. Example

```
<button id="btn">Click Me</button>



<script>

document.getElementById("btn")

  .addEventListener("click", () => {

    alert("Button was clicked!");

  });

</script>
```

4. Multiple Listeners

- You can attach multiple listeners to the same element without overwriting.

```
let btn = document.getElementById("btn");


btn.addEventListener("click", () => console.log("First"));

btn.addEventListener("click", () => console.log("Second"));
```

5. Removing Listeners

- Use removeEventListener() with the same function reference.

```
function greet() {
 alert("Hello!");
}


btn.addEventListener("click", greet);

btn.removeEventListener("click", greet);
```

6. Event Object

- Every event has an object with details.

```
document.addEventListener("keydown", (event) => {
 console.log(event.key);  // which key pressed
 console.log(event.type); // "keydown"
});
```

7. Event Propagation

- Bubbling → event travels from child → parent → root. (default)
- Capturing → event travels from root → parent → child.

```
element.addEventListener("click", handler, true); // capture phase
```

8. Common Events

- Mouse: click, dblclick, mouseover, mouseout, mousemove
- Keyboard: keydown, keyup, keypress
- Form: submit, change, focus, blur

- **Window: load, resize, scroll**

✍️ Short Notes (for Revision)

- **Event Listener = waits & executes code when event occurs.**
- **Syntax: element.addEventListener("event", callback, useCapture).**
- **Supports multiple listeners (unlike onclick).**
- **Can be removed using removeEventListener().**
- **Event object → gives details (event.key, event.type).**
- **Propagation: Bubbling (default) & Capturing.**
- **Common events: mouse, keyboard, form, window events.**