* Advanced Javascript (All ES6 Versions)
  + ✔ Variable Scoping

JavaScript mai **variable scoping** ka matlab hota hai **"ek variable kis jagah tak accessible hai"**, yaani **kis area ya block ke andar aap us variable ko use kar sakte ho.**

JavaScript mai **3 types ke scope** hote hain:

**✅ 1. Global Scope**

* Jo variable function ke bahar banaya jata hai.
* Wo har jagah accessible hota hai (poore code mai).

Example:

let name = "Bilal"; // global scope

function greet() {

console.log(name); // accessible here

}

greet();

console.log(name); // accessible here too

**✅ 2. Function Scope**

* Jo variable function ke andar banaya jata hai, wo sirf us function ke andar hi accessible hota hai.

Example:

function greet() {

let message = "Hello"; // function scope

console.log(message);

}

greet();

// console.log(message); ❌ error: message is not defined

**✅ 3. Block Scope (ES6 ke baad)**

* let aur const se banaye gaye variables {} (curly braces) ke andar hi kaam karte hain.

Example:

{

let a = 10;

const b = 20;

console.log(a, b); // ✅ works here

}

// console.log(a); ❌ error

// console.log(b); ❌ error

**Note:** var block scope ko follow nahi karta, sirf function scope follow karta hai.

**🔄 Summary Table:**

| **Keyword** | **Function Scope** | **Block Scope** |
| --- | --- | --- |
| var | ✅ Yes | ❌ No |
| let | ✅ Yes | ✅ Yes |
| const | ✅ Yes | ✅ Yes |

* + ✔ Closure

**🔒 Closure kya hota hai?**

* **Closure** ka matlab hai:
* Jab koi **inner function** apne **outer function** ke variables ko **yaad rakhta hai aur access kar sakta hai**, **chahe outer function already khatam ho chuka ho** — is behavior ko closure kehte hain.

**✅ Example:**

function outerFunction() {

let name = "Bilal";

function innerFunction() {

console.log("Hello " + name); // 'name' is from outer function

}

return innerFunction;

}

const greet = outerFunction(); // outerFunction executes

greet(); // "Hello Bilal"

* Yahan innerFunction() ek **closure** hai kyun ke wo name variable ko **yaad rakhta hai**, jo outerFunction() ke andar tha, **even after** outerFunction complete ho chuki hai.

**🔍 Real-life Example: Counter**

function createCounter() {

let count = 0;

return function () {

count++;

console.log(count);

};

}

const counter = createCounter();

counter(); // 1

counter(); // 2

counter(); // 3

* Yahan count variable **closure ke through yaad rakha gaya hai**. Har dafa counter() call karne par, wo usi count ko update karta hai.

**💡 Summary:**

| **Point** | **Description** |
| --- | --- |
| Closure kya hai? | Inner function jo outer function ke variables ko access karta hai |
| Kyun useful hai? | Data ko private rakhne ke liye (encapsulation) |
| Kab banta hai? | Jab ek function ke andar doosra function define hota hai |

* + ✔ Template literals

**template literals** JavaScript ka ek feature hai jo **string banane ko easy aur powerful** banata hai. Ye **backticks (`)** ka use karta hai instead of normal quotes (' ' ya " ").

**✅ Template Literals kya hota hai?**

Template literals allow:

1. **String interpolation** (variables ko directly string ke andar lagana)
2. **Multi-line strings** (asaani se multi-line likhna)
3. **Expressions** ka use directly string ke andar

**🔤 1. Basic Syntax**

Example:

let name = "Bilal";

let message = `Hello, ${name}!`;

console.log(message); // Hello, Bilal!

${variable} ka matlab hota hai "variable ko string ke andar inject karna".

**🔁 2. Expressions bhi likh sakte ho**

Example:

let a = 5;

let b = 10;

console.log(`The sum is ${a + b}`); // The sum is 15

**📄 3. Multi-line Strings**

Example:

let msg = `This is

a multi-line

message.`;

console.log(msg);

Agar aap ' ' ya " " use karte to \n likhna padta, lekin **template literals mai seedha multi-line likh sakte ho**.

**🔐 Compare with Normal String:**

Example:

let name = "Bilal";

let normal = "Hello, " + name + "!";

let template = `Hello, ${name}!`;

console.log(normal); // Hello, Bilal!

console.log(template); // Hello, Bilal!

**💡 Use Karne Ka Faida:**

| **Feature** | **Normal String** | **Template Literal** |
| --- | --- | --- |
| Variable insert | ❌ Hard | ✅ Easy with ${} |
| Multi-line string | ❌ \n needed | ✅ No \n needed |
| Expression support | ❌ Not direct | ✅ Yes with ${} |

* + ✔ Destructuring

**Destructuring** JavaScript ka ek **short aur smart** tareeqa hai **objects ya arrays ke andar ke values ko nikaal kar variables mai rakhne** ka.

**✅ Destructuring ka Matlab:**

**Destructuring** ka matlab hai **"todna"** — yaani array ya object ko tod kar uske values ko **alag-alag variables mai assign karna**.

**🔹 1. Array Destructuring**

Example:

const arr = [10, 20, 30];

// Normal way

let a = arr[0];

let b = arr[1];

// Destructuring way

let [x, y, z] = arr;

console.log(x); // 10

console.log(y); // 20

console.log(z); // 30

**🔹 2. Object Destructuring**

Example:

const person = {

name: "Bilal",

age: 25,

};

// Normal way

// let name = person.name;

// let age = person.age;

// Destructuring way

let { name, age } = person;

console.log(name); // Bilal

console.log(age); // 25

**🔹 3. Rename During Destructuring**

Example:

const user = {

username: "bilal123",

email: "bilal@example.com",

};

let { username: userName, email: userEmail } = user;

console.log(userName); // bilal123

console.log(userEmail); // bilal@example.com

**🔹 4. Destructuring Function Parameters**

Example:

function greet({ name, age }) {

console.log(`Hello ${name}, you are ${age} years old`);

}

greet({ name: "Bilal", age: 25 });

**🔹 5. Skipping Items in Array**

Example:

const nums = [1, 2, 3, 4];

const [first, , third] = nums;

console.log(first); // 1

console.log(third); // 3

**💡 Summary:**

| **Type** | **Syntax Example** |
| --- | --- |
| Array | let [a, b] = [1, 2]; |
| Object | let {name, age} = {name: "Ali", age: 20} |
| Rename variable | let {name: myName} = person; |
| Function param | function greet({name}) {} |

* + ✔ Default parameters
* **default parameter** JavaScript ka wo feature hai jo aapko **function ke parameter ke liye ek default value set karne** ki facility deta hai — agar user koi value **na de**, to ye default value use hoti hai.

**✅ Default Parameter Kya Hota Hai?**

Jab function call karte waqt koi argument **na diya jaye**, to **function ek default value use karta hai** — is feature ko **default parameter** kehte hain.

**🔹 Syntax:**

function greet(name = "Guest") {

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

}

**🔍 Example 1: Value Di Gayi Hai**

greet("Bilal"); // Hello, Bilal

**🔍 Example 2: Value Nahi Di Gayi**

greet(); // Hello, Guest

Kyun? Kyun ke name ki default value "Guest" hai.

**🧮 Example with Numbers**

function add(a = 1, b = 2) {

return a + b;

}

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

console.log(add(5)); // 5 + 2 = 7

console.log(add()); // 1 + 2 = 3

**⚠️ Without Default Parameter (Old Way)**

Pehle aise likhte the:

function greet(name) {

name = name || "Guest";

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

}

But ab ES6 ke baad, seedha default set kar sakte ho parameter mai hi:

function greet(name = "Guest") {

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

}

**💡 Summary:**

| **Point** | **Example** |
| --- | --- |
| Default value for string | function hi(name = "Ali") |
| Default value for number | function sum(a = 0) |
| Default value used when | Argument is undefined |

* + ✔ Rest parameter

**rest parameter** JavaScript ka ek feature hai jo aapko allow karta hai ke aap **ek hi parameter mai multiple (unlimited) arguments ko collect** kar sako **as an array**.

**✅ Rest Parameter Kya Hota Hai?**

**Rest parameter** ... (3 dots) ka use karta hai, aur function ke andar jitne bhi extra arguments diye gaye ho, un sab ko **ek array ke andar collect** kar leta hai.

**🔹 Syntax:**

function funcName(...rest) {

// rest is an array

}

**🔍 Example 1: Simple Rest Parameter**

function showNames(...names) {

console.log(names);

}

showNames("Bilal", "Ali", "Ahmed");

// Output: ["Bilal", "Ali", "Ahmed"]

Yahan ...names ek array ban gaya jisme saare arguments aa gaye.

**🔍 Example 2: Rest + Regular Parameters**

function greet(greeting, ...names) {

console.log(`${greeting} to: ${names.join(", ")}`);

}

greet("Hello", "Bilal", "Ali", "Sara");

// Output: Hello to: Bilal, Ali, Sara

greeting first argument le raha hai, baaqi sab ...names mai chala gaya.

**🔍 Example 3: Sum Function with Rest Parameter**

function sum(...numbers) {

let total = 0;

for (let num of numbers) {

total += num;

}

return total;

}

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

console.log(sum(5, 10, 15, 20)); // 50

Kitne bhi numbers ho, ...numbers un sab ko array bana deta hai.

**❗ Important Note:**

* **Rest parameter hamesha last mai hota hai**

Example:

function wrong(a, ...b, c) {} // ❌ Error

**💡 Summary:**

| **Feature** | **Description** |
| --- | --- |
| Syntax | function(...args) {} |
| Result | args is an array of all arguments |
| Use Case | Jab arguments ka number unknown ho |
| Rule | Hamesha **function ke last** mai likhna hota hai |

* + ✔ Spread Operator

**spread operator (...)** JavaScript ka ek powerful feature hai jo aapko **array ya object ke elements ya properties ko "phailaane"** (spread karne) ki facility deta hai.

Yeh bilkul **rest parameter jaise dikhta hai (...)**, **lekin kaam alag karta hai**.

**✅ Spread Operator Kya Hota Hai?**

**Spread operator** ka kaam hota hai:  
**"Array ya object ke elements/properties ko ek-ek karke kisi naye array, object, ya function mai daalna."**

**🔹 1. Array Spread**

const arr1 = [1, 2, 3];

const arr2 = [4, 5];

const combined = [...arr1, ...arr2];

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

...arr1 aur ...arr2 ne dono arrays ke elements ko naya array mai phaila diya.

**🔹 2. Copy Array**

const nums = [10, 20, 30];

const copy = [...nums];

console.log(copy); // [10, 20, 30]

Yeh simple aur safe tareeqa hai array ki **copy banane ka**, bina reference share kiye.

**🔹 3. Spread in Function Arguments**

Example:

function sum(a, b, c) {

return a + b + c;

}

const values = [1, 2, 3];

console.log(sum(...values)); // 6

Array ke values ko function ke arguments mai spread kar diya.

**🔹 4. Object Spread**

Example:

const person = { name: "Bilal", age: 25 };

const updated = { ...person, city: "Karachi" };

console.log(updated);

// { name: 'Bilal', age: 25, city: 'Karachi' }

Existing object ke saath naya data add/update karne ka **best** tareeqa.

**🔹 5. Merging Objects**

Example:

const obj1 = { a: 1 };

const obj2 = { b: 2 };

const merged = { ...obj1, ...obj2 };

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

**🧠 Spread vs Rest — Difference:**

| **Feature** | **Looks Like** | **Works In** | **Purpose** |
| --- | --- | --- | --- |
| Rest parameter | ... | Function parameters | **Collect** arguments as array |
| Spread operator | ... | Arrays/Objects/Calls | **Expand** values from array/object |

**💡 Summary:**

| **Use Case** | **Example** |
| --- | --- |
| Merge arrays | [...a, ...b] |
| Merge objects | {...obj1, ...obj2} |
| Copy arrays/objects | [...arr], {...obj} |
| Function arguments | func(...array) |

* + ✔ Arrow functions

**arrow function** JavaScript ka ek **short aur modern** tareeqa hai function likhne ka. Yeh ES6 (ECMAScript 2015) mai introduce hua tha, aur isse code **simple aur readable** ban jaata hai.

**✅ Arrow Function Kya Hota Hai?**

**Arrow function** ek short syntax hai jo normal function ko likhne ka alternative hai.  
Iska symbol hota hai: => (arrow)

**🔹 Normal Function vs Arrow Function**

Example:

// Normal function

function add(a, b) {

return a + b;

}

// Arrow function

const add = (a, b) => {

return a + b;

};

**🔹 1. Short One-liner Arrow Function**

Agar function sirf **ek line ka ho** to {} aur return bhi hata sakte ho:

Example:

const square = (x) => x \* x;

console.log(square(5)); // 25

Agar **ek hi parameter** ho to () bhi optional hai:

Example:

const greet = name => `Hello, ${name}`;

console.log(greet("Bilal")); // Hello, Bilal

**🔹 2. No Parameter Function**

Example:

const sayHello = () => console.log("Hello!");

sayHello(); // Hello!

**🔹 3. Arrow Function in Arrays / Loops**

Example:

const numbers = [1, 2, 3, 4];

const doubled = numbers.map(num => num \* 2);

console.log(doubled); // [2, 4, 6, 8]

**⚠️ Arrow Function ka Important Note:**

**❌ this ka behavior change ho jaata hai**

Arrow functions **apna this nahi banate**, wo **parent ka this inherit** karte hain.

Example:

const person = {

name: "Bilal",

greet: () => {

console.log(`Hello, ${this.name}`); // ❌ undefined

}

};

person.greet();

👆 Is case mai this.name undefined ho jata hai.

Isliye arrow functions **methods** ke liye nahi use karte, balkay callbacks, array methods, short functions ke liye use karte hain.

**💡 Summary:**

| **Feature** | **Arrow Function** |
| --- | --- |
| Syntax | const fn = () => {} |
| Shorter | ✅ Yes |
| Auto return | ✅ If one line, no {} needed |
| this Binding | ❌ Does not have its own this |
| Best for | Callbacks, array methods |

* + Enhanced object literals
* **Enhanced Object Literals** JavaScript ka ek modern feature hai jo **object banane ka tareeqa asaan aur powerful** bana deta hai. Yeh feature **ES6 (ECMAScript 2015)** mai introduce hua tha.

**✅ Enhanced Object Literals Kya Hota Hai?**

* Jab aap **object banate ho**, to **shorter syntax**, **dynamic keys**, aur **functions ko directly define karne ka tareeqa** milta hai — is feature ko **Enhanced Object Literals** kehte hain.

**🔥 Features of Enhanced Object Literals:**

**1. Property shorthand**

**2. Method shorthand**

**3. Computed property names**

**🔹 1. Property Shorthand**

Agar variable ka naam aur object key ka naam same ho, to aap sirf ek naam likh saktay ho.

Example:

const name = "Bilal";

const age = 25;

// Old way

const user1 = {

name: name,

age: age

};

// Enhanced way

const user2 = {

name,

age

};

console.log(user2); // { name: 'Bilal', age: 25 }

**🔹 2. Method Shorthand**

Function ko object ke andar define karne ka chhota tareeqa.

Example:

// Old way

const user = {

greet: function () {

console.log("Hello!");

}

};

// Enhanced way

const user2 = {

greet() {

console.log("Hello!");

}

};

user2.greet(); // Hello!

**🔹 3. Computed Property Names**

Aap object ke key ko **dynamic (expression se)** bana sakte ho.

Example:

const key = "email";

const user = {

name: "Bilal",

[key]: "bilal@example.com"

};

console.log(user);

// { name: 'Bilal', email: 'bilal@example.com' }

**🧠 Summary Table:**

| **Feature** | **Example** |
| --- | --- |
| Property shorthand | { name, age } |
| Method shorthand | greet() {} |
| Computed property names | [key]: value |

**📦 Full Example:**

const city = "Karachi";

const population = 15000000;

const info = {

city,

population,

showInfo() {

console.log(`${this.city} has ${this.population} people.`);

},

["capital" + "Status"]: false

};

console.log(info);

// { city: 'Karachi', population: 15000000, showInfo: [Function: showInfo], capitalStatus: false }

info.showInfo(); // Karachi has 15000000 people.

* + Iterators & For..of
* **"Iterators"** aur **for...of loop** dono JavaScript mai **collections (like arrays, strings, etc.) ko step-by-step process karne** ke liye use hote hain.

**✅ Iterator Kya Hota Hai?**

* **Iterator** ek aisa object hota hai jo aapko **step by step** kisi collection (array, string, etc.) ke elements par loop chalane deta hai.  
  Har step par next() function se **ek value** milti hai aur pata chalta hai ke loop khatam hua ke nahi.

**🔹 Simple Example:**

Example:

const arr = [10, 20, 30];

// Iterator banaye:

const iterator = arr[Symbol.iterator]();

console.log(iterator.next()); // { value: 10, done: false }

console.log(iterator.next()); // { value: 20, done: false }

console.log(iterator.next()); // { value: 30, done: false }

console.log(iterator.next()); // { value: undefined, done: true }

done: true ka matlab ke elements khatam ho gaye hain.

**✅ for...of Loop Kya Hota Hai?**

for...of ek loop hai jo automatically **iterator use karta hai** aur **array, string, set, map** jese iterable objects ke har item ko directly access karta hai.

**🔹 Example 1: Array**

Example:

const numbers = [1, 2, 3];

for (const num of numbers) {

console.log(num);

}

// Output:

// 1

// 2

// 3

**🔹 Example 2: String**

Example:

const name = "Bilal";

for (const char of name) {

console.log(char);

}

// Output:

// B

// i

// l

// a

// l

**💡 Summary Table:**

| **Concept** | **Description** |
| --- | --- |
| **Iterator** | Object with next() method to loop items |
| **Iterable** | Any item usable with iterator (array, string, map, set) |
| **for...of** | Easy way to loop over any iterable object |
| for...in | ❗ Loops over object **keys**, not values |

**❗ for...of vs for...in (Don't confuse):**

| **Loop Type** | **Used For** | **Loops Over** |
| --- | --- | --- |
| for...of | Arrays, Strings, Sets, etc. | **Values** |
| for...in | Objects | **Keys / indexes** |

Example:

const obj = { a: 1, b: 2 };

for (const key in obj) {

console.log(key); // a, b

console.log(obj[key]); // 1, 2

}

* + Generators
  + Modules
  + Map
  + Array methods
  + Higher-order function
  + CallBack
  + Promises
  + Exponentiation Operator
  + Classes
  + Ternary Operator
  + Optional chaining
* TypeScript
  + TS Compiler
  + Type Annotations
  + Interfaces
  + Classes
  + Generics
  + Enums
  + Type Inference
  + Union and Intersection Types
  + Type Guards
  + Decorators
* Advance Github
  + Branches
  + PR
  + PR review
  + Merge
  + Rebase
  + Cherry Pick
* GSAP Animations
* Supabase
  + Authentication
    - Email
    - Google
  + Database
  + Storage