**Day 2 — Functions, Scope & Closures** 🚀

**01) Functions (basic) ----**

**Function declaration:**

* // Defining the function:

function sum(num1, num2) {

return num1 + num2;

}

// Calling the function:

sum(3, 6); // 9

* Isko **hoisting** milti hai → function ko upar likhne se pehle bhi call kar sakte ho.

**Anonymous Functions**

Anonymous functions in JavaScript do not have a name property. They can be defined using the function keyword, or as an arrow function. See the code example for the difference between a named function and an anonymous function.

// Named function

function rocketToMars() {

return 'BOOM!';

}

// Anonymous function

const rocketToMars = function() {

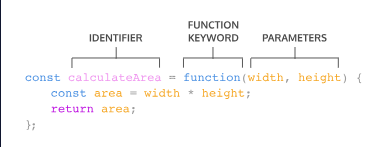
return 'BOOM!';

}

**Function expression:**

In a function expression, the function name is usually omitted. A function with no name is called an anonymous function. A function expression is often stored in a variable in order to refer to it.

* Ye **hoisted nahi** hota (agar const/let use kiya to TDZ lagega).

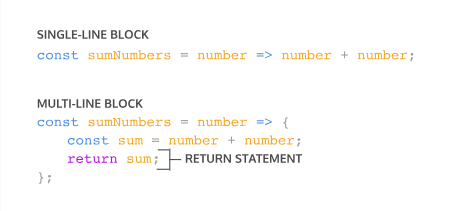


const dog = function() {

return 'Woof!';

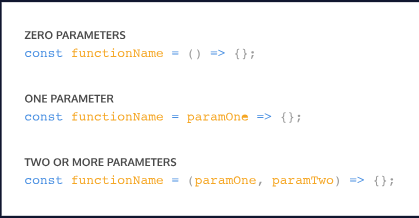
}

## 02) Arrow functions (quick difference)



## *Single-line raha toh no need of{}.*

## *Multi-line me {}+return statement chahiye!.*



## Ex:

// Arrow function with two parameters

const sum = (firstParam, secondParam) => {

return firstParam + secondParam;

};

console.log(sum(2,5)); // Prints: 7

// Arrow function with no parameters

const printHello = () => {

console.log('hello');

};

printHello(); // Prints: hello

// Arrow functions with a single parameter

const checkWeight = weight => {

console.log(`Baggage weight : ${weight} kilograms.`);

};

checkWeight(25); // Prints: Baggage weight : 25 kilograms.

**Important differences:**

### 1. Shorter syntax

// Normal

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

// Arrow

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

### 2. this behavior (lexical this)

* Normal fn → apna this banata hai.
* Arrow fn → parent ka this use karta hai.

let obj = {

name: "Bhai",

normalFn: function () {

console.log(this.name); // "Bhai"

},

arrowFn: () => {

console.log(this.name); // undefined

}

};

obj.normalFn(); // Bhai

obj.arrowFn(); // undefined

### 3. No arguments object

* Normal fn me arguments hota hai (array-like).
* Arrow fn me nahi hota → use **rest parameters** instead.

function test(a, b) {

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

}

test(1,2,3);

let arrow = (a, b) => {

console.log(arguments); // ❌ Error

};

✅ Rest parameter (modern way):

function sum(...nums) {

return nums.reduce((s, n) => s + n, 0);

}

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

### 4. Cannot be used as constructor

* Normal fn → new ke saath object ban sakta.
* Arrow fn → ❌ error (kyunki apna this nahi hota).

function Person(name) {

this.name = name;

}

let p1 = new Person("Bhai"); // ✅

let Person2 = (name) => { this.name = name; };

let p2 = new Person2("Bhai"); // ❌ TypeError

## 03) Function parameters — default & rest

**Default parameter:** *function is called* ***without a value***

function greet(name = "Dost") {

console.log("Hi " + name);

}

greet(); // Hi Dost

greet("Anu"); // Hi Anu

 If you pass name, that value is used.

 If you don’t pass anything, "Dost" is used.

**Rest parameter (collects extra args into array):**

### function sum(...nums) {

### return nums.reduce((s, n) => s + n, 0); // 👉 Array ke sabhi elements ko ek single value me convert karna (reduce karna). Matlab array → ek number / string / object / kuch bhi ek value.

### }

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

**Spread vs Rest quick:**

* ... in function definition = **rest** (collects) *yane ki sabhi argu ko ek me collect krna .*
* ... in call = **spread** (expand array into args) *Ek array ko fhaila dena alag alag element m.*

**return Keyword:**

* Functions return (pass back) values using the return keyword. return ends function execution and returns the specified value to the location where it was called. A common mistake is to forget the return keyword, in which case the function will return undefined by default.
* // With return
* function sum(num1, num2) {
* return num1 + num2;
* }
* // Without return, so the function doesn't output the sum
* function sum(num1, num2) {
* num1 + num2;
* }

## 04) Scope — global, function, block (simple)

#### *Scope* is a concept that refers to where values and functions can be accessed.

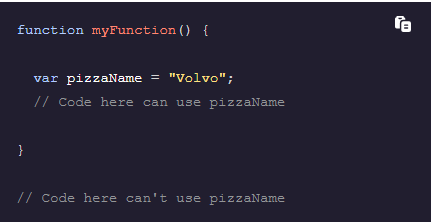
#### Various scopes include:

#### *Global* scope (a value/function in the global scope can be used anywhere in the entire program)

#### *File* or *module* scope (the value/function can only be accessed from within the file)

#### *Function* scope (only visible within the function),

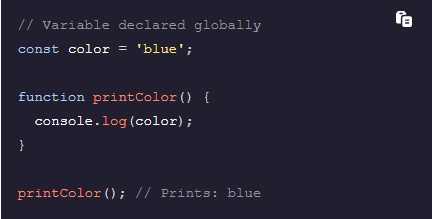
#### *Code block* scope (only visible within a { ... } codeblock)



* **Global scope:** var/let/const declared outside any function.

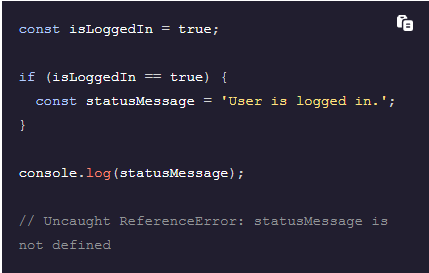
#### JavaScript variables that are declared outside of blocks or functions can exist in the global scope, which means they are accessible throughout a program. Variables declared outside of smaller block or function scopes are accessible inside those smaller scopes.

#### **Note:** It is best practice to keep global variables to a minimum.



* **Function scope:** variables declared inside a function — sirf waha valid.
* **Block scope:** {} ke andar — let/const block scoped; var nahi.

### const and let are *block scoped* variables, meaning they are only accessible in their block or nested blocks. In the given code block, trying to print the statusMessage using the console.log() method will result in a ReferenceError. It is accessible only inside that if block.



Example:

if (true) {

var x = 1;

let y = 2;

}

console.log(x); // 1

console.log(y); // ReferenceError: y is not defined

* var is function-scoped (or global if not in function) → so bahar bhi dikh jaata.
* let/const are block-scoped → safe for blocks.

**🔹 Hoisting kya hai?**

JavaScript code chalane se pehle, **variables aur functions ki declaration ko upar le aata hai** (compile phase me).  
Matlab tu likhe chahe neeche, JS unko “pehle se jaanta hai” — par **kaise jaanta hai, wo depend karta hai var / let / const / function pe**.

**🔹 Example 1: var hoisting**

Var a;

console.log(a); // undefined (kyunki declare upar ho gaya, par value assign baad me)

var a = 10;

console.log(a); // 10

**Kya hua andar se?**  
JS isko aise treat karta hai:

var a; // declaration upar aa gayi

console.log(a); // undefined

a = 10; // assignment baad me

console.log(a); // 10

**🔹 Example 2: let / const hoisting**

console.log(b); // ❌ ReferenceError (TDZ: Temporal Dead Zone)

let b = 20;

* let aur const bhi technically **hoist hote hain** (upar le jaaye jaate hain).
* Par unko **initialize** nahi kiya jaata → jab tak actual line pe nahi pahuchte, unka use karoge to error milega.
* Isko bolte hain **Temporal Dead Zone (TDZ)**.

**🔹 Example 3: Function hoisting**

sayHi(); // ✅ chalega

function sayHi() {

console.log("Hello!");

}

👉 Function declarations **fully hoist** hote hain (definition + body dono upar chali jaati hai).

Par function expressions nahi:

sayBye(); // ❌ Error: sayBye is not a function

var sayBye = function() {

console.log("Bye!");

};

**⚡ Quick Summary**

| **Keyword** | **Hoisting Behavior** |
| --- | --- |
| **Var** | Declaration hoist hoti hai, value undefined ho jaati hai. |
| **let/const** | Declaration hoist hoti hai, par TDZ me rehte hain (use karne par error). |
| **Function** | Pure function (definition + body) hoist hoti hai, kahin se bhi call kar sakte ho. |
| **function expr / arrow** | Var/let/const rules follow karte hain (expr jaisa behave karte hain). |

## *(practise que for hoisting:)*

## ❓ Q1

console.log(x);

var x = 5;

console.log(x);

👉 Output:

undefined

5

**Reason:**

* var x hoist hota hai (memory me create hota hai, default value = undefined)
* Line 1 → console.log(x) → abhi undefined
* Line 2 → assign 5
* Line 3 → console.log(x) → 5

## ❓ Q2

console.log(y);

let y = 10;

console.log(y);

👉 Output:

ReferenceError: Cannot access 'y' before initialization

**Reason:**

* let y hoist hota hai but **TDZ (Temporal Dead Zone)** me rehta hai initialization tak.
* Line 1 → access karne ki koshish → **error**
* Aage ka code execute hi nahi hoga cause error agaya.

## ❓ Q3

sayHello();

function sayHello() {

console.log("Hello World");

}

👉 Output:

Hello World

Reason:

Function declaration poora hoist hota hai memory me.

Isliye line 1 pe direct call possible.

## ❓ Q4

sayHi();

var sayHi = function() {

console.log("Hi Bhai");

};

👉 Output:

TypeError: sayHi is not a function

**Reason:**

* var sayHi hoist hota hai as undefined.
* Line 1 pe → sayHi() call karte hai → but abhi sayHi = undefined hai.
* undefined() call karne par → **TypeError**.

❓ Q5

var a = 1;

function test() {

console.log(a);

var a = 2;

console.log(a);

}

test();

👉 Output:

undefined

2

**Reason:**

* Global a = 1 hai.
* Lekin function ke andar **apna hi var a** declare hua hai (hoisted with undefined).
* Execution:
  + Line 1 (inside func) → console.log(a) → local a hai → abhi undefined
  + Line 2 → assign 2
  + Line 3 → console.log(a) → 2

## 05) this — basics (ekdum simple rules)

## *(this ka matlab hai* ***“current execution context ka object”*** *— matlab jo object ya environment function ke liye active hai, wahi this hoga.)*

* **Method call:** object ke method me this hota hai object.
* **Standalone function call:** strict mode me this = undefined, non-strict me window (browser).

### *🔹 1. Standalone Function Call*

### *function showThis() {*

### *console.log(this);*

### *}*

### *showThis();*

### *🔹 2.* ***Non-strict mode (by default in browser)***

### *Jab tu direct showThis() call karega, bina object ke,*

### *to this ka reference* ***global object*** *ban jata hai.*

### *Browser me global object = window.*

### *👉 Output:*

### *Window {...} // pura window object print hoga*

### *🔹 3.* ***Strict mode (****"use strict"****)***

### *Agar function ke top me "use strict"; likha ho,*

### *to JS bolta hai: "bhai, ab aise free me global mat pakdo".*

### *Isliye this = undefined ho jaata hai.*

### *👉 Example:*

### *"use strict";*

### *function showThis() {*

### *console.log(this);*

### *}*

### *showThis(); // undefined*

### *🔑 Why?*

### ***Non-strict*** *→ backward compatibility (purana JS aise hi tha).*

### ***Strict*** *→ safe programming (galti se global pollution avoid).*

* **Arrow function:** apna this nahi banata — surrounding this use karta hai.
* **call/apply/bind** se this set kar sakte ho.

Examples:

**1️.Method Call**

const obj = {

num: 10,

regular: function() {

console.log(this.num);

}

};

obj.regular(); // 10

**Explanation:**

* Function regular ko **obj ke through call kiya** → this = obj
* Isliye this.num = obj.num = 10

**2️. Standalone Function Call**

function standalone() {

console.log(this);

}

standalone();

**Behaviour:**

* **Non-strict mode (default browser):** this = window (global object)
* **Strict mode ('use strict'):** this = undefined

**Example:**

'use strict';

function strictFn() {

console.log(this);

}

strictFn(); // undefined

### *👉 Kyon undefined?*

### *"use strict" ke andar JavaScript bolta hai:*

### *"Agar function ko bina object ke call kiya, to this automatically global object ko point nahi karega."*

### *Matlab safety ke liye JS ne bola: “bhai, mujhe object ke bina this mat do, warna bug banega.”*

### *Isliye this = undefined.*

**3️. Arrow Function**

const obj = {

num: 10,

arrow: () => console.log(this.num)

};

obj.arrow(); // undefined (or window.num)

**Explanation:**

* Arrow function **apna this nahi banata**
* Lexical scoping use karta hai → surrounding context ka this
* Agar arrow kisi object me hai bhi, fir bhi **outer scope ka this** lega → mostly global ya undefined.

## ****4.1 call()**** – turant call, arguments comma-separated

Socho ek function hai:

function greet(greeting, punctuation) {

console.log(greeting + ', ' + this.name + punctuation);

}

const person = { name: "Amit" };

* Agar tu normal greet() call karega → this.name undefined ho jaayega.
* call ka fayda → tu manually this set kar sakta hai.

greet.call(person, "Hello", "!");

Step by step:

1. this = person → matlab function ke andar this.name = "Amit"
2. greeting = "Hello", punctuation = "!"
3. Function turant execute → console.log("Hello, Amit!")  
   ✅ Output: Hello, Amit!

**Key point:** arguments ko comma se alag-alag bhejna hai.

## ****4.2️ apply()**** – turant call, arguments array me

greet.apply(person, ["Hi", "!!"]);

* this = person → same as call
* Arguments ko **array** me pass karte hain  
  Step by step:

1. this = person → this.name = "Amit"
2. Array → [ "Hi", "!!" ] → greeting = "Hi", punctuation = "!!"
3. Function turant execute → "Hi, Amit!!"  
   ✅ Output: Hi, Amit!!

**Difference from call:**

* call → arguments alag-alag
* apply → arguments ek array me

## ****4.3️ bind()**** – turant nahi, baad me call

const sayHi = greet.bind(person, "Hi");

sayHi("!!!");

* bind ek **naya function return karta hai**, turant execute nahi karta
* this fix ho gaya → person
* Pehla argument preset → "Hi"
* Baaki arguments call ke time add kar sakte ho → "!!!"

Step by step:

1. sayHi = naya function, this = person, first arg = "Hi"
2. Call sayHi("!!!") → function executes
3. greeting = "Hi", punctuation = "!!!"
4. Output → "Hi, Amit!!!" ✅

**Key point:** bind → function ko future ke liye ready kar deta hai, turant nahi call karta

* later)

***💡 Key learning:***

***Normal function me this ka value automatic depend karta hai → kaise call kar rahe ho.***

***call/apply/bind se tum khud decide karte ho ki this kisko point kare.***

# **06. Closure ka basic idea**

**Definition (seedha):**

Jab ek function apne **outer function ke variables ko yaad rakh ke use karta hai**, to ye **closure** kehlata hai.

* Matlab inner function ko **outer function ka scope yaad rehta hai**, chahe outer function execute ho ke chala bhi gaya ho.
* Ye ek tarah se **private variable** jaisa kaam karta hai.

## Example – Counter

function makeCounter() {

let count = 0; // private variable

return function() {

count++;

return count;

};

}

const c = makeCounter();

console.log(c()); // 1

console.log(c()); // 2

console.log(c()); // 3

### Step by step:

1. makeCounter() call hota hai
   * count = 0 create hua
   * Ek **inner function** return hota hai
2. makeCounter() chala ke gaya, **par inner function ke paas count yaad hai**
3. c() call karte hi:
   * First call → count = 1
   * Second call → count = 2
   * Third call → count = 3

💡 Point: **count private hai** — koi bahar se directly change nahi kar sakta.

# 🔹 Problem: var vs let in loops + closure

### Case 1 – var ke saath

for (var i = 1; i <= 3; i++) {

setTimeout(() => console.log(i), 100);

}

**Step by step (what happens in memory):**

1. var i **ek hi variable hai** jo pura loop share karta hai
   * Matlab iteration 1, 2, 3 → sab same i use karenge
2. setTimeout ek **asynchronous callback** hai
   * Matlab 100ms ke baad function call hoga
3. Jab 100ms ke baad callback run hota hai → loop already complete ho chuka
   * i = 4 (loop exit condition)
4. Isliye **sab callbacks** same i ko access karenge → 4,4,4

✅ Output:

4

4

4

### Case 2 – let ke saath

for (let i = 1; i <= 3; i++) {

setTimeout(() => console.log(i), 100);

}

**Step by step:**

1. let i **har iteration ke liye alag binding create karta hai**
   * Iteration 1 → i=1
   * Iteration 2 → i=2
   * Iteration 3 → i=3
2. setTimeout ke andar closure capture kar leta hai **us iteration ka i**
   * Matlab callback ke paas apni copy hoti hai
3. 100ms ke baad callback call hota hai → correct i print hota hai

✅ Output:

1

2

3

PRACTISE:

## Q1. Multiply with arguments

function multiply() {

// arguments use karke product nikal

}

console.log(multiply(2, 3, 4)); // 24

👉 Ab isko **rewrite kar** using ...rest.

OUTPUT:

function multiply(…nums) {

return nums.reduce((acc,curr) => acc\*curr,1);

}

Console.log(multiply(2,3,4)); //24

## Q2. Arrow + Outer Arguments

(function(x, y){

const arrow = () => console.log(arguments.length);

arrow();

})(10, 20);

👉 Output guess kar! [2]

## Q3. Rest parameters with named args

function show(a, ...rest) {

console.log(a);

console.log(rest);

}

show(1, 2, 3, 4);

👉 Output? [1 nd 2,3,4]

## Q4. Arguments vs Rest difference

function compare(a, b) {

console.log(arguments[2]); //15

}

compare(5, 10, 15);

function compareRest(a, b, ...rest) {

console.log(rest[0]); //15

}

compareRest(5, 10, 15);

👉 Dono me kya print hoga?

## Q5. Arrow + Rest

const sum = (...nums) => nums.reduce((s,n)=>s+n,0);

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

**SHORT NOTES:**

| **Topic** | **Normal Function** | **Arrow Function** | **Notes / Tips** |
| --- | --- | --- | --- |
| **Declaration / Expression** | Function declaration → hoisted (can call before definition) Function expression → not hoisted if const/let | Always expression → not hoisted | Anonymous fn usually expression |
| **Syntax** | function sum(a,b){ return a+b; } | const sum = (a,b) => a+b | Single-line → implicit return |
| **this behavior** | Dynamic → depends on **caller** | Lexical → uses surrounding scope | Arrow fn inside object → cannot access object via this |
| **arguments object** | ✅ Available | ❌ Not available | Use ...rest in arrow |
| **Constructor / new** | ✅ Can use new | ❌ Cannot use new | Arrow fn has no this binding |
| **Hoisting** | ✅ Declaration + body hoist | ❌ Follows var/let rules | TDZ for let/const |
| **Scope** | Function scoped (var inside fn) | Same | Use let/const for block-scope |
| **Default & Rest Params** | function greet(name="Dost") | const sum = (...nums) => nums.reduce((a,b)=>a+b,0) | Rest collects extra args, Spread expands array |
| **call / apply / bind** | Can manually set this | Can still use but mostly normal fn | call → args comma-separated, apply → args array, bind → returns new fn |
| **Closures** | Inner fn can access outer scope variables | Same | Can create private variables / counters |

💡 **Quick tips for interview:**

1. Var vs let inside loops → closure traps (var → all callbacks same, let → each callback its own value).
2. Arrow fn + this → always think **lexical.**
3. Function expressions → hoisting trap → TypeError if called too early.
4. call/apply/bind → arguments passing difference yaad rakhna.

**INDEX:**

## ****Day 2 – Functions, Scope & Closures (ongoing)****

1. **Functions Basics**
   * Function declaration vs function expression
   * Hoisting (function declarations hoist hote hain, expressions nahi)
   * Scope basics (global, function, block)
2. **Arrow Functions**
   * Shorter syntax
   * this → lexical (surrounding scope ka this use hota hai)
   * arguments object ❌ nahi hota
   * Constructor ke saath use ❌ (new ArrowFn())
3. **Arguments & Rest Parameters**
   * Normal function me arguments object hota hai
   * Arrow function me nahi → solution: ...rest parameters
   * Example: multiply(...args).reduce()
4. **call, apply, bind**
   * call(this, a, b) → immediate, args comma separated
   * apply(this, [a, b]) → immediate, args array me
   * bind(this, a, b) → new function return, later call
5. **Closures (intro)**
   * Definition: function + lexical environment
   * Counter example (private variable)
   * Bank account example (deposit, withdraw, getBalance)
   * Loops me closure (var vs let)
     + var → sab callbacks same value (4,4,4).
     + let → har iteration apna alag value (1,2,3).