JavaScript Foundation – Training Day-3

```
<script>
if(age > 19){
    alert(“Adult”);
} else{
    alert(“Teenager”);
}
</script>
```

Course content and duration:

Duration: 16 hours | Schedule: 8 days @ 2 hours/day

S. No	Day	Module	Topics
1	WED (10/12/2025)	Module 1: Introduction to JavaScript and Basics	JavaScript Overview, Syntax, and Variables
2	FRI (12/12/2025)	Module 2: Control Flow and Loops	Conditional Statements and Iteration
3	MON (15/12/2025)	Module 3: Functions and Scope	Function Declaration, Expressions, and Scope
4	TUE (16/12/2025)	Module 4: Arrays and Array Methods	Array Manipulation and Higher-Order Functions
5	WED (17/12/2025)	Module 5: Objects and Object-Oriented Programming	Objects, Properties, Methods, and Prototypes
6	THU (18/12/2025)	Module 6: DOM Manipulation and Events	Document Object Model and Event Handling
7	FRI (19/12/2025)	Module 7: Asynchronous JavaScript	Callbacks, Promises, and Async/Await
8	MON (22/12/2025)	Module 8: ES6+ Features and Best Practices	Modern JavaScript Features and Code Quality

Day3: Function Declaration, Expressions, and Scope (JavaScript)

Contents

What is a Function?

A function is a reusable block of code designed to perform a specific task.

```
// Function = input → processing → output
```

Function Declaration:

A **function declaration** defines a named function using the function keyword.

```
function functionName(parameters) {
  // function body
  return value;
}
```

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

console.log(add(5, 3)); // 8
```

Key Characteristics

✅ Has a **name**
✅ **Hoisted** (can be called before it is defined)
✅ Commonly used for reusable, general-purpose logic

Function Expression:

A **function expression** stores a function inside a variable.

```
const functionName = function(parameters) {
  // function body
};
```

```
const multiply = function(a, b) {
  return a * b;
};

console.log(multiply(4, 5)); // 20
```

Key Characteristics

❌ Not hoisted
✅ Can be anonymous
✅ Useful for conditional logic and callbacks

Named vs Anonymous Function Expressions

Anonymous Function Expression

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

Named Function Expression

```
const greet = function sayHello() {
  console.log("Hello");
};
```

Comparison Example:

Using Function Declaration

```
function calculateTotal(price, tax) {
  return price + tax;
}
```

Using Function Expression

```
const calculateTotal = function(price, tax) {
  return price + tax;
};
```

Both work the same **after definition**, but differ in **hoisting**.

Function Declaration is like a **permanent position** —available anytime.
Function Expression is like a **contract position** —available only after declaration.

When to use **Function Declaration** and **Function Expression**

- Use **function declarations** for reusable logic
- Use **function expressions** for callbacks and conditional functions
- Hoisting is the **main difference**
- Expressions give **more control**

Next topic: Arrow functions

Arrow Functions in JavaScript (ES6)

What is an Arrow Function?

An **arrow function** is a **shorter and more modern syntax** for writing functions in JavaScript, introduced in **ES6 (2015)**.

It is mainly used for:

- Cleaner code
- Short callbacks
- Preserving this context

Basic Syntax

Traditional Function

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

Arrow Function

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

Shorter Version

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

Arrow functions replace the function keyword with =>, making code concise and readable.

Anatomy of an Arrow Function

```
(parameters) => { function body }
```

- () → parameters
- => → arrow
- {} → function body

Arrow Function Variations:

a) No Parameters

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

b) Single Parameter (Parentheses Optional)

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

Parentheses can be omitted if **only one parameter** exists.

c) Multiple Parameters

```
const multiply = (a, b) => a * b;
```

d) Multi-line Function Body

```
const calculate = (a, b) => {
  const sum = a + b;
  return sum;
};
```

If {} are used, **return must be explicit**.

e) Implicit Return (Single Expression)

```
const greet = name => "Hello " + name;
```

Arrow Functions vs Function Expressions

Function Expression:

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

Arrow Function:

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

Arrow functions are:

- Shorter
- Cleaner
- Easier for callbacks

Arrow Functions and this (Very Important)

Traditional Function (this depends on caller)

```
const obj = {
  value: 10,
  show: function() {
    console.log(this.value);
  }
};
```

Arrow Function (this is lexical)

```
const obj = {
  value: 10,
  show: () => {
    console.log(this.value);
  }
};
```

Key Rule:

Arrow functions **do not have their own this**
They inherit this from the surrounding scope.

Where Arrow Functions Are Best Used

✓ Callbacks

```
function greet(name, callback) {
  console.log("Hello " + name);
  callback();
}

function sayBye() {
  console.log("Goodbye!");
}

greet("Srikanth", sayBye);
```

✓ Array methods

```
const numbers = [1, 2, 3, 4];
const squares = numbers.map(n => n * n);
```

Where Arrow Functions Should NOT Be Used

Object methods (when this is needed)

```
const Person = (name) => {
  this.name = name;
};
```

Arrow functions cannot be constructors

What is a Constructor?

A **constructor function** is used to create objects using the new keyword.

```
function Person(name) {
  this.name = name;
}
const p1 = new Person("Ram");
console.log(p1.name); // Ram
```

✓ Works because:

- Person has its own this
- JavaScript sets up a new object when new is used

Trying the Same with an Arrow Function

```
const Person = (name) => {
  this.name = name;
};
const p1 = new Person("Ram");
```

Reason:

Arrow functions do **not** have a prototype and do **not** create their own this.

Because constructors rely on:

1. A new object being created
2. this pointing to that new object
3. A prototype chain

Next topic: Parameters vs Arguments

Parameters vs Arguments:

Parameters

Defined in function definition.

Arguments

Values we pass during function call.

```javascript
function multiply(a, b) {  // a, b = parameters
  return a * b;
}

multiply(5, 2, 4); // 5, 2 = arguments
```

JavaScript allows fewer or more arguments than parameters; extras go to arguments (for non-arrow functions) or can be collected with rest syntax (...rest).

```javascript
function greet(name) {
  console.log("Hello " + name);
}
greet("Ravi"); // Hello Ravi
greet();       // Hello undefined
```

arguments

```javascript
function showArgs() {
  console.log(arguments[0], arguments.length);
}
showArgs("a", "b"); // "a", 2
```

Default values combined with destructuring:

```javascript
function connect({ host = 'localhost', port = 80 } = {}) {
  console.log(host, port);
}
connect(); // "localhost", 80
connect({ host: 'example.com' }); // "example.com", 80
```

Rest parameters (preferred):

```
function sum(...nums) {
  return nums.reduce((s, n) => s + n, 0);
}
console.log(sum(1,2,3)); // 6
```

This code uses reduce() to add all numbers in the array, starting from 0.

reduce() is an **array method** used to **reduce an array to a single value**.

Common uses:

- sum of numbers
- product
- flatten arrays
- build objects

It has 2 main arguments:

1. **Callback function** → (s, n) => s + n
2. **Initial value** → 0

This callback runs **once for each array element**.

Parameter	Meaning
s	accumulator (running total)
n	current array element

On each step: s = s + n

s starts at 0 **Initial Value (0)**

This is important for safety and clarity

Next topic: Return values and default parameters

Return values and default parameters

Explanation

- return stops execution and gives a value back.
- No return → function returns undefined.
- Default parameters provide fallback values.

Examples

Early return pattern:

```
function divide(a, b) {
  if (b === 0) return null; // early guard
  return a / b;
}
```

Default params and expressions:

```
function multipl(a = 1, b = a) {
  return a * b;
}
console.log(multipl());      // 1
console.log(multipl(5));     // 25 (b defaults to a)
```

Return of complex values:

```
function createUser(name) {
  return { id: Date.now(), name };
}
const u = createUser("Me");
```

If a function reaches return, it sends back a value and exits;

if it reaches the end without return, JavaScript returns undefined.

What does return do?

- Sends a value **back to the caller**
- **Immediately stops** function execution

Example

```
function add(a, b) {
  return a + b;
  console.log("This will NOT run");
}

const result = add(2, 3);
console.log(result);
```

Falling Off the End of a Function

If a function **does not explicitly return anything**, JavaScript **automatically returns undefined**.

Example

```
function add(a, b) {
  const sum = a + b;
}

const result = add(2, 3);
console.log(result);
```

Conditional Return vs Falling Off

```
function checkAge(age) {
  if (age >= 18) {
    return "Allowed";
  }
}

console.log(checkAge(20));
console.log(checkAge(15));
```

Missing return paths lead to **unexpected undefined**.

Always return **something meaningful** if the function is expected to produce a value.

Better Version

```
function checkAge(age) {
  if (age >= 18) {
    return "Allowed";
  }
  return "Not allowed";
}
```

side effects vs pure functions

What is a Side Effect?

A side effect happens when a function:

- Changes something outside itself, or
- Depends on external state

📌 Examples of side effects:

- Modifying global variables
- Changing object/array arguments
- Logging to console
- Making API calls
- Updating DOM

Function WITH Side Effects

Global variable modification

```
let total = 0;
function addToTotal(n) {
  total = total + n; // side effect
}
addToTotal(5);
addToTotal(3);

console.log(total);
```

Why this has side effects Function changes external state (total)

What is a Pure Function?

A **pure function**:

1. Returns the **same output** for the same input
2. Has **no side effects**

✓ No external state
✓ No mutation

Example: Pure Function

```
function add(a, b) {
  return a + b;
}
add(2, 3); // always 5
```

Why this is pure

- Output depends only on inputs
- Does not modify anything outside

Next topic: Function scope and closures

Function scope and closures

What is Scope?

Scope determines **where a variable can be accessed** in your code.

In JavaScript, scope: From where can I use this variable?

Types of Scope (Quick Overview)

Scope Type	Description
Global Scope	Accessible everywhere
Function Scope	Accessible only inside the function
Block Scope	Accessible only inside {} (let/const)
Lexical Scope	Scope determined by code location

Function Scope (Core Concept)

Definition

A variable declared **inside a function** is **function-scoped** and **cannot be accessed outside** that function.

```
function greet() {
  let message = "Hello";
  console.log(message); // ✅ Accessible
}

greet();
console.log(message); // ❌ ReferenceError
```

Key Rule

Variables declared inside a function live only inside that function.

Function Scope with var, let, and const

Using var

var is **function-scoped**, not block-scoped.

```
function testVar() {
  if (true) {
    var x = 10;
  }
  console.log(x); // ✅ 10
}
```

Using let / const

let and const are **block-scoped**.

```
function testLet() {
  if (true) {
    let y = 20;
  }
  console.log(y); // ❌ ReferenceError
}
```

Interview Tip

var ignores blocks, but **respects functions**.

Global Scope vs Function Scope

```
let count = 0; // Global

function increment() {
  let count = 10; // Function scope
  console.log(count);
}

increment();       // 10
console.log(count); // 0
```

Explanation

- Inner count **shadows** outer count
- They are **different variables**

Nested Functions and Scope Chain

JavaScript uses **lexical scoping**.

```
function outer() {
  let outerVar = "I am outer";

  function inner() {
    console.log(outerVar); // ✅ Access outer scope
  }

  inner();
}

outer();
```

Scope Chain

inner()

→ outer()

→ global

JavaScript searches **inside** → **outside** → **global**

What is a Closure?

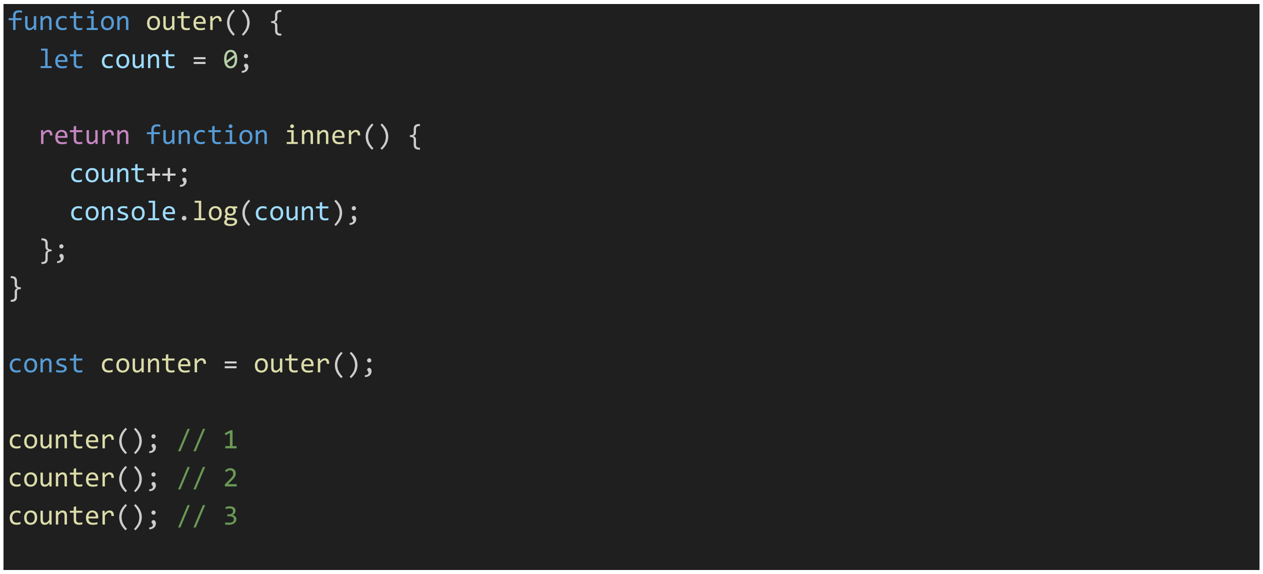
Simple Definition

A **closure** is created when a function **remembers variables from its outer scope**, even after the outer function has finished executing.

Formal Definition

A closure is a function bundled together with its lexical environment.

Closure Example (Classic)



```
function outer() {
  let count = 0;

  return function inner() {
    count++;
    console.log(count);
  };
}

const counter = outer();

counter(); // 1
counter(); // 2
counter(); // 3
```

What's Happening?

1. outer() runs and returns inner
2. outer() finishes execution
3. inner() **still remembers count**
4. count is preserved in memory

This memory retention = closure

Visualizing a Closure:

```
counter → inner()
           |
           └── count = 0 → 1 → 2 → 3
```

Even though outer() is gone, count lives on.

Why Closures Are Powerful

Data Encapsulation (Private Variables)

```
function createBankAccount() {
  let balance = 0;

  return {
    deposit(amount) {
      balance += amount;
    },
    getBalance() {
      return balance;
    }
  };
}

const account = createBankAccount();
account.deposit(100);
console.log(account.getBalance()); // 100
```

balance cannot be accessed directly.

Common Closure Mistake (Loop Problem)

Using var

```
for (var i = 1; i <= 3; i++) {
  setTimeout(() => console.log(i), 1000);
}
// Output: 4 4 4
```

var is function-scoped

One shared i

Fix with let

```
for (let i = 1; i <= 3; i++) {
  setTimeout(() => console.log(i), 1000);
}
// Output: 1 2 3
```

Each iteration gets its **own closure**.

Closure vs Normal Function

Feature	Normal Function	Closure
Access outer variables	❌	✅
Preserves state	❌	✅
Memory retention	❌	✅

Performance Note (Important)

Closures **keep variables in memory**

- Avoid creating unnecessary closures in loops
- Release references when not needed

Next topic: Global vs Local Scope in JavaScript

Global vs Local Scope in JavaScript

What is Scope?

Scope defines **where a variable can be accessed** in your program.

“Who can see this variable?”

Global Scope

Definition

A variable declared **outside of all functions and blocks** is in **global scope**.

```
let appName = "MyApp"; // Global variable

function showApp() {
  console.log(appName); // ✅ Accessible
}

showApp();
console.log(appName); // ✅ Accessible
```

Key Characteristics

✅ Accessible **anywhere**
❌ Can be modified from anywhere
⚠ Risk of name collisions

Local Scope

Definition

A variable declared **inside a function or block** is **local** to that scope.

```
function login() {
  let user = "Admin"; // Local variable
  console.log(user); // ✅ Accessible
}

login();
console.log(user); // ❌ ReferenceError
```

Rule

Local variables **exist only inside their scope**.

Function Scope (Local Scope Type)

Variables declared with var, let, or const **inside a function** are **function-scoped**.

```
function calculate() {
  var x = 10;
  let y = 20;
  const z = 30;

  console.log(x, y, z); // ✅
}
calculate();
console.log(x); // ❌
```

Block Scope vs Function Scope

Block Scope (let, const)

```
if (true) {
  let a = 5;
  const b = 10;
}

console.log(a); // ❌
console.log(b); // ❌
```

Function Scope (var)

```
function test() {
  if (true) {
    var x = 100;
  }
  console.log(x); // ✅
}
```

⚠ var **ignores block scope**

Global vs Local Scope – Side by Side

```
let count = 1; // Global

function update() {
  let count = 5; // Local (shadows global)
  console.log(count);
}

update();           // 5
console.log(count); // 1
```

Scope Chain (Very Important)

JavaScript looks for variables in this order:

Local scope

→ Parent scope

→ Global scope

```
let site = "Google";

function outer() {
  let page = "Home";

  function inner() {
    console.log(site); // Global
    console.log(page); // Outer local
  }

  inner();
}
```

Accidental Global Variables (Common Mistake)

❌ Without let, var, or const

```
function test() {
  score = 100; // ❌ Becomes global!
}
test();
console.log(score); // 100
```

Real-World Analogy

🏠 **Global Scope** → Living room (everyone can access)
🛏 **Local Scope** → Bedroom (private access)

Next topic: Hoisting Concepts

Hoisting Concepts in JavaScript (In-Depth Explanation)

Hoisting is one of the most **important JavaScript concepts**.

What is Hoisting?

Hoisting is JavaScript's behavior of moving declarations to the top of their scope during the compilation phase.

⚠️ Important:

- **Only declarations are hoisted**
- **Assignments are NOT hoisted**
- Hoisting happens **before code execution**

Think of JavaScript as doing **two passes**:

1. **Memory Creation Phase (Hoisting)**
2. **Execution Phase**

JavaScript Execution Context & Hoisting

Every time JavaScript runs code, it creates an **Execution Context**.

Memory Creation Phase

- Variables are registered
- Functions are registered
- Space is allocated in memory

Execution Phase

- Code is executed line by line
- Values are assigned
- Functions are called

Hoisting with var

Example

```
console.log(a);
var a = 10;
// What actually happens internally
var a;          // hoisted
console.log(a);
a = 10;
```

Explanation

- var a is hoisted
- Value assignment (= 10) stays in place
- Variable exists but has value undefined

✅ **var is hoisted and initialized with undefined**

Hoisting with let and const

Example

```
console.log(b);
let b = 20;
```

Output

ReferenceError: Cannot access 'b' before initialization

Why?

- let and const are **hoisted**
- BUT they are placed in the **Temporal Dead Zone (TDZ)**

Temporal Dead Zone (TDZ)

TDZ is the time between variable creation and initialization where access is forbidden.

```
// TDZ starts
console.log(x); // ❌ Error
let x = 5;      // TDZ ends
```

✅ Prevents bugs caused by accidental early access

✅ Makes code more predictable

Hoisting Comparison: var vs let vs const

Feature	var	let	const
Hoisted	✅ Yes	✅ Yes	✅ Yes
Initialized during hoisting	✅ undefined	❌ No	❌ No
Temporal Dead Zone	❌ No	✅ Yes	✅ Yes
Re-declaration allowed	✅ Yes	❌ No	❌ No
Block scoped	❌ No	✅ Yes	✅ Yes

Function Hoisting

Function Declaration (Fully Hoisted)

```
sayHello();

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

✅ Works perfectly

📌 Function declarations are **fully hoisted (body + name)**

Function Expression (NOT Fully Hoisted)

```
sayHi();

var sayHi = function () {
  console.log("Hi!");
};
```

Output

TypeError: sayHi is not a function

Why?

- var sayHi is hoisted as undefined
- Function assignment happens later

Arrow Function Hoisting

```
greet();

const greet = () => {
  console.log("Hello");
};
```

❌ ReferenceError (TDZ)

Function Hoisting Summary

Function Type	Hoisted?	Callable before definition?
Function declaration	✅ Yes	✅ Yes
Function expression (var)	Partial	❌ No
Arrow function (let/const)	Partial	❌ No

Global vs Local Hoisting

```
var x = 10;

function test() {
  console.log(x);
  var x = 20;
}

test();
```

Output

undefined

Explanation

- Local var x is hoisted inside test
- Shadows global variable

Best Practices to Avoid Hoisting Issues

✅ Always declare variables at the **top of the block**

✅ Prefer **let and const over var**

✅ Define functions **before using them**

✅ Avoid relying on hoisting behavior

✅ Use **strict mode**

"use strict";

Next topic: Callback Functions

Callback Functions – Introduction (JavaScript)

A **callback function** is a **core JavaScript concept** and the foundation for understanding **asynchronous programming**.

What is a Callback Function?

A callback function is a function that is passed as an argument to another function and is executed later.

📌 In simple words:

- You **don't call** the function yourself
- You **give it to another function**
- That function **calls it back** at the right time

Why Do We Need Callbacks?

JavaScript:

- Is **single-threaded**
- Executes code **one task at a time**

Callbacks allow JavaScript to:

- Handle **async operations**
- Avoid blocking execution
- Run code **after** something finishes (timer, API, event)

Basic Callback Example (Synchronous)

```
function greet(name, callback) {
  console.log("Hello " + name);
  callback();
}

function sayBye() {
  console.log("Goodbye!");
}

greet("Srikanth", sayBye);
```

Output

Hello Srikanth

Goodbye!

✅ sayBye is passed
✅ Executed **inside** greet

Callback with Anonymous Function

```javascript
function calculate(a, b, operation) {
  operation(a, b);
}

calculate(5, 3, function (x, y) {
  console.log(x + y);
});
```

Arrow Function as Callback

```javascript
calculate(10, 5, (x, y) => {
  console.log(x * y);
});
```

✔ Cleaner
✔ More readable

Asynchronous Callback Example (setTimeout)

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

setTimeout(() => {
  console.log("Inside callback");
}, 2000);
console.log("End");
```

Output

Start

End

Inside callback

📌 Callback runs **after delay**, not immediately

Callback vs Normal Function

Feature	Normal Function	Callback Function
Called directly	✅ Yes	❌ No
Passed as argument	❌ No	✅ Yes
Execution timing	Immediate	Controlled by another function
Common usage	General logic	Async & event handling

Callback Hell (Problem)

When callbacks are **nested deeply**, code becomes hard to read.

```
setTimeout(() => {
  console.log("Task 1");
  setTimeout(() => {
    console.log("Task 2");
    setTimeout(() => {
      console.log("Task 3");
    }, 1000);
  }, 1000);
}, 1000);
```

How to Avoid Callback Hell

✅ Use **named functions**

✅ Use **Promises** (A Promise represents a value that will be **available now, later, or never**.)

✅ Use **async / await**

```
async function runTasks() {
  await task1();
  await task2();
  await task3();
}
```