## JavaScript 2025

Developers HandBook

## ✅ JavaScript Deep Dive Curriculum (2025 Edition)

### 📘 Chapter Flow

1. Introduction to JavaScript & Setup
2. Data Types & Type Conversion
3. Variables: let, const, var
4. Operators (Arithmetic, Logical, Comparison)
5. **Conditional Statements** 🔥
6. **Loops & Iteration** 🔥
7. **Switch Statements** 🔥
8. **Arrays: Basics to Advanced** 🔥
9. Strings & String Methods
10. Objects & Nested Structures
11. Functions (Complete ✅)
12. DOM Manipulation
13. Events & Event Delegation
14. Error Handling & Debugging
15. ES6+ Features Overview
16. Modules & Imports
17. Local Storage & Session Storage
18. JSON & Fetch API
19. Async JavaScript (Complete ✅)
20. Building Interactive UI Logic (for React readiness)

## ✅ JavaScript Introduction & Setup (2025)

📘 Section 1: INTRODUCTION TO JAVASCRIPT

🔹 1. What is JavaScript?  
A high-level, interpreted programming language primarily used for web development.  
• Runs in the browser and server (via Node.js).  
• Dynamically typed, prototype-based, single-threaded with event-driven concurrency.

🔹 2. JavaScript Evolution  
• ES5 (2009): The classic version most browsers supported for years.  
• ES6 / ES2015+: Introduced let, const, arrow functions, classes, modules, etc.  
• Modern JS (ES2020+): Top-level await, nullish coalescing, optional chaining, etc.

## ✅ JavaScript Data Types & Variables (2025)

📘 Section 2: DATA TYPES & VARIABLES

🔹 1. Primitive Data Types  
• string, number, boolean, undefined, null, bigint, symbol

let name = "LoneCoder"; // string

let age = 25; // number

let isActive = true; // boolean

let score; // undefined

let empty = null; // null

let big = 12345678901234567890n; // bigint

let unique = Symbol("id"); // symbol

🔹 2. Non-Primitive Types  
• Object, Array, Function – All derived from Object.

🔹 3. typeof Operator  
Identifies the type of a value.

console.log(typeof name); // "string"

console.log(typeof null); // "object" (legacy quirk)

🔹 4. Variable Declaration: var, let, const  
• var: function-scoped, hoisted (not block-scoped)  
• let: block-scoped, no hoisting issues  
• const: block-scoped, cannot be reassigned

let x = 5;

x = 6; // ✅

const y = 10;

y = 12; // ❌ Error

🔹 5. Dynamic Typing  
JavaScript allows variables to change type at runtime.

let value = 42;

value = "Now I'm a string!";

🔹 6. Null vs Undefined  
• undefine   
• null: a deliberate assignment of "no value".

🧠 Assignments: Data Types

1. Declare one variable of each primitive type and log their types.
2. Use typeof to check the type of null, NaN, and a function.
3. Try changing a let variable from a number to a string.
4. Explain in comments why typeof null is "object".
5. Create an object with at least 3 properties, then access one using dot notation.

## ✅ JavaScript Operators Deep Dive (2025)

📘 Section 3: OPERATORS

🔹 1. Arithmetic Operators

+, -, \*, /, %, \*\* (exponentiation)

let sum = 5 + 3; // 8

🔹 2. Assignment Operators

=, +=, -=, \*=, /=, %=

let x = 10;

x += 5; // 15

🔹 3. Comparison Operators

== (loose equality)

=== (strict equality)

!=, !==, >, >=, <, <=

🔹 4. Logical Operators

&& (AND), || (OR), ! (NOT)

🔹 5. Ternary Operator

let age = 18;

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

🔹 6. Nullish Coalescing (??)  
Returns right-hand side only if the left-hand side is null or undefined.

let username = null;

let display = username ?? "Guest"; // "Guest"

🔹 7. Optional Chaining (?.)  
Safely access deeply nested properties.

let user = {};

console.log(user.profile?.name); // undefined, doesn't throw error

🔹 8. Type Coercion in Operators

console.log("5" + 1); // "51"

console.log("5" - 1); // 4

console.log("5" \* "2"); // 10

🧠 Assignments: Operators

1. Use arithmetic operators to calculate area of a circle.
2. Write a ternary expression that checks if a number is even or odd.
3. Demonstrate use of ?? with null, undefined, and 0.
4. Log the results of "5" + 5, "5" - 5, true + true.
5. Use optional chaining to safely access a nested object value.

## 🔥 Chapter 5: Conditional Statements in JavaScript

Conditional logic is how your program makes decisions. JS provides if, else if, else, and ternary for branching logic.

### 🔹 1. if Statement

let score = 85;

if (score > 80) {

console.log("Excellent");

}

### 🔹 2. if...else

let age = 20;

if (age >= 18) {

console.log("Adult");

} else {

console.log("Minor");

}

### 🔹 3. else if Ladder

let marks = 75;

if (marks >= 90) {

console.log("Grade A");

} else if (marks >= 70) {

console.log("Grade B");

} else {

console.log("Grade C or below");

}

### 🔹 4. Nesting Conditions

let isLoggedIn = true;

let isAdmin = false;

if (isLoggedIn) {

if (isAdmin) {

console.log("Show admin panel");

} else {

console.log("Show user dashboard");

}

}

### 🔹 5. Ternary Operator

Shorter if...else for assignments or inline decisions.

let age = 25;

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

console.log(status); // Adult

### 🔹 6. Logical Operators in Conditions

let isMember = true;

let hasCoupon = false;

if (isMember || hasCoupon) {

console.log("You get a discount!");

}

### 🧠 Assignments: Conditional Statements

1. Write a program that checks if a number is positive, negative, or zero.
2. Use if...else to determine if a person can vote based on age.
3. Build a grade calculator using if...else if...else.
4. Use a nested if to check if a user is logged in and has admin access.
5. Rewrite a simple if...else using the ternary operator.

## 🔄 Chapter 6: Loops & Iteration in JavaScript

Loops help you repeat actions—whether iterating over data or executing a block multiple times.

### 🔹 1. for Loop

Most common loop for precise repetitions.

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

console.log("Step:", i);

}

### 🔹 2. while Loop

Runs while the condition is true.

let count = 0;

while (count < 3) {

console.log("Count:", count);

count++;

}

### 🔹 3. do...while Loop

Runs at least once even if condition is false.

let num = 1;

do {

console.log("Value:", num);

num++;

} while (num <= 3);

### 🔹 4. break Statement

Exits the loop immediately.

for (let i = 1; i <= 10; i++) {

if (i === 5) break;

console.log(i); // 1 to 4

}

### 🔹 5. continue Statement

Skips current iteration and continues loop.

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

if (i === 3) continue;

console.log(i); // skips 3

}

### 🔹 6. for...of Loop (for Iterables)

Elegant way to loop through arrays and strings.

const colors = ["red", "green", "blue"];

for (let color of colors) {

console.log(color);

}

### 🔹 7. for...in Loop (for Objects)

Iterates over keys (not values) in an object.

const user = { name: "LoneCoder", age: 30 };

for (let key in user) {

console.log(`${key}: ${user[key]}`);

}

### 🔹 8. Nested Loops

Loops within loops — useful for matrices or combinations.

for (let i = 1; i <= 2; i++) {

for (let j = 1; j <= 2; j++) {

console.log(`i=${i}, j=${j}`);

}

}

### 🧠 Assignments: Loops

1. Print numbers 1 to 10 using a for loop.
2. Create a multiplication table for 5 using a while loop.
3. Loop through an array using for...of and print each item.
4. Loop through an object using for...in and print key: value pairs.
5. Use nested loops to print a 3x3 grid of \*.

## 🔀 Chapter 7: switch Statements in JavaScript

The switch statement is a cleaner alternative to long chains of if...else if...else.

### 🔹 1. Basic switch Syntax

let day = 3;

switch (day) {

case 1:

console.log("Monday");

break;

case 2:

console.log("Tuesday");

break;

case 3:

console.log("Wednesday");

break;

default:

console.log("Other day");

}

💡 Without break, execution "falls through" to the next case.

### 🔹 2. default Case

Executed if none of the cases match. Optional but recommended.

let role = "guest";

switch (role) {

case "admin":

console.log("Welcome, Admin");

break;

case "user":

console.log("Hello, User");

break;

default:

console.log("Unknown role");

}

### 🔹 3. Grouping Multiple Cases

Use fall-through intentionally to group conditions.

let fruit = "apple";

switch (fruit) {

case "apple":

case "pear":

console.log("Pome fruit");

break;

case "banana":

console.log("Tropical fruit");

break;

}

### 🔹 4. Replacing if-else if with switch

let score = 85;

switch (true) {

case score >= 90:

console.log("Grade: A");

break;

case score >= 80:

console.log("Grade: B");

break;

case score >= 70:

console.log("Grade: C");

break;

default:

console.log("Fail");

}

### 🔹 5. switch vs if...else

| switch | if...else |
| --- | --- |
| Best for checking equality of a value | Best for complex conditions |
| Clean & readable with many cases | Flexible for range or complex logic |
| Executes from first match until break | Evaluates each if/else if |

### 🧠 Assignments: switch

1. Create a switch block to print day names based on 1–7.
2. Create a grading system using switch(true) like shown above.
3. Use switch to categorize vehicles: "car", "bike", "truck".
4. Implement a simple calculator using switch with +, -, \*, /.
5. Convert an existing if...else block into a switch.

## 🧱 Chapter 8: ****Arrays in JavaScript****

Arrays are ordered, index-based collections used to store multiple values in a single variable.

### 🔹 1. Declaring Arrays

const fruits = ["apple", "banana", "cherry"];

const numbers = [1, 2, 3, 4, 5];

const mixed = [42, "hello", true, null];

### 🔹 2. Accessing Elements

console.log(fruits[0]); // "apple"

console.log(fruits[2]); // "cherry"

### 🔹 3. Array Length

console.log(fruits.length); // 3

### 🔹 4. Looping Over Arrays

for (let i = 0; i < fruits.length; i++) {

console.log(fruits[i]);

}

for (let fruit of fruits) {

console.log(fruit);

}

fruits.forEach((item, index) => {

console.log(index, item);

});

### 🔹 5. Modifying Arrays

fruits.push("date"); // Add to end

fruits.pop(); // Remove from end

fruits.unshift("kiwi"); // Add to beginning

fruits.shift(); // Remove from beginning

### 🔹 6. Common Array Methods

const nums = [10, 20, 30, 40, 50];

nums.includes(30); // true

nums.indexOf(40); // 3

nums.slice(1, 4); // [20, 30, 40]

nums.splice(2, 1); // removes 30

const doubled = nums.map(n => n \* 2); // [20, 40, 80, 100]

const filtered = nums.filter(n => n > 25);// [40, 50]

const total = nums.reduce((a, b) => a + b);// 150

### 🔹 7. Sorting and Reversing

const letters = ["b", "c", "a"];

letters.sort(); // ["a", "b", "c"]

letters.reverse(); // ["c", "b", "a"]

**Numeric Sort (important!)**

const nums = [10, 5, 20];

nums.sort(); // Incorrect: [10, 20, 5] (compares as strings)

nums.sort((a, b) => a - b); // Correct: [5, 10, 20]

### 🔹 8. Nesting Arrays (2D Arrays)

const matrix = [

[1, 2],

[3, 4],

[5, 6]

];

console.log(matrix[1][0]); // 3

### 🔹 9. Spread Operator with Arrays

const arr1 = [1, 2];

const arr2 = [3, 4];

const combined = [...arr1, ...arr2]; // [1, 2, 3, 4]

### 🔹 10. Destructuring Arrays

const [first, second] = ["one", "two", "three"];

console.log(first); // "one"

console.log(second); // "two"

### 🧠 Assignments: Arrays

1. Create an array of your top 5 favorite movies. Log the first and last one.
2. Write a loop to print only even numbers from an array.
3. Use map() to square all elements in a number array.
4. Use filter() to return only names longer than 4 characters.
5. Merge two arrays using spread and sort them alphabetically.

## 🧱 Chapter 9: ****Objects in JavaScript****

Objects are key-value pair collections used to store structured data. They form the backbone of most JavaScript code, especially in Node.js and React.

### 🔹 1. Creating Objects

const user = {

name: "LoneCoder",

age: 30,

isActive: true

};

### 🔹 2. Accessing Properties

console.log(user.name); // "LoneCoder"

console.log(user["age"]); // 30

### 🔹 3. Updating & Adding Properties

user.age = 31;

user.email = "lone@coder.com";

### 🔹 4. Deleting Properties

delete user.isActive;

### 🔹 5. Checking for a Property

console.log("name" in user); // true

### 🔹 6. Looping Through Objects

for (let key in user) {

console.log(`${key}: ${user[key]}`);

}

Or using Object.entries():

Object.entries(user).forEach(([key, value]) => {

console.log(`${key}: ${value}`);

});

### 🔹 7. Nested Objects

const company = {

name: "TechCorp",

address: {

city: "Delhi",

zip: "110001"

}

};

console.log(company.address.city); // "Delhi"

### 🔹 8. Object Methods

const person = {

first: "Lone",

last: "Coder",

fullName() {

return `${this.first} ${this.last}`;

}

};

console.log(person.fullName()); // "Lone Coder"

### 🔹 9. Object Destructuring

const { name, age } = user;

console.log(name); // "LoneCoder"

### 🔹 10. Spread Operator & Object Copy

const newUser = { ...user, isAdmin: true };

🔥 Note: This performs a **shallow copy**.

### 🔹 11. Object.freeze() and Object.seal()

const config = Object.freeze({

apiKey: "123456"

});

config.apiKey = "000000"; // Ignored — cannot modify

### 🧠 Assignments: Objects

1. Create a book object with title, author, and year. Add a method getSummary().
2. Write a function that takes an object and prints all key-value pairs.
3. Use destructuring to extract name and email from a user object.
4. Merge two objects: profile and settings.
5. Create a nested object representing a laptop and access its battery capacity.

## 🔤 Chapter 10: ****Strings in JavaScript****

Strings are sequences of characters used to store and manipulate text. JavaScript provides powerful methods to work with them.

### 🔹 1. Creating Strings

let name = "LoneCoder";

let greeting = 'Hello';

let message = `Welcome, ${name}`; // Template literal

### 🔹 2. String Length

console.log(name.length); // 9

### 🔹 3. Accessing Characters

console.log(name[0]); // 'L'

console.log(name.charAt(1)); // 'o'

### 🔹 4. String Immutability

name[0] = "X";

console.log(name); // Still "LoneCoder" — strings are immutable

### 🔹 5. Changing Case

console.log(name.toUpperCase()); // "LONECODER"

console.log(name.toLowerCase()); // "lonecoder"

### 🔹 6. Searching Strings

console.log(name.indexOf("Coder")); // 4

console.log(name.includes("one")); // true

console.log(name.startsWith("Lo")); // true

console.log(name.endsWith("er")); // true

### 🔹 7. Extracting Substrings

console.log(name.slice(4)); // "Coder"

console.log(name.slice(0, 4)); // "Lone"

console.log(name.substring(0, 4)); // "Lone"

console.log(name.substr(4, 5)); // "Coder"

🧠 slice() supports negative indices; substring() doesn’t.

### 🔹 8. Trimming Whitespace

let input = " hello world ";

console.log(input.trim()); // "hello world"

console.log(input.trimStart()); // "hello world "

console.log(input.trimEnd()); // " hello world"

### 🔹 9. Replacing Text

let sentence = "The sky is blue";

console.log(sentence.replace("blue", "gray")); // "The sky is gray"

console.log(sentence.replace(/the/i, "A")); // Regex-based replacement

### 🔹 10. Splitting & Joining

let tags = "html,css,js";

let tagArray = tags.split(",");

console.log(tagArray); // ['html', 'css', 'js']

console.log(tagArray.join(" | ")); // "html | css | js"

### 🔹 11. Repeat & Pad

console.log("Hi! ".repeat(3)); // "Hi! Hi! Hi! "

console.log("42".padStart(5, "0")); // "00042"

console.log("42".padEnd(5, ".")); // "42..."

### 🔹 12. Escaping Characters

let quote = "He said, \"JavaScript is awesome!\"";

### 🔹 13. Template Literals (Backticks ``)

const lang = "JavaScript";

const msg = `I love ${lang}!`;

console.log(msg); // I love JavaScript!

Multi-line strings:

let poem = `

Roses are red,

Violets are blue,

Code is poetry,

Especially for you.`;

### 🧠 Assignments: Strings

1. Create a string and print its first and last characters.
2. Write a function to count how many times a word appears in a sentence.
3. Extract the domain from an email using slice() and indexOf().
4. Create a slug generator: convert "My Blog Post" into "my-blog-post".
5. Write a program that reverses a string.

## 🧱 JavaScript Deep Dive (2025)

### 📘 Chapter: OBJECTS & NESTED STRUCTURES

### 🔹 1. Object Basics

Objects are key-value pairs used to represent real-world entities.

const user = {

name: "LoneCoder",

age: 30,

active: true

};

console.log(user.name); // LoneCoder

console.log(user["age"]); // 30

### 🔹 2. Modifying and Adding Properties

user.age = 31;

user.country = "India";

console.log(user);

// { name: 'LoneCoder', age: 31, active: true, country: 'India' }

### 🔹 3. Deleting Properties

delete user.active;

console.log(user); // { name: 'LoneCoder', age: 31, country: 'India' }

### 🔹 4. Methods in Objects

Functions stored as values are called **methods**.

const calculator = {

add(a, b) {

return a + b;

}

};

console.log(calculator.add(3, 4)); // 7

### 🔹 5. The this Keyword

this refers to the object that is executing the function.

const person = {

name: "Neo",

greet() {

return `Hi, I'm ${this.name}`;

}

};

console.log(person.greet()); // Hi, I'm Neo

### 🔹 6. Nested Objects

Objects can contain other objects.

const company = {

name: "CyberWorks",

founder: {

firstName: "Lara",

lastName: "Elon"

}

};

console.log(company.founder.firstName); // Lara

### 🔹 7. Object Destructuring

Easily extract values from objects.

const { name, age } = user;

console.log(name, age); // LoneCoder 31

const {

founder: { firstName }

} = company;

console.log(firstName); // Lara

### 🔹 8. Optional Chaining (?.)

Prevents runtime errors when accessing deeply nested properties.

const jobTitle = company?.ceo?.title;

console.log(jobTitle); // undefined (no error)

### 🔹 9. Nullish Coalescing (??)

Returns the right-hand value only if the left is null or undefined.

const status = user.status ?? "offline";

console.log(status); // offline

### 🔹 10. Looping Over Objects

for (let key in user) {

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

}

### 🔹 11. Object Utility Methods

const keys = Object.keys(user); // ['name', 'age', 'country']

const values = Object.values(user); // ['LoneCoder', 31, 'India']

const entries = Object.entries(user); // [['name', 'LoneCoder'], ...]

### 🔹 12. Merging Objects (Spread Operator)

const defaults = { theme: "dark", notifications: true };

const settings = { notifications: false };

const config = { ...defaults, ...settings };

console.log(config); // { theme: 'dark', notifications: false }

### 🔹 13. JSON: Stringify and Parse

To send/receive objects as strings.

const json = JSON.stringify(user); // Converts to string

const obj = JSON.parse(json); // Back to object

### 🧠 Assignments: Objects & Nested Structures

1. Create a book object with title, author, and rating.
2. Add a method to book called summary() that returns a string.
3. Create a nested object student with name, and address containing city and zip.
4. Use destructuring to extract city and zip.
5. Loop through all keys and values of the book object using for...in.

## 🧠 Section: Functions & Scope Revisited

### 🔹 1. this in JavaScript (Refresher)

In regular functions, this is dynamic — it depends on how a function is called.

function show() {

console.log(this);

}

const obj = { name: "LoneCoder", show };

obj.show(); // logs: { name: "LoneCoder", show: f }

show(); // logs: window (in browsers) or undefined (in strict mode)

### 🔹 2. Arrow Functions Don’t Have Their Own this

Arrow functions lexically bind this — they inherit from their surrounding scope.

const person = {

name: "LoneCoder",

greet: function() {

setTimeout(() => {

console.log(`Hi, ${this.name}`); // Correctly prints "Hi, LoneCoder"

}, 1000);

}

};

person.greet();

Compare with a regular function:

setTimeout(function() {

console.log(this.name); // `this` is not `person`, it's global

}, 1000);

✅ **Rule:** Avoid arrow functions when you need a dynamic this (like in event handlers or object methods).

### 🔹 3. Arrow Function Caveats

* 🚫 No this, arguments, super, or new.target
* 🚫 Can’t be used as constructors (new won't work)
* 🚫 No prototype property

const Arrow = () => {};

console.log(typeof new Arrow()); // ❌ TypeError: Arrow is not a constructor

### 🔹 4. Lexical Scope vs Dynamic Scope

JavaScript uses **lexical scope**, meaning variables are resolved by where functions are defined, not where they're called.

let a = "outer";

function outer() {

let a = "inner";

function inner() {

console.log(a);

}

return inner;

}

const fn = outer();

fn(); // inner

### 🔹 5. Closures Revisited 🔁

A closure is a function that "remembers" the variables from the scope where it was defined.

function makeCounter() {

let count = 0;

return function() {

count++;

return count;

};

}

const counter1 = makeCounter();

console.log(counter1()); // 1

console.log(counter1()); // 2

✅ The inner function holds onto the count variable, even after makeCounter() has finished executing.

### 🔹 6. Closures in Loops (Fix with let or IIFE)

#### ❌ Common Bug with var:

for (var i = 0; i < 3; i++) {

setTimeout(() => console.log(i), 1000); // logs: 3, 3, 3

}

#### ✅ Fixed with let:

for (let i = 0; i < 3; i++) {

setTimeout(() => console.log(i), 1000); // logs: 0, 1, 2

}

Or use an IIFE:

for (var i = 0; i < 3; i++) {

(function(i) {

setTimeout(() => console.log(i), 1000);

})(i);

}

### 🔹 7. Function Scope vs Block Scope

* var is **function-scoped**
* let and const are **block-scoped**

function test() {

if (true) {

var x = 10;

let y = 20;

}

console.log(x); // 10

console.log(y); // ❌ ReferenceError

}

### 🔹 8. Dynamic Function Creation (Rare but Important)

const sum = new Function("a", "b", "return a + b;");

console.log(sum(3, 4)); // 7

✅ Used in metaprogramming and some libraries, but can be dangerous.

### 🔹 9. Scope Chain & Variable Lookup

let a = 1;

function foo() {

let b = 2;

function bar() {

let c = 3;

console.log(a, b, c);

}

bar();

}

foo(); // 1 2 3

✅ JavaScript looks "up" the scope chain if a variable isn’t found locally.

## 🧪 Assignments: Functions & Scope Revisited

1. Create an object with a method that logs its name property using both regular and arrow functions. Compare results.
2. Write a function that returns a closure that squares numbers.
3. Create a loop with var that prints 0 to 2 after 1 second — fix it using let.
4. Build a counter factory function that gives each counter its own private count.
5. Demonstrate lexical vs dynamic scoping with a real-world analogy (comments or code).

**Here is the JavaScript Deep Dive: ES6+ and Array Methods (**map**,** filter**,** reduce**,** find**) — a high-utility toolkit used heavily in Node.js and React.**

## 🔹 Section: ES6+ Features & Core Array Methods

### ✅ 1. ****ES6+ Enhancements Quick Recap****

Before diving into array methods, a few essential ES6+ features:

// let & const

const name = "LoneCoder";

let count = 0;

// Arrow Functions

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

// Template Literals

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

// Destructuring

const [first, second] = [10, 20];

const {x, y} = {x: 5, y: 10};

// Spread & Rest

const arr1 = [1, 2];

const arr2 = [...arr1, 3, 4];

function sum(...nums) {

return nums.reduce((a, b) => a + b);

}

## 📦 Core Array Methods

### 🔹 2. map(): Transform Each Element

const numbers = [1, 2, 3];

const squared = numbers.map(n => n \* n);

console.log(squared); // [1, 4, 9]

🧠 Purpose: Create a new array of the same length with each item transformed.

### 🔹 3. filter(): Select Specific Elements

const ages = [12, 17, 25, 30];

const adults = ages.filter(age => age >= 18);

console.log(adults); // [25, 30]

🧠 Purpose: Create a new array with only items that match a condition.

### 🔹 4. reduce(): Accumulate Values

const scores = [10, 20, 30];

const total = scores.reduce((acc, curr) => acc + curr, 0);

console.log(total); // 60

🧠 Purpose: Reduce array to a **single value** — number, object, etc.

// Reduce into object:

const items = ["apple", "banana", "apple"];

const count = items.reduce((acc, fruit) => {

acc[fruit] = (acc[fruit] || 0) + 1;

return acc;

}, {});

console.log(count); // { apple: 2, banana: 1 }

### 🔹 5. find(): Return First Match

const users = [{id: 1}, {id: 2}, {id: 3}];

const user = users.find(u => u.id === 2);

console.log(user); // {id: 2}

🧠 Purpose: Return the first item matching the condition (or undefined).

### 🔹 6. Bonus: some() and every()

const nums = [10, 20, 30];

console.log(nums.some(n => n > 25)); // true

console.log(nums.every(n => n > 5)); // true

### 🔹 7. Chain Them Like a Pro 💪

const people = [

{ name: "Alice", age: 25 },

{ name: "Bob", age: 17 },

{ name: "Carol", age: 30 },

];

const names = people

.filter(p => p.age >= 18)

.map(p => p.name.toUpperCase());

console.log(names); // ["ALICE", "CAROL"]

### 🔹 8. Arrow Function Shorthand Reminders

arr.map(x => x \* 2); // OK

arr.map(function(x) { return x \* 2 }); // Also OK

arr.map(x => { return x \* 2 }); // OK with brackets and return

### 📌 When to Use Which:

| Method | Use When You Want To... | Returns |
| --- | --- | --- |
| map | Transform each element | New array |
| filter | Keep elements that match a condition | New array |
| reduce | Compute a single value from many | One value (any type) |
| find | Find the first match | Single element |
| some | Check if **any** match | Boolean |
| every | Check if **all** match | Boolean |

## 🧪 Assignments: Practice with Array Methods

1. Given [2, 4, 6, 8], return an array of half the value using .map().
2. Given [12, 19, 8, 25, 17], filter out all numbers under 18.
3. Reduce an array of names like ["a", "b", "c"] into a single string "abc".
4. From an array of users with {id, active} properties, return the first active user using .find().
5. Chain .filter() and .map() to find names of users over 21 and capitalize them.

## 📘 Chapter: Working with Objects & Functional Patterns

**(Destructuring, Spread/Rest, Deep Object Loops)**

### 🔹 1. Object Destructuring

Extract specific properties into variables with clean syntax.

const user = { name: "LoneCoder", age: 30, role: "dev" };

const { name, age } = user;

console.log(name); // "LoneCoder"

console.log(age); // 30

💡 You can also rename:

const { name: username } = user;

console.log(username); // "LoneCoder"

And with defaults:

const { location = "Unknown" } = user;

console.log(location); // "Unknown"

### 🔹 2. Nested Object Destructuring

const person = {

id: 1,

profile: {

firstName: "Neo",

address: {

city: "Zion"

}

}

};

const { profile: { firstName, address: { city } } } = person;

console.log(firstName); // "Neo"

console.log(city); // "Zion"

### 🔹 3. Array Destructuring (Quick Refresher)

const coords = [10, 20];

const [x, y] = coords;

console.log(x, y); // 10 20

You can skip items:

const [ , second ] = [1, 2, 3];

console.log(second); // 2

### 🔹 4. The Spread Operator ...

Expands or copies values.

#### In arrays:

const a = [1, 2];

const b = [...a, 3];

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

#### In objects:

const user = { name: "Neo" };

const details = { ...user, age: 30 };

console.log(details); // { name: "Neo", age: 30 }

### 🔹 5. The Rest Operator ...

Gathers remaining items.

#### In function parameters:

function sum(...nums) {

return nums.reduce((a, b) => a + b);

}

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

#### In object destructuring:

const { name, ...rest } = { name: "Neo", age: 30, role: "One" };

console.log(rest); // { age: 30, role: "One" }

### 🔹 6. Deep Object Iteration (Nested Loops)

const data = {

user1: { score: 10 },

user2: { score: 20 },

user3: { score: 30 }

};

for (let [key, value] of Object.entries(data)) {

console.log(`${key} scored ${value.score}`);

}

### 🔹 7. Object.keys / values / entries Recap

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

console.log(Object.keys(obj)); // ['a', 'b']

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

console.log(Object.entries(obj));// [['a', 1], ['b', 2]]

### 🔹 8. Functional Object Utilities

#### Using Object.entries() with map():

const userScores = { Alice: 20, Bob: 30 };

const scoreList = Object.entries(userScores).map(

([name, score]) => `${name} has ${score} points`

);

console.log(scoreList);

// ['Alice has 20 points', 'Bob has 30 points']

### 🧠 Assignments

1. Destructure a nested object to extract city and email.
2. Use spread to combine two objects with overlapping keys (observe the result).
3. Write a function that accepts any number of scores and returns their average using rest + reduce.
4. Create a loop that prints all keys and values from a deeply nested object.
5. Use Object.entries and map() to convert an object to an array of strings like "key: value".

## ✅ Chapter 12: DOM Manipulation

The **DOM (Document Object Model)** represents your HTML as a structured tree. JavaScript can access and manipulate this structure to create dynamic web pages.

### 🔹 1. Accessing DOM Elements

#### getElementById, getElementsByClassName, getElementsByTagName

const title = document.getElementById("main-title");

const buttons = document.getElementsByClassName("btn");

const paragraphs = document.getElementsByTagName("p");

#### querySelector, querySelectorAll (modern and flexible)

const heading = document.querySelector("#main-title"); // First match

const allItems = document.querySelectorAll(".list-item"); // NodeList

### 🔹 2. Changing Content

#### textContent vs innerHTML

title.textContent = "New Title!";

title.innerHTML = "<span style='color:red'>Updated</span>";

#### .value (for input fields)

const nameInput = document.querySelector("#name");

console.log(nameInput.value);

### 🔹 3. Changing Styles

title.style.color = "blue";

title.style.fontSize = "2rem";

You can also toggle classes:

title.classList.add("highlight");

title.classList.remove("highlight");

title.classList.toggle("highlight");

### 🔹 4. Creating & Inserting Elements

const newPara = document.createElement("p");

newPara.textContent = "I am new!";

document.body.appendChild(newPara);

const container = document.querySelector(".container");

container.prepend(newPara); // Insert at the beginning

### 🔹 5. Removing Elements

const unwanted = document.querySelector(".ad");

unwanted.remove();

### 🔹 6. Traversing the DOM

const parent = title.parentElement;

const children = container.children;

const first = container.firstElementChild;

const next = title.nextElementSibling;

### 🔹 7. Attributes

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

link.getAttribute("href"); // Read

link.setAttribute("href", "/new"); // Update

You can also use dataset:

<button data-user-id="42">Click Me</button>

const btn = document.querySelector("button");

console.log(btn.dataset.userId); // "42"

### 🔹 8. Templates & Cloning

<template id="user-template">

<div class="user-card">

<h3></h3>

</div>

</template>

const tmpl = document.querySelector("#user-template");

const clone = tmpl.content.cloneNode(true);

clone.querySelector("h3").textContent = "LoneCoder";

document.body.appendChild(clone);

### 🧠 Assignments: DOM Manipulation

1. Create a function that adds a new <li> item to a list when a button is clicked.
2. Write a function that changes the background color of the page based on user input.
3. Add and remove a CSS class from a div when hovering.
4. Build a dynamic form that adds input fields when a button is clicked.
5. Use a template to generate a user card with name and age.

## ✅ Chapter 13: Events & Event Delegation

JavaScript events let you respond to user interactions like clicks, keystrokes, mouse movement, and more. Event delegation lets you handle events efficiently, especially for dynamically added elements.

### 🔹 1. Basic Event Listeners

Use addEventListener to respond to events:

const btn = document.querySelector("#submit-btn");

btn.addEventListener("click", function () {

alert("Button clicked!");

});

You can also use arrow functions or named functions:

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

### 🔹 2. Common Event Types

| Event | Description |
| --- | --- |
| click | Mouse click |
| dblclick | Double click |
| mouseover | Hover over element |
| keydown | Key is pressed |
| submit | Form submission |
| input | Input field is changed |
| change | Input/select value is changed |

### 🔹 3. Event Object

Every event gives you an event object:

btn.addEventListener("click", function (e) {

console.log(e.target); // The clicked element

});

You can use this to prevent default behavior:

document.querySelector("form").addEventListener("submit", function (e) {

e.preventDefault(); // Stop form from reloading the page

console.log("Form submitted!");

});

### 🔹 4. Removing Event Listeners

function handleClick() {

console.log("Clicked!");

}

btn.addEventListener("click", handleClick);

btn.removeEventListener("click", handleClick);

### 🔹 5. Event Delegation (⚡ Performance Tip)

Instead of attaching listeners to each item, attach it to a **parent** and use e.target:

<ul id="task-list">

<li>Task 1</li>

<li>Task 2</li>

</ul>

const list = document.getElementById("task-list");

list.addEventListener("click", function (e) {

if (e.target.tagName === "LI") {

e.target.style.textDecoration = "line-through";

}

});

This works for new elements added later too.

### 🔹 6. Keyboard & Input Events

document.addEventListener("keydown", (e) => {

console.log("Key pressed:", e.key);

});

const input = document.querySelector("#username");

input.addEventListener("input", (e) => {

console.log("Typed:", e.target.value);

});

### 🔹 7. Event Propagation: Bubbling & Capturing

* **Bubbling**: Events move from child → parent
* **Capturing**: Events move from parent → child

child.addEventListener("click", () => console.log("child"), true); // capture

parent.addEventListener("click", () => console.log("parent")); // bubble

Use e.stopPropagation() to prevent bubbling further.

### 🔹 8. Debouncing Input Events

Use this to optimize frequent input events (e.g., search boxes):

let timeout;

input.addEventListener("input", (e) => {

clearTimeout(timeout);

timeout = setTimeout(() => {

console.log("Search for:", e.target.value);

}, 300);

});

### 🧠 Assignments: Events & Delegation

1. Create a list where clicking on an item toggles a "done" class.
2. Build a form that doesn’t reload the page and logs input values.
3. Add a keydown listener to track typed characters in an input field.
4. Create a dynamic search bar with debouncing.
5. Use event delegation to remove list items when clicked.

## ✅ Chapter 14: Error Handling & Debugging

Every developer writes buggy code — the key is knowing how to **handle errors** and **debug effectively**.

### 🔹 1. Types of Errors

| Type | Example | When It Happens |
| --- | --- | --- |
| Syntax Error | if (a > b (missing closing brace) | At parse time |
| Runtime Error | undefinedVar + 1 | During execution |
| Logical Error | Wrong formula or wrong logic used | Produces wrong output |

### 🔹 2. Try...Catch

Use try...catch to handle errors and prevent crashes:

try {

const x = someUndefinedFunction();

} catch (err) {

console.log("Caught error:", err.message);

}

Optional finally block always runs:

try {

// risky code

} catch (e) {

// handle error

} finally {

console.log("Cleanup or closing");

}

### 🔹 3. Throwing Custom Errors

You can throw your own errors:

function validateAge(age) {

if (age < 0) throw new Error("Age can't be negative");

return age;

}

try {

validateAge(-5);

} catch (e) {

console.error(e.message);

}

### 🔹 4. Debugging with console

Common console methods:

console.log("Normal message");

console.warn("Warning!");

console.error("Something went wrong");

console.table([{name: "Alice"}, {name: "Bob"}]);

### 🔹 5. Using the Browser DevTools

In Chrome or Firefox:

* Open DevTools (Right click → Inspect → Console).
* Go to **Sources** tab to add breakpoints.
* Use **Call Stack**, **Watch**, and **Scope** panels for deeper debugging.
* debugger; keyword acts like a manual breakpoint:

function sum(a, b) {

debugger;

return a + b;

}

### 🔹 6. Stack Trace & Error Object

Example:

function a() {

b();

}

function b() {

throw new Error("Something broke");

}

a();

DevTools shows the full call stack and error line for tracking.

### 🔹 7. Optional Chaining & Nullish Coalescing

Modern JavaScript tools to avoid runtime errors:

const user = null;

console.log(user?.name); // undefined (no crash)

console.log(user?.address?.city); // safe

const age = user?.age ?? 18; // default to 18 if null/undefined

### 🔹 8. Defensive Programming

Avoid errors by checking before acting:

if (Array.isArray(users) && users.length) {

console.log(users[0]);

}

### 🧠 Assignments: Error Handling

1. Write a function divide(a, b) that throws if b === 0.
2. Create a try...catch block around JSON parsing:

JSON.parse('{"valid": true}');

JSON.parse('{invalid JSON}');

1. Use debugger inside a loop and inspect values.
2. Use optional chaining on a nested object safely.
3. Display a custom error message if a form input is empty.

## ✅ Chapter 15: ES6+ Features Overview

Modern JavaScript (ES6 and beyond) introduced powerful syntax and patterns that make code cleaner, faster, and more expressive.

### 🔹 1. let and const

* let: block-scoped, reassignable
* const: block-scoped, **not reassignable**

let name = "LoneCoder";

const age = 25;

Prefer const by default. Use let only when you need reassignment.

### 🔹 2. Arrow Functions

Concise function syntax:

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

If only one parameter:

const greet = name => console.log("Hi", name);

⚠️ **Arrow Caveats:**

* No this, arguments, or super
* Not suitable as constructors

### 🔹 3. Template Literals

Multiline strings and variable embedding:

const name = "LoneCoder";

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

### 🔹 4. Default Parameters

Set default values for parameters:

function greet(name = "Stranger") {

return `Hello, ${name}`;

}

### 🔹 5. Destructuring

Extract values from arrays/objects:

const [a, b] = [1, 2];

const { name, age } = { name: "Ana", age: 30 };

### 🔹 6. Spread & Rest

**Spread** expands:

const nums = [1, 2, 3];

const more = [...nums, 4, 5];

**Rest** collects:

function sum(...args) {

return args.reduce((a, b) => a + b, 0);

}

### 🔹 7. Enhanced Object Literals

const x = 10;

const obj = {

x, // shorthand property

greet() { return "Hi"; }, // shorthand method

};

### 🔹 8. Promises & Async/Await (Recap)

const delay = ms => new Promise(res => setTimeout(res, ms));

async function load() {

await delay(1000);

console.log("Loaded");

}

### 🔹 9. For...of Loop

const nums = [10, 20, 30];

for (const num of nums) {

console.log(num);

}

Works on iterables like arrays, strings, maps, etc.

### 🔹 10. Object.entries(), .values(), .keys()

const user = { name: "Lone", age: 30 };

Object.entries(user); // [['name', 'Lone'], ['age', 30]]

### 🔹 11. Optional Chaining + Nullish Coalescing (Recap)

const city = user?.address?.city ?? "Unknown";

### 🧠 Assignments: ES6+ Features

1. Convert a regular function to an arrow function with default parameters.
2. Use template literals to log a sentence with embedded variables.
3. Destructure an object and an array in a single line.
4. Use rest parameters to create a multiplyAll(...nums) function.
5. Loop over an array using for...of and log squared values.

## ✅ Chapter 16: Modules & Imports (ES6 Modules)

Modular code keeps your JavaScript maintainable, scalable, and reusable. Modern JavaScript uses the **ES6 module system** with export and import statements.

### 🔹 1. What Are Modules?

* A **module** is any JavaScript file.
* Variables and functions declared inside are **scoped locally**.
* To make code **reusable**, you export it.
* To use exported code in another file, you import it.

### 🔹 2. Exporting Code

#### 👉 Named Exports

// file: mathUtils.js

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

export const multiply = (a, b) => a \* b;

You can export multiple values from a module using named exports.

#### 👉 Default Export

// file: greet.js

export default function greet(name) {

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

}

Each file can have **only one default export**.

### 🔹 3. Importing Code

#### 👉 Named Imports

// file: main.js

import { add, multiply } from './mathUtils.js';

console.log(add(2, 3));

You can also alias them:

import { add as addition } from './mathUtils.js';

#### 👉 Default Import

// file: main.js

import greet from './greet.js';

greet("LoneCoder");

You can combine both:

import greet, { add, multiply } from './mathUtils.js';

### 🔹 4. File Paths

* Use **relative paths** (./, ../)
* Or **bare module specifiers** in advanced setups (e.g., via bundlers or Node with type="module")

### 🔹 5. Combining with HTML

<!-- index.html -->

<script type="module" src="main.js"></script>

type="module" is required to enable ES6 imports in browser.

### 🔹 6. Live Bindings

Exports are **live references**. If a value changes in the module, the change reflects in all imports.

### 🔹 7. Module Scope

Modules are always in **strict mode** and have their own top-level scope (no global pollution).

### 🔹 8. Dynamic Imports

Load modules asynchronously:

const module = await import('./mathUtils.js');

console.log(module.add(1, 2));

Useful for code-splitting and performance optimization.

### 🧠 Assignments: Modules & Imports

1. Create a file math.js and export add, subtract, and multiply functions.
2. In main.js, import all functions and call them with sample inputs.
3. Export a default greeting function and import it in another file.
4. Create an object with constants (e.g., PI, E) and export/import it.
5. Try dynamic import and log something after a timeout.

## ✅ Chapter 17: Local Storage & Session Storage

Modern browsers provide **Web Storage APIs** to store key-value data directly in the user's browser — no server needed.

### 🔹 1. Two Types of Web Storage

| Type | Persistence | Scope |
| --- | --- | --- |
| localStorage | Until manually cleared | Shared across tabs |
| sessionStorage | Cleared when tab closes | Specific to a tab |

### 🔹 2. Storing Data

localStorage.setItem('username', 'LoneCoder');

sessionStorage.setItem('authToken', 'abc123');

* Both methods accept **string keys and values only**.
* You must JSON.stringify() objects before storing.

const user = { name: 'LoneCoder', age: 30 };

localStorage.setItem('user', JSON.stringify(user));

### 🔹 3. Retrieving Data

const name = localStorage.getItem('username'); // "LoneCoder"

const user = JSON.parse(localStorage.getItem('user')); // { name: "LoneCoder", age: 30 }

### 🔹 4. Removing & Clearing Data

localStorage.removeItem('username');

sessionStorage.clear(); // removes all session data

### 🔹 5. Checking Key Existence

if (localStorage.getItem('theme')) {

console.log("Theme preference found!");

}

### 🔹 6. Practical Example

// Save theme preference

function setTheme(theme) {

localStorage.setItem('theme', theme);

document.body.className = theme;

}

// Load theme on page load

window.onload = () => {

const theme = localStorage.getItem('theme') || 'light';

document.body.className = theme;

};

### 🔹 7. Storage Event

Triggered when localStorage is changed in another tab.

window.addEventListener('storage', (e) => {

console.log(`Key ${e.key} changed from ${e.oldValue} to ${e.newValue}`);

});

Note: Only works **across tabs**, not in the same one.

### 🚧 Best Practices

* Use **localStorage for preferences**.
* Use **sessionStorage for temp data like auth in single tabs**.
* Avoid storing sensitive data directly (e.g., passwords, tokens).

### 🧠 Assignments: Local & Session Storage

1. Store a user's name in localStorage, retrieve and display it.
2. Save an object (user profile) and load it after refresh.
3. Create a button to clear all local storage data.
4. Build a light/dark theme toggle using localStorage.
5. Use sessionStorage to track how many times a user reloads the current tab.

## ✅ Chapter 18: JSON & Fetch API

This chapter focuses on:

* Understanding **JSON** (JavaScript Object Notation)
* Using the **Fetch API** to make HTTP requests (GET/POST)
* Handling responses and errors
* Connecting to real APIs

### 🔹 1. What is JSON?

* JSON is a **lightweight** format for storing and transporting data.
* It's based on JavaScript syntax but is language-independent.

{

"name": "LoneCoder",

"age": 30,

"skills": ["JS", "Python"]

}

### 🔹 2. JSON in JavaScript

#### Converting JS object → JSON string:

const obj = { name: "LoneCoder", active: true };

const jsonStr = JSON.stringify(obj);

#### Converting JSON string → JS object:

const jsonStr = '{"name":"LoneCoder","active":true}';

const obj = JSON.parse(jsonStr);

### 🔹 3. Fetch API – Making Requests

The Fetch API returns a Promise and is used to call URLs:

#### Basic GET Request:

fetch('https://jsonplaceholder.typicode.com/posts/1')

.then(response => response.json()) // parse JSON

.then(data => console.log(data))

.catch(error => console.error('Error:', error));

### 🔹 4. Making a POST Request

fetch('https://jsonplaceholder.typicode.com/posts', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({

title: 'Hello',

body: 'This is a post.',

userId: 1

})

})

.then(response => response.json())

.then(data => console.log('Post created:', data))

.catch(error => console.error('Error:', error));

### 🔹 5. Async/Await Version

async function loadPost() {

try {

const response = await fetch('https://jsonplaceholder.typicode.com/posts/2');

const data = await response.json();

console.log(data);

} catch (err) {

console.error('Failed to fetch:', err);

}

}

### 🔹 6. Handling Non-200 Responses

fetch('https://api.example.com/data')

.then(response => {

if (!response.ok) throw new Error('Network response was not ok');

return response.json();

})

.then(data => console.log(data))

.catch(err => console.error('Fetch failed:', err));

### 🔹 7. Real API Example (GET & POST)

You can try using:

* https://jsonplaceholder.typicode.com/posts (Fake REST API)
* https://reqres.in (Fake API for testing POST/PUT/DELETE)

### 🧠 Assignments: JSON & Fetch API

1. Fetch and display a list of posts from the JSONPlaceholder API.
2. Make a form that sends a POST request to create a new post.
3. Build a function using async/await to fetch a single user’s data.
4. Store a fetched JSON object in localStorage.
5. Add error handling for 404 or 500 response codes using .ok flag.

✅ Once you're ready, we’ll move on to the **final chapter**:  
**19. Async JavaScript (already marked complete)**.

Would you like a **summary of all completed chapters** and a printable **JavaScript Deep Dive PDF-style TOC** next?

## ✅ ****Completed JavaScript Deep Dive Chapters****

### 🟢 ****Core Foundations****

1. **Introduction to JavaScript**
2. **JavaScript Data Types (Primitives & Objects)**
3. **Variables, Constants & Operators**
4. **Conditional Statements (if, else, ternary)**
5. **Loops (for, while, do-while, for...of)**
6. **Switch Case Statements**
7. **Arrays & Core Methods**
8. **Strings & String Manipulation**
9. **Working with Objects & Nested Structures**

### 🟦 ****Functional Programming & Modern Patterns****

1. **Functions & Scope Revisited**  
     - Function declarations, expressions, arrow functions  
     - Closures, lexical scope, context binding
2. **ES6+ Array Methods**  
     - map, filter, reduce, find, some, every
3. **Working with Objects & Functional Patterns**  
     - Destructuring, spread/rest, deep looping

### 🟣 ****Browser Environment & Interaction****

1. **DOM Manipulation**  
     - Selectors, innerHTML, style, element creation
2. **Events & Event Delegation**  
     - Event listeners, bubbling/capturing, delegation
3. **Error Handling & Debugging**  
     - try...catch, finally, throw, dev tools
4. **ES6+ Features Overview**  
     - let, const, arrow functions, template strings, classes
5. **Modules & Imports**  
     - export, import, default, ES modules
6. **Local Storage & Session Storage**  
     - localStorage.setItem/getItem, sessionStorage, JSON usage
7. **JSON & Fetch API**  
     - JSON parsing/stringifying, Fetch API (GET/POST), async/await

### ✅ Bonus (Completed Previously)

1. **Async JavaScript**  
     - Promises, async/await, error chaining, race/all, microtasks