JavaScript Foundation – Training Day-8

```
<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

Day7: Callbacks, Promises, and Async/Await (JavaScript)

Contents

- Template literals and string interpolation
- Destructuring assignment (arrays and objects)
- Spread and rest operators
- Modules: import and export
- Map, Set, and WeakMap data structures
- Error handling with try-catch
- JavaScript best practices and coding standards
- Debugging techniques and tools

Template Literals & String Interpolation (JavaScript)

1️⃣ What are Template Literals?

Template literals are a modern way to create strings in JavaScript using **backticks (`)** instead of quotes.

They allow you to:

- Embed variables directly inside strings
- Write multi-line strings easily
- Add expressions inside strings

📌 Introduced in **ES6 (2015)**

Syntax

```
`This is a template literal`
```

2️⃣ What is String Interpolation?

String interpolation means **injecting variables or expressions directly into a string** using:

```
${expression}
```

Example:

```
const name = "Rahul";

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

3️⃣ Why Use Template Literals?

❌ Old way (String concatenation)

```
const name = "Rahul";

const age = 25;
```

```
console.log("My name is " + name + " and I am " + age + " years old.");
```

❗ Problems:

- Hard to read
- Easy to make mistakes
- Messy for long strings

✅ New way (Template literals)

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

✓ Cleaner
✓ More readable
✓ Less error-prone

4️⃣ When to Use Template Literals?

Use template literals when you need to:

✓ Insert variables into strings
✓ Write readable dynamic messages
✓ Create multi-line strings
✓ Build HTML templates
✓ Log meaningful debug messages

5️⃣ Basic String Interpolation Example

```
const product = "Laptop";

const price = 55000;

console.log(`The price of ${product} is ₹${price}`);
```

📌 Output:

```
The price of Laptop is ₹55000
```

6️⃣ Expressions Inside Template Literals

You can place **any valid JavaScript expression** inside ${}.

```
const a = 10;

const b = 20;

console.log(`Sum is ${a + b}`);

console.log(`Is a greater than b? ${a > b}`);
```

📌 Output:

```
Sum is 30

Is a greater than b? false
```

7️⃣ Multi-line Strings (Very Important)

❌ Old way

```
const msg = "Hello\nWelcome to\nJavaScript";
```

✅ Template literals

```
const msg = `

Hello

Welcome to

JavaScript

`;

console.log(msg);
```

✓ Natural formatting
✓ Perfect for emails, messages, HTML

8️⃣ Template Literals with Functions

```
function greet(name) {
  return `Hello ${name}, welcome!`;
}
console.log(greet("Anita"));
```

📌 Output:

```
Hello Anita, welcome!
```

9️⃣ Building HTML Using Template Literals (Very Common)

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

const html = `
  <div>
    <h2>${user.name}</h2>
    <p>Age: ${user.age}</p>
  </div>
`;

document.body.innerHTML = html;
```

✓ Widely used in:

- DOM manipulation
- Fetch API UI rendering
- React / Angular templates

🔟 Template Literals in Console Logs (Debugging)

```
const status = "Success";

const time = "10:30 AM";


console.log(`Status: ${status} | Time: ${time}`);
```

✓ Cleaner logs
✓ Easy debugging

1️⃣1️⃣ Common Beginner Mistakes 🚫

❌ Using quotes instead of backticks

```
console.log("Hello ${name}"); // ❌ won't work
```

✅ Correct

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

❌ Forgetting ${}

```
console.log(`Hello name`); // literal text
```

1️⃣2️⃣ Comparison Table

Feature	String Concatenation	Template Literals
Readability	❌ Poor	✅ Excellent
Multi-line support	❌ No	✅ Yes
Expressions	❌ Hard	✅ Easy
HTML generation	❌ Messy	✅ Clean

1️⃣3️⃣ When NOT to Use Template Literals?

Avoid them when:

- Writing **very simple static strings**
- Supporting **very old browsers** (pre-ES6)

1️⃣4️⃣ Real-World Use Cases

✓ Showing API data in UI
✓ Form validation messages
✓ Error messages
✓ Dynamic emails
✓ Logs & debugging

✅ Final Summary

Template literals are:

- A modern, clean way to work with strings
- Essential for readable JavaScript
- Widely used in real-world applications

👉 **If your string is dynamic → use template literals**

🧩 Destructuring Assignment – Practice Questions with Answers

(Arrays & Objects)

🟢 LEVEL 1: Basics

1️⃣ Array Basics

Question
Given:

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

Use array destructuring to store "red" in firstColor and "green" in secondColor.

Answer

```
const [firstColor, secondColor] = colors;
```

2️⃣ Object Basics

Question
Given:

```
const user = { name: "Anita", age: 25 };
```

Extract name and age using object destructuring.

Answer

```
const { name, age } = user;
```

3️⃣ Skip Values

Question
Given:

```
const numbers = [10, 20, 30, 40];
```

Extract only 20 and 40.

Answer

```
const [, second, , fourth] = numbers;
```

4 Default Values

Question

Given:

```
const scores = [95];
```

Destructure so that math = 95 and science = 0.

Answer

```
const [math, science = 0] = scores;
```

🟡 LEVEL 2: Intermediate

5 Swapping Values

Question

Swap values without using a temporary variable.

```
let a = 5;

let b = 10;
```

Answer

```
[a, b] = [b, a];
```

6 Rename Object Properties

Question

Given:

```
const employee = { id: 101, role: "Developer" };
```

Rename id to employeeId and role to jobRole.

Answer

```
const { id: employeeId, role: jobRole } = employee;
```

7️⃣ Nested Object Destructuring

Question
Given:

```
const user = {
  name: "Rahul",
  address: {
    city: "Pune",
    pin: 411001
  }
};
```

Extract city and pin.

Answer

```
const {
  address: { city, pin }
} = user;
```

8️⃣ Rest Operator (Array)

Question
Given:

```
const nums = [1, 2, 3, 4, 5];
```

Extract first value and remaining values.

Answer

```
const [first, ...rest] = nums;
```

🟠 LEVEL 3: Real-World Scenarios

9️⃣ API-like Object

Question
Given:

```
const response = {
  status: 200,
  data: {
    users: ["A", "B", "C"]
  }
};
```

Extract status and users.

Answer

```
const {
  status,
  data: { users }
} = response;
```

🔟 Function Parameters

Question
Write a function using destructuring to print:

Name: Sita, Age: 30

Answer

```
function printUser({ name, age }) {
  console.log(`Name: ${name}, Age: ${age}`);
}


printUser({ name: "Sita", age: 30 });
```

1️⃣1️⃣ Safe Destructuring

Question
Given:

```
const user = {};
```

Safely extract city from user.address.

Answer

```
const { city } = user.address ?? {};
```

1️⃣2️⃣ Rest Operator (Object)

Question
Given:

```
const product = {
  name: "Phone",
  price: 20000,
  brand: "ABC"
};
```

Extract name and store remaining properties in details.

Answer

```
const { name, ...details } = product;
```

🔴 LEVEL 4: Advanced

1️⃣3️⃣ Multiple Return Values

Question
Return sum and difference, then destructure the result.

Answer

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

const [sum, difference] = calculate(10, 5);
```

1️⃣4️⃣ Loop with Destructuring

Question
Given:

```
const users = [
  { name: "A", age: 20 },
  { name: "B", age: 25 }
];
```

Print:

```
A is 20
B is 25
```

Answer

```
for (const { name, age } of users) {
  console.log(`${name} is ${age}`);
}
```

1️⃣5️⃣ Mixed Destructuring

Question
Given:

```
const data = {
  id: 1,
  scores: [80, 90]
};
```

Extract id, firstScore, and secondScore.

Answer

```
const {
  id,
  scores: [firstScore, secondScore]
} = data;
```

🎯 Challenge

Convert Traditional Code

Question

```
const city = user.address.city;
const pin = user.address.pin;
```

Answer

```
const {
  address: { city, pin }
} = user;
```

✅ Wrap-up

✓ Arrays → position based
✓ Objects → property-name based
✓ Defaults prevent undefined
✓ ?? {} prevents runtime errors
✓ Essential for modern JavaScript & React

🧩 Spread & Rest Operators – Practice Problems

🟢 LEVEL 1: Basics

1️⃣ Copy an Array (Spread)

Question
Given:

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

Create a **new copy** of the array using spread.

Answer

```
const copy = [...nums];
```

2️⃣ Merge Two Arrays (Spread)

Question
Given:

```
const a = [10, 20];

const b = [30, 40];
```

Merge both arrays into one.

Answer

```
const merged = [...a, ...b];
```

3️⃣ Add Elements While Copying (Spread)

Question
Insert 1 at the beginning and 5 at the end.

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

Answer

```
const updated = [1, ...nums, 5];
```

4️⃣ Copy an Object (Spread)

Question
Given:

```
const user = { name: "Amit", age: 25 };
```

Create a shallow copy.

Answer

```
const copy = { ...user };
```

🟡 LEVEL 2: Intermediate

5️⃣ Merge Objects (Spread)

Question
Given:

```
const personal = { name: "Ravi" };
```

```
const job = { role: "Developer" };
```

Merge into one object.

Answer

```
const profile = { ...personal, ...job };
```

6️⃣ Override Object Property (Spread)

Question
Update age to 31.

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

Answer

```
const updatedUser = { ...user, age: 31 };
```

7 Spread in Function Call

Question
Pass array values to Math.max.

```
const nums = [5, 15, 25];
```

Answer

```
Math.max(...nums);
```

---

8 Spread a String

Question
Convert string into array of characters.

```
const lang = "JS";
```

Answer

```
const chars = [...lang];
```

🟠 LEVEL 3: Rest Operator (Destructuring)

---

9 Rest with Arrays

Question
Extract first element and store remaining.

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

Answer

```
const [first, ...rest] = nums;
```

---

10 Rest with Objects

Question
Extract name and group remaining properties.

```
const user = { name: "Neha", age: 28, city: "Delhi" };
```

Answer

```
const { name, ...details } = user;
```

1️⃣1️⃣ Rest in Function Parameters

Question
Write a function that accepts any number of values and logs them.

Answer

```
function logValues(...values) {
  console.log(values);
}

logValues(1, 2, 3);
```

1️⃣2️⃣ Sum Using Rest Operator

Question
Create a function that sums all arguments.

Answer

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

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

🔴 LEVEL 4: Mixed (Spread + Rest)

1️⃣3️⃣ Spread While Calling, Rest While Receiving

Question
Given:

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

Call a function using spread and collect values using rest.

Answer

```
function demo(...values) {
  console.log(values);
}


demo(...nums);
```

1️⃣4️⃣ Exclude Property Using Rest

Question
Remove password from object.

```
const user = {
  username: "admin",
  password: "1234",
  role: "manager"
};
```

Answer

```
const { password, ...safeUser } = user;
```

1️⃣5️⃣ Clone and Extend Object (Real-World)

Question
Add isActive: true without mutating original object.

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

Answer

```
const newUser = { ...user, isActive: true };
```

🎯 Challenge (Thinking Question)

Why does this create a new array but this does not?

```
const copy1 = [...arr]; // ✅ new array

const copy2 = arr;     // ❌ same reference
```

Answer

- Spread (...) creates a **new array**
- Direct assignment copies the **reference**, not the data

✅ Final Recap

✓ **Spread** → expands values
✓ **Rest** → collects values
✓ Same syntax (...), different meaning
✓ Essential for modern JavaScript & React

JavaScript Modules – import & export

1️⃣ What are JavaScript Modules?

A **module** is a **separate JavaScript file** that:

- Contains related code (variables, functions, classes)
- Can **export** parts of its code
- Can **import** code from other files

📌 Each module has its **own scope**
📌 Introduced officially in **ES6 (2015)**

Simple Meaning

One file = one responsibility

2️⃣ Why Do We Need Modules?

❌ Problems without modules

- All code in one file
- Global variable conflicts
- Hard to maintain large projects
- Difficult teamwork

✅ Benefits of modules

✓ Better code organization
✓ Reusable code
✓ Avoid global pollution
✓ Easier debugging & testing
✓ Industry standard (React, Angular, Node.js)

3️⃣ When Should You Use Modules?

Use modules when:
✓ Project grows beyond one file
✓ Multiple developers work together
✓ You want reusable utilities
✓ Building modern web apps

👉 **Real-world apps ALWAYS use modules**

🔷 EXPORT (Sharing Code)

4️⃣ What is export?

export is used to **make variables, functions, or classes available** to other files.

5️⃣ Types of Exports

There are **two main types**:

1. **Named export**
2. **Default export**

🔷 NAMED EXPORT

6️⃣ What is Named Export?

A **named export** allows you to export **multiple items** from a file using their names.

Example: math.js

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

export const subtract = (a, b) => a - b;
```

✓ Multiple exports
✓ Must use exact names when importing

7️⃣ Why Use Named Exports?

✓ Clear what is exported
✓ Multiple utilities per file
✓ Better for libraries

8️⃣ When to Use Named Exports?

Use when:

- File contains **multiple related functions**
- You want **explicit imports**
- Building utility/helper files

🔷 IMPORT (Using Code)

9️⃣ Importing Named Exports

Example: app.js

```
import { add, subtract } from "./math.js";


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

console.log(subtract(5, 3)); // 2
```

📌 Curly braces {} are required
📌 Names must match exactly

🔟 Renaming While Importing

```
import { add as sum } from "./math.js";


console.log(sum(2, 3)); // 5
```

✓ Useful to avoid name conflicts

🔷 DEFAULT EXPORT

1️⃣1️⃣ What is Default Export?

A **default export** allows a file to export **only one main value**.

Example: calculator.js

```
export default function multiply(a, b) {
  return a * b;
}
```

✓ One export per file
✓ Import name can be anything

1️⃣2️⃣ Importing Default Export

```
import multiply from "./calculator.js";
console.log(multiply(4, 5)); // 20
```

📌 No curly braces
📌 Name is flexible

1️⃣3️⃣ Why Use Default Export?

✓ File has **one primary responsibility**
✓ Cleaner import syntax
✓ Common in components (React)

1️⃣4️⃣ When to Use Default Export?

Use when:

- File exports **one main thing**
- Component files
- Main utility functions

🔷 MIXED EXPORTS

1️⃣5️⃣ Combining Named + Default Export

Example: utils.js

```
export const PI = 3.14;

export default function area(radius) {

  return PI * radius * radius;

}
```

Importing

```
import area, { PI } from "./utils.js";

console.log(area(5));

console.log(PI);
```

1️⃣6️⃣ Common Beginner Mistakes 🚫

❌ Forgetting {} for named imports

```
import add from "./math.js"; // ❌ wrong
```

✅ Correct

```
import { add } from "./math.js";
```

❌ Wrong file path

```
import { add } from "math.js"; // ❌
```

✅ Correct

```
import { add } from "./math.js";
```

❌ Not using type="module" in HTML

```
<script src="app.js"></script> <!-- ❌ -->
```

✅ Correct

```
<script type="module" src="app.js"></script>
```

🔷 MODULES IN HTML (Browser)

1️⃣7️⃣ Using Modules in HTML

```
<!DOCTYPE html>
<html>
<body>
  <script type="module" src="app.js"></script>
</body>
</html>
```

✓ Enables import/export
✓ Runs in strict mode automatically

1️⃣8️⃣ Module Rules to Remember

✓ Modules are **strict mode by default**
✓ Each module has its **own scope**
✓ File extension .js is mandatory
✓ Modules load asynchronously

1️⃣9️⃣ Real-World Use Cases

✓ Utility/helper files
✓ API services
✓ UI components
✓ State management
✓ Large-scale applications

2️⃣0️⃣ Summary Table

Feature	**Named Export**	**Default Export**
Number per file	Multiple	One
Import syntax	{} required	No {}
Name flexibility	❌ No	✅ Yes
Common usage	Utilities	Components

✅ Final Summary (For Students)

👉 **Use modules to organize your code**
👉 **Use named exports for multiple utilities**
👉 **Use default export for one main feature**

🧩 JavaScript Modules – Practice Problems

(import & export)

🟢 LEVEL 1: Basics

1️⃣ Named Export (Single Value)

Question
Create a file math.js that exports a function add.

Answer

```
// math.js

export function add(a, b) {

  return a + b;

}
```

2️⃣ Import Named Export

Question
Import the add function from math.js and use it.

Answer

```
// app.js

import { add } from "./math.js";


console.log(add(2, 3)); // 5
```

3️⃣ Multiple Named Exports

Question
Export two functions: add and subtract.

Answer

```
// math.js

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

export const subtract = (a, b) => a - b;
```

4️⃣ Import Multiple Named Exports

Question
Import both add and subtract.

Answer

```
import { add, subtract } from "./math.js";
```

🟡 LEVEL 2: Default Export

5️⃣ Default Export Function

Question
Create calculator.js with a default export function multiply.

Answer

```
// calculator.js

export default function multiply(a, b) {

  return a * b;

}
```

6️⃣ Import Default Export

Question
Import multiply and use it.

Answer

```
import multiply from "./calculator.js";
```

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

7️⃣ Rename Default Import

Question

Import default export but rename it to mul.

Answer

```
import mul from "./calculator.js";


console.log(mul(3, 4)); // 12
```

🟠 LEVEL 3: Named + Default Together

8️⃣ Mixed Exports

Question

Create a file that exports:

- PI (named)
- area() (default)

Answer

```
// utils.js
export const PI = 3.14;


export default function area(r) {
  return PI * r * r;
}
```

9️⃣ Import Mixed Exports

Question
Import both PI and area.

Answer

```
import area, { PI } from "./utils.js";


console.log(area(5));

console.log(PI);
```

🟠 LEVEL 4: Renaming & Organization

🔟 Rename Named Import

Question
Rename add to sum while importing.

Answer

```
import { add as sum } from "./math.js";


console.log(sum(2, 3)); // 5
```

1️⃣1️⃣ Import Everything as Namespace

Question
Import all exports from math.js.

Answer

```
import * as math from "./math.js";


console.log(math.add(5, 3));

console.log(math.subtract(5, 3));
```

🔴 LEVEL 5: Browser Modules

1️⃣2️⃣ Enable Modules in HTML

Question

How do you enable import/export in HTML?

Answer

```
<script type="module" src="app.js"></script>
```

1️⃣3️⃣ Why Does This Fail?

Question

```
import { add } from "math.js";
```

Answer

❌ Missing relative path

✅ Correct:

```
import { add } from "./math.js";
```

1️⃣4️⃣ Common Mistake

Question

```
import add from "./math.js";
```

Why is this wrong?

Answer

- add is a **named export**
- Named imports **require {}**

✅ Correct:

```
import { add } from "./math.js";
```

🎯 Challenge Questions

1️⃣5️⃣ Convert to Module Code

Question

Convert this global code into modules:

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

Answer

```
// greet.js
export function greet(name) {
  return `Hello ${name}`;
}


// app.js
import { greet } from "./greet.js";
```

✅ Final Recap

✓ Use **named exports** for multiple utilities

✓ Use **default export** for one main feature

✓ Always use type="module" in HTML

✓ Use ./ for local files

✓ Modules are the foundation of modern JavaScript

JavaScript Data Structures: Map, Set, WeakMap

1️⃣ Why New Data Structures?

Before ES6, JavaScript mainly used:

- **Objects** → key–value storage
- **Arrays** → ordered lists

❌ Limitations:

- Object keys are always strings/symbols
- No guaranteed insertion order (historically)
- Difficult to manage uniqueness
- Memory management issues for temporary references

✅ **ES6 introduced Map, Set, WeakMap** to solve these problems.

🔷 MAP

2️⃣ What is a Map?

A **Map** is a collection of **key–value pairs** where:

- Keys can be **any data type** (object, function, number, etc.)
- Insertion order is preserved
- Size can be easily checked

Basic Syntax

```
const map = new Map();
```

3️⃣ Why Use Map Instead of Object?

Feature	Object	Map
Key types	String / Symbol only	Any type
Order guaranteed	❌	✅
Size property	❌	✅
Easy iteration	❌	✅

4️⃣ How to Use Map (Examples)

Adding & Getting Values

```
const userRoles = new Map();

userRoles.set("Amit", "Admin");
userRoles.set("Neha", "Editor");

console.log(userRoles.get("Amit")); // Admin
```

Using Objects as Keys

```
const user = { id: 1 };

const map = new Map();
map.set(user, "Active");

console.log(map.get(user)); // Active
```

✓ Impossible with plain objects (safely)

Checking & Deleting

```
map.has(user);    // true

map.delete(user);

map.size;         // number of entries
```

5️⃣ Iterating Over Map

```
for (const [key, value] of map) {

  console.log(key, value);

}
```

6️⃣ When to Use Map?

Use **Map** when:
✓ Keys are not strings
✓ You need insertion order
✓ Frequent add/remove operations
✓ Storing metadata related to objects

📌 Common use cases

- Caching
- Configuration storage
- Mapping DOM elements to data

🔷 SET

7️⃣ What is a Set?

A **Set** is a collection of **unique values**.

- No duplicates allowed
- Values can be of any type

- Maintains insertion order

Basic Syntax

```
const set = new Set();
```

8️⃣ Why Use Set?

❌ Array allows duplicates:

```
[1, 1, 2, 2, 3]
```

✅ Set automatically removes duplicates:

```
new Set([1, 1, 2, 2, 3]); // {1, 2, 3}
```

9️⃣ How to Use Set (Examples)

Adding & Checking Values

```
const ids = new Set();

ids.add(101);

ids.add(102);

ids.add(101); // ignored

ids.has(101); // true
```

Removing Values

```
ids.delete(102);

ids.clear();
```

Iterating Over Set

```
for (const value of ids) {

  console.log(value);
}
```

🔟 Convert Set ↔ Array

```
const uniqueNums = [...new Set([1, 2, 2, 3])];
```

✓ Very common interview question

1️⃣1️⃣ When to Use Set?

Use **Set** when:
✓ You need **unique values only**
✓ Removing duplicates from arrays
✓ Tracking visited items
✓ Membership checking (fast lookups)

📌 Real-world examples

- Unique user IDs
- Selected items
- Tags/categories

🔷 WEAKMAP

1️⃣2️⃣ What is a WeakMap?

A **WeakMap** is similar to Map, but:

- **Keys must be objects**
- Keys are held **weakly**
- Not iterable
- No size property

Basic Syntax

```
const weakMap = new WeakMap();
```

1️⃣3️⃣ Why WeakMap Exists?

👉 To **avoid memory leaks**

When an object key is no longer referenced anywhere else:

- It is **automatically garbage collected**
- Entry is removed from WeakMap

❌ Map keeps object references forever
✅ WeakMap releases memory safely

1️⃣4️⃣ WeakMap Example

```
let user = { name: "Ravi" };


const wm = new WeakMap();

wm.set(user, "LoggedIn");


user = null; // object eligible for GC
```

✓ No manual cleanup required

1️⃣5️⃣ What Can’t WeakMap Do? (Important)

❌ No iteration

```
wm.forEach(); // ❌
```

❌ No .size
❌ Keys must be objects

1️⃣6️⃣ When to Use WeakMap?

Use **WeakMap** when:
✓ Storing **temporary metadata**
✓ Associating data with objects
✓ Avoiding memory leaks
✓ Working with DOM elements

📌 Common use cases

- Private data
- Caching DOM-related info
- Framework internals

🔷 COMPARISON SUMMARY

1️⃣7️⃣ Map vs Set vs WeakMap

Feature	**Map**	**Set**	**WeakMap**
Stores	Key–Value	Unique Values	Key–Value
Duplicate allowed	❌ keys	❌ values	❌
Key type	Any	N/A	Object only
Iterable	✅	✅	❌
Memory safe	❌	❌	✅

1️⃣8️⃣ When NOT to Use Them?

❌ Use Object when:

- Simple key–value with string keys
- JSON data

❌ Use Array when:

- Order & duplicates matter

❌ Avoid WeakMap when:

- You need iteration or size

1️⃣9️⃣ Common Beginner Mistakes 🚫

❌ Expecting WeakMap to be iterable
❌ Using objects as keys in plain objects
❌ Using Set when duplicates are needed

2️⃣0️⃣ Final Summary (For Students)

- **Map** → flexible key–value storage
- **Set** → unique value collection
- **WeakMap** → memory-safe object metadata

⭐ Golden Rule

Map for mapping, Set for uniqueness, WeakMap for memory safety

🧩 Map, Set & WeakMap – Practice Problems

🟢 LEVEL 1: Basics

1️⃣ Create a Map

Question
Create a Map and store:

- "Amit" → "Admin"
- "Neha" → "Editor"

Answer

```
const roles = new Map();
```

```
roles.set("Amit", "Admin");

roles.set("Neha", "Editor");
```

2️⃣ Read from a Map

Question
Get the role of "Amit" from the map above.

Answer

```
roles.get("Amit"); // "Admin"
```

3️⃣ Check Existence in Map

Question
Check if "Neha" exists as a key.

Answer

```
roles.has("Neha"); // true
```

4️⃣ Size of a Map

Question
How do you find the number of entries in a Map?

Answer

```
roles.size;
```

🟡 LEVEL 2: Map – Practical Usage

5️⃣ Object as Map Key

Question
Use an object as a key in a Map.

Answer

```
const user = { id: 1 };
const statusMap = new Map();
statusMap.set(user, "Active");

statusMap.get(user); // "Active"
```

6️⃣ Iterate Over a Map

Question
Loop through keys and values in a Map.

Answer

```
for (const [key, value] of roles) {
  console.log(key, value);
}
```

🟠 LEVEL 3: Set Basics

7️⃣ Create a Set with Unique Values

Question
Create a Set from [1, 2, 2, 3, 3].

Answer

```
const uniqueSet = new Set([1, 2, 2, 3, 3]);
// {1, 2, 3}
```

8️⃣ Add & Check Values in Set

Question
Add 10 to a Set and check if it exists.

Answer

```
uniqueSet.add(10);

uniqueSet.has(10); // true
```

9️⃣ Remove Duplicates from Array

Question
Remove duplicates from [5, 6, 6, 7].

Answer

```
const unique = [...new Set([5, 6, 6, 7])];

// [5, 6, 7]
```

🔟 Iterate Over a Set

Question
Loop through all values in a Set.

Answer

```
for (const value of uniqueSet) {

  console.log(value);

}
```

🔴 LEVEL 4: WeakMap

1️⃣1️⃣ Create a WeakMap

Question
Create a WeakMap and add an object key.

Answer

```
const wm = new WeakMap();


let obj = { name: "Ravi" };
```

```
wm.set(obj, "LoggedIn");
```

1️⃣2️⃣ Why WeakMap Does Not Cause Memory Leak?

Question
Explain with code.

Answer

```
let user = { id: 10 };


const wm = new WeakMap();

wm.set(user, "Active");


user = null; // object can be garbage collected
```

✓ Entry is automatically removed
✓ No manual cleanup needed

1️⃣3️⃣ Invalid WeakMap Usage

Question
Why is this wrong?

```
wm.set("name", "value");
```

Answer
❌ WeakMap keys must be **objects**, not primitives.

1️⃣4️⃣ Can We Iterate WeakMap?

Question
Can you loop over a WeakMap?

Answer

```
// ❌ No
```

✓ WeakMap is **not iterable**
✓ No .size, .keys(), .values()

🟣 LEVEL 5: Comparison & Thinking

1️⃣5️⃣ Choose the Right Data Structure

Question
Which should you use and why?

Scenario	Correct Choice
Unique values	Set
Object → metadata	WeakMap
Key–value with any key type	Map

Answer

Set → uniqueness

Map → flexible keys

WeakMap → memory-safe object metadata

1️⃣6️⃣ Convert Object to Map

Question

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

Answer

```
const map = new Map(Object.entries(obj));
```

🎯 Challenge Question

Why Use Map Instead of Object Here?

Question

const map = new Map();

map.set({ id: 1 }, "User Data");

Answer

✓ Objects can be keys
✓ No string coercion
✓ Safe & reliable reference mapping

✅ Final Recap

✓ **Map** → key–value with flexible keys
✓ **Set** → unique values only
✓ **WeakMap** → memory-safe object associations

Error Handling in JavaScript – try…catch

1️⃣ What is Error Handling?

Error handling is the process of **detecting, managing, and responding to runtime errors** so that:

- The program doesn’t crash
- Users see meaningful messages
- Developers can debug issues easily

JavaScript uses:

```
try { ... }

catch (error) { ... }

finally { ... }
```

2️⃣ Why Do We Need Error Handling?

❌ Without error handling:

- App crashes
- Blank screens
- Poor user experience

✅ With error handling:
✓ Graceful failures
✓ Better debugging
✓ Stable applications
✓ Cleaner control flow

3️⃣ When Should You Use try…catch?

Use try…catch when:
✓ Code may fail at runtime
✓ Working with user input

✓ Parsing JSON
✓ Calling APIs
✓ Using async/await
✓ Accessing unknown object properties

⚠️ **Do NOT use try…catch for normal control flow**

4️⃣ Basic Syntax

```
try {
  // risky code
} catch (error) {
  // handle error
} finally {
  // always runs (optional)
}
```

5️⃣ What is an Error Object?

When an error occurs, JavaScript creates an **Error object** with:

- name
- message
- stack

```
try {
  undefinedFunction();
} catch (error) {
  console.log(error.message);
}
```

6️⃣ Simple Example

```
try {
  let result = 10 / x;
} catch (err) {
  console.log("Something went wrong");
}
```

📌 Prevents app crash

7️⃣ Catching Specific Errors

```
try {
  JSON.parse("{ bad json }");
} catch (error) {
  console.log("Invalid JSON format");
}
```

✓ Very common real-world use case

8️⃣ Throwing Custom Errors

```
function withdraw(amount) {
  if (amount <= 0) {
    throw new Error("Invalid amount");
  }
  return "Success";
}
```

✓ Used for validation
✓ Improves clarity

9️⃣ Using finally

```
try {
  console.log("Try block");
} catch {
  console.log("Catch block");
} finally {
  console.log("Always runs");
}
```

📌 Useful for:

- Closing connections
- Cleaning resources
- Logging

🔷 try…catch with async/await

🔟 Why Needed in Async Code?

Promises reject asynchronously → must be handled.

```
async function loadData() {
  try {
    const res = await fetch("invalid-url");
  } catch (error) {
    console.log("Network error");
  }
}
```

✓ Prevents unhandled promise rejection

1️⃣1️⃣ try…catch vs .then().catch()

Aspect	**try…catch**	**.catch()**
Async/Await	✅ Best	❌ Not used
Promise chain	❌	✅
Readability	✅ High	⚠️ Medium

🧩 PRACTICE QUESTIONS WITH ✅ ANSWERS

🟢 LEVEL 1: Basics

1️⃣ Handle Reference Error

Question

```
console.log(x);
```

Answer

```
try {
  console.log(x);
} catch (error) {
  console.log("Variable is not defined");
}
```

2️⃣ Catch JSON Parsing Error

Question

```
const data = "{name:'John'}";
```

Answer

```
try {
  JSON.parse(data);
} catch {
  console.log("Invalid JSON");
}
```

3️⃣ Using finally

Question
Demonstrate finally.

Answer

```
try {
  console.log("Start");
} catch {
  console.log("Error");
} finally {
  console.log("End");
}
```

🟡 LEVEL 2: Throwing Errors

4️⃣ Custom Error

Question
Throw error if age is below 18.

Answer

```
function checkAge(age) {
  if (age < 18) {
    throw new Error("Not eligible");
  }
  return "Eligible";
}
```

5️⃣ Handle Custom Error

Question

Handle error from checkAge().

Answer

```
try {
  checkAge(15);
} catch (e) {
  console.log(e.message);
}
```

🟠 LEVEL 3: Real-World Scenarios

6️⃣ Safe Object Access

Question

```
const user = {};
console.log(user.address.city);
```

Answer

```
try {
  console.log(user.address.city);
} catch {
  console.log("Address not available");
}
```

7️⃣ Division Validation

Question
Throw error when dividing by zero.

Answer

```
function divide(a, b) {
  if (b === 0) {
    throw new Error("Cannot divide by zero");
  }
  return a / b;
}
```

8️⃣ Handle API Failure (Mock)

Question
Simulate API error handling.

Answer

```
async function fetchData() {
  try {
    throw new Error("Server down");
  } catch (e) {
    console.log("API Error:", e.message);
  }
}
```

🔴 LEVEL 4: Understanding Behavior

9️⃣ Does try…catch Catch Syntax Errors?

Question

```
try {
  let a = ;
} catch {}
```

Answer
❌ No
✓ Syntax errors happen before execution

🔟 Catch Without Error Variable

Question

Answer

```
try {
  JSON.parse("bad");
} catch {
  console.log("Parsing failed");
}
```

🎯 Interview-Style Questions

1️⃣1️⃣ When NOT to Use try…catch?

Answer

- For normal conditions
- For simple if/else logic
- Inside tight loops (performance)

1️⃣2️⃣ try…catch vs if/else

Answer

- if/else → predictable conditions
- try…catch → unpredictable runtime errors

✅ Final Summary

✓ try → risky code
✓ catch → handle runtime errors
✓ finally → cleanup
✓ throw → custom validation errors
✓ Essential for async code

⭐ Golden Rule

Use try…catch for unexpected failures, not normal logic

JavaScript Best Practices & Coding Standards

1️⃣ What Are JavaScript Best Practices?

Best practices are **recommended ways of writing JavaScript code** that make it:

- Readable
- Maintainable
- Scalable
- Less error-prone

They are **not rules enforced by the language**, but **standards followed by professional developers and teams**.

2️⃣ Why Are Coding Standards Important?

Without standards:

- Code is hard to read
- Bugs are difficult to trace
- Team collaboration suffers

With standards:
✓ Cleaner code
✓ Fewer bugs
✓ Easier debugging
✓ Better teamwork
✓ Production-ready applications

👉 **Good code is read more times than it is written**

3️⃣ When Should You Follow Best Practices?

You should follow them:

- From **day one of learning**
- In **real projects**

- During **code reviews**
- For **interviews**
- In **team environments**

📌 **Small projects become big projects**

🔷 CODE STRUCTURE & READABILITY

4️⃣ Use Meaningful Variable & Function Names

❌ **Bad**

```
let x = 10;

function a(b) {}
```

✅ **Good**

```
let totalPrice = 10;

function calculateTax(amount) {}
```

✓ Code explains itself
✓ Less comments needed

5️⃣ Follow Consistent Naming Conventions

Type	**Convention**	**Example**
Variables	camelCase	userName
Functions	camelCase	getUserData()
Classes	PascalCase	UserService
Constants	UPPER_CASE	MAX_LIMIT

6️⃣ Keep Functions Small (Single Responsibility)

❌ Bad (does too much):

```
function handleUser() {
  validate();
  save();
  sendEmail();
}
```

✅ Better:

```
function validateUser() {}
function saveUser() {}
function sendWelcomeEmail() {}
```

📌 **One function = one job**

🔷 VARIABLES & DATA HANDLING

7️⃣ Prefer const and let Over var

❌ **Avoid**

```
var count = 10;
```

✅ **Use**

```
const MAX_USERS = 100;
let currentUsers = 5;
```

✓ Block scoped
✓ Prevents bugs

8️⃣ Avoid Global Variables

❌ Bad

```
userCount = 10;
```

✅ Good

```
function updateCount() {
  let userCount = 10;
}
```

📌 Globals cause conflicts and bugs

9️⃣ Use Strict Equality (===)

❌ Bad

```
if (age == "18") {}
```

✅ Good

```
if (age === 18) {}
```

✓ No type coercion
✓ Predictable behavior

🔷 FUNCTIONS & LOGIC

🔟 Use Default Parameters

❌ Bad

```
function greet(name) {
  name = name || "Guest";
}
```

✅ Good

```
function greet(name = "Guest") {}
```

1️⃣1️⃣ Avoid Deep Nesting (Pyramid of Doom)

❌ Bad

```
if (a) {
  if (b) {
    if (c) {}
  }
}
```

✅ Good

```
if (!a) return;
if (!b) return;
if (!c) return;
```

📌 Early returns improve readability

1️⃣2️⃣ Use Arrow Functions Carefully

✅ Good for callbacks:

```
items.map(item => item.price);
```

❌ Avoid when this is needed:

```
// inside object methods
```

🔷 ERROR HANDLING & SAFETY

1️⃣3️⃣ Always Handle Errors

❌ Bad

```
JSON.parse(data);
```

✅ Good

```
try {
  JSON.parse(data);
} catch {
  console.log("Invalid JSON");
}
```

1️⃣4️⃣ Validate Inputs Early

```
function withdraw(amount) {
  if (amount <= 0) {
    throw new Error("Invalid amount");
  }
}
```

✓ Fail fast
✓ Safer code

🔷 ASYNC & PERFORMANCE

1️⃣5️⃣ Prefer async/await Over Promise Chains

❌ Hard to read

```
fetch(url).then().then().catch();
```

✅ Cleaner

```
try {
  const res = await fetch(url);
} catch {}
```

1️⃣6️⃣ Avoid Blocking Code

❌ Bad

```
while(true) {}
```

✅ Use async operations

🔷 MODERN JAVASCRIPT PRACTICES

1️⃣7️⃣ Use Destructuring

❌ Bad

```
const name = user.name;
```

✅ Good

```
const { name } = user;
```

1️⃣8️⃣ Use Spread for Immutability

```
const updatedUser = { ...user, age: 31 };
```

✓ Prevents accidental mutation

1️⃣9️⃣ Use Map / Set When Appropriate

- Map → key–value with any key
- Set → unique values
- Avoid forcing everything into arrays or objects

🔷 COMMENTS, FORMATTING & TOOLS

2️⃣0️⃣ Write Useful Comments (Not Obvious Ones)

❌ Bad

```
// add two numbers

a + b;
```

✅ Good

```
// Apply discount only for premium users
```

2️⃣1️⃣ Use Consistent Formatting

✓ Indentation
✓ Line breaks
✓ One statement per line

👉 Use tools like:

- ESLint
- Prettier

2️⃣2️⃣ Remove Debug Code

❌ Bad

```
console.log("test");
```

✅ Remove before production

🔷 SECURITY & MAINTAINABILITY

2️⃣3️⃣ Never Trust User Input

```
if (typeof age !== "number") return;
```

2️⃣4️⃣ Avoid Magic Numbers

❌ Bad

```
if (score > 90) {}
```

✅ Good

```
const PASS_MARK = 90;
```

🔷 FINAL CHECKLIST (Quick Review)

✅ Use const / let
✅ Meaningful names
✅ Small functions
✅ Handle errors
✅ Avoid globals
✅ Use modern syntax
✅ Follow consistency

⭐ Golden Rules for Students

Code for humans first, computers second
Readable code is maintainable code
Best practices are habits, not shortcuts

🧩 JavaScript Best Practices – Practice Problems with ✅ Answers

🟢 LEVEL 1: Readability & Naming

1️⃣ Improve Variable Names

Question
Refactor the code to follow best naming practices:

```
let x = 500;

let y = 0.18;
```

Answer

```
const totalAmount = 500;

const taxRate = 0.18;
```

2️⃣ Function Naming

Question
Rename the function to reflect its purpose:

```
function a(b) {

  return b * 2;

}
```

Answer

```
function doubleValue(value) {

  return value * 2;

}
```

🟡 LEVEL 2: Variables & Scope

3️⃣ Replace var

Question
Refactor using modern variable declarations:

```
var count = 10;

count = 20;
```

Answer

```
let count = 10;

count = 20;
```

4️⃣ Use const Where Possible

Question

```
let PI = 3.14;
```

Answer

```
const PI = 3.14;
```

5️⃣ Avoid Global Variables

Question

Fix the global variable issue:

```
total = 100;
```

Answer

```
function calculateTotal() {
  const total = 100;
}
```

🟠 LEVEL 3: Conditions & Logic

6️⃣ Use Strict Equality

Question

```
if (age == "18") {
  console.log("Adult");
}
```

Answer

```
if (age === 18) {
  console.log("Adult");
}
```

7️⃣ Avoid Deep Nesting

Question

```
if (isLoggedIn) {
  if (isAdmin) {
    showDashboard();
  }
}
```

Answer

```
if (!isLoggedIn) return;
if (!isAdmin) return;

showDashboard();
```

8️⃣ Avoid Magic Numbers

Question

```
if (score > 90) {
  grade = "A";
}
```

Answer

```
const A_GRADE_SCORE = 90;

if (score > A_GRADE_SCORE) {
  grade = "A";
}
```

🔴 LEVEL 4: Functions & Structure

9️⃣ Use Default Parameters

Question

```
function greet(name) {
  name = name || "Guest";
  return "Hello " + name;
}
```

Answer

```
function greet(name = "Guest") {
  return `Hello ${name}`;
}
```

🔟 Small Functions (Single Responsibility)

Question

Refactor:

```
function processUser() {
  validateUser();
  saveUser();
  sendEmail();
}
```

Answer

```
function validateUser() {}
function saveUser() {}
function sendEmail() {}
```

🟣 LEVEL 5: Error Handling & Safety

1️⃣1️⃣ Add Error Handling

Question

```
JSON.parse(data);
```

Answer

```
try {
  JSON.parse(data);
} catch {
  console.log("Invalid JSON data");
}
```

1️⃣2️⃣ Validate Input Early

Question

```
function withdraw(amount) {
  balance -= amount;
}
```

Answer

```
function withdraw(amount) {
  if (amount <= 0) {
    throw new Error("Invalid amount");
  }
  balance -= amount;
}
```

🟤 LEVEL 6: Modern JavaScript

1️⃣3️⃣ Use Destructuring

Question

```
const name = user.name;

const age = user.age;
```

Answer

```
const { name, age } = user;
```

1️⃣4️⃣ Use Spread for Immutability

Question

```
user.age = 31;
```

Answer

```
const updatedUser = { ...user, age: 31 };
```

1️⃣5️⃣ Prefer async/await

Question

```
fetch(url)

  .then(res => res.json())

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

Answer

```
try {

  const res = await fetch(url);

  const data = await res.json();

  console.log(data);

} catch (e) {

  console.error(e);

}
```

🎯 Scenario-Based Questions

1️⃣6️⃣ Which is Better and Why?

Question

```
if (!user) {
  return;
}
console.log(user.name);
```

Answer

✓ Early return avoids nested logic
✓ Improves readability

1️⃣7️⃣ What’s Wrong Here?

Question

```
console.log("Debugging...");
```

Answer

❌ Debug code left in production
✓ Should be removed or replaced with proper logging

Debugging Techniques & Tools (JavaScript)

1️⃣ What is Debugging?

Debugging is the process of **finding, understanding, and fixing errors (bugs)** in your code.

A **bug** can be:

- A runtime error (app crashes)
- A logical error (wrong output)
- A performance issue (slow app)

👉 Debugging is **not just fixing errors**, it's **understanding why the code behaves the way it does**.

2️⃣ Why is Debugging Important?

Without debugging:

- Applications crash
- Wrong results are shown
- Bugs reach production
- User trust is lost

With proper debugging:
✓ Faster bug fixing
✓ Better code understanding
✓ Higher code quality
✓ Confident development

📌 **Professional developers spend more time debugging than writing code**

3️⃣ When Do You Debug?

You debug when:
✓ Code throws errors
✓ Output is incorrect
✓ App behaves unexpectedly
✓ Performance is slow
✓ You refactor existing code
✓ You integrate APIs

🔷 BASIC DEBUGGING TECHNIQUES

4️⃣ Using console.log() (Most Common)

What

Printing values to the console to inspect code execution.

Example

```
function add(a, b) {
  console.log("a:", a);
  console.log("b:", b);
  return a + b;
}
```

Why

✓ Quick inspection
✓ Beginner-friendly

When

- Small bugs
- Learning phase
- Temporary checks

⚠️ **Remove logs before production**

5️⃣ Using console.error(), console.warn()

console.warn("This is a warning");

console.error("This is an error");

✓ Better visibility
✓ Color-coded messages

6️⃣ Reading Error Messages Carefully

Example

Uncaught TypeError: Cannot read properties of undefined

Why

- Error messages usually **tell you what and where**
- Line number + file name helps locate bug

📌 **Never ignore error messages**

🔷 BROWSER DEBUGGING TOOLS

7️⃣ Browser Developer Tools (DevTools)

What

Built-in tools in browsers for debugging JavaScript.

Why

✓ Real-time debugging
✓ Powerful inspection
✓ Industry standard

When

- Frontend development
- UI-related bugs

- Performance analysis

8️⃣ Breakpoints (MOST IMPORTANT)

What

Pause code execution at a specific line.

Example

```
function calculateTotal(price) {
  debugger; // manual breakpoint
  return price * 1.18;
}
```

Or set breakpoints directly in DevTools.

Why

✓ Inspect variables step-by-step
✓ Understand execution flow

When

- Complex logic
- Loops
- Conditional issues

9️⃣ Step Controls in Debugger

Action	**Purpose**
Step Over	Execute next line
Step Into	Go inside function
Step Out	Exit function
Resume	Continue execution

🔷 DEBUGGING COMMON ERROR TYPES

🔟 Debugging Reference Errors

```
console.log(x);
```

❌ x is not defined

Fix

```
let x = 10;

console.log(x);
```

1️⃣1️⃣ Debugging Type Errors

```
user.name.toUpperCase();
```

❌ user is undefined

Fix

```
if (user && user.name) {
  console.log(user.name.toUpperCase());
}
```

1️⃣2️⃣ Debugging Logical Errors (Hardest)

Buggy Code

```
function isEven(num) {
  return num % 2 === 1;
}
```

Debug Technique

```
console.log(num % 2);
```

Fix

```
return num % 2 === 0;
```

🔷 DEBUGGING ASYNC CODE

1️⃣3️⃣ Debugging Promises

```
fetch(url)
  .then(res => console.log(res))
  .catch(err => console.error(err));
```

✓ Always add .catch()

1️⃣4️⃣ Debugging async/await

```
async function loadData() {
  try {
    const res = await fetch(url);
    console.log(res);
  } catch (error) {
    console.error("Fetch failed", error);
  }
}
```

✓ Prevents silent failures

🔷 ADVANCED DEBUGGING TECHNIQUES

1️⃣5️⃣ Using debugger Keyword

```
for (let i = 0; i < 5; i++) {
  debugger;
  console.log(i);
}
```

✓ Pauses loop execution
✓ Inspect iteration behavior

1️⃣6️⃣ Debugging with DevTools Watch & Scope

- Watch variables
- Inspect call stack
- Check local & global scope

📌 Helps understand **how values change over time**

1️⃣7️⃣ Network Debugging (APIs)

What to Check

✓ Request URL
✓ Status code
✓ Response data
✓ Headers

📌 Many bugs are **backend or network related**

🔷 DEBUGGING BEST PRACTICES

1️⃣8️⃣ Isolate the Problem

❌ Debug everything at once
✅ Comment out sections and test step-by-step

1️⃣9️⃣ Reproduce the Bug Consistently

✓ Same steps
✓ Same input
✓ Same environment

📌 If you can reproduce it, you can fix it.

2️⃣0️⃣ Fix Root Cause, Not Symptoms

❌ Patch over error
✅ Understand why it happens

🔷 TOOLS USED IN INDUSTRY

2️⃣1️⃣ Common Debugging Tools

Tool	Usage
Browser DevTools	Frontend debugging
VS Code Debugger	Breakpoints & stepping
ESLint	Catch bugs early
Source Maps	Debug minified code
Logging tools	Production debugging

2️⃣2️⃣ Debugging Mindset (Very Important)

✓ Be patient
✓ Read errors slowly
✓ Assume code is wrong, not the computer
✓ One fix at a time

✅ Final Summary

- Debugging is a **core developer skill**
- Tools help, but **thinking matters more**
- Breakpoints > console.log for complex bugs
- Always handle errors explicitly

⭐ Golden Rule

Don’t guess — debug