JavaScript Foundation – Training Day-7

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
Synchronous vs asynchronous programming
Callback functions and callback hell
Promises: creation and consumption
Promise chaining and error handling
Async/await syntax
Fetch API for HTTP requests
Working with JSON data
Error handling in async code

Synchronous Programming

🔷What is Synchronous Programming?

Synchronous programming means **code runs line by line**, and **each task must finish before the next one starts**.

- Execution is **blocking**
- The program **waits** for the current task to complete
- Follows a **top-to-bottom** order

🔁 How it works

1. Task 1 starts
2. Task 1 finishes
3. Task 2 starts
4. Task 2 finishes

🔷 Example (Synchronous)

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

function syncTask() {
  for (let i = 0; i < 3; i++) {
    console.log("Processing", i);
  }
}
syncTask();
console.log("End");
```

📌 Output

Start

Processing 0

Processing 1

Processing 2

End

➡️ **End waits** until syncTask() finishes.

🔷 Why use Synchronous Programming?

✅ Simple to **understand and debug**
✅ Predictable execution order
✅ Good for **small, fast operations**

🔷 When to use Synchronous Programming?

Use synchronous code when:

- Operations are **quick**
- No waiting for external resources
- Logic must execute in **strict order**

🧠 Common use cases

- Calculations
- Loops
- Data transformations
- Validation logic

❌ Problem with Synchronous Code

If a task takes **too long**, everything else **freezes**.

```
alert("This blocks everything!");
```

👉 In browsers, this can **freeze the UI**.

Asynchronous Programming

🔷 What is Asynchronous Programming?

Asynchronous programming allows **long-running tasks** to run **in the background**, without blocking the main program.

- Execution is **non-blocking**

- Other code continues running
- Results are handled **later**

🔷 Example (Asynchronous)

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

setTimeout(() => {
  console.log("Async task done");
}, 2000);

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

📌 Output

Start

End

Async task done

➡️ The program **does not wait** for setTimeout.

🔷 Why use Asynchronous Programming?

✅ Keeps applications **responsive**
✅ Prevents UI freezing
✅ Handles **slow operations efficiently**

🔷 When to use Asynchronous Programming?

Use async code when:

- Waiting for **network requests**
- Accessing **files or databases**
- Using **timers**
- Handling **user interactions**

🧠 Common use cases

- Fetching data from API
- Reading files
- setTimeout / setInterval
- Event handling
- Animations

Real-World Example (Sync vs Async)

🛒 Synchronous (Bad UX)

```
// User waits until data loads
const data = getDataFromServer(); // blocks
display(data);
```

❌ Page freezes until data arrives

🛒 Asynchronous (Good UX)

```
fetch("https://api.example.com/data")
  .then(response => response.json())
  .then(data => {
    display(data);
  });

console.log("Page is still responsive");
```

✅ Page works while data loads

Key Differences (Interview-Friendly Table)

Feature	Synchronous	Asynchronous
Execution	One after another	Background execution
Blocking	Yes	No
Speed (UX)	Can be slow	Faster & smoother
Complexity	Simple	Slightly complex
Use cases	Calculations	APIs, timers, events

JavaScript Is…

⚠️ Important Concept

JavaScript is:

- **Single-threaded**
- Uses **asynchronous mechanisms** to avoid blocking

➡️ Achieved using:

- Callbacks
- Promises
- async/await
- Event loop

Simple Teaching Analogy 🎓

🧍 Synchronous

One person using an ATM
Everyone waits in line

🚗 Asynchronous

Multiple online banking users
Everyone continues independently

Quick Summary

- **Synchronous** = wait → execute → move on
- **Asynchronous** = start → continue → handle result later
- Sync is **simple**, async is **powerful**
- Modern web apps rely heavily on **asynchronous programming**

Next topic: Callback Functions and Callback Hell

Callback Functions and Callback Hell (async JavaScript)

Callback Functions

🔷 What is a Callback Function?

A **callback function** is a function that is **passed as an argument** to another function and is **executed later**, usually **after a task completes**.

👉 “Call me back when you’re done”

🔷 Simple Example (Synchronous Callback)

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

function sayBye() {
  console.log("Goodbye!");
}
greet("Srikanth", sayBye);
```

📌 Output

Hello Srikanth

Goodbye!

➡️ sayBye is passed **without ()** and executed later.

🔷 Why use Callback Functions?

Callbacks are used to:

✅ Control **execution order**
✅ Handle **asynchronous operations**
✅ Avoid blocking the main thread
✅ Execute code **after a task finishes**

🔷 When to use Callback Functions?

Use callbacks when:

- You don't know **when a task will finish**
- You need a **result later**
- Working with:
  - Timers
  - Events
  - Async APIs

🔷 Asynchronous Callback Example

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

setTimeout(function () {
  console.log("This runs after 2 seconds");
}, 2000);
console.log("End");
```

📌 Output

Start

End

This runs after 2 seconds

➡️ The callback runs **after** the timer completes.

Callbacks in Real-World Scenarios

📌 Event Handling

```
button.addEventListener("click", function () {
  console.log("Button clicked");
});
```

➡️ The callback runs **only when the event occurs**.

📌 API Simulation with Callback

```
function getData(callback) {
  setTimeout(() => {
    callback("Data received");
  }, 1000);
}
getData(function (result) {
  console.log(result);
});
```

Callback Hell

🔥 What is Callback Hell?

Callback Hell happens when **callbacks are nested inside other callbacks**, making code:

❌ Hard to read
❌ Hard to debug
❌ Hard to maintain

It is also called:

- **Pyramid of Doom**
- **Arrow-shaped code**

🔷 Example of Callback Hell

```
setTimeout(() => {
  console.log("Step 1");
  setTimeout(() => {
    console.log("Step 2");
    setTimeout(() => {
      console.log("Step 3");
      setTimeout(() => {
        console.log("Step 4");
      }, 1000);
    }, 1000);
  }, 1000);
}, 1000);
```

📉 Code goes **rightward**, readability drops.

🔷 Why Callback Hell is a Problem?

❌ Deep nesting
❌ Error handling is messy
❌ Difficult to reuse logic
❌ Debugging becomes painful

🔷 When Does Callback Hell Occur?

It occurs when:

- Multiple async tasks depend on each other
- Using callbacks for **sequential async logic**
- No structure or abstraction

Real-World Callback Hell Example

```
login(user, () => {
  getProfile(() => {
    getOrders(() => {
      makePayment(() => {
        console.log("Order complete");
      });
    });
  });
});
```

➡️ Very common in **old JavaScript codebases**

How to Reduce Callback Hell

✅ Solution 1: Named Functions

```
function step1() {
  console.log("Step 1");
  setTimeout(step2, 1000);
}
function step2() {
  console.log("Step 2");
  setTimeout(step3, 1000);
}
function step3() {
  console.log("Step 3");
}
step1();
```

✔ Improves readability

✖ Still callback-based

✅ Solution 2: Promises

```
doStep1()
  .then(doStep2)
  .then(doStep3)
  .catch(error => console.log(error));
```

➡ Flat structure

➡ Better error handling

✅ Solution 3: async / await (Best)

```
async function runSteps() {
  await doStep1();
  await doStep2();
  await doStep3();
}
runSteps();
```

✔ Looks synchronous

✔ Easy to read and debug

When Should You Still Use Callbacks?

Callbacks are still useful for:

✅ Event listeners
✅ Simple async tasks
✅ Array methods (map, filter, forEach)
✅ Libraries expecting callbacks

🔷 Example: Array Callback

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

numbers.forEach(function (num) {
  console.log(num * 2);
});
```

Key Differences (Exam / Interview Table)

Feature	Callback	Callback Hell
Structure	Flat	Deeply nested
Readability	Good	Poor
Maintainability	Easy	Difficult
Error Handling	Simple	Complex

Quick Summary

- Callback = function passed to another function
- Used heavily in async operations
- Callback Hell = excessive nesting
- Avoid using callbacks for **complex async flows**
- Prefer **Promises / async-await** for scalability

Next topic: Promises: creation and consumption

Promises: Creation and Consumption (async JavaScript)

Promises (Overview)

🔷 What is a Promise?

A **Promise** is a JavaScript object that represents the **eventual completion or failure** of an asynchronous operation.

➡️ It acts as a **placeholder for a future value**.

A promise can be in one of three states:

State	Meaning
pending	Operation is ongoing
fulfilled	Operation completed successfully
rejected	Operation failed

🔷 Why Promises Were Introduced?

Promises solve problems of **callbacks**, especially:

❌ Callback hell
❌ Difficult error handling
❌ Hard-to-read nested code

✅ Promises provide:

- Flat, readable code
- Centralized error handling
- Better async flow control

🔷 When to Use Promises?

Use promises when:

- Handling **asynchronous operations**
- Multiple async steps depend on each other
- You want **clean, maintainable code**

Common use cases

- API calls
- File operations
- Database queries
- Timers
- Any async task that may succeed or fail

Creating a Promise

🔷 What Does "Creating a Promise" Mean?

Creating a promise means defining:

- **What async work to do**
- **When to resolve**
- **When to reject**

🔷 Syntax

```
const promise = new Promise((resolve, reject) => {
  // async operation
});
```

- resolve(value) → success
- reject(error) → failure

🔷 Simple Promise Creation Example

```
const myPromise = new Promise((resolve, reject) => {
  const success = true;

  if (success) {
    resolve("Operation successful");
  } else {
    reject("Operation failed");
  }
});
```

➡️ Promise is **created**, but not yet consumed.

🔷 Real Async Example (with setTimeout)

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Data received");
    }, 2000);
  });
}
```

➡️ This promise resolves **after 2 seconds**.

Consuming a Promise

🔷 What is Promise Consumption?

Consuming a promise means **using its result** once it is:

- fulfilled → .then()
- rejected → .catch()

🔷 Basic Consumption Example

```
fetchData()
  .then(result => {
    console.log(result);
  })
  .catch(error => {
    console.error(error);
  });
```

📌 Output (after 2 seconds)

Data received

🔷 Why Consume Promises This Way?

- .then() handles success
- .catch() handles errors
- Code stays **flat and readable**

Promise Chaining

🔷 What is Promise Chaining?

Promise chaining allows **multiple async operations** to run **in sequence**, where each step depends on the previous one.

🔷 Example: Promise Chaining

```
function step1() {
  return Promise.resolve("Step 1 done");
}

function step2(prev) {
  return Promise.resolve(prev + " → Step 2 done");
}

function step3(prev) {
  return Promise.resolve(prev + " → Step 3 done");
}
```


```
step1()
  .then(step2)
  .then(step3)
  .then(result => console.log(result))
  .catch(err => console.error(err));
```

📌 **Output**

Step 1 done → Step 2 done → Step 3 done

🔷 Why Use Chaining?

✅ Avoids nested callbacks
✅ Cleaner logic flow
✅ Single error handling

Promise Rejection & Error Handling

🔷 Rejecting a Promise

```
function login() {
  return new Promise((resolve, reject) => {
    reject("Invalid credentials");
  });
}
```

🔷 Handling Errors with .catch()

```
login()
  .then(result => console.log(result))
  .catch(error => console.error("Error:", error));
```

➡️ .catch() catches **any error** in the chain.

🔷 .finally() (Cleanup Code)

```
fetchData()
  .then(data => console.log(data))
  .catch(err => console.log(err))
  .finally(() => {
    console.log("Operation completed");
  });
```

✔ Runs **always** (success or failure)

Promise vs Callback (Quick Comparison)

Feature	Callback	Promise
Readability	Low	High
Nesting	Deep	Flat
Error handling	Messy	Clean
Chaining	Hard	Easy

When NOT to Use Promises?

Avoid promises when:

❌ Task is synchronous
❌ Simple logic doesn't need async
❌ Over-engineering small code

Summary

- Promises are **eager** (start immediately)
- .then() returns a new promise
- Errors bubble to nearest .catch()
- Promises improve **code maintainability**

Quick Summary

- Promise = future value
- Created using new Promise()
- Consumed using .then() and .catch()
- Chaining solves callback hell
- Foundation for async / await

Promise Chaining and Error Handling

Promise Chaining

🔷 What is Promise Chaining?

Promise chaining is the process of **linking multiple .then() calls**, where **each .then() returns a new promise** and passes its result to the next one.

➡️ Each step waits for the **previous promise to resolve**.

🔷 Why Promise Chaining Exists?

Promise chaining solves:

❌ Callback hell
❌ Deeply nested async code
❌ Hard-to-manage execution order

✅ Provides:

- Linear, readable async flow
- Better maintainability
- Centralized error handling

🔷 When to Use Promise Chaining?

Use promise chaining when:

- Async tasks must run **in sequence**
- Each step depends on the previous result
- You want **clean, flat async code**

Common scenarios

- Login → fetch profile → fetch orders
- Load config → load data → render UI
- File upload → process → save result

🔷 Basic Promise Chaining Example

```
function step1() {
  return Promise.resolve("Step 1 completed");
}

function step2(data) {
  return Promise.resolve(data + " → Step 2 completed");
}

function step3(data) {
  return Promise.resolve(data + " → Step 3 completed");
}

step1()
  .then(step2)
  .then(step3)
  .then(result => console.log(result));
```

📌 Output

Step 1 completed → Step 2 completed → Step 3 completed

🔷 Important Rule

Always return a promise or value inside .then()

❌ Wrong

```
.then(data => {
  step2(data);
})
```

✅ Correct

```
.then(data => {
  return step2(data);
})
```

Promise Error Handling

🔷 What is Promise Error Handling?

Promise error handling is the mechanism to **catch and handle failures** using:

- .catch()
- .finally()

Errors can occur due to:

- Network failure
- Invalid data
- Manual reject()
- JavaScript runtime errors

🔷 Why Proper Error Handling Is Important?

Without proper error handling:
❌ App may crash
❌ Bugs go unnoticed
❌ User experience suffers

With .catch():
✅ One place to handle all errors
✅ Clean and predictable behavior

🔷 When to Use .catch()?

Use .catch():

- At the **end of the promise chain**
- To handle **any error** in the chain

🔷 Basic Error Handling Example

```
function getData() {
  return new Promise((resolve, reject) => {
    reject("Server not reachable");
  });
}

getData()
  .then(data => console.log(data))
  .catch(error => console.error("Error:", error));
```

📌 Output

Error: Server not reachable

Error Propagation in Promise Chains

🔷 What is Error Propagation?

If a promise **fails at any step**, JavaScript:

➡️ Skips remaining .then() blocks

➡️ Jumps directly to .catch()

🔷 Example: Error Stops the Chain

```
function step1() {
  return Promise.resolve("Step 1 done");
}

function step2() {
  return Promise.reject("Step 2 failed");
}

function step3() {
  return Promise.resolve("Step 3 done");
}
step1()
  .then(step2)
  .then(step3)
  .catch(error => console.error(error));
```

📌 Output

Step 2 failed

➡️ step3() never runs.

Handling Errors at Specific Steps

🔷 Use .catch() in the Middle

```
step1()
  .then(step2)
  .catch(err => {
    console.log("Handled step2 error:", err);
    return "Recovered value";
  })
  .then(step3)
  .then(result => console.log(result));
```

✔️ Allows **recovery and continuation**

Throwing Errors Manually

🔷 What Does throw Do in Promises?

Throwing an error inside .then():
➡️ Automatically sends control to .catch()

🔷 Example

```
fetchData()
  .then(data => {
    if (!data) {
      throw new Error("No data found");
    }
    return data;
  })
  .catch(err => console.error(err.message));
```

.finally() – Cleanup Logic

🔷 What is .finally()?

.finally() executes **regardless of success or failure**.

✔ Ideal for:

- Hiding loaders
- Closing connections
- Logging completion

🔷 Example

```
fetchData()
  .then(data => console.log(data))
  .catch(err => console.error(err))
  .finally(() => {
    console.log("Operation finished");
  });
```

7️⃣ Real-World Example (End-to-End)

```
function login() {
  return Promise.resolve("User logged in");
}
function fetchProfile() {
  return Promise.reject("Profile service down");
}
function fetchOrders() {
  return Promise.resolve(["Order1", "Order2"]);
}
login()
  .then(fetchProfile)
  .then(fetchOrders)
  .then(orders => console.log(orders))
  .catch(err => console.error("Flow error:", err))
  .finally(() => console.log("Process completed"));
```

Common Mistakes

❌ Forgetting return inside .then()
❌ Multiple .catch() without purpose
❌ Swallowing errors silently
❌ Overusing .then() instead of async/await

Promise Chaining vs Callback Hell

Feature	Callback Hell	Promise Chaining
Structure	Nested	Flat
Readability	Poor	Excellent
Error handling	Messy	Centralized
Debugging	Hard	Easier

Quick Summary (Slide-Ready)

- Promise chaining = sequential async execution
- Each .then() returns a new promise
- Errors propagate automatically
- .catch() handles all failures
- .finally() runs always
- Cleaner alternative to callbacks

Async / Await

🔷 Async / Await Syntax (JavaScript)

✅ WHAT is Async / Await?

async/await is **syntactic sugar over Promises** that lets you write **asynchronous code that looks and behaves like synchronous code**.

- async is used before a function
- await pauses execution **until a Promise resolves or rejects**

```
async function getData() {
  const result = await somePromise();
  return result;
}
```

📌 Internally, it still uses **Promises**.

✅ WHY Async / Await is Used?

Problems with Promises alone

- Harder to read when chaining many .then() calls
- Error handling can become confusing
- Logic feels “inside-out”

Async/await solves this by:

✓ Making code **more readable**
✓ Writing async logic **top-to-bottom**
✓ Using **try/catch** like synchronous code
✓ Easier debugging and teaching

✅ WHEN to Use Async / Await?

Use async/await when:

✓ You work with **APIs (fetch, axios)**
✓ You have **multiple async steps**
✓ You want **clean error handling**
✓ You want readable and maintainable code

🚫 Avoid when:

- You need simple one-step Promise
- You don't want to block logical flow (parallel work needed)

✅ BASIC SYNTAX

```
async function fetchData() {
  const response = await fetch(url);
  const data = await response.json();
  return data;
}
```

Key Rules

Rule	Explanation
async function always returns a Promise	Even if you return a value
await works only inside async	Except top-level (modules)
await pauses execution	Until Promise resolves/rejects

✅ SIMPLE EXAMPLE (Comparison)

Using Promises

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

Using Async/Await

```
async function loadData() {
  try {
    const res = await fetch(url);
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
```

✓ Same logic
✓ Cleaner flow
✓ Easier error handling

✅ ERROR HANDLING with Async / Await

Using try...catch

```
async function getUsers() {
  try {
    const res = await fetch("/users");
    if (!res.ok) {
      throw new Error("Failed to fetch users");
    }
    const users = await res.json();
    console.log(users);
  } catch (error) {
    console.error("Error:", error.message);
  }
}
```

try/catch handles **both network and logic errors**

✅ MULTIPLE AWAITS (Sequential Execution)

```
async function processOrder() {
  const user = await getUser();
  const orders = await getOrders(user.id);
  const payment = await processPayment(orders);
}
```

📌 Each step waits for the previous one.

✅ PARALLEL EXECUTION with Promise.all

```
async function loadDashboard() {
  const [users, products] = await Promise.all([
    fetch("/users").then(r => r.json()),
    fetch("/products").then(r => r.json())
  ]);
  console.log(users, products);
}
```

✓ Faster

✓ Independent tasks

✅ REAL-WORLD FETCH EXAMPLE

```
async function loadUsers() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/users");
    const users = await response.json();
    console.log(users);
  } catch (error) {
    console.error("Failed to load users");
  }
}
```

COMMON MISTAKES

Using await outside async

Forgetting await

Not handling errors

```
const data = await fetch(url); // ❌ ERROR
const data = fetch(url); // ❌ Promise, not data
async function load() {
  const data = await fetch(url); // ❌ unhandled rejection
}
```

✅ ASYNC FUNCTION RETURNS PROMISE (Important Concept)

```
async function sum() {
  return 10;
}
sum().then(result => console.log(result)); // 10
```

“async wraps the return value in a Promise automatically.”

✅ ASYNC/AWAIT vs PROMISES

Feature	Promises	Async/Await
Readability	Medium	High
Error Handling	.catch()	try/catch
Debugging	Harder	Easier
Learning Curve	Medium	Easier

Fetch API for HTTP Requests (JavaScript)

✅ WHAT is the Fetch API?

The **Fetch API** is a modern JavaScript interface used to make **HTTP requests** (GET, POST, PUT, DELETE) from the browser or Node.js.

It replaces older techniques like **XMLHttpRequest (XHR)** and provides a **Promise-based** way to work with APIs.

```
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data));
```

📌 Fetch is **built into modern browsers**.

✅ WHY Fetch API is Used?

Problems with older approaches (XHR)

- Complex syntax
- Callback-based
- Hard to read and maintain

Fetch API advantages

✓ Uses **Promises**
✓ Cleaner syntax
✓ Works well with **async/await**
✓ Supports modern web standards
✓ Easy JSON handling

✅ WHEN to Use Fetch API?

Use Fetch when:

✓ Calling **REST APIs**
✓ Loading data dynamically
✓ Submitting forms without page reload

✓ CRUD operations (Create, Read, Update, Delete)
✓ Communicating with backend services

🚫 Avoid Fetch when:

- You need IE support (very old browsers)
- You require request cancellation (use AbortController carefully)

✅ BASIC FETCH SYNTAX

```
fetch("https://api.example.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Fetch returns:

✓ A **Promise**
✓ Resolves even for HTTP errors (404, 500)
❗ Only rejects for **network errors**

✅ HTTP METHODS with Fetch

🔷 GET (Read Data)

```
fetch("/users")
  .then(res => res.json())
  .then(users => console.log(users));
```

🔷 POST (Create Data)

```
fetch("/users", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({
    name: "John",
    email: "john@example.com"
  })
});
```

🔷 PUT / PATCH (Update Data)

```
fetch("/users/1", {
  method: "PUT",
  body: JSON.stringify({ name: "Updated Name" })
});
```

🔷 DELETE (Remove Data)

```
fetch("/users/1", {
  method: "DELETE"
});
```

✅ RESPONSE HANDLING (IMPORTANT)

Response Object

```
fetch(url).then(response => {
  console.log(response.status); // 200, 404, 500
  console.log(response.ok);     // true / false
});
```

📌 Fetch does **not** throw error on 404/500 by default.

✅ ERROR HANDLING (CORRECT WAY)

Promise Chaining

```
fetch(url)
  .then(response => {
    if (!response.ok) {
      throw new Error("HTTP Error " + response.status);
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error(error.message));
```

Async / Await

```
async function loadData() {
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error("Request failed");
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

✅ FETCH + UI EXAMPLE (Simple)

```
async function loadUsers() {
  const res = await fetch("/users");
  const users = await res.json();
  displayUsers(users);
}
```

✅ WORKING with JSON

```
fetch(url)
  .then(res => res.json())   // convert JSON → JS object
  .then(data => console.log(data));
```

📌 Common mistake: forgetting .json()

❌ COMMON FETCH MISTAKES (EXAM FAVORITE)

1️⃣ Assuming fetch rejects on 404

```
fetch("/wrong-url").then(res => {
  // still resolved
});
```

2️⃣ Forgetting return

```
.then(response => response.json()); // correct
```

3️⃣ Not setting headers for POST

```
headers: { "Content-Type": "application/json" }
```

✅ BEST PRACTICES

✓ Always check response.ok
✓ Use try/catch with async/await
✓ Separate fetch logic and UI logic
✓ Show loading indicators
✓ Handle errors gracefully
✓ Log errors for debugging

✅ FETCH vs AXIOS (Quick Comparison)

Feature	Fetch	Axios
Built-in	Yes	No
JSON parsing	Manual	Automatic
Error handling	Manual	Better defaults
Interceptors	❌	✓
Browser support	Modern	Excellent

Fetch API is a modern Promise-based API for making HTTP requests in JavaScript, commonly used to communicate with RESTful services.

Working with JSON Data (JavaScript)

✅ WHAT is JSON?

JSON (JavaScript Object Notation) is a **lightweight data format** used to **store and exchange data** between a client (browser) and a server.

- Text-based
- Language-independent
- Easy for humans to read
- Easy for machines to parse

Example JSON:

```
{
  "id": 1,
  "name": "Alice",
  "email": "alice@example.com"
}
```

📌 JSON looks like JavaScript objects, but it is **not JavaScript code**.

✅ WHY JSON is Used?

JSON is used because it is:

✓ Lightweight
✓ Faster to transmit over network
✓ Easy to parse and stringify
✓ Universally supported (JS, Java, Python, PHP, etc.)
✓ Standard format for REST APIs

🧠 Teaching sentence:

“JSON is the common language spoken between frontend and backend.”

✅ WHEN to Use JSON?

Use JSON when:

✓ Sending data from server to client
✓ Receiving API responses
✓ Storing configuration data
✓ Saving structured data (localStorage, files)
✓ Communicating between services

🚫 Do not use JSON when:

- You need to store functions or methods
- You need circular references

✅ JSON vs JavaScript Object (IMPORTANT)

Feature	JSON	JavaScript Object
Keys	Must be in double quotes	Quotes optional
Values	No functions / undefined	Can have functions
Comments	❌ Not allowed	✓ Allowed
Data type	String format	Runtime object

✅ CONVERTING JSON ↔ JAVASCRIPT

🔷 JSON → JavaScript Object (JSON.parse)

```
const jsonString = '{"name":"Bob","age":25}';
const obj = JSON.parse(jsonString);
console.log(obj.name); // Bob
```

📌 Converts **string → object**

🔷 JavaScript Object → JSON (JSON.stringify)

```
const user = { name: "Bob", age: 25 };
const json = JSON.stringify(user);

console.log(json);
// {"name":"Bob","age":25}
```

📌 Converts **object → string**

✅ WORKING WITH JSON FROM FETCH API

Example: Fetch + JSON

```
fetch("/users")
  .then(response => response.json())
  .then(users => {
    console.log(users);
  });
```

📌 response.json():

- Reads response body
- Parses JSON automatically
- Returns a Promise

✅ ACCESSING JSON DATA

Example JSON

```
{
  "id": 1,
  "name": "Alice",
  "address": {
    "city": "Chennai",
    "zip": "600001"
  }
}
```

Access in JS

```
console.log(user.name);
console.log(user.address.city);
```

✅ WORKING WITH JSON ARRAYS

JSON Array

```
[
  { "id": 1, "name": "Alice" },
  { "id": 2, "name": "Bob" }
]
```

Looping

```
users.forEach(user => {
  console.log(user.name);
});
```

✅ MODIFYING JSON DATA (Client Side)

```
users.push({ id: 3, name: "Charlie" });
users[0].name = "Updated Name";
```

📌 JSON data becomes **normal JS objects** after parsing.

✅ SENDING JSON TO SERVER (POST Request)

```
fetch("/users", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({
    name: "John",
    email: "john@example.com"
  })
});
```

📌 Always stringify before sending.

❌ COMMON MISTAKES (VERY IMPORTANT)

1️⃣ Forgetting to parse JSON

```
const data = response; // ❌
const data = await response.json(); // ✓
```

2️⃣ Treating JSON string as object

```
json.name; // ❌ undefined
JSON.parse(json).name; // ✓
```

3️⃣ Sending object without stringify

```
body: user; // ❌
body: JSON.stringify(user); // ✓
```

4️⃣ Using single quotes in JSON

```
{ 'name': 'Alice' } // ❌ Invalid JSON
```

✅ ERROR HANDLING WITH JSON

Invalid JSON Parse Error

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

✅ BEST PRACTICES

✓ Always validate JSON structure
✓ Use try/catch when parsing
✓ Keep JSON flat when possible
✓ Use meaningful property names
✓ Avoid deeply nested JSON
✓ Handle missing properties safely

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

✅ REAL-WORLD USE CASES

✓ API responses
✓ Configuration files
✓ localStorage / sessionStorage

✓ AJAX requests
✓ Microservices communication

🎯 QUESTIONS & ANSWERS

Q: What is JSON?

JSON is a lightweight, text-based data format used for data exchange between client and server.

Q: Difference between JSON and object?

JSON is a string format, while JavaScript objects exist at runtime.

Q: Why use JSON.stringify?

To convert JavaScript objects into JSON strings before sending to server.

🧠 KEY ONE-LINER

“JSON is the bridge between frontend and backend, and JavaScript works with it by parsing and stringifying.”

Error Handling in Async Code (JavaScript)

✅ WHAT is Error Handling in Async Code?

Error handling in async code is the process of **detecting, managing, and responding to errors** that occur during asynchronous operations such as:

- API calls (fetch)
- Promises
- async / await
- Timers (setTimeout)
- Network operations

Unlike synchronous code, async errors **do not propagate normally** and must be handled explicitly.

✅ WHY Error Handling is Important?

Without proper error handling:

❌ App crashes silently
❌ UI stays in loading state
❌ Users see blank screens
❌ Bugs become hard to debug

With proper error handling:

✓ App remains stable
✓ Errors are shown clearly
✓ Loaders are cleaned up
✓ Debugging is easier

🧠 Teaching line:

“Async code fails more often than sync code because it depends on networks, APIs, and external systems.”

✅ WHEN to Handle Errors in Async Code?

You must handle errors when:

✓ Calling APIs
✓ Parsing JSON
✓ Using Promises
✓ Using await
✓ Doing async business logic
✓ Updating UI based on async results

📌 **Every async operation must assume failure is possible.**

✅ TYPES OF ERRORS IN ASYNC CODE

1 Network Errors

- No internet
- Server down
- DNS failure

2 HTTP Errors

- 404 (Not Found)
- 500 (Server Error)

3 Parsing Errors

- Invalid JSON

4 Logical Errors

- Empty data
- Invalid format

✅ ERROR HANDLING WITH PROMISES

🔷 Using .catch()

```
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => {
    console.error("Error:", error);
  });
```

📌 .catch() handles:

- Rejected Promises
- Errors thrown inside .then()

🔷 Throwing Errors Manually

```
fetch(url)
  .then(response => {
    if (!response.ok) {
      throw new Error("HTTP Error " + response.status);
    }
    return response.json();
  })
  .catch(error => console.error(error.message));
```

🧠 Teaching point:

“Promises only fail when rejected or when an error is thrown.”

✅ ERROR HANDLING WITH ASYNC / AWAIT

🔷 Using try / catch

```
async function loadUsers() {
  try {
    const response = await fetch(url);

    if (!response.ok) {
      throw new Error("Failed to fetch users");
    }

    const users = await response.json();
    console.log(users);

  } catch (error) {
    console.error("Error:", error.message);
  }
}
```

📌 try/catch works **only with await**.

🔷 finally Block (Cleanup)

```
async function loadData() {
  try {
    await fetch(url);
  } catch (error) {
    console.error(error);
  } finally {
    hideLoader();
  }
}
```

✓ Runs always

✓ Perfect for loaders & cleanup

✅ HANDLING FETCH ERRORS (VERY IMPORTANT)

❗ Fetch does NOT reject on HTTP errors

```
fetch("/wrong-url")
  .then(response => {
    console.log(response.ok); // false
  });
```

✓ Fetch resolves

❌ You must manually check response.ok

✅ Correct Fetch Error Handling

```
async function fetchData() {
  const response = await fetch(url);

  if (!response.ok) {
    throw new Error("HTTP Error " + response.status);
  }

  return response.json();
}
```

✅ ERROR HANDLING IN MULTIPLE ASYNC STEPS

```
async function processOrder() {
  try {
    const user = await getUser();
    const orders = await getOrders(user.id);
    const payment = await processPayment(orders);
  } catch (error) {
    console.error("Process failed:", error.message);
  }
}
```

📌 Any error jumps directly to catch.

✅ HANDLING PARALLEL ASYNC ERRORS

🔷 Promise.all (Fails Fast)

```
try {
  const results = await Promise.all([
    fetchUsers(),
    fetchProducts()
  ]);
} catch (error) {
  console.error("One request failed");
}
```

📌 If one fails → all fail.

🔷 Promise.allSettled (Advanced)

```
const results = await Promise.allSettled([
  fetchUsers(),
  fetchProducts()
]);

results.forEach(result => {
  if (result.status === "rejected") {
    console.error(result.reason);
  }
});
```

✓ Handles partial success

✓ Very interview-relevant

❌ COMMON MISTAKES (EXAM FAVORITE)

1️⃣ Forgetting try/catch

await fetch(url); // ❌ unhandled rejection

2️⃣ Assuming fetch throws on 404

await fetch("/404"); // ❌ no error thrown

3️⃣ Catching too late

.then(step1)

.then(step2)

.catch(...) // ❌ hard to trace

4️⃣ Swallowing errors

catch (e) { } // ❌ bad practice

✅ BEST PRACTICES

✓ Always check response.ok
✓ Use try/catch/finally
✓ Show user-friendly error messages
✓ Log errors for debugging
✓ Separate API logic and UI logic
✓ Never leave loaders running

🎯 QUESTIONS & ANSWERS

Q: How do you handle errors in async/await?

Using try/catch blocks around awaited code.

Q: Does fetch reject on 404?

No, fetch resolves; we must manually throw errors.

Q: Difference between Promise.all and allSettled?

Promise.all fails fast, allSettled reports all results.

🧠 KEY TEACHING ONE-LINER

“In async JavaScript, errors don’t disappear — they just move. Good error handling brings them back under control.”