JS

Getting Started with JavaScript (JS Fundamentals)

Definition:
JavaScript is a **lightweight, interpreted, scripting language** used to make web pages **interactive and dynamic.**

Why JavaScript is Needed

- HTML → structure
- CSS → styling
- **JavaScript → behavior & logic**
  Used for **form validation, dynamic content, API calls, events**, etc.

Key Fundamentals of JavaScript

a) Variables

Used to store data.

```
let name = "Gaffoor";
const age = 21;
var city = "Hyderabad";
```

- let → changeable
- const → fixed value
- var → old (avoid in modern JS)

---

b) Data Types

- **Primitive:** String, Number, Boolean, Undefined, Null
- **Non-Primitive:** Object, Array, Function

```
let isActive = true;
let marks = 85;
let skills = ["Java", "JS"];
```

---

c) Operators

- Arithmetic: + - * / %
- Comparison: == === != > <
- Logical: && || !

```
if (marks > 50 && isActive) {
  console.log("Passed");
}
```

---

d) Conditional Statements

Used to make decisions.

```
if (age >= 18) {
  console.log("Eligible");
} else {
  console.log("Not Eligible");
}
```

---

e) Loops

Used to repeat tasks.

```
for (let i = 1; i <= 3; i++) {
  console.log(i);
}
```

---

f) Functions
Reusable block of code.

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

Technical / IT Example
Form validation:

```
if (password.length < 6) {
  alert("Password too short");
}
```

Interview-Ready Points

- JavaScript is **single-threaded** and **event-driven**
- Runs in the **browser** and also on **server (Node.js)**
- === checks **value + type**, == checks only value

JavaScript Variables (var, let, const)

Definition:
Variables are containers used to **store data values** so they can be reused and updated in a program.

Why Variables are Needed

- To store user input
- To perform calculations
- To control application logic
- To avoid hard-coding values

Types of Variables in JavaScript

---

1. var (Old way)
Key Points

- Function-scoped
- Can be **redeclared** and **updated**
- Hoisted (initialized as undefined)
- **Not recommended** in modern JS

```
var x = 10;
var x = 20; // allowed
```

Problem with var

```
if (true) {
  var a = 5;
}
console.log(a); // 5 (unexpected)
```

---

2. let (Modern & Preferred)
Key Points

- Block-scoped
- Can be **updated** but **not redeclared**
- Safer than var
- Hoisted but not initialized (TDZ)

```
let count = 1;
```

```
count = 2; // allowed
let count = 1;
let count = 2; // error
```

---

3. const (Fixed Value)
Key Points

- Block-scoped
- **Cannot be updated or redeclared**
- Must be initialized
- Used for constants

```
const PI = 3.14;
PI = 3.15; // error
For objects & arrays:
const user = { name: "Ali" };
user.name = "Ahmed"; // allowed
```

Real-Life Example

- var → open water tap (uncontrolled)
- let → tap with control
- const → sealed water bottle

Technical / IT Example

```
const API_URL = "https://api.example.com";
let loginCount = 0;

function login() {
  loginCount++;
}
```

Interview Comparison Table

Feature	var	let	const
Scope	Function	Block	Block
Redeclare	Yes	No	No
Update	Yes	Yes	No
Hoisting	Yes	Yes (TDZ)	Yes (TDZ)
Use Today	No	No	No

Interview-Ready Lines

- Use const by default, let when value changes
- Avoid var due to scope issues
- const protects reference, not object content

JavaScript Data Types & Operators

PART 1: JavaScript Data Types

Definition:
Data types define the **type of value** a variable can store in JavaScript.

Why Data Types are Needed

- To perform correct operations
- To manage memory efficiently
- To avoid runtime errors
- Helps JS engine understand data behavior

Types of Data Types in JavaScript

A) Primitive Data Types

Stored by **value** (immutable)

1. String

```
let name = "Gaffoor";
```

2. Number

```
let age = 21;
```

3. Boolean

```
let isLoggedIn = true;
```

4. Undefined

```
let x;
```

5. Null

```
let y = null;
```

6. Symbol (ES6)

```
let id = Symbol("id");
```

B) Non-Primitive (Reference) Data Types

Stored by **reference**

1. Object

```
let user = { name: "Ali", age: 25 };
```

2. Array

```
let skills = ["JS", "React"];
```

3. Function

```
function greet() {}
```

Real-Life Example

- Primitive → written note (copy changes nothing)
- Object → shared Google Doc (changes visible everywhere)

Technical Example

```
let a = 10;
let b = a;
b = 20; // a remains 10
let obj1 = { x: 1 };
let obj2 = obj1;
obj2.x = 2; // obj1.x also becomes 2
```

Interview-Ready Points (Data Types)

- JavaScript is **dynamically typed**
- typeof null returns "object" (known bug)
- Primitive → value copy, Non-primitive → reference copy

PART 2: JavaScript Operators

Definition:

Operators are symbols used to **perform operations** on values and variables.

Types of Operators

1. Arithmetic Operators

```
+  -  *  /  %
let sum = 10 + 5;
```

---

2. Assignment Operators

```
=  +=  -=  *=  /=
x += 5; // x = x + 5
```

---

3. Comparison Operators

```
==   ===   !=   !==   >   <
5 == "5"   // true
5 === "5"  // false
```

Important: Always use ===

---

4. Logical Operators

```
&&   ||   !
if (age > 18 && isLoggedIn) {}
```

---

5. Ternary Operator

```
let result = age > 18 ? "Adult" : "Minor";
```

---

6. Type Operators

```
typeof
instanceof
typeof 10; // "number"
```

Real-Life Example

- && → both switch ON → light ON
- || → any switch ON → light ON

Technical / IT Example

```
if (password.length >= 6 && isActive) {
  login();
}
```

Interview-Ready Points (Operators)

- === checks **value + type**
- && stops at first false (short-circuit)
- Ternary improves readability for simple conditions

Control Flow in JavaScript (if-else, switch, loops)

Control Flow

Definition:

Control flow determines the **order in which statements are executed** in a program.

Why Control Flow is Needed

- To make decisions
- To repeat tasks
- To control program execution logically
- Core concept in **real projects & interviews**

Interview Line
Control flow allows a program to execute different blocks of code based on conditions or repetitions.

---

if-else Statement
Definition:
Used to execute code **based on a condition.**

Syntax

```
if (condition) {
  // true block
} else {
  // false block
}
```

Key Points

- Condition must return true or false
- Can use else if for multiple conditions
- Most commonly used decision structure

Example

```
let marks = 75;

if (marks >= 50) {
  console.log("Pass");
} else {
  console.log("Fail");
}
```

Real-Life Example

- If traffic light is **green** → go
- Else → stop

Interview Tip

- Use if-else when **range-based or complex conditions** exist.

---

switch Statement
Definition:
Used to execute code based on **fixed values (choices).**

Syntax

```
switch (day) {
  case 1:
    console.log("Monday");
    break;
  default:
    console.log("Invalid");
}
```

Key Points

- Uses === internally

- break prevents **fall-through**
- Cleaner than multiple else if

Example

```
let role = "admin";

switch (role) {
  case "admin":
    console.log("Full Access");
    break;
  case "user":
    console.log("Limited Access");
    break;
  default:
    console.log("No Access");
}
```

Real-Life Example

- Menu selection in ATM
- Remote control buttons

Interview Tip

- Use switch when conditions depend on **exact values.**

Loops

Definition:

Loops are used to **repeat a block of code** multiple times.

Types of Loops in JavaScript

---

1. for Loop

```
for (let i = 1; i <= 3; i++) {
  console.log(i);
}
```

Known number of iterations

---

2. while Loop

```
let i = 1;
while (i <= 3) {
  console.log(i);
  i++;
}
```

Condition-based repetition

---

3. do-while Loop

```
let i = 1;
do {
  console.log(i);
  i++;
} while (i <= 3);
```

Executes **at least once**

---

4. for...of (Arrays)

```
for (let skill of skills) {
```

```
  console.log(skill);
}
```

---

5. for...in (Objects)

```
for (let key in user) {
  console.log(key, user[key]);
}
```

Real-Life Example (Loops)

- Attendance calling
- Printing bills
- Repeating alarms

Technical Example

```
let users = ["Ali", "Ahmed", "Sara"];

for (let user of users) {
  console.log(user);
}
```

Interview Comparison

Use Case	Best Choice
Fixed count	for
Condition-based	while
At least once	do-while
Array	for...of
Object	for...in

Interview-Ready Lines

- break stops loop execution
- continue skips current iteration
- Infinite loops cause performance issues

JavaScript Objects (Creation, Properties, Methods)

Object
Definition:
An **object** is a collection of **key–value pairs** used to store **related data and behavior** together.

Why Objects are Needed

- To represent real-world entities
- To group related data
- Foundation of **OOP in JavaScript**
- Widely used in **APIs, JSON, React, backend**

Interview Line
Objects allow us to store data and functions together in a structured way.

Object Creation

1. Object Literal (Most Common)

```
let user = {
  name: "Gaffoor",
  age: 21
};
```

Simple and preferred

---

2. Using new Object()

```
let user = new Object();
user.name = "Ali";
```

Rarely used

---

3. Constructor Function

```
function User(name, age) {
  this.name = name;
  this.age = age;
}

let u1 = new User("Ahmed", 25);
```

---

4. Using ES6 class

```
class User {
  constructor(name) {
    this.name = name;
  }
}
```

Interview Tip

- Object literal is **best for simple objects**
- class is preferred for **multiple objects**

Object Properties

Definition:

Properties are variables stored inside an object.

Accessing Properties

```
user.name;        // dot notation
user["age"];     // bracket notation
```

Adding / Updating Properties

```
user.city = "Hyderabad";
user.age = 22;
```

Deleting Properties

```
delete user.city;
```

Real-Life Example

- Person → name, age, address
- Car → model, price, color

Object Methods

Definition:
Methods are functions stored inside an object.

Example

```
let user = {
  name: "Gaffoor",
  greet: function () {
    return "Hello " + this.name;
  }
};
```

---

this Keyword

- Refers to **current object**

```
this.name
```

Shorthand Method (ES6)

```
let user = {
  greet() {
    console.log("Hello");
  }
};
```

Real-Life Example

- Mobile phone
  - Properties → brand, price
  - Methods → call(), message()

Technical / IT Example

```
let product = {
  id: 101,
  price: 500,
  getDiscount() {
    return this.price * 0.1;
  }
};
```

Interview-Ready Points

- Objects store data in **key-value pairs**
- Methods are functions inside objects
- this refers to object calling the method
- Objects are **reference types**

Common Interview Questions

Q) Dot vs Bracket notation?

A) Bracket allows dynamic keys

Q) Can object have functions?

A) Yes, called methods

THIS KEYWORD IN JS

Definition:
this refers to the **object that is currently calling a function.**

Why this is Needed

- To access object properties inside methods
- To reuse functions for multiple objects
- Core concept in **OOP, events, classes, React**

Behavior of this (Important Cases)
1. this inside an Object Method
Refers to the **object itself**

```
let user = {
  name: "Gaffoor",
  greet() {
    console.log(this.name);
  }
};
user.greet(); // Gaffoor
```

Most common use

---

2. this in Global Scope

```
console.log(this);
```

- Browser → window
- Strict mode → undefined

---

3. this inside a Normal Function

```
function test() {
  console.log(this);
}
test();
```

- Non-strict → window
- Strict → undefined

Common interview trap

---

4. this inside Arrow Function
Does NOT have its own this
Takes this from parent scope

```
let user = {
  name: "Ali",
  greet: () => {
    console.log(this.name);
  }
};
undefined
```

Arrow functions are not for object methods

---

5. this in Event Handling

```
button.onclick = function () {
  console.log(this);
};
```

Refers to the **HTML element**

---

6. this in Classes

```
class User {
  constructor(name) {
    this.name = name;
  }
}
```
Refers to current object instance

Real-Life Example

- “I” changes based on **who is speaking**
- Same with this

Technical / IT Example

```
let product = {
  price: 1000,
  getPrice() {
    return this.price;
  }
};
```

Interview Comparison

Scenario	this refers to
Object method	Object
Arrow function	Parent scope
Event handler	HTML element
Class	Instance
Strict function	undefined

Interview-Ready Lines

- this depends on **how a function is called**
- Arrow functions don’t bind this
- Avoid arrow functions for object methods

Common Interview Question

Q) Why arrow functions shouldn’t be used as object methods?

A) Because they don’t have their own this

JavaScript Arrays & Array Methods

Definition:

Array is a special variable that **stores multiple values** in a **single ordered list.**

```
let fruits = ["Apple", "Banana", "Mango"];
```

Why Arrays are Needed

- To store **related data** together
- To **iterate or manipulate** multiple values easily
- Useful in **lists, tables, API data**

Key Points

- Indexed from **0**
- Can store **any data type**

- Dynamic in size
- **Reference type** → copied by reference

Array Methods (Important for Freshers & Interviews)

Method	Description	Example
push()	Add element at end	arr.push("Orange")
pop()	Remove last element	arr.pop()
shift()	Remove first element	arr.shift()
unshift()	Add element at start	arr.unshift("Grapes")
splice()	Add/remove element at index	arr.splice(1, 1, "Kiwi")
slice()	Copy part of array	arr.slice(0,2)
indexOf()	Find element index	arr.indexOf("Banana")
includes()	Check existence	arr.includes("Mango")
forEach()	Loop over array	arr.forEach(item => console.log(item))
map()	Return new array	arr.map(x => x.toUpperCase())
filter()	Filter elements	arr.filter(x => x.includes("a"))
reduce()	Reduce to single value	arr.reduce((a,b)=>a+b)
sort()	Sort elements	arr.sort()
reverse()	Reverse order	arr.reverse()

Real-Life Example

- Grocery list → ["Milk", "Eggs", "Rice"]
- Add, remove, filter items easily

Technical / IT Example

```
let users = ["Ali", "Ahmed", "Sara"];
users.push("Zara");
users.forEach(u => console.log(u));
let upperUsers = users.map(u => u.toUpperCase());
```

Interview-Ready Points

- Arrays are **zero-indexed**
- push & pop → **fast at end**, shift & unshift → **slower at start**
- Use map, filter, reduce → **functional programming**
- Arrays are **objects in JS**

JavaScript Array Iteration Methods: forEach, map, filter, reduce

Definition:

These are **array methods** used to **process elements** without writing traditional loops.

- forEach → iterate
- map → transform
- filter → select
- reduce → aggregate

Why They Are Needed

- Cleaner, readable code
- Functional programming style
- Avoid manual for loops

- Essential for **React, Node.js, APIs**

Key Points

Method	Returns	Mutates Original?	Use Case
forEach	undefined	No	Execute function for each element
map	New array	No	Transform elements
filter	New array	No	Select elements based on condition
reduce	Single value	No	Aggregate / combine elements

Syntax & Examples

1. forEach()

```
let nums = [1, 2, 3];
nums.forEach(num => console.log(num * 2));
// Output: 2, 4, 6
```

---

2. map()

```
let nums = [1, 2, 3];
let squares = nums.map(num => num * num);
console.log(squares); // [1, 4, 9]
```

---

3. filter()

```
let nums = [1, 2, 3, 4, 5];
let even = nums.filter(num => num % 2 === 0);
console.log(even); // [2, 4]
```

---

4. reduce()

```
let nums = [1, 2, 3, 4];
let sum = nums.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // 10
```

- acc → accumulator (cumulative value)
- curr → current element
- 0 → initial value

Real-Life Example

- map → convert all prices to USD
- filter → show only available products
- reduce → calculate total bill
- forEach → print all product names

Technical / IT Example

```
let users = [
  { name: "Ali", age: 21 },
  { name: "Sara", age: 25 },
  { name: "Ahmed", age: 18 }
];

users.filter(u => u.age >= 21)
     .map(u => u.name)
     .forEach(name => console.log(name));
// Output: Ali, Sara

let totalAge = users.reduce((sum, u) => sum + u.age, 0);
console.log(totalAge); // 64
```

Interview-Ready Lines

- map → **transform**, filter → **select**, reduce → **aggregate**
- Prefer functional array methods over loops for **readability & performance**
- forEach **does not return a new array**

JavaScript Functions & Arrow Functions

Function Declaration
Definition:
A **function** is a **block of reusable code** that performs a specific task.

Why Functions Are Needed

- Avoid code repetition
- Modular & maintainable code
- Can accept inputs and return outputs

Syntax

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

Key Points

- **Hoisted** → can call before declaration
- Can return a value with return
- Parameters are optional

Real-Life Example

- Recipe → “Make tea” function
  - Inputs → water, tea leaves
  - Output → tea

Technical / IT Example

```
function greetUser(name) {
  return `Hello, ${name}`;
}
console.log(greetUser("Ali")); // Hello, Ali
```

Interview-Ready Points

- Function declarations are **hoisted**
- Good for **reusable and readable code**

Arrow Functions (ES6)

Definition:
Arrow functions are a **shorter syntax** for writing functions, introduced in ES6.

Syntax

```
const add = (a, b) => a + b;
console.log(add(2, 3)); // 5
```

Key Points

- **No this binding** → inherits from parent scope
- Can omit braces {} for single return
- Cannot be used as constructors

Comparison with Function Declaration

Feature	Function Declaration	Arrow Function
Syntax	function add(a,b){}	const add=(a,b)=>a+b
this	Own this	Inherits this from parent
Hoisting	Yes	No
Constructor	Yes	No

Real-Life Example

- Function → "Switch ON fan"
- Arrow → "Quick action like doubling numbers"

Technical / IT Example

```
// Callback example
const nums = [1,2,3];
const squares = nums.map(n => n*n);
console.log(squares); // [1,4,9]
```

Interview-Ready Lines

- Arrow functions are **shorter, concise, and lexical this**
- Use arrow functions in **callbacks, map, filter, reduce**

JavaScript Advanced Functions: Callbacks, Closures & IIFE

Callbacks
Definition:
A **callback** is a **function passed as an argument** to another function, executed later.

Why Callbacks Are Needed

- To handle **asynchronous tasks**
- Execute code after an **event or operation**
- Core in **APIs, setTimeout, event handling**

Syntax & Example

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

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

greet("Ali", sayBye);
// Output: Hello Ali
//         Goodbye!
```

Real-Life Example

- Ordering food → callback = deliver food after cooking
- Event listener → execute function on click

Interview Tip

- Callbacks are **foundation for async programming**
- Avoid "callback hell" → use **Promises / async-await**

Closures

Definition:

A **closure** is a function that **remembers its outer scope**, even after the outer function has executed.

Why Closures Are Needed

- Encapsulation & private variables
- Maintain state between function calls
- Common in **modules and event handling**

Syntax & Example

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    return count;
  };
}

const counter = outer();
console.log(counter()); // 1
console.log(counter()); // 2
```

Real-Life Example

- Bank account → private balance, can increase/decrease via functions
- Keeps internal data safe

Interview Tip

- Closures allow **data privacy** in JS
- Often asked in **scope & memory questions**

IIFE (Immediately Invoked Function Expression)

Definition:

An **IIFE** is a function that **runs immediately after it is defined.**

Why IIFE Is Needed

- Create **private scope**
- Avoid polluting global namespace
- Useful in **modules, scripts, and closures**

Syntax & Example

```
(function() {
  let message = "Hello!";
  console.log(message);
})();
// Output: Hello!
```

ES6 Version with Arrow Function:
```
(() => console.log("Hi!"))();
```

Real-Life Example

- Quick setup code for a webpage without affecting global scope
- Initializing config values in JS library

Interview-Ready Lines

- Callbacks → pass function as argument
- Closures → function + remembered scope
- IIFE → runs immediately, creates private scope

JavaScript DOM: Selecting Elements & Modifying Content

Selecting Elements
Definition:
DOM (Document Object Model) allows JS to **access and manipulate HTML elements.**

Why Selecting Elements is Needed

- To **read or update content**
- Handle **events** like click, input
- Build **dynamic web pages**

Key Methods

Method	Description	Example
getElementById	Select element by id	document.getElementById("title")
getElementsByClassName	Select elements by class	document.getElementsByClassName("item")
getElementsByTagName	Select elements by tag	document.getElementsByTagName("p")
querySelector	First matching CSS selector	document.querySelector(".item")
querySelectorAll	All matching CSS selectors	document.querySelectorAll(".item")

Real-Life Example

- Selecting input box → validate user input
- Selecting button → perform action on click

Modifying Content

Definition:
After selecting, JS can **change content, style, attributes** of elements.

Properties to Modify Content

Property	Description	Example
innerHTML	Change HTML content	el.innerHTML = "Hello"
textContent	Change text only	el.textContent = "Hi"
value	For inputs	input.value = "New value"

Property	Description	Example
style	Change CSS	el.style.color = "red"
setAttribute	Change attributes	el.setAttribute("src","img.jpg")

Example

```
// Selecting element
let heading = document.getElementById("title");

// Changing text
heading.textContent = "Welcome to JS";

// Changing style
heading.style.color = "blue";
heading.style.fontSize = "24px";
```

Real-Life Example

- Change greeting based on time
- Highlight a selected menu item

Interview-Ready Lines

- Use querySelector / querySelectorAll → modern & flexible
- innerHTML → can insert HTML; textContent → plain text
- style allows direct CSS modification

JavaScript Event Handling

Definition:
Event handling is the process of **responding to user actions** (clicks, typing, hover) in a web page.

Why Event Handling is Needed

- To make web pages **interactive**
- Respond to **user input**
- Core concept in **dynamic websites, forms, and apps**

Key Events

Event Type	Description	Example
click	When element is clicked	Button click
mouseover	Mouse pointer enters element	Hover effect
mouseout	Mouse leaves element	Hover out effect
keydown	Key pressed	Input field
keyup	Key released	Input validation
submit	Form submission	Validate before submit

Ways to Handle Events

1. Inline Event Handler (HTML)

```
<button onclick="sayHi()">Click Me</button>
<script>
function sayHi() {
```

```
  alert("Hello!");
}
</script>
```

---

2. DOM Property Event Handler

```
let btn = document.getElementById("btn");
btn.onclick = function() {
  alert("Button Clicked");
};
```

---

3. addEventListener (Recommended)

```
let btn = document.getElementById("btn");
btn.addEventListener("click", () => {
  console.log("Clicked!");
});
```

- Allows multiple events on same element
- Modern and flexible

Event Object

```
document.getElementById("input").addEventListener("keydown", (e) => {
  console.log(e.key); // Shows which key was pressed
});
```

- e → event object
- Contains info like type, target, key, clientX/Y

Real-Life Example

- Click → open menu
- Input → validate username
- Hover → change button color

Technical / IT Example

```
let btn = document.querySelector("#submit");
btn.addEventListener("click", function(e) {
  e.preventDefault(); // Prevent form submission
  console.log("Form submitted!");
});
```

Interview-Ready Lines

- addEventListener is preferred over inline handlers
- Event object gives **details about the event**
- Supports multiple events for the same element

JavaScript Asynchronous Programming: Promises, Async/Await & Fetch API

Promises

Definition:

A **Promise** is an object representing the **eventual completion or failure** of an asynchronous operation.

Why Promises Are Needed

- Handle **asynchronous tasks** like API calls or timers
- Avoid **callback hell**

- Clean and readable code

Syntax

```
let promise = new Promise((resolve, reject) => {
  let success = true;
  if(success) resolve("Done!");
  else reject("Error!");
});

promise
  .then(result => console.log(result))
  .catch(error => console.log(error));
```

Key Points

- **Pending** → initial state
- **Resolved / Fulfilled** → success
- **Rejected** → failure

Real-Life Example

- Ordering online → success → item delivered
- Failure → item out of stock

Async / Await

Definition:
async function returns a **Promise**.
await pauses execution until the promise resolves.

Syntax & Example

```
async function fetchData() {
  try {
    let response = await fetch("https://api.example.com/data");
    let data = await response.json();
    console.log(data);
  } catch(err) {
    console.log("Error:", err);
  }
}
fetchData();
```

Key Points

- Makes **async code look synchronous**
- Always use try-catch for errors
- Works only inside async function

Real-Life Example

- Wait for food delivery → then eat
- Wait for API response → then update page

Fetch API

Definition:
Fetch API is used to **make HTTP requests** from JS.

Syntax

```
fetch(url, options)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(err => console.log(err));
```

- url → API endpoint
- options → method, headers, body

Example

```
fetch("https://jsonplaceholder.typicode.com/posts")
  .then(res => res.json())
  .then(posts => console.log(posts[0]))
  .catch(err => console.log("Error:", err));
```

Real-Life Example

- Get weather data from API
- Load user posts in social media app

Interview-Ready Lines

- Promise → handle async results
- async/await → cleaner syntax
- fetch() → modern way to call APIs
- Always handle errors with .catch() or try-catch