javascript

Here is a **complete JavaScript interview question bank**—covering fundamentals → ES6+ → Functions → Async/Await → DOM → OOP in JS → Promises → Event Loop → Hoisting → Closures → Scope → Arrays → API Calls → Real-time scenarios.

These are the questions you absolutely need to prepare.

🔥 JAVASCRIPT — FULL INTERVIEW QUESTION LIST

📌 1. JavaScript Basics

✅ 1) What is JavaScript and why is it called a scripting language?

■ Explanation (simple, conceptual)

JavaScript is a lightweight, interpreted programming language used mainly for adding interactivity to web pages. It runs directly inside the browser without needing compilation, which is why it's called a *scripting* language.

■ When/Why is it used?

Used for DOM manipulation, event handling, user interactions, API calls, animations, and building dynamic single-page applications like Angular/React.

■ Example / Code snippet

```
console.log("Hello JavaScript!");
```

■ One-line summary:

> JavaScript is a browser-based interpreted language called a script because it runs without compilation.

✅ 2) Difference between JavaScript and TypeScript?

■ Explanation (simple, conceptual)

JavaScript is a dynamic, interpreted language; TypeScript is a superset of JavaScript that adds static typing, interfaces, generics, and compile-time error checking.

■ When/Why is it used?

JS is used for quick scripting; TypeScript is used for scalable applications (Angular) because it catches errors early and makes the code more maintainable.

■ Example / Code snippet

```
// TypeScript
let age: number = 25;

// JavaScript
let age = 25;
```

■ One-line summary:

> JavaScript is dynamic, while TypeScript adds type safety and compiles to JavaScript.

✅ 3) What are primitive data types in JavaScript?

■ Explanation (simple, conceptual)

Primitive types are basic data types that store single values and are immutable.

■ When/Why is it used?

Used to represent simple values like numbers, strings, or booleans that don't require objects or complex structures.

■ Example / List

The 7 primitive types are:

1. `string`
2. `number`
3. `boolean`
4. `null`
5. `undefined`
6. `symbol`
7. `bigint`

■ Example Code

```
let name = "John";       // string
let age = 25;            // number
let isActive = true;     // boolean
```

■ One-line summary:

> JavaScript has 7 primitive types like string, number, boolean, null, undefined, symbol, and bigint.

✅ 4) Difference between `==` and `===` ?

■ Explanation (simple, conceptual)

`==` compares values after performing *type coercion*, while `===` compares values *without* type conversion and is therefore more strict.

■ When/Why is it used?

Use `===` in modern JS to avoid unexpected results caused by automatic type conversion.

■ Example / Code snippet

```
5 == "5"   // true  → type coercion
5 === "5"  // false → strict comparison
```

■ One-line summary:

> == allows type conversion; === checks both value and type strictly.

✅ 5) What are truthy and falsy values?

■ Explanation (simple, conceptual)

Truthy and falsy values are non-boolean values that JS implicitly converts to `true` or `false` when used in conditions.

■ When/Why is it used?

Used in `if` conditions, loops, and logical operations to simplify checks without writing explicit comparisons.

■ Example / Code snippet

Falsy values (only 6):

- `0`
- `""` (empty string)
- `null`
- `undefined`
- `NaN`
- `false`

Everything else is **truthy**, e.g., `"hello"`, `[]`, `{}`, `1`, etc.

```
if ("") console.log("truthy");  // won't run
if ("hello") console.log("truthy"); // runs
```

■ One-line summary:

> Falsy values automatically become false in conditions; everything else is truthy.

✅ 6) What is `var`, `let`, and `const`? Differences?

■ Explanation (simple, conceptual)

`var`, `let`, and `const` are keywords used to declare variables in JavaScript, but they differ in scope, re-assignment, and behavior.

■ When/Why is it used?

- Use `var` only for legacy code.
- Use `let` for variables that change.
- Use `const` for fixed values or objects that shouldn't be reassigned.

■ Example / Code snippet

```
var a = 10;   // function-scoped, can be redeclared
let b = 20;   // block-scoped, cannot be redeclared
const c = 30; // block-scoped, cannot be reassigned
```

Key Differences:

Feature	var	let	const
Scope	Function	Block	Block
Redeclare	Yes	No	No
Reassign	Yes	Yes	No
Hoisting	Yes	Yes	Yes

■ One-line summary:

> var is function-scoped, let is block-scoped, and const is block-scoped + non-reassignable.

✅ 7) Why is `var` function-scoped and not block-scoped?

■ Explanation (simple, conceptual)

`var` was designed in earlier versions of JavaScript where only functions created a new scope. Blocks (`if`, `for`) didn't create scopes, so variables declared with `var` were hoisted to the nearest function scope, not the nearest block.

■ When/Why is it used?

This behavior is mostly legacy. Modern JavaScript prefers `let` and `const` to avoid accidental variable leaking outside blocks.

■ **Example / Code snippet**

```
if (true) {
  var x = 10;
}
console.log(x); // 10 (leaks outside block)
```

But with `let`:

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

■ **One-line summary:**

> var is function-scoped because JavaScript originally treated only functions as scopes, not blocks.

✅ 8) What is Hoisting in JavaScript?

■ **Explanation (simple, conceptual)**

Hoisting means JavaScript moves variable and function declarations to the top of their scope before executing code.

But **only declarations are hoisted, not initializations**.

■ **When/Why is it used?**

It explains why we can call functions before they are declared and why `var` behaves differently from `let` and `const`.

■ **Example / Code snippet**

Example with `var`:

```
console.log(a); // undefined
var a = 10;
```

Example with `let` / `const`:

```
console.log(b); // ReferenceError (Temporal Dead Zone)
let b = 20;
```

Function hoisting:

```
greet();        // works
function greet(){ console.log("Hello"); }
```

■ **One-line summary:**

> Hoisting lifts declarations to the top, but let/const stay in a Temporal Dead Zone until initialized.

✅ 9) What are template literals?

■ **Explanation (simple, conceptual)**

Template literals are a cleaner way to create strings using backticks (`` ` ``). They support multi-line strings and embedded expressions using `${}`.

■ **When/Why is it used?**

Used to create dynamic strings, build HTML templates, and avoid messy string concatenations.

■ **Example / Code snippet**

```
let name = "Mohini";
let msg = `Hello ${name}, welcome to JavaScript!`;

let multiLine = `
This is line 1
This is line 2
`;
```

■ **One-line summary:**

> Template literals allow cleaner string formatting using backticks and ${} expressions.

✅ 10) What is strict mode `"use strict"` ?

■ Explanation (simple, conceptual)

`"use strict"` enables strict mode, which makes JavaScript more secure by preventing silent errors. It disallows bad practices like using undeclared variables, duplicate parameters, and automatically binding `this` to the global object.

■ When/Why is it used?

Used to catch bugs early, write cleaner code, and avoid accidental global variable creation.

■ Example / Code snippet

```
"use strict";

x = 10; // ❌ Error: x is not defined
```

Another example:

```
function test(a, a){ } // ❌ Error in strict mode
```

■ One-line summary:

> Strict mode prevents unsafe JavaScript behavior by disallowing silent errors and forcing secure practices.

📌 2. Functions & Scope

✅ 1) Difference between normal function vs arrow function

■ Explanation (simple, conceptual)

A normal function has its own `this` context, supports `arguments` object, can be used as a constructor, and has function-level binding.

An arrow function does **not** have its own `this`, does **not** have `arguments`, and cannot be used as a constructor.

■ When/Why is it used?

Arrow functions are used for short callbacks, array methods, and when we want to use the surrounding `this` (lexical `this`).

Normal functions are used when we need dynamic `this`, constructors, or advanced features.

■ Example / Code snippet

```
// Normal function
function show() {
  console.log(this);
}

// Arrow function
const show2 = () => {
  console.log(this); // takes parent's 'this'
};
```

■ One-line summary:

> Normal functions have their own this, arrow functions inherit this from the parent (lexical this).

✅ 2) What is a callback function?

■ Explanation (simple, conceptual)

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

■ When/Why is it used?

Used for asynchronous operations like API calls, timers, event listeners, or defining custom logic to run after a task completes.

■ Example / Code snippet

```
function greet(name, callback){
  callback(name);
}

greet("Mohini", (n) => console.log("Hello " + n));
```

■ **One-line summary:**

> A callback is a function passed into another function to run after some operation.

✅ 3) What is closure? Real-life example.

■ **Explanation (simple, conceptual)**

A closure is a function that remembers variables from its outer lexical scope even after the outer function has finished executing.

■ **When/Why is it used?**

Used in private variables, event handlers, modules, data hiding, and maintaining state.

■ **Example / Code snippet**

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

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

Real-life example:

A shopping cart count that increases even after the function is done.

■ **One-line summary:**

> Closure allows a function to access outer scope variables even after the outer function is done.

✅ 4) What is lexical scope?

■ **Explanation (simple, conceptual)**

Lexical scope means a function can access variables from its parent function based on where it is written in the code (not how it's called).

■ **When/Why is it used?**

Important for closures, nested functions, and understanding how variables behave in inner functions.

■ **Example / Code snippet**

```
function outer(){
  let x = 10;
  function inner(){
    console.log(x); // accesses outer scope
  }
  inner();
}
outer();
```

■ **One-line summary:**

> Lexical scope means inner functions can access variables from their outer function.

✅ 5) What is IIFE? Why used?

■ **Explanation (simple, conceptual)**

IIFE stands for **Immediately Invoked Function Expression** — a function that runs immediately after it is created.

■ **When/Why is it used?**

Used to avoid global variable pollution, create private scope, and run setup code only once.

■ **Example / Code snippet**

```
(function(){
  console.log("I run immediately!");
})();
```

■ **One-line summary:**

> IIFE is a self-executing function used to create private scope and avoid global variables.

✅ 6) Pure function vs Impure function

■ **Explanation (simple, conceptual)**

A **pure function** always returns the same output for the same input and has no side effects.

An **impure function** depends on external data or modifies external state.

■ **When/Why is it used?**

Pure functions help in predictable code, testing, and functional programming.

Impure functions are used when we need side effects like updating state, modifying DOM, or changing variables.

■ **Example / Code snippet**

```
// Pure → no side effects
function add(a, b){
  return a + b;
}

// Impure → modifies external variable
let count = 0;
function increment(){
  count++;
}
```

■ **One-line summary:**

> Pure functions have no side effects and always return consistent results; impure functions modify external state.

📌 3. OOP in JavaScript

✅ 1) What is prototype in JavaScript?

■ **Explanation (simple, conceptual)**

Every JavaScript object has a hidden property called **prototype**, which contains shared methods and properties. When you access a property on an object, JS looks for it on the object and then in its prototype chain.

■ **When/Why is it used?**

Used for inheritance, memory optimization (methods shared across objects), and extending built-in objects.

■ **Example / Code snippet**

```
function Person(name){
  this.name = name;
}

Person.prototype.sayHi = function(){
  console.log("Hi " + this.name);
};

const p = new Person("Mohini");
p.sayHi();  // uses prototype method
```

■ **One-line summary:**

> Prototype is an object used for inheritance and sharing methods across instances.

✅ 2) Difference between Classical vs Prototypal inheritance?

■ Explanation (simple, conceptual)

Classical inheritance (Java, C#) is based on classes creating objects.

Prototypal inheritance (JavaScript) is based on objects inheriting from other objects, using the prototype chain.

■ When/Why is it used?

JS uses prototypal inheritance because it is more flexible—objects can inherit directly without needing classes.

■ Example / Code snippet

```
// Prototypal inheritance
const parent = { greet(){ console.log("hello"); } };
const child = Object.create(parent);
child.greet();
```

■ One-line summary:

> Classical uses classes; prototypal uses objects inheriting from other objects via prototypes.

✅ 3) How to create class and objects in JS? (ES6 class)

■ Explanation (simple, conceptual)

ES6 introduced the `class` keyword as a cleaner syntax over prototype-based object creation.

■ When/Why is it used?

Used to create reusable object blueprints and perform inheritance in a cleaner, modern syntax.

■ Example / Code snippet

```
class Task {
  constructor(title, priority){
    this.title = title;
    this.priority = priority;
  }

  show(){
    console.log(`Task: ${this.title}`);
  }
}

const t1 = new Task("Learn JS", "High");
t1.show();
```

■ **One-line summary:**

> ES6 classes are syntactic sugar over prototypes to create objects and structure code cleanly.

✅ 4) What is `this` keyword? How it behaves differently?

■ **Explanation (simple, conceptual)**

`this` refers to the current execution context.

Its value depends on **how** the function is called, not where it is defined.

■ **When/Why is it used?**

Used to access object properties inside methods, event handlers, or constructors.

■ **Example / Code snippet**

```
const obj = {
  name: "Mohini",
  show(){ console.log(this.name); }
```

```
};
obj.show();  // this → obj
```

Inside normal function

```
function test(){
  console.log(this); // global (window) in browser
}
```

■ **One-line summary:**

> this depends on the calling context — object, function, event, or global.

✅ 5) Arrow functions & `this` — difference?

■ **Explanation (simple, conceptual)**

Arrow functions do **not** have their own `this`.

They inherit `this` from the surrounding (lexical) scope.

■ **When/Why is it used?**

Used when you want to preserve the parent context inside callbacks or event handlers.

■ **Example / Code snippet**

```
const obj = {
  value: 10,
  show: function(){
    setTimeout(() => {
      console.log(this.value); // arrow inherits obj's this
    }, 1000);
  }
};
obj.show();
```

Normal function loses `this`:

```
setTimeout(function(){
  console.log(this.value); // undefined
}, 1000);
```

■ **One-line summary:**

> Arrow functions inherit this from the parent; normal functions have their own this.

✅ 6) Constructor & Inheritance using `extends`

■ **Explanation (simple, conceptual)**

JavaScript classes use `constructor()` to initialize objects.

`extends` allows one class to inherit properties/methods from another class.

■ **When/Why is it used?**

Used to build reusable hierarchical models—e.g., BaseUser → AdminUser.

■ **Example / Code snippet**

```
class Task {
  constructor(title){
    this.title = title;
  }
  show(){ console.log(this.title); }
}

class UrgentTask extends Task {
  constructor(title, priority){
    super(title);         // calls parent constructor
    this.priority = priority;
  }

  showPriority(){
    console.log(this.priority);
  }
```

```
}

const ut = new UrgentTask("Pay Bills", "High");
ut.show();        // inherited method
ut.showPriority(); // child method
```

■ **One-line summary:**

> constructor initializes objects, and extends + super() enable class-based inheritance in JavaScript.

📌 4. Async Programming (VERY IMPORTANT)

✅ 1) What is event loop?

■ **Explanation (simple, conceptual)**

The **event loop** is the mechanism in JavaScript that allows asynchronous code (like setTimeout, promises, API calls) to run in a single-threaded environment.

It continuously checks the call stack and pushes pending callbacks from the task queues when the stack becomes empty.

■ **When/Why is it used?**

Used for handling asynchronous operations efficiently without blocking the main thread — making JavaScript non-blocking.

■ **Example / Code snippet**

```
console.log("start");
setTimeout(() => console.log("async"), 0);
console.log("end");

// Output: start, end, async
```

■ **One-line summary:**

> Event loop manages asynchronous tasks in JS, ensuring non-blocking execution in a single thread.

✅ 2) What is call stack?

■ **Explanation (simple, conceptual)**

The **call stack** is a data structure where JavaScript stores function execution contexts.

When a function is called, it's pushed onto the stack, and when it finishes, it's popped off.

■ **When/Why is it used?**

It determines the execution order of synchronous code and is key to understanding stack overflow errors.

■ **Example / Code snippet**

```
function a(){ b(); }
function b(){ console.log("Hello"); }

a(); // a() goes into stack → b() goes into stack → executes
```

■ **One-line summary:**

> Call stack stores the order of function executions and handles synchronous code.

✅ 3) What are Web APIs?

■ **Explanation (simple, conceptual)**

Web APIs are browser-provided features (not part of JavaScript itself) like **DOM APIs, setTimeout, fetch, localStorage**, etc.

JavaScript delegates asynchronous tasks to these Web APIs.

■ **When/Why is it used?**

Used when interacting with the browser — manipulating DOM, making network calls, timers, storage, geolocation, etc.

■ **Example / Code snippet**

```
setTimeout(() ⇒ console.log("done"), 1000); // Web API timer handles dela
```

```
y
```

■ **One-line summary:**

> Web APIs are browser-provided functionalities that JS uses for DOM, timers, fetch, and asynchronous operations.

✅ 4) What are Promises in JS?

■ **Explanation (simple, conceptual)**

A Promise is an object that represents a value that will be available **now, later, or never**.

It handles asynchronous operations more cleanly than callbacks.

■ **When/Why is it used?**

Used for API calling, reading files, time-based tasks, and avoiding callback hell.

■ **Example / Code snippet**

```
let p = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Done"), 1000);
});

p.then(res => console.log(res));
```

■ **One-line summary:**

> A Promise represents the result of an asynchronous operation with cleaner handling than callbacks.

✅ 5) Promise states → pending, fulfilled, rejected

■ **Explanation (simple, conceptual)**

A Promise has **three states**:

1. **Pending** → initial state, not completed

2. **Fulfilled** → completed successfully, calls `resolve()`
3. **Rejected** → failed, calls `reject()`

■ When/Why is it used?

Used in asynchronous flows like API responses, where success and failure must be handled separately.

■ Example / Code snippet

```
const promise = new Promise((resolve, reject) => {
  const success = true;
  success ? resolve("Success") : reject("Error");
});
```

■ One-line summary:

> Promises move from pending → fulfilled or rejected based on asynchronous success or failure.

✅ 6) `.then()` vs `.catch()` vs `.finally()`

■ Explanation (simple, conceptual)

- `.then()` handles successful promise resolution.
- `.catch()` handles errors or rejected promises.
- `.finally()` runs regardless of success or failure (cleanup logic).

■ When/Why is it used?

Use `.then()` for success results, `.catch()` for error handling, and `.finally()` for cleanup tasks like hiding loaders.

■ Example / Code snippet

```
fetch("/api/tasks")
  .then(res => res.json())       // success
  .catch(err => console.error(err)) // error
  .finally(() => console.log("Done")); // always runs
```

■ One-line summary:

> .then() handles success, .catch() handles errors, .finally() always runs for cleanup.

✅ 7) async vs await

■ Explanation (simple, conceptual)

- `async` marks a function as asynchronous and ensures it returns a Promise.
- `await` pauses the execution inside an async function until a Promise resolves.

■ When/Why is it used?

Used to write asynchronous code in a clean, readable, synchronous-looking style.

■ Example / Code snippet

```
async function loadData(){
  const res = await fetch("/api/tasks");
  const data = await res.json();
  console.log(data);
}
loadData();
```

■ One-line summary:

> async makes a function return a Promise; await pauses until the Promise resolves.

✅ 8) Why async/await is better than Promises?

■ Explanation (simple, conceptual)

Async/await makes code look linear and readable, avoiding nested `.then()` chains. It also simplifies error handling using normal `try/catch`.

■ When/Why is it used?

Used for complex async workflows like fetching multiple APIs, sequential operations, or cleaner error handling.

■ **Example / Code snippet**

Using Promises

```
fetch("/api")
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.log(err));
```

Using async/await

```
try {
  const res = await fetch("/api");
  const data = await res.json();
} catch (err) {
  console.log(err);
}
```

■ **One-line summary:**

> Async/await is cleaner, readable, and handles errors better than chained Promises.

✅ 9) What is setTimeout and setInterval?

■ **Explanation (simple, conceptual)**

- `setTimeout()` runs a function **once** after a delay.
- `setInterval()` runs a function **repeatedly** at fixed intervals.

■ **When/Why is it used?**

Used for timers, periodic updates, delayed actions, scheduling UI updates, animations, and API polling.

■ **Example / Code snippet**

```
setTimeout(() => console.log("run after 1 sec"), 1000);

setInterval(() => console.log("run every sec"), 1000);
```

■ **One-line summary:**

> setTimeout runs once after delay; setInterval repeats execution at intervals.

✅ 10) What is callback hell?

■ **Explanation (simple, conceptual)**

Callback hell is a situation where multiple nested callbacks create deep, unreadable, and hard-to-maintain code—often called “pyramid of doom.”

■ **When/Why is it used?**

Occurs when doing sequential async operations using callbacks—like multiple API calls, file reads, or time-based events.

■ **Example / Code snippet**

```
getUser(function(user){
  getOrders(user.id, function(orders){
    getItems(orders[0], function(items){
      console.log(items); // deeply nested → callback hell
    });
  });
});
```

Better approach: Promises or async/await

■ **One-line summary:**

> Callback hell is deeply nested asynchronous callbacks that make code messy; solved using Promises or async/await.

📌 5. Array & String Operations

✅ 1) Difference between `map` vs `forEach`

■ Explanation (simple, conceptual)

`map()` creates and returns a **new array**, applying a function to each element.

`forEach()` simply **iterates** over the array and does **not** return anything.

■ When/Why is it used?

Use `map` when you want to transform data into a new array.

Use `forEach` for side effects like logging or updating UI.

■ Example / Code snippet

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

let doubled = nums.map(n => n * 2);   // [2, 4, 6]

nums.forEach(n => console.log(n));    // prints values, no return
```

■ One-line summary:

> map returns a new array; forEach is only for iteration without returning.

✅ 2) filter(), reduce(), find(), some(), every()

■ Explanation (simple, conceptual)

These are commonly used array methods:

- **filter()** → returns all elements matching a condition
- **find()** → returns the first matching element
- **reduce()** → reduces array to a single value
- **some()** → returns true if **at least one** element satisfies a condition
- **every()** → returns true if **all** elements satisfy a condition

■ When/Why is it used?

Used for searching, checking conditions, transforming arrays, and aggregating values.

■ **Example / Code snippet**

```
let arr = [10, 20, 30, 40];

arr.filter(x => x > 20);      // [30, 40]
arr.find(x => x === 20);      // 20
arr.some(x => x < 0);         // false
arr.every(x => x > 5);        // true
arr.reduce((sum, x) => sum + x, 0);  // 100
```

■ **One-line summary:**

> filter (many), find (first), some (at least one), every (all), reduce (aggregate).

✅ 3) Convert array to string and vice versa

■ **Explanation (simple, conceptual)**

JavaScript provides built-in methods to convert arrays to strings and strings back to arrays.

■ **When/Why is it used?**

Used in serialization, storing arrays in localStorage, sending arrays in API requests, or splitting CSV-like data.

■ **Example / Code snippet**

Array → String

```
let arr = ["a", "b", "c"];
let str = arr.join(","); // "a,b,c"
```

String → Array

```
let s = "a,b,c";
let arr2 = s.split(","); // ["a","b","c"]
```

■ **One-line summary:**

> join() converts array → string, split() converts string → array.

✅ 4) Sort numbers properly in JS

■ **Explanation (simple, conceptual)**

JavaScript's default `sort()` treats values as **strings**, leading to incorrect numeric sorting.

We must provide a compare function for correct numeric sorting.

■ **When/Why is it used?**

Used to sort numbers, dates, prices, or numeric IDs correctly.

■ **Example / Code snippet**

```
let nums = [10, 50, 2, 5];

// WRONG (string sort)
nums.sort();       // [10, 2, 5, 50]

// RIGHT
nums.sort((a, b) ⇒ a - b);  // ascending: [2, 5, 10, 50]
nums.sort((a, b) ⇒ b - a);  // descending
```

■ **One-line summary:**

> Use a compare function sort((a,b)⇒a-b) to sort numbers correctly.

✅ 5) Remove duplicates from an array

■ **Explanation (simple, conceptual)**

You can remove duplicates using modern JS features like `Set`, or functions like `filter()`.

■ **When/Why is it used?**

Used when cleaning API data, merging arrays, and removing redundant items.

■ Example / Code snippet

Using Set (best & simplest):

```
let arr = [1,2,2,3,4,4];
let unique = [...new Set(arr)]; // [1,2,3,4]
```

Using filter():

```
let unique2 = arr.filter((v, i) => arr.indexOf(v) === i);
```

Using reduce():

```
let unique3 = arr.reduce((acc, curr) =>
  acc.includes(curr) ? acc : [...acc, curr], []);
```

■ One-line summary:

> The easiest way to remove duplicates is using new Set().

✅ 6) Flatten array using `flat()`

■ Explanation (simple, conceptual)

`flat()` is a built-in array method that reduces nested arrays into a single-level array.

You can specify depth like `flat(1)`, `flat(2)`, or use `flat(Infinity)` for deep flattening.

■ When/Why is it used?

Used when you receive nested data from APIs or when merging hierarchical arrays into a simple, flat list.

■ Example / Code snippet

```
let arr = [1, [2, 3], [4, [5, 6]]];

arr.flat();         // [1, 2, 3, 4, [5, 6]]
arr.flat(2);        // [1, 2, 3, 4, 5, 6]
arr.flat(Infinity); // completely flatten
```

■ **One-line summary:**

> flat() removes nested levels from an array and returns a flattened version.

✅ 7) Merge two arrays using spread operator

■ **Explanation (simple, conceptual)**

The spread operator (`...`) expands array elements, making it easy to merge arrays into one.

■ **When/Why is it used?**

Used to combine datasets, append lists, or merge API results without mutating original arrays.

■ **Example / Code snippet**

```
let arr1 = [1, 2];
let arr2 = [3, 4];

let merged = [...arr1, ...arr2]; // [1, 2, 3, 4]
```

■ **One-line summary:**

> Use [..., ...] with spread operator to merge arrays cleanly.

✅ 8) Reverse words of a string

■ **Explanation (simple, conceptual)**

You can reverse the words by splitting the string, reversing the array, and joining it back.

■ **When/Why is it used?**

Useful in formatting text, search optimization, and string manipulation problems in coding rounds.

■ **Example / Code snippet**

```
let str = "I love JavaScript";

let reversed = str.split(" ").reverse().join(" ");
console.log(reversed); // "JavaScript love I"
```

■ **One-line summary:**

> Split → Reverse → Join is the simplest way to reverse words in a string.

✅ 9) Find common elements in two arrays

■ **Explanation (simple, conceptual)**

We can find intersection by checking which elements of one array appear in the other—using `filter()`, `Set`, or both.

■ **When/Why is it used?**

Useful in comparing lists, permissions, filters, and real-world logic like "matching items".

■ **Example / Code snippet**

Using filter + includes (simple):

```
let a = [1,2,3,4];
let b = [3,4,5];

let common = a.filter(x => b.includes(x)); // [3, 4]
```

Using Set (faster for large arrays):

```
let setB = new Set(b);
let intersection = a.filter(x => setB.has(x));
```

■ **One-line summary:**

> Use filter() with includes or Set to get the intersection of two arrays.

📌 6. Objects in JavaScript

✅ 1) What is object destructuring?

■ Explanation (simple, conceptual)

Object destructuring allows you to extract properties from an object and assign them to variables in a clean, shorter way.

■ When/Why is it used?

Used to clean up code, reduce repetition, unpack API responses, and access only required fields.

■ Example / Code snippet

```
const user = { name: "Mohini", age: 23, city: "Hyderabad" };

const { name, age } = user;

console.log(name); // Mohini
console.log(age);  // 23
```

■ One-line summary:

> Object destructuring extracts specific properties from an object into variables cleanly.

✅ 2) What is spread operator `{...obj}` ?

■ Explanation (simple, conceptual)

The spread operator expands an object or array into individual elements. For objects, `{...obj}` creates a shallow copy.

■ When/Why is it used?

Used for cloning objects, merging objects, updating state in React/Angular, and copying arrays without mutation.

■ Example / Code snippet

```
const user = { name: "Mohini", age: 23 };

const updated = { ...user, city: "Hyderabad" };
```

■ **One-line summary:**

> Spread operator expands objects/arrays and is mainly used for copying or merging.

✅ 3) Freeze vs Seal vs PreventExtensions

■ **Explanation (simple, conceptual)**

Feature	Object.freeze()	Object.seal()	Object.preventExtensions()
Add new props	❌ No	❌ No	❌ No
Delete props	❌ No	❌ No	✓ Yes
Modify existing props	❌ No	✓ Yes	✓ Yes
Make non-configurable	✓ Yes	✓ Yes	❌ No

■ **When/Why is it used?**

Used to control mutability of objects (freeze = immutable, seal = partially locked, preventExtensions = stop adding props).

■ **Example / Code snippet**

```
const obj = {a:1};
Object.freeze(obj);
// obj.a = 10; // ❌ ignored
```

■ **One-line summary:**

> Freeze prevents all changes, Seal allows edits but no adds/deletes, PreventExtensions only blocks additions.

✅ 4) Shallow copy vs Deep copy

■ Explanation (simple, conceptual)

- **Shallow copy:** Copies only top-level properties; nested objects share references.
- **Deep copy:** Completely copies all levels, creating fully independent objects.

■ When/Why is it used?

Shallow copy is fine for simple objects; deep copy is used when object contains nested objects and mutations must not affect original.

■ Example / Code snippet

Shallow Copy (spread operator):

```
let obj1 = { a:1, b:{ c:2 } };
let copy1 = { ...obj1 };
copy1.b.c = 99;

console.log(obj1.b.c); // 99 (changed)
```

Deep Copy (structuredClone / JSON):

```
let copy2 = structuredClone(obj1);
copy2.b.c = 50;

console.log(obj1.b.c); // 2 (not affected)
```

■ One-line summary:

> Shallow copies copy top-level only; deep copies duplicate nested objects too.

✅ 5) How to clone object properly?

■ Explanation (simple, conceptual)

You can clone objects using shallow or deep copy methods depending on whether the object contains nested data.

■ When/Why is it used?

Used to avoid accidental mutation, especially in state management (React, Angular).

■ Example / Code snippet

✔ Shallow Clone

```
let clone1 = { ...obj };
```

✔ Deep Clone (best)

```
let deep = structuredClone(obj);
```

✔ Deep Clone (fallback)

```
let deep2 = JSON.parse(JSON.stringify(obj));
```

■ One-line summary:

> Use structuredClone() for accurate deep cloning; spread operator for shallow copying.

✅ 6) What is JSON.stringify & JSON.parse used for?

■ Explanation (simple, conceptual)

`JSON.stringify()` converts an object into a JSON string.

`JSON.parse()` converts a JSON string back to an object.

■ When/Why is it used?

Used for API communication, storing data in localStorage, deep cloning, and sending objects over the network.

■ Example / Code snippet

```
let obj = { name: "Mohini", age: 23 };
```

```
let str = JSON.stringify(obj);       // object → string
let newObj = JSON.parse(str);        // string → object
```

■ **One-line summary:**

> stringify() converts object → JSON string, parse() converts JSON string → object.

📌 7. DOM + Browser Concepts

✅ 1) What is DOM?

■ **Explanation (simple, conceptual)**

DOM (Document Object Model) is a programming interface that represents the structure of a web page as a tree of nodes. JavaScript can use DOM to read, modify, and update HTML/CSS dynamically.

■ **When/Why is it used?**

Used when updating UI, changing content, handling events, or creating interactive webpages.

■ **Example / Code snippet**

```
document.body.style.background = "lightblue";
```

■ **One-line summary:**

> DOM is a tree structure of the webpage that JavaScript uses to access and modify UI elements.

✅ 2) querySelector vs getElementById

■ **Explanation (simple, conceptual)**

- `getElementById()` → selects element by **ID** (fastest).
- `querySelector()` → selects using **CSS selectors** (ID, class, tag, attribute).

■ **When/Why is it used?**

Use `getElementById` for single known IDs.

Use `querySelector` for flexible selections (classes, nested elements).

■ **Example / Code snippet**

```
document.getElementById("title");      // by ID
document.querySelector(".btn-primary"); // CSS selector
```

■ **One-line summary:**

> getElementById is faster but limited; querySelector is more flexible using CSS selectors.

✅ 3) How to access and modify HTML elements using JS?

■ **Explanation (simple, conceptual)**

You can use DOM methods to select elements and modify text, attributes, or styles dynamically.

■ **When/Why is it used?**

Used for updating UI after user actions, API responses, or form submissions.

■ **Example / Code snippet**

```
let title = document.querySelector("#title");
title.textContent = "Updated text";
title.style.color = "red";
title.setAttribute("data-id", "10");
```

■ **One-line summary:**

> Use DOM selectors to access elements and update text, style, or attributes.

✅ 4) Add/remove HTML dynamically

■ **Explanation (simple, conceptual)**

JavaScript can create new elements, append them to the DOM, or remove elements dynamically.

■ When/Why is it used?

Used for dynamic UI updates like adding rows, cards, lists, notifications, and removing items.

■ Example / Code snippet

Add Element

```
let div = document.createElement("div");
div.innerText = "New Task Added";
document.body.appendChild(div);
```

Remove Element

```
let element = document.querySelector("#removeMe");
element.remove();
```

■ One-line summary:

> Use createElement, appendChild, and remove to dynamically add or remove UI elements.

✅ 5) What is event bubbling vs capturing?

■ Explanation (simple, conceptual)

- **Bubbling (default):** Event goes from **child → parent**
- **Capturing:** Event goes from **parent → child**

■ When/Why is it used?

Used for event delegation, handling nested elements, or controlling where event execution should start.

■ Example / Code snippet

```
div.addEventListener("click", handler, false); // bubbling (default)
div.addEventListener("click", handler, true);  // capturing
```

■ **One-line summary:**

> Bubbling flows from child to parent; capturing flows from parent to child.

✅ 6) What are Event Listeners?

■ **Explanation (simple, conceptual)**

Event listeners are functions that run when a specific event occurs (click, input, keypress, submit).

■ **When/Why is it used?**

Used for user interactions—buttons, forms, keystrokes, API calls on UI actions.

■ **Example / Code snippet**

```
document.querySelector("#btn")
  .addEventListener("click", () ⇒ alert("Clicked!"));
```

■ **One-line summary:**

> Event listeners execute functions when a user interacts with the webpage.

✅ 7) localStorage vs sessionStorage vs cookies

■ **Explanation (simple, conceptual)**

Feature	localStorage	sessionStorage	Cookies
Lifetime	Permanent until cleared	Ends when tab closes	Custom expiry
Capacity	~5–10 MB	~5 MB	~4 KB
Sent to server?	❌ No	❌ No	✔ Yes (by default)
Use-case	Preferences, token	Per-tab data	Auth, tracking

■ **When/Why is it used?**

- `localStorage` → save user settings, theme, token
- `sessionStorage` → store data per browser tab
- `cookies` → authentication, server communication

■ Example / Code snippet

```
localStorage.setItem("name", "Mohini");
sessionStorage.setItem("sessionId", "123");
document.cookie = "token=abc; max-age=3600";
```

■ One-line summary:

> localStorage is permanent, sessionStorage lasts per tab, cookies are small and sent to server automatically

---

📌 8. Error Handling & Debugging

✅ 1) try/catch usage

■ Explanation (simple, conceptual)

`try/catch` is used to handle runtime errors without crashing the application. Code inside `try` runs normally; if an error occurs, control immediately moves to `catch`.

■ When/Why is it used?

Used when executing risky operations like API calls, JSON parsing, DOM manipulation, or user input validation.

■ Example / Code snippet

```
try {
  let data = JSON.parse("{ invalid json }");
} catch (err) {
  console.log("Error happened:", err.message);
}
```

■ One-line summary:

> try/catch prevents crashes by catching runtime errors and allowing graceful recovery.

✅ 2) Custom errors in JS

■ Explanation (simple, conceptual)

JavaScript allows creating custom error classes by extending the built-in `Error` class. This helps define domain-specific errors.

■ When/Why is it used?

Used when building applications that need meaningful error messages, better debugging, or layered error handling.

■ Example / Code snippet

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

try {
  throw new ValidationError("Invalid email format");
} catch (err) {
  console.log(err.name, err.message);
}
```

■ One-line summary:

> Custom errors help create meaningful, specific error types for better debugging and control.

✅ 3) Real-time debugging tools in browser

■ Explanation (simple, conceptual)

Browsers like Chrome provide developer tools for debugging JavaScript, inspecting DOM, monitoring network calls, analyzing performance, and detecting memory leaks.

■ **When/Why is it used?**

Used during development to find errors quickly, check variables, set breakpoints, profile performance, and inspect API calls.

■ **Example / Tools**

- **Console** → log variables, errors
- **Sources tab** → breakpoints, step-through debugging
- **Network tab** → monitor API requests
- **Elements tab** → inspect HTML/CSS
- **Performance tab** → detect lag
- **Memory tab** → find leaks

■ **Code snippet (console debugging example)**

```
console.log("Value:", value);
console.table(users);
debugger; // pauses execution in Sources tab
```

■ **One-line summary:**

> Chrome DevTools lets you debug JS in real-time using breakpoints, console, network inspection, and performance tools.

✅ 4) Common runtime vs logical JS errors

■ **Explanation (simple, conceptual)**

- **Runtime errors:** Occur during execution due to invalid operations.
- **Logical errors:** Code runs but gives incorrect results due to wrong logic.

■ **When/Why is it used?**

Understanding this helps decide whether to use try/catch (runtime errors) or debugging and test cases (logical errors).

■ **Example / Code snippet**

Runtime error

```
console.log(x); // ReferenceError: x is not defined
```

Logical error

```
let price = "100";
console.log(price + 20);  // outputs "10020" (wrong result)
```

■ **One-line summary:**

> Runtime errors break execution; logical errors produce wrong output without crashing.

📌 9. ES6+ Concepts (Must Know)

✅ 1) let, const, arrow functions

■ **Explanation (simple, conceptual)**

- **let** → block-scoped variable, can be reassigned.
- **const** → block-scoped constant, cannot be reassigned.
- **Arrow functions** → shorter syntax, no own `this`, no `arguments` object.

■ **When/Why is it used?**

Use `let` for changing values, `const` for constants or objects, and arrow functions for cleaner callbacks.

■ **Example**

```
let count = 1;
const PI = 3.14;

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

■ **One-line summary:**

> let = variable, const = fixed, arrow functions = shorter, no-own-this.

✅ 2) Spread vs Rest Operators

■ **Explanation (simple, conceptual)**

- **Spread (`...`)** → expands elements.
- **Rest (`...`)** → collects remaining elements.

■ **When/Why is it used?**

Spread: copying/merging arrays & objects.

Rest: handling variable parameters in functions.

■ **Example**

```
// Spread
let arr = [1, 2];
let newArr = [...arr, 3]; // [1,2,3]

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

■ **One-line summary:**

> Spread expands items; Rest gathers them.

✅ 3) Destructuring arrays/objects

■ **Explanation (simple, conceptual)**

Destructuring extracts specific values directly from arrays/objects into variables.

■ **When/Why is it used?**

Useful when handling API responses or cleaner variable extraction.

■ **Example**

```
// Array
let [a, b] = [10, 20];

// Object
let { name, age } = { name: "Mohini", age: 23 };
```

■ **One-line summary:**

> Destructuring cleanly extracts values from arrays/objects.

✅ 4) Template strings

■ **Explanation (simple, conceptual)**

Template strings use backticks (`` ` ``) and `${}` to create dynamic strings.

■ **When/Why is it used?**

Used for multi-line strings, embedding variables, and building UI templates.

■ **Example**

```
let user = "Mohini";
let msg = `Hello ${user}, welcome!`;
```

■ **One-line summary:**

> Template strings allow dynamic strings using ${}.

✅ 5) Classes & inheritance

■ **Explanation (simple, conceptual)**

ES6 `class` provides a cleaner syntax for prototypal inheritance.

`extends` allows child classes to inherit from parent classes.

■ **When/Why is it used?**

Used to organize code, create reusable object blueprints, and model hierarchies.

■ Example

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

class Admin extends User {
  constructor(name, role) {
    super(name);
    this.role = role;
  }
}
```

■ One-line summary:

> Classes simplify object creation and extends enables inheritance.

✅ 6) Modules — import/export

■ Explanation (simple, conceptual)

Modules allow splitting JS code into separate files using `export` and `import`.

■ When/Why is it used?

Used to organize large projects, reuse utilities, and improve maintainability.

■ Example

```
// utils.js
export let add = (a,b) => a+b;

// main.js
import { add } from './utils.js';
```

■ One-line summary:

> Modules let you export/import code across files for better structure.

✅ 7) Default parameters

■ Explanation (simple, conceptual)

Default parameters provide fallback values if no argument is passed.

■ When/Why is it used?

Useful for safe functions, optional parameters, and preventing `undefined` errors.

■ Example

```
function greet(name = "Guest") {
  console.log("Hello", name);
}
greet(); // Hello Guest
```

■ One-line summary:

> Default parameters give automatic fallback values to function arguments.

✅ 8) Generators

■ Explanation (simple, conceptual)

Generators are special functions using `function*` that can pause and resume execution using `yield`.

■ When/Why is it used?

Used for on-demand data generation, async flows, and custom iterators.

■ Example

```
function* count() {
  yield 1;
  yield 2;
  yield 3;
}

let c = count();
console.log(c.next().value); // 1
```

■ **One-line summary:**

> Generators are pausable functions using yield for controlled execution.

✅ 9) Symbol type

■ **Explanation (simple, conceptual)**

`Symbol` is a unique and immutable primitive used as a unique object property key.

■ **When/Why is it used?**

Used to create private-like properties, avoid naming conflicts, and support meta-programming.

■ **Example**

```
let id = Symbol("id");
let user = { [id]: 101 };
```

■ **One-line summary:**

> Symbol creates unique keys to avoid property conflicts and provide hidden properties.

📌 10. Memory & Performance

✅ 1) Garbage Collection Concept

■ **Explanation (simple, conceptual)**

Garbage Collection (GC) in JavaScript is an automatic memory management process where the JS engine (like V8) removes objects from memory when they are no longer reachable.

JavaScript uses the **"mark and sweep"** algorithm — during GC, the engine marks all reachable objects starting from the global root (`window`), and anything not reachable is removed.

■ **When/Why is it used?**

Garbage collection ensures efficient memory usage and prevents application crashes due to memory overload. Developers don't manually free memory like in C/C++; GC automatically handles it, making JS easier and safer to write.

■ **Example / Code snippet**

```
function createUser() {
  let user = { name: "Mohini" }; // allocated in memory
  return user;
}

let u = createUser();
u = null; // removing reference → GC can clean it
```

Once `u` is set to null, there's no reference to the object → GC cleans it.

■ **One-line summary:**

> JavaScript automatically frees memory by removing unreachable objects using the mark-and-sweep algorithm.

✅ 2) Memory Leaks in JS — how to avoid?

■ **Explanation (simple, conceptual)**

A memory leak happens when your program unintentionally keeps references to data that is no longer needed, preventing garbage collection.

Common leak sources include global variables, unused timers, accidental closures, DOM references, and event listeners not removed.

■ **When/Why is it used?**

Preventing memory leaks is crucial for long-running apps like dashboards, SPAs (Angular), and pages that update frequently — otherwise the browser becomes slow or crashes.

■ **Example / Code snippet**

Common Causes & Fixes

1️⃣ Uncleared Timers (setInterval leak)

```
let interval = setInterval(() => { /* work */ }, 1000);
clearInterval(interval); // FIX: always clear
```

2️⃣ Event listeners not removed

```
function subscribe() {
  window.addEventListener("resize", handler);
}

function unsubscribe() {
  window.removeEventListener("resize", handler); // FIX
}
```

3️⃣ Global variables leak

```
leaked = "hello"; // becomes global automatically ❌
"use strict"; // FIX: prevents this
```

4️⃣ Detached DOM nodes

```
let div = document.getElementById("myDiv");
div.remove();
div = null; // FIX: remove reference after removing DOM element
```

5️⃣ Overuse of closures

```
function outer() {
  let bigArr = new Array(1000000);
  return function inner() {
    console.log("hello");
  };
}
let fn = outer(); // FIX: heavy data remains in scope unnecessarily
```

■ One-line summary:

> Memory leaks occur when unused objects stay referenced—avoid them by clearing timers, removing event listeners, avoiding accidental globals, and

> cleaning DOM references.

✅ 3) Debouncing vs Throttling for performance improvement

■ Explanation (simple, conceptual)

Both techniques limit how often a function executes, improving performance during high-frequency events like scroll, resize, or keypress.

Debouncing:

Executes the function **after the last event**.

Resets the timer every time the event fires.

Throttling:

Executes the function at **regular intervals**, ignoring excessive events.

■ When/Why is it used?

Technique	When Used	Why
Debounce	Search box typing, form validation, resize UI	Avoid unnecessary repeated calls
Throttle	Scroll events, infinite scrolling, window resize, drag events	Control event frequency for performance

■ Example / Code snippet

Debounce (runs after stop typing)

```
function debounce(fn, delay) {
  let timer;
  return function(...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn.apply(this, args), delay);
  };
}

window.addEventListener("resize", debounce(() => {
```

```
  console.log("Resize event handled!");
}, 300));
```

Throttle (runs every 500 ms only)

```
function throttle(fn, limit) {
  let lastCall = 0;
  return function(...args) {
    let now = Date.now();
    if (now - lastCall >= limit) {
      lastCall = now;
      fn.apply(this, args);
    }
  };
}

window.addEventListener("scroll", throttle(() => {
  console.log("Scroll event handled!");
}, 500));
```

■ One-line summary:

> Debouncing waits for a pause to run; throttling runs at fixed intervals—both improve performance during rapid events.

📌 11. API Calls with JS / Fetch

1) How to make GET/POST API call using `fetch`?

■ Explanation (simple, conceptual)

`fetch` is a built-in browser API that returns a Promise and is used to make HTTP requests. For GET you call `fetch(url)` and for POST you pass a config object with `method`, `headers`, and a JSON `body`. Always check `response.ok` before using `response.json()` because `fetch` only rejects on network failure, not HTTP errors.

■ When/Why is it used

Use `fetch` for lightweight API calls in the browser when you don't want an extra dependency. Prefer `fetch` for standard REST calls; use `AbortController` to cancel

long requests and handle timeouts.

■ **Example / Code snippet (async/await)**

GET

```
async function getTasks() {
  const res = await fetch('/api/tasks', { method: 'GET', credentials: 'same-or
igin' });
  if (!res.ok) throw new Error(`API error ${res.status}`);
  const data = await res.json();
  return data;
}
```

POST

```
async function createTask(task) {
  const res = await fetch('/api/tasks', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(task),
    credentials: 'same-origin'
  });
  if (!res.ok) {
    const errBody = await res.text();
    throw new Error(`Create failed (${res.status}): ${errBody}`);
  }
  return await res.json(); // created resource
}
```

Timeout with AbortController

```
const controller = new AbortController();
const id = setTimeout(() => controller.abort(), 5000); // 5s timeout
try {
  const res = await fetch('/api/long', { signal: controller.signal });
  clearTimeout(id);
  // ...
} catch (err) {
```

```
  if (err.name === 'AbortError') console.log('Request timed out');
}
```

■ **One-line summary:**

> Use fetch(url, {method, headers, body}) with response.ok checks and AbortController for timeouts when performing GET/POST requests.

---

2) `fetch` vs `axios`?

■ **Explanation (simple, conceptual)**

`fetch` is native and minimal; `axios` is a popular third-party library that wraps HTTP requests and adds features like automatic JSON transform, request/response interceptors, timeout handling, and easier cancellation.

■ **When/Why is it used**

Use `fetch` to avoid extra dependencies for simple use-cases. Use `axios` when you need built-in timeouts, interceptors for auth tokens/refresh, automatic response parsing, or a consistent API across browsers (older browsers require polyfills for `fetch`).

■ **Example / Code snippet**

Axios GET

```
import axios from 'axios';

async function getTasksAxios() {
  const res = await axios.get('/api/tasks'); // res.data has parsed JSON
  return res.data;
}
```

Axios interceptor (attach JWT)

```
axios.interceptors.request.use(config => {
  const token = authService.getToken();
  if (token) config.headers.Authorization = `Bearer ${token}`;
  return config;
});
```

Summary of trade-offs

- `fetch`: no dependency, low-level, needs manual error handling & timeouts.
- `axios`: extra features (interceptors, timeout, automatic JSON), smaller custom code.

■ **One-line summary:**

> fetch is native and minimal; axios adds convenience (timeouts, interceptors, automatic parsing) which speeds up real-world API work.

---

3) Handle API error gracefully

■ **Explanation (simple, conceptual)**

Graceful error handling means: detect HTTP errors, parse error payloads, show friendly UI messages, log details (with correlation id), and optionally retry on transient failures. Always avoid exposing raw server stack traces to users.

■ **When/Why is it used**

Used to improve UX (show meaningful messages), enable observability (log correlation id), and recover from transient issues (retries/backoff). Essential for production-grade APIs and dashboards.

■ **Example / Code snippet**

```
async function safeFetch(url, options = {}) {
 try {
  const res = await fetch(url, options);
  if (!res.ok) {
   // try to read a structured error body
   let errText = await res.text();
   try { errText = JSON.parse(errText); } catch {}
   // Map status codes to user-friendly messages
   const userMsg = res.status >= 500 ? 'Server error, try again later' :
                res.status === 401 ? 'Please login again' :
                (errText.message || 'Request failed');
   // log for devs (send to logging service)
   console.error('API error', { url, status: res.status, body: errText });
   throw new Error(userMsg);
  }
```

```
    return await res.json();
  } catch (err) {
    if (err.name === 'AbortError') throw new Error('Request timed out');
    // network error or thrown above
    throw err; // let caller show friendly UI
  }
}

// Usage in UI
try {
  const tasks = await safeFetch('/api/tasks');
} catch (err) {
  // show toast or inline error: err.message
}
```

■ **One-line summary:**

> Check response.ok, parse the error body, map HTTP codes to user-friendly messages, log details for ops, and apply retries for transient failures.

---

4) Convert JSON response to object

■ **Explanation (simple, conceptual)**

When a server returns JSON, use `response.json()` (fetch) or `res.data` (axios) to parse the JSON into a JS object. For raw strings, use `JSON.parse()`; to send objects to the server, use `JSON.stringify()`.

■ **When/Why is it used**

Used whenever exchanging structured data with backend APIs—parsing responses into objects to access fields, or serializing objects to send as request bodies.

■ **Example / Code snippet**

```
// fetch
const res = await fetch('/api/tasks');
const data = await res.json(); // data is JS object/array

// axios
```

```
const res2 = await axios.get('/api/tasks');
const data2 = res2.data;

// parse raw string
const raw = '{"id":1,"title":"task"}';
const obj = JSON.parse(raw);

// stringify before POST
await fetch('/api/tasks', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ title: 'New' })
});
```

■ **One-line summary:**

> Use response.json() or JSON.parse() to turn JSON text into JS objects, and JSON.stringify() to serialize objects for POST.

---

5) What if server returns 500? How to handle?

■ **Explanation (simple, conceptual)**

HTTP 500 indicates a server-side error. The client's response should be: show a friendly error message, optionally retry if safe (with exponential backoff for transient faults), log the event (with correlation id), and provide a path for the user (retry button, contact support) — but never reveal server internals.

■ **When/Why is it used**

Handle 500s to maintain good UX, avoid blind retries for non-idempotent ops, and ensure operations are auditable. For critical operations prefer safe server-side compensations rather than client retries.

■ **Example / Code snippet (retry with backoff & limit)**

```
async function retryFetch(url, options = {}, maxRetries = 3) {
  let attempt = 0;
  while (attempt < maxRetries) {
    try {
      const res = await fetch(url, options);
```

```
      if (res.ok) return await res.json();
      if (res.status >= 500) {
        attempt++;
        const wait = 200 * Math.pow(2, attempt); // exponential backoff
        await new Promise(r => setTimeout(r, wait));
        continue; // retry
      } else {
        // client error or auth issue: throw immediately
        const body = await res.text();
        throw new Error(body || `HTTP ${res.status}`);
      }
    } catch (err) {
      if (err.name === 'AbortError') throw new Error('Request timed out');
      if (attempt >= maxRetries - 1) throw new Error('Server error, try later');
      attempt++;
      await new Promise(r => setTimeout(r, 200 * Math.pow(2, attempt)));
    }
  }
}
```

UI behavior:

- Show a non-technical message like “Something went wrong — please try again later.”
- Offer “Retry” or save user's work offline (optimistic UI / queue) for later sync.
- Log error with server correlation id and request context for backend teams.

■ **One-line summary:**

> On 500 errors show a friendly message, log details, optionally retry with exponential backoff for transient issues, and avoid exposing server internals to users.

📌 12. Real-World Scenarios

1) Load dropdown values using API — how?

■ **Explanation (simple, conceptual)**

Call a backend endpoint that returns the dropdown items (id/label). In the frontend call this API on component init (or lazy on first open), map the response to the dropdown model, and optionally cache the results in a shared service so multiple components reuse the list.

■ When/Why is it used

Use this for dynamic lists (priority list, categories, teams) that may be maintained server-side, so UI always shows current values and respects central admin changes.

■ Example / Code snippet (Angular)

Service

```
@Injectable({providedIn: 'root'})
export class LookupService {
  private priorities$ = new BehaviorSubject<Lookup[]>(null);

  constructor(private http: HttpClient) {}

  getPriorities() {
    if (!this.priorities$.value) {
      this.http.get<Lookup[]>('/api/lookups/priorities')
        .subscribe(list => this.priorities$.next(list));
    }
    return this.priorities$.asObservable();
  }
}
```

Component

```
ngOnInit() {
  this.lookupService.getPriorities().subscribe(list => {
    this.priorities = list;
  });
}
```

Backend (.NET)

`GET /api/lookups/priorities` returns `[{ id: 1, label: "High" }, ...]`

■ **One-line summary:**

> Fetch dropdown values from a lookup API on init, map them in a shared service, and cache via a BehaviorSubject to reuse across components.

---

2) Validate email/password in JS

■ **Explanation (simple, conceptual)**

Perform client-side validation for UX (format, length) then always validate again on server side. For email use a conservative regex or HTML5 `type="email"`. For password validate length, complexity rules (min length, letters+digits/special) and check on server for breach/strength.

■ **When/Why is it used**

Client validation gives immediate feedback. Server validation is required for security and to prevent malicious input or bypass.

■ **Example / Code snippet (JS + Angular Reactive Form)**

Reactive form validators

```
this.loginForm = this.fb.group({
  email: ['', [Validators.required, Validators.email]],
  password: ['', [Validators.required, Validators.minLength(8),
                  Validators.pattern(/(?=.*[0-9])(?=.*[A-Za-z])/)]]
});
```

Simple email regex (practical)

```
const emailRe = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
emailRe.test('user@example.com'); // true
```

Server-side: re-validate, hash password with bcrypt, rate-limit login attempts.

■ **One-line summary:**

> Validate email/password on the client for UX and always re-validate & securely hash/check on the server for security.

---

3) Show loader until API response returns

■ Explanation (simple, conceptual)

Show a global or component-level spinner while an API call is pending. Best practice: use an HTTP interceptor to count active requests and expose a shared `isLoading` observable, so loader logic is centralized and consistent.

■ When/Why is it used

Used to indicate progress during slow network operations, avoid duplicate actions, and improve perceived performance/UX.

■ Example / Code snippet (Angular interceptor + service)

Loading service

```
@Injectable({ providedIn: 'root' })
export class LoadingService {
  private count = 0;
  private subj = new BehaviorSubject<boolean>(false);
  isLoading$ = this.subj.asObservable();

  show() { if (++this.count === 1) this.subj.next(true); }
  hide() { if (--this.count === 0) this.subj.next(false); }
}
```

Interceptor

```
intercept(req, next) {
  this.loadingService.show();
  return next.handle(req).pipe(finalize(() => this.loadingService.hide()));
}
```

Component template

```
<app-spinner *ngIf="loadingService.isLoading$ | async"></app-spinner>
```

■ One-line summary:

> Use a centralized HTTP interceptor + a shared loading service to show/hide a spinner while requests are in-flight.

---

4) Store user session token securely

■ Explanation (simple, conceptual)

Prefer storing auth tokens in **secure, HttpOnly cookies** (server-set) to mitigate XSS; use refresh-token rotation for long sessions. If cookies are not possible, store tokens in memory (short-lived) and use refresh tokens via a secure, HttpOnly cookie. Avoid localStorage for sensitive tokens unless you accept XSS risk and implement strict CSP.

■ When/Why is it used

Used to authenticate requests and maintain sessions. Proper storage prevents token theft (XSS) and CSRF; cookies + SameSite + secure attributes protect against many attacks.

■ Example / Best-practice patterns

- **Recommended:** Server issues access token in response body and sets refresh token as HttpOnly cookie; client stores short-lived access token in-memory and uses refresh cookie to obtain new access tokens.
- **Angular interceptor** attaches access token from memory and backend validates refresh via cookie.

Cookie set (server-side)

```
Response.Cookies.Append("refreshToken", token, new CookieOptions {
  HttpOnly = true, Secure = true, SameSite = SameSiteMode.Strict, Expires
= DateTime.UtcNow.AddDays(14)
});
```

■ One-line summary:

> Prefer HttpOnly, Secure cookies + refresh-token rotation; avoid localStorage for long-lived sensitive tokens to reduce XSS risk.

---

5) How JavaScript communicates with backend (.NET / Java REST API)?

■ Explanation (simple, conceptual)

JavaScript calls backend REST endpoints over HTTP(S) using `fetch` or `HttpClient` (Angular). Data is serialized as JSON (Content-Type: application/json). Authentication is via tokens (JWT) or cookies. The client handles responses,

status codes, and retries; server handles routing, validation, and returns structured responses.

■ **When/Why is it used**

Used for all CRUD operations, authentication, and integration. The contract is HTTP + JSON making front-end/back-end decoupled and language-agnostic.

■ **Example / Code snippet (Angular)**

```
// GET
this.http.get<Task[]>('/api/tasks').subscribe(data ⇒ this.tasks = data);

// POST with token via interceptor
this.http.post('/api/tasks', task).subscribe();
```

Server (.NET) exposes REST controllers:

```
[HttpGet("api/tasks")]
public IActionResult Get() ⇒ Ok(_service.GetAll());
```

■ **One-line summary:**

> JS communicates with REST APIs over HTTPS using JSON (via fetch/HttpClient), with tokens or cookies for auth and semantic HTTP status codes.

---

6) Open modal on button click → how DOM handles this?

■ **Explanation (simple, conceptual)**

When a user clicks a button, an event listener triggers logic that either toggles a CSS class or creates/inserts a modal DOM node. For accessibility, focus should move into the modal, keyboard navigation trapped inside, and backdrop should block background interaction.

■ **When/Why is it used**

Modals are used for confirmations, forms, or detailed views without leaving the page. Proper DOM handling ensures accessibility and prevents background interaction.

■ **Example / Code snippet (vanilla + Angular approach)**

Vanilla

```
<button id="open">Open</button>
<div id="modal" class="hidden"> ... </div>
```

```
document.getElementById('open').addEventListener('click', () => {
  document.getElementById('modal').classList.remove('hidden');
  document.getElementById('modal').focus(); // focus management
});
```

Angular (recommended): use Angular Material or Bootstrap modal. Simple toggle:

```
<button (click)="showModal = true">Open</button>
<app-modal *ngIf="showModal" (close)="showModal = false"></app-modal>
```

Accessibility: set `aria-modal`, trap focus, return focus to opener when closed.

■ **One-line summary:**

> Button click triggers an event that toggles/inserts a modal DOM node; ensure focus management, backdrop, and accessibility (aria/modal, focus trap).

---

7) Pagination for large data — how?

■ **Explanation (simple, conceptual)**

Implement **server-side pagination**: the client requests page/size (or uses keyset cursor) and server returns only that subset plus metadata (total count, next cursor). For large data use keyset (cursor) pagination to avoid OFFSET cost. Combine with filtering and sorting at DB level.

■ **When/Why is it used**

Used when datasets are large (thousands+ rows). Server-side pagination reduces network payload, DB load, and keeps UI responsive.

■ **Example / Code snippet (API + client)**

API endpoint

```
GET /api/tasks?page=2&size=50
Response: { items: [...], totalCount: 10234 }
```

Keyset (preferred for deep paging)

```
GET /api/tasks?lastId=2345&pageSize=50
Response: { items: [...], lastId: 2987 }
```

Angular service

```
getTasks(page:number, size:number) {
  return this.http.get(`/api/tasks?page=${page}&size=${size}`);
}
```

Client UI

- Use server totalCount for pagination UI or "Load more" with keyset.
- Use lazy loading/infinite scroll for better UX, but keep accessible paging as option.

DB hints

- Use indexes for sort/filter columns, prefer keyset conditions (`WHERE (createdAt < @lastCreatedAt)`), and avoid large OFFSET on enormous tables.

■ One-line summary:

> Use server-side pagination (page+size or keyset cursor), apply DB-level filters/indexes, and let the client request/append pages to keep UI performant.

📌 13. Short Rapid-Fire Round

✅ 1) `"5" + 2` vs `"5" - 2` output?

■ Explanation

`+` performs **string concatenation** when one operand is a string.

- `-` forces **numeric conversion** because subtraction is not defined for strings.

■ When/Why

Used in type-coercion questions to test JS implicit conversions.

■ Example

```
"5" + 2   // "52"
"5" - 2   // 3
```

■ One-line summary:

> + concatenates strings, but - converts string to number.

✅ 2) `typeof null` ?

■ Explanation

`typeof null` returns `"object"` — this is a long-standing **bug** in JavaScript's initial design but kept for backward compatibility.

■ When/Why

Important to know in interviews because it's a classic JS quirk.

■ Example

```
typeof null; // "object"
```

■ One-line summary:

> typeof null is "object" due to a historical JS bug.

✅ 3) `[] == []` ?

■ Explanation

Each array is a separate **object reference**. `==` compares references, not values → two different arrays are never equal.

■ When/Why

Used to check understanding of reference comparison.

■ Example

```
[] == []  // false
```

■ **One-line summary:**

> Two arrays are never equal because object references differ.

✅ 4) `"2" + 2 + 2` vs `2 + 2 + "2"`

■ **Explanation**

Evaluation is left-to-right:

- First case: `"2" + 2` becomes `"22"`, then `"22" + 2` → `"222"`
- Second case: `2 + 2` becomes `4`, then `4 + "2"` → `"42"`

■ **When/Why**

Checks understanding of JS coercion & evaluation order.

■ **Example**

```
"2" + 2 + 2;  // "222"
2 + 2 + "2";  // "42"
```

■ **One-line summary:**

> String first → full concatenation; number first → math happens first.

✅ 5) NaN type?

■ **Explanation**

`NaN` stands for “Not a Number”, but its type is actually **number**.

■ **When/Why**

Used to check JS quirks; also important for handling invalid numeric inputs.

■ **Example**

```
typeof NaN;  // "number"
```

■ **One-line summary:**

> NaN means invalid number, but its type is "number".

✅ 6) Is JS single-threaded or multi-threaded?

■ **Explanation**

JavaScript is **single-threaded** — it has **one call stack**, executing one task at a time.

But the browser provides **background threads** via Web APIs (Timers, Fetch, DOM events).

■ **When/Why**

Important for async programming, event-loop understanding, and avoiding blocking UI.

■ **Example**

```
console.log("A");
setTimeout(()⇒console.log("B"),0);
console.log("C");
// A C B
```

■ **One-line summary:**

> JavaScript is single-threaded, but browser APIs run tasks in background threads.

✅ 7) How async code runs in single thread?

■ **Explanation**

JS delegates async tasks (timers, fetch, async I/O) to **Web APIs**, which run in background.

Once done, callbacks are queued and the **event loop** pushes them to the call stack when it's free.

■ **When/Why**

This mechanism achieves concurrency without multiple JavaScript threads.

■ **Example**

```
setTimeout(()⇒console.log("Done"), 0);
console.log("Next");
// Output: Next → Done
```

■ **One-line summary:**

> Async tasks run via Web APIs, event loop, and callback queues—keeping JS non-blocking despite single thread.

✅ 8) Optional chaining `?.` usage

■ **Explanation**

Optional chaining safely accesses nested properties without throwing errors if a value is `null` or `undefined`.

■ **When/Why**

Used when dealing with API responses, deep objects, or optional values to prevent "Cannot read property of undefined".

■ **Example**

```
let user = { profile: null };
console.log(user.profile?.email); // undefined (no error)
```

■ **One-line summary:**

> Optional chaining (?.) safely checks nested properties without crashing on null/undefined.
