**Most asked JavaScript interview questions**

**1. Difference Between**var**,**let**, and**const**?**

* var is **function-scoped** and can be re-declared.
* let is **block-scoped** and cannot be re-declared in the same scope.
* const is also **block-scoped**, but it **cannot be reassigned**.

function example() {  
 var a = 10;   
 let b = 20;   
 const c = 30;  
 b = 25; // Allowed  
 c = 35; // Error ❌  
}

**2. What is Hoisting?**

Hoisting means that JavaScript moves function and variable **declarations** to the top of their scope **before execution**.

console.log(a); // undefined (hoisted but not assigned)  
var a = 10;

* Functions are **fully hoisted**:

hello(); // Works  
function hello() {  
 console.log("Hello!");  
}

**3. What are Closures?**

A closure is when a function **remembers** variables from its parent scope even after the parent has finished execution.

function outer() {  
 let count = 0;  
 return function inner() {  
 count++;  
 console.log(count);  
 };  
}  
const counter = outer();  
counter(); // 1  
counter(); // 2

**4. Difference Between Shallow Copy & Deep Copy?**

* **Shallow copy** copies only references (not actual values).
* **Deep copy** creates a **new independent object**.

let obj1 = { a: 1, b: { c: 2 } };  
let shallow = { ...obj1 }; // Shallow copy  
shallow.b.c = 100; // Also changes obj1.b.c ❌  
  
let deep = JSON.parse(JSON.stringify(obj1)); // Deep copy  
deep.b.c = 200; // No effect on obj1 ✅

**5. Difference Between**==**and**===**?**

* == only checks **values** (allows type conversion).
* === checks **values and types** (strict comparison).

console.log(5 == "5"); // true (type conversion)  
console.log(5 === "5"); // false (different types)

**6. How Does the Event Loop Work?**

The **event loop** allows JavaScript to handle asynchronous tasks like **setTimeout**, **Promises**, and **fetch**.

* **Call Stack** → Runs functions synchronously.
* **Task Queue** → Holds asynchronous callbacks.
* **Microtask Queue** → Holds **Promise callbacks** (higher priority).

console.log("Start");  
setTimeout(() => console.log("Timeout"), 0);  
Promise.resolve().then(() => console.log("Promise"));  
console.log("End");  
  
  
// Output:  
// Start  
// End  
// Promise  
// Timeout

**7. What are Promises?**

A **Promise** is an object that represents a value that will be available **in the future**.

const myPromise = new Promise((resolve, reject) => {  
 setTimeout(() => resolve("Success!"), 1000);  
});  
myPromise.then(result => console.log(result)); // Success!

**8. What is**async/await**?**

* async functions always return a **Promise**.
* await waits for the **Promise** to resolve.

async function fetchData() {  
 let data = await fetch("https://jsonplaceholder.typicode.com/todos/1");  
 let json = await data.json();  
 console.log(json);  
}  
fetchData();

**9. What is Memoization?**

Memoization stores **previous results** to speed up repeated calculations.

function memoizedAdd() {  
 let cache = {};  
 return function (num) {  
 if (num in cache) return cache[num];  
 cache[num] = num + 10;  
 return cache[num];  
 };  
}  
const add10 = memoizedAdd();  
console.log(add10(5)); // 15 (calculates)  
console.log(add10(5)); // 15 (from cache)

**10. What are Higher-Order Functions?**

A **higher-order function** takes another function as a **parameter** or **returns a function**.

function multiplyBy(factor) {  
 return function (num) {  
 return num \* factor;  
 };  
}  
const double = multiplyBy(2);  
console.log(double(5)); // 10

**11. What is Prototypal Inheritance?**

JavaScript objects **inherit** properties from their **prototype**.

function Person(name) {  
 this.name = name;  
}  
Person.prototype.greet = function () {  
 console.log(`Hello, I am ${this.name}`);  
};  
const john = new Person("John");  
john.greet(); // Hello, I am John

**12. Difference Between**Object.create()**and Class Inheritance?**

* Object.create(proto) creates a new object **with a specific prototype**.
* Class inheritance uses extends.

let person = { greet() { console.log("Hello"); } };  
let student = Object.create(person);  
student.greet(); // Hello

**13. How Does**this**Work?**

this refers to **different objects** based on how a function is called.

const obj = {   
 name: "John",   
 greet() { console.log(this.name); }   
};  
obj.greet(); // John  
  
const greet = obj.greet;  
greet(); // undefined (in strict mode)

**14. Getters and Setters in JavaScript?**

let person = {  
 firstName: "John",  
 lastName: "Doe",  
 get fullName() { return `${this.firstName} ${this.lastName}`; }  
};  
console.log(person.fullName); // John Doe

**15. Difference Between**call()**,**apply()**, and**bind()**?**

* call() calls a function with arguments **separately**.
* apply() calls with arguments **as an array**.
* bind() returns a **new function**.

function greet(greeting) {  
 console.log(`${greeting}, ${this.name}`);  
}  
const user = { name: "Alice" };  
greet.call(user, "Hello"); // Hello, Alice  
greet.apply(user, ["Hi"]); // Hi, Alice  
const boundGreet = greet.bind(user);  
boundGreet("Hey"); // Hey, Alice

**16. What is Currying?**

Currying is when a function **takes multiple arguments one at a time** instead of all at once.

function add(a) {  
 return function (b) {  
 return a + b;  
 };  
}  
console.log(add(5)(3)); // 8

**17. What is Functional Programming?**

Functional programming is a way of writing code using **pure functions**, **higher-order functions**, and **immutability**.

const numbers = [1, 2, 3, 4];  
const doubled = numbers.map(num => num \* 2); // [2, 4, 6, 8]

**18. What are Pure Functions?**

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

function add(a, b) {  
 return a + b; // No external modification  
}

**19. What is Debouncing and Throttling?**

* **Debouncing** delays a function execution until after a set time **without repeated calls**.
* **Throttling** ensures a function runs **at most once per interval**.

function debounce(func, delay) {  
 let timer;  
 return function (...args) {  
 clearTimeout(timer);  
 timer = setTimeout(() => func(...args), delay);  
 };  
}

**20. What is Lazy Loading?**

Lazy loading **delays loading resources** until needed, improving performance.

const image = new Image();  
image.src = "large-image.jpg"; // Loads only when required

**21. What are Web Workers?**

Web Workers allow running **JavaScript in the background** without blocking the main thread.

const worker = new Worker("worker.js");  
worker.postMessage("Hello");

**22. How Does**setTimeout**and**setInterval**Work?**

* setTimeout(func, delay) runs **once after the delay**.
* setInterval(func, delay) runs **repeatedly at intervals**.

setTimeout(() => console.log("Hello after 2s"), 2000);  
setInterval(() => console.log("Repeating"), 1000);

**23. What are Microtasks and Macrotasks?**

* **Microtasks** (Promises, MutationObserver) run **before macrotasks**.
* **Macrotasks** (setTimeout, setInterval) run **after microtasks**.

setTimeout(() => console.log("Macrotask"), 0);  
Promise.resolve().then(() => console.log("Microtask"));  
  
// Output:  
// Microtask  
// Macrotask

**24. Difference Between**Promise.all()**and**Promise.race()**?**

* Promise.all() waits for **all promises** to resolve.
* Promise.race() returns **the first resolved/rejected promise**.

Promise.all([fetch(url1), fetch(url2)]).then(console.log);  
Promise.race([fetch(url1), fetch(url2)]).then(console.log);

**25. How to Handle Errors in**async/await**?**

Use **try…catch** to handle errors in async/await.

async function fetchData() {  
 try {  
 let response = await fetch("invalid-url");  
 let data = await response.json();  
 } catch (error) {  
 console.log("Error:", error);  
 }  
}

**26. How Does JavaScript Garbage Collection Work?**

JavaScript automatically removes **unused variables and objects** using **reference counting** and **mark-and-sweep algorithms**.

let obj = { name: "John" };  
obj = null; // Now eligible for garbage collection

**27. What are WeakMap and WeakSet?**

* WeakMap and WeakSet allow **garbage collection** of keys that are no longer used.

let weakMap = new WeakMap();  
let obj = {};  
weakMap.set(obj, "Hello");  
obj = null; // Automatically removed

**28. What are Generators and Iterators?**

* **Iterators** allow looping over collections (for...of).
* **Generators** use function\* to **pause and resume execution**.

function\* generator() {  
 yield "Hello";  
 yield "World";  
}  
const gen = generator();  
console.log(gen.next().value); // Hello  
console.log(gen.next().value); // World

**29. What are ES6 Modules?**

ES6 Modules allow **importing and exporting** functions between files.

// file1.js  
export function greet() {  
 return "Hello!";  
}  
  
// file2.js  
import { greet } from "./file1.js";  
console.log(greet()); // Hello!

**30. How to Prevent Memory Leaks in JavaScript?**

* **Avoid global variables**.
* **Use WeakMap for large objects**.
* **Remove event listeners when no longer needed**.

let element = document.getElementById("btn");  
function handleClick() {  
 console.log("Clicked");  
}  
element.addEventListener("click", handleClick);  
element.removeEventListener("click", handleClick); // Prevents memory leak