1. **What is JavaScript?**

**JavaScript (JS)** is a **high-level, dynamic, interpreted programming language** mainly used to make web pages **interactive and dynamic**.

1. **Template literals**

## 🔹 What are Template Literals?

Template literals are a modern way (introduced in **ES6/ES2015**) to work with **strings** in JavaScript.  
They are enclosed by **backticks** (`), instead of single quotes (') or double quotes (").

`Hello, ${name}!`

## what is hoisting in javascript?

Great question! Let’s break down **Hoisting in JavaScript** step by step 🚀

## 🔹 What is Hoisting?

Hoisting is JavaScript’s **default behavior of moving declarations to the top** of the current scope **before the code is executed**.

👉 In simple words:  
You can **use a variable or function before it is declared** because JavaScript “hoists” (lifts) the declaration part to the top of its scope.

## 🔹 Example 1: Variable Hoisting (with var)

console.log(a); // undefined

var a = 10;

👉 What happens behind the scenes:

var a; // declaration is hoisted to the top

console.log(a); // undefined (not assigned yet)

a = 10; // initialization stays in place

⚠️ With var, the **declaration** is hoisted, but the **value assignment is not**. That’s why we get undefined, not an error.

## 🔹 Example 2: Function Hoisting

sayHello(); // ✅ works

function sayHello() {

console.log("Hello World!");

}

👉 Behind the scenes:

function sayHello() { // entire function is hoisted

console.log("Hello World!");

}

sayHello();

✅ Function **declarations** are fully hoisted, so you can call them before defining.

## 🔹 Example 3: let and const (Temporal Dead Zone ⚡)

console.log(x); // ❌ ReferenceError

let x = 5;

console.log(y); // ❌ ReferenceError

const y = 10;

* let and const are **also hoisted**, but they are placed in a **Temporal Dead Zone (TDZ)** until the line of declaration is reached.
* You cannot access them before declaration → results in **ReferenceError**.

## 🔹 Summary

* **var** → Hoisted, initialized with undefined.
* **function declaration** → Fully hoisted (can call before defining).
* **let / const** → Hoisted, but in **TDZ**, so cannot be used before declaration.

1. Explain difffrence between var, let and const?

Perfect 👌 this is a classic JavaScript interview question. Let’s carefully compare **var, let, and const**:

## 🔹 1. Scope

* **var** → **Function-scoped** (visible inside the whole function where declared).
* **let & const** → **Block-scoped** (only visible inside { } block).

👉 Example:

if (true) {

var a = 10;

let b = 20;

const c = 30;

}

console.log(a); // ✅ 10 (function/global scoped)

console.log(b); // ❌ ReferenceError (block scoped)

console.log(c); // ❌ ReferenceError (block scoped)

## 🔹 2. Hoisting

* **var** → Hoisted and initialized with undefined.
* **let & const** → Hoisted but in **Temporal Dead Zone (TDZ)** → cannot access before declaration.

👉 Example:

console.log(x); // undefined

var x = 5;

console.log(y); // ❌ ReferenceError

let y = 10;

console.log(z); // ❌ ReferenceError

const z = 15;

## 🔹 3. Re-declaration

* **var** → Can be re-declared in the same scope (❌ risky).
* **let & const** → Cannot be re-declared in the same scope.

👉 Example:

var a = 10;

var a = 20; // ✅ Allowed

let b = 30;

// let b = 40; // ❌ Error

const c = 50;

// const c = 60; // ❌ Error

## 🔹 4. Re-assignment

* **var** → Can be reassigned.
* **let** → Can be reassigned.
* **const** → ❌ Cannot be reassigned (constant).

👉 Example:

var x = 1;

x = 2; // ✅ allowed

let y = 3;

y = 4; // ✅ allowed

const z = 5;

z = 6; // ❌ Error (assignment to constant variable)

⚠️ **Note**: For const objects/arrays, the **reference** cannot change, but the **contents** can:

const arr = [1, 2];

arr.push(3); // ✅ allowed

// arr = [4, 5]; // ❌ Error (can't reassign reference)

## 🔹 5. Global Object Property

* **var** → Declared globally, becomes a property of window (in browsers).
* **let & const** → Do NOT attach to global window.

👉 Example (in browser console):

var x = 10;

let y = 20;

console.log(window.x); // 10 ✅

console.log(window.y); // undefined ❌

## 🔹 Quick Comparison Table

| **Feature** | **var 🟡** | **let 🔵** | **const 🔴** |
| --- | --- | --- | --- |
| Scope | Function | Block | Block |
| Hoisting | Yes (undefined) | Yes (TDZ) | Yes (TDZ) |
| Re-declaration | ✅ Allowed | ❌ Not allowed | ❌ Not allowed |
| Re-assignment | ✅ Allowed | ✅ Allowed | ❌ Not allowed |
| Global window | ✅ Attaches | ❌ No | ❌ No |

✅ **Rule of thumb**:

* Use **const** by default.
* Use **let** if the value will change.
* Avoid **var** (old behavior, can cause bugs).

👉 Do you want me to also show **real-world bugs** caused by var (like inside loops with async calls)?

##### You said:

var is global scope that means in other functions also we can use ?

###### ChatGPT said:

Good catch 👍 Let’s break this down carefully because this is where many get confused.

## 🔹 var is ****function-scoped****, not truly global always.

* If you declare var **inside a function**, it belongs only to that function (NOT visible outside).
* If you declare var **outside any function**, it becomes **global** (and attaches to window in browsers).

### ✅ Example 1: var inside a function (NOT accessible outside)

function test() {

var x = 10; // function-scoped

console.log("Inside:", x);

}

test();

console.log("Outside:", x); // ❌ ReferenceError

### ✅ Example 2: var outside any function (Global)

var y = 20; // global variable

function demo() {

console.log("Inside function:", y); // ✅ 20 accessible

}

demo();

console.log("Outside function:", y); // ✅ 20 accessible

### ✅ Example 3: Global var is also a window property (in browser)

var z = 100;

console.log(window.z); // ✅ 100

## ⚠️ BUT Careful!

Even though var declared globally is accessible in other functions, it’s risky because:

* It can be **overwritten accidentally** (polluting global scope).

1. Multiple functions may change the same variable → hard-to-track bugs.

👉 That’s why **modern JS best practice** is:

* Use let or const inside blocks/functions.
* Avoid var unless maintaining old code.

🔑 **Answer to your question:**  
Yes, if you declare var globally (outside any function), you can use it inside **other functions too**.  
But if you declare var inside one function, it’s **not available in other functions** (because it’s function-scoped, not block-scoped).

Do you want me to show you with a **side-by-side comparison** of var, let, and const inside loops and functions (where scope difference is very clear)?

1. **How to iterate array in javascripyt?**

for (let i = 0; i < arr.length; i++) {

console.log(arr[i]); // prints 10, 20, 30, 40, 50

}

1. **Data types available javascript?**

Good one 👍 Let’s go step by step.

In **JavaScript**, data types are divided into two categories:

**🔹 1. Primitive Data Types (simple, immutable, stored by value)**

1. **Number** → for all numbers (integer, float, NaN, Infinity)
2. let x = 10; // integer
3. let y = 3.14; // float
4. let z = Infinity; // Infinity
5. let n = NaN; // Not-a-Number
6. **String** → text (inside single, double, or backticks)
7. let name = "Ramu";
8. let greeting = `Hello ${name}`; // template literal
9. **Boolean** → true or false
10. let isAdmin = true;
11. let isLoggedIn = false;
12. **Undefined** → declared but not assigned a value
13. let a;
14. console.log(a); // undefined
15. **Null** → intentional "empty" value
16. let b = null;
17. **Symbol** → unique identifier (ES6)
18. let s1 = Symbol("id");
19. let s2 = Symbol("id");
20. console.log(s1 === s2); // false (always unique)
21. **BigInt** → very large integers (beyond Number.MAX\_SAFE\_INTEGER)
22. let big = 123456789012345678901234567890n;

**🔹 2. Non-Primitive (Reference) Data Types**

These store **references (addresses in memory)**, not actual values.

1. **Object** → collection of key-value pairs
2. let person = { name: "Ramu", age: 25 };
3. **Array** → ordered list of values
4. let numbers = [1, 2, 3, 4];
5. **Function** → first-class object in JS

function greet() { console.log("Hello!"); }

1. **== and ===**

###### ChatGPT said:

Nice question 👍 This is one of the most common interview doubts in JavaScript.

## 🔹 == (Loose Equality)

* Compares **values only**, not data types.
* If types are different, it does **type coercion** (tries to convert before comparing).

Example:

console.log(5 == "5"); // true (string "5" converted to number 5)

console.log(0 == false); // true (false converted to 0)

console.log(null == undefined); // true (special rule)

## 🔹 === (Strict Equality)

* Compares **values + data types**.
* No type coercion (must be exactly the same).

Example:

console.log(5 === "5"); // false (number vs string)

console.log(0 === false); // false (number vs boolean)

console.log(null === undefined); // false

console.log(5 === 5); // true

1. What is the purpose of NAN

console.log(0 / 0); // NaN

console.log("abc" - 10); // NaN

console.log(undefined + 5); // NaN

console.log(Math.sqrt(-1)); // NaN

console.log(isNaN(NaN)); // true

console.log(isNaN(123)); // false

console.log(isNaN("hello")); // true (because "hello" -> NaN when converted)

console.log(isNaN("123")); // false ("123" -> 123 number)

1. **Undefined and null**

**undefined**

* Means **a variable has been declared but not assigned a value**.
* It is **set by JavaScript automatically**.
* Type: undefined

Example:

let a;

console.log(a); // undefined

**🔹 null**

* Means **"no value" or "empty value"**, and it is **explicitly assigned by the developer**.
* Type: object (this is a long-standing bug in JavaScript, but kept for backward compatibility).

Example:

let b = null;

console.log(b); // null

**typeOf()**

console.log(typeof 123); // "number"

console.log(typeof "Hello"); // "string"

console.log(typeof true); // "boolean"

console.log(typeof undefined); // "undefined"

console.log(typeof null); // "object" ❌ (weird bug in JS)

console.log(typeof {}); // "object"

console.log(typeof []); // "object" (arrays are objects in JS)

console.log(typeof function(){}); // "function"

**Medium Questions**

1. **Purpose of map method?**

const numbers=[1,2,3,4,5];

const double=numbers.map(x=>x\*2);

console.log(double);

1. **Event bubbling and event capturing in java?**

**Event Bubbling**

* **Default behavior** in JavaScript.
* When an event happens on an element, it first runs the handler on that element, **then bubbles up** to its parent, grandparent, and so on up to document.
* Example: Click on a <button> inside a <div> inside <body> → event goes:  
  button → div → body → document

<div id="parent">

<button id="child">Click Me</button>

</div>

<script>

document.getElementById("parent").addEventListener("click", () => {

console.log("Parent clicked");

});

document.getElementById("child").addEventListener("click", () => {

console.log("Child clicked");

});

</script>

👉 Output when button clicked:

Child clicked

Parent clicked // event bubbled up

**🔹 Event Capturing (a.k.a. Trickling)**

* Opposite of bubbling.
* Event is **captured from the outermost element to the target element**.
* Order:  
  document → body → div → button

document.getElementById("parent").addEventListener("click", () => {

console.log("Parent capturing");

}, true); // 👈 true enables capturing

document.getElementById("child").addEventListener("click", () => {

console.log("Child clicked");

});

👉 Output when button clicked:

Parent capturing

Child clicked

**🔹 Stopping Propagation**

You can stop the event from bubbling/capturing further with:

event.stopPropagation();

**✅ Summary**

* **Event Bubbling (default):** Target → Parent → Ancestors
* **Event Capturing:** Ancestors → Parent → Target
* Controlled with the **third parameter** in addEventListener:
  + false (default) → bubbling
  + true → capturing

**Bubbling**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

<div id="parent"> parent

    <div id="mid"> mid

        <div id="child"> child

        </div>

    </div>

</div>

<script>

document.getElementById("parent").addEventListener(("click"),()=>{

    console.log("parent clicked");

})

</script>

<script>

document.getElementById("mid").addEventListener(("click"),()=>{

    console.log("mid clicked");

})

</script>

<script>

document.getElementById("child").addEventListener(("click"),()=>{

    console.log("child clicked");

})

</script>

</body>

</html>

**For Capturing add only extra true**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

<div id="parent"> parent

    <div id="mid"> mid

        <div id="child"> child

        </div>

    </div>

</div>

<script>

document.getElementById("parent").addEventListener(("click"),()=>{

    console.log("parent clicked");

},true)

</script>

<script>

document.getElementById("mid").addEventListener(("click"),()=>{

    console.log("mid clicked");

},true)

</script>

<script>

document.getElementById("child").addEventListener(("click"),()=>{

    console.log("child clicked");

},true)

</script>

</body>

</html>

1. **Higher order functions**

### Definition

A **higher-order function** is a function that does **at least one** of these:

1. **Takes another function as an argument** (callback).
2. **Returns another function** as its result.

👉 In short: **Functions that work with other functions**.

### 🔹 Example 1: Passing a function as an argument

function greet(name) {

return `Hello, ${name}!`;

}

function processUserInput(callback) {

let name = "Ramu";

return callback(name); // passing function

}

console.log(processUserInput(greet));

// Output: Hello, Ramu!

Here, processUserInput is a higher-order function because it accepts another function (greet) as input.

### 🔹 Example 2: Returning a function

function multiplier(factor) {

return function (x) {

return x \* factor;

};

}

const double = multiplier(2);

console.log(double(5)); // 10

const triple = multiplier(3);

console.log(triple(5)); // 15

Here, multiplier returns another function → making it a higher-order function.

### 🔹 Example 3: Built-in Higher-Order Functions in JS

Many **array methods** are HOFs because they take callbacks:

const numbers = [1, 2, 3, 4, 5];

// map → transforms each element

const squares = numbers.map(n => n \* n);

console.log(squares); // [1, 4, 9, 16, 25]

// filter → keeps elements that match a condition

const evens = numbers.filter(n => n % 2 === 0);

console.log(evens); // [2, 4]

// reduce → accumulates values

const sum = numbers.reduce((acc, curr) => acc + curr, 0);

console.log(sum); // 15

✅ **Summary**:

* Higher-order function = a function that **takes functions as arguments or returns a function**.
* Examples: map, filter, reduce, event listeners, middleware functions, etc.

1. **IIFE**

### What is IIFE?

**IIFE** stands for **Immediately Invoked Function Expression**.

It’s a function that is:

1. **Defined**
2. **Immediately executed** (without calling it later).

### 🔹 Syntax of IIFE

(function () {

console.log("I am an IIFE!");

})();

👉 Notice:

* Function is wrapped in () → (function(){...})
* Then immediately executed with () → (...)()

### 🔹 Example 1: Basic IIFE

(function () {

let msg = "Hello from IIFE!";

console.log(msg);

})();

// Output: Hello from IIFE!

Here, the function executes right away, and variables inside are **private** (not polluting global scope).

### 🔹 Example 2: IIFE with parameters

(function (name) {

console.log("Hello " + name);

})("Ramu");

// Output: Hello Ramu

### 🔹 Example 3: Returning values

const result = (function (a, b) {

return a + b;

})(5, 10);

console.log(result); // 15

### 🔹 Why use IIFE?

1. **Avoid global scope pollution**
2. (function () {
3. let counter = 0;
4. console.log("Counter is private:", counter);
5. })();
6. // console.log(counter); // ❌ Error: counter is not defined

Keeps variables **private**.

1. **Execute code immediately**  
   Useful for initialization logic (like configs, setup code).
2. **Module pattern** (before ES6 modules)  
   Developers used IIFEs to create private scopes.

✅ **Summary:**

* **IIFE = Immediately Invoked Function Expression**.
* Runs immediately after definition.
* Helps with **privacy**, **encapsulation**, and **avoiding global scope pollution**.

1. **Closures?**

### ****Closures in JavaScript****

👉 A **closure** is created when a **function “remembers” the variables from its outer scope** even after that outer function has finished executing.

In simple terms:

* A function can "close over" (capture) the variables outside of it.
* That’s why it’s called a **closure**.

### ****Example 1: Basic Closure****

function outer() {

let count = 0; // variable inside outer function

function inner() {

count++;

console.log(count);

}

return inner; // returning the inner function

}

const closureFn = outer(); // outer() runs, returns inner

closureFn(); // 1

closureFn(); // 2

closureFn(); // 3

✅ Even though outer() has finished, the inner function **remembers** count and keeps updating it.

### ****Example 2: Real-Life Analogy****

Think of a closure like a backpack 🎒:

* When a function is created, it packs the variables around it into a “backpack”.
* Even if the outer function is gone, the inner function still carries its backpack.

### ****Example 3: Private Variables****

Closures let you create private data.

function createCounter() {

let count = 0; // private variable

return {

increment: () => ++count,

decrement: () => --count,

get: () => count

};

}

const counter = createCounter();

console.log(counter.increment()); // 1

console.log(counter.increment()); // 2

console.log(counter.decrement()); // 1

console.log(counter.get()); // 1

Here, nobody can directly access count — only through the returned functions.

### ****Key Points****

1. Closures happen **automatically** whenever you nest a function inside another.
2. They allow **data hiding** and **stateful functions**.
3. They are heavily used in:
   * Event handlers
   * Callbacks
   * setTimeout / setInterval
   * Functional programming
   * React hooks (useState, useEffect internally use closures)
4. **setTimeOut and setTimeInterval?**

function greet(name) {

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

}

setTimeout(greet, 3000, "Ramu"); // after 3 sec → "Hello, Ramu"

clearTimeout(timer)

setInterval(() => {

console.log("Repeating every 1 second");

}, 1000);

clearInterval(interval);

**Canceling an interval:**

let counter = 0;

let interval = setInterval(() => {

counter++;

console.log("Count: " + counter);

if (counter === 5) {

clearInterval(interval); // stop after 5 times

}

}, 1000);

1. **Promise**

const data=new Promise((resolve,reject)=>{

    let success= true;

    if(success){

        resolve("Data succesfully retrived");

    }else{

        reject("Data fetching failed");

    }

});

data.then(x=>{

    console.log(x)

    return x+"hellooooooo123";

}).then(x=>{

    console.log(x);

}).catch(err=>{

    console.error(err);

});

# ****What is a Promise?****

A **Promise** in JavaScript represents a **value that will be available in the future** (like a placeholder for an asynchronous result).

It is used to handle **asynchronous operations** such as:

* API calls (fetching data from server),
* File reading,
* setTimeout, etc.

## ****Promise States****

A Promise has **3 states**:

1. **Pending** → Initial state (operation still running).
2. **Fulfilled** → Operation completed successfully (resolve called).
3. **Rejected** → Operation failed (reject called).

## ****Creating a Promise****

const myPromise = new Promise((resolve, reject) => {

let success = true;

if (success) {

resolve("Operation successful ✅");

} else {

reject("Something went wrong ❌");

}

});

## ****Consuming a Promise****

You use .then() and .catch():

myPromise

.then(result => {

console.log(result); // "Operation successful ✅"

})

.catch(error => {

console.log(error); // "Something went wrong ❌"

})

.finally(() => {

console.log("Operation completed (success or fail)");

});

## ****Example with setTimeout****

function getData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

let data = { name: "Ramu", age: 25 };

resolve(data); // success

// reject("Data not found"); // failure

}, 2000);

});

}

getData()

.then(user => console.log("User:", user))

.catch(err => console.log("Error:", err));

## ****Promise Chaining****

You can chain multiple .then() calls:

new Promise(resolve => resolve(2))

.then(num => num \* 2)

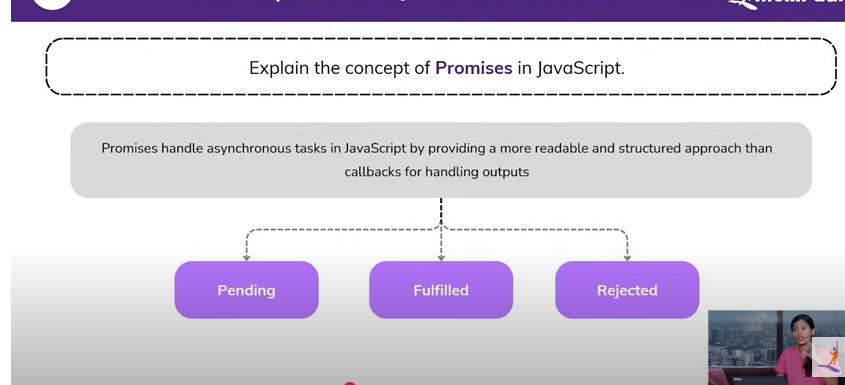
.then(num => num \* 3)

.then(result => console.log(result)); // 12

## ****Promise Methods****

* **Promise.all([p1, p2, p3])** → Waits for all promises to finish. (Fails if any rejects.)
* **Promise.race([p1, p2, p3])** → Returns the first promise that settles (fulfilled or rejected).
* **Promise.allSettled([p1, p2])** → Returns results of all promises (success or fail).
* **Promise.any([p1, p2])** → Returns the first fulfilled promise (ignores rejections).

✅ **In short:** A Promise is like saying  
"I promise to return you the result later. If everything goes well, you get a resolve. If not, you get a reject."



1. Async/Await

### ****What is**** async****?****

* When you declare a function with async, it **always returns a Promise**.
* Even if you return a normal value, it gets wrapped in a resolved promise.

### ****What is**** await****?****

* await can **only be used inside async functions**.
* It pauses execution until the Promise resolves, then returns the result.
* While waiting, JavaScript can continue executing other code (non-blocking).

async function fetchData() {

try {

let response = await fetch("https://jsonplaceholder.typicode.com/posts/1");

let data = await response.json();

console.log(data);

} catch (error) {

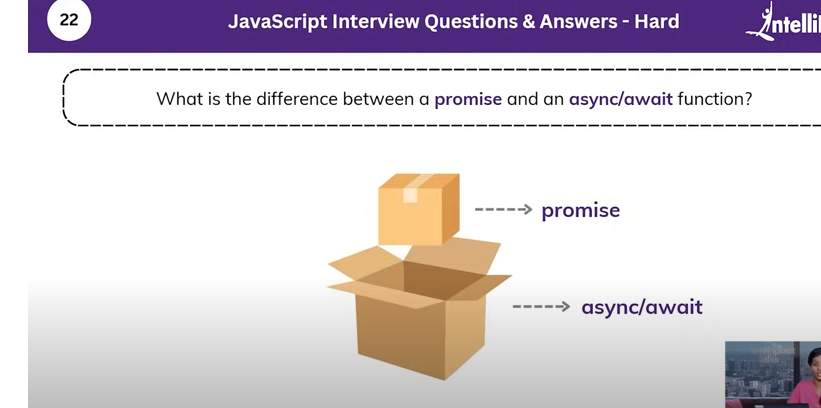
console.error("Error:", error);

}

}

fetchData();

1. **Call, apply and bind**
2. **Event Delegation**
3. **Event loop**
4. **Diffrence between aync/await and promise**



1. **Reduce**

**const numbers = [1, 2, 3, 4, 5];**

**const sum = numbers.reduce((acc, curr) => acc + curr, 0);**

**console.log(sum); // 15**

1. **Currying**

## Normal Function

function add(a, b, c) {

return a + b + c;

}

console.log(add(1, 2, 3)); // 6

## 🔹 Curried Function

function add(a) {

return function(b) {

return function(c) {

return a + b + c;

};

};

}

console.log(add(1)(2)(3)); // 6

1. **Generator function**
2. **Weak maps and weak sets**

**let weakMap = new WeakMap();**

**let obj = { name: "Ramu" };**

**weakMap.set(obj, "Some Value");**

**console.log(weakMap.get(obj)); // "Some Value"**

**obj = null; // remove reference**

**// After GC, the {name:"Ramu"} key will be removed automatically**

1. **Memory management**
2. **Shallow and deep copying**