js Interview prep

Explain this in JavaScript. How does it behave in arrow functions vs normal functions?

- `this` is a special keyword that refers to the execution context (the object that is calling the function).
- Its value depends on how the function is called, not where it is defined.

`this` in Normal Functions

In a regular function, `this` depends on the caller:

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

const obj = {
  name: "Suraj",
  sayHi: normalFunc,
};

normalFunc();   // In strict mode: undefined, otherwise: window (in
browser)
obj.sayHi();    // { name: "Suraj", sayHi: f } → obj is caller
```

Key Points:

- If called as a method → `this` = the object.
- If called standalone → `this` = `undefined` (strict mode) or `window` (non-strict).
- `this` changes depending on how the function is invoked.

🔹 `this` in Arrow Functions

Arrow functions don't have their own `this`.

- Instead, they inherit `this` from the surrounding scope (lexical binding).

```
const obj = {
  name: "Suraj",
  arrowFunc: () => {
    console.log(this);
  },
```

```
};
obj.arrowFunc();  // window / undefined (NOT obj!)
```

Here, `this` inside `arrowFunc` doesn't bind to `obj`. Instead, it looks one level up (the scope where `obj` was created).

Key Points:

- Arrow functions don't create a new `this`.
- `this` is determined by where the arrow function is defined, not called.
- Very useful in callbacks (e.g., inside `map`, `setTimeout`, React components).

What are closures? Can you give an example?

Closure is when a function remembers variables from its parent scope, even after the parent has finished execution.
It allows us to create private data and functions. For example, a counter function can keep its count variable hidden using closure

Simple Explanation of Hoisting

Hoisting in JavaScript means that variable and function declarations are moved ("hoisted") to the top of their scope (global or function scope) during the memory creation phase of execution.
But → how they are initialized is different depending on whether you use `var`, `let`, `const`, or a function.

How Each Works

1. Function Declarations
   - Fully hoisted.
   - You can call them before they are defined.
     `greet(); // ✅ works function greet() { console.log("Hello"); }`
2. `var` variables
   - Hoisted but initialized with `undefined`.
   - If you access them before assignment → you get `undefined` (not error).
     `console.log(a); // undefined var a = 10;`
3. `let` and `const` variables
   - Hoisted too, but placed in the Temporal Dead Zone (TDZ).
   - If you try to access them before assignment → you get a ReferenceError.
     `console.log(b); // ❌ ReferenceError let b = 20;`

Interview-Ready Summary

"In JavaScript, during the first phase (memory creation), variables and functions are hoisted.

- **Function declarations** are fully hoisted (you can use them before defining).
- `var` **variables** are hoisted but initialized to `undefined`.
- `let` and `const` are hoisted too but kept in a *Temporal Dead Zone* until their declaration line is executed, so accessing them early throws a ReferenceError.

What is the Temporal Dead Zone (TDZ)?

The **Temporal Dead Zone** is the time between when a variable is **hoisted** and when it is **actually initialized** in code.

- Variables declared with `let` and `const` are hoisted but not given a value (`undefined` like `var`).
- Instead, they stay in the **TDZ** until the line of code that declares them is executed.
- If you try to access them before that → you get a **ReferenceError**.
  Here's what's happening step by step:

1. `a` is hoisted (JavaScript knows it exists).
2. From the start of the scope until the `let a = 5` line → `a` is in **TDZ**.
3. When the code reaches that line, `a` is initialized, and TDZ ends.

Explain scope: block scope vs function scope vs global scope.

Scope in JavaScript defines where a variable can be accessed.

1. **Global Scope** → Variables declared outside any function or block are accessible everywhere in the program.
2. **Function Scope** → Variables declared with `var` inside a function are accessible only within that function.
3. **Block Scope** → Variables declared with `let` and `const` are accessible only within the block `{ }` where they are defined.

- `var` → function-scoped.
- `let` and `const` → block-scoped.
- Global variables are available throughout the entire code, but too many globals can cause conflicts.

What are higher-order functions?

Higher-order functions in JavaScript are functions that can do either (or both):

1. **Take another function as an argument (callback)**
2. **Return a function as their result**

This makes them powerful for abstraction, reusability, and functional programming patterns.
Examples include built-in methods like `map`, `filter`, and `reduce`.

Synchronous JavaScript

- Code runs **line by line** in a single thread.
- Each task must **finish completely** before the next one starts.
- If a task takes too long (e.g., a big loop or heavy computation), it **blocks the main thread**, and nothing else can run in the meantime.

Asynchronous JavaScript

- Code does **not wait** for a task to finish.
- Long-running tasks (like API calls, file reads, setTimeout) are **delegated to the browser or Node.js APIs**.
- Meanwhile, the main thread keeps running other code.
- When the async task is done, the result is handled later using **callbacks, promises, or async/await**.
- This makes apps **non-blocking and faster** for I/O operations.

Event Loop in JavaScript

- JavaScript is **single-threaded** (one call stack).
- To handle **asynchronous tasks** (timers, promises, I/O), JavaScript uses the **event loop mechanism**, built on **libuv** (in Node.js).
- The event loop's job is to **continuously check**:
  1. **Call Stack** → Is it empty? If yes, it can take the next task.
  2. **Task Queues** → It looks at callback queues (macrotasks like setTimeout, I/O) and microtask queues (promises, `process.nextTick`).
- When an async task finishes, its **callback is queued**. The event loop then pushes it back into the call stack when the stack is free.
- This makes JS **non-blocking**, even though it runs in a single thread.

Debouncing

- **What**: Groups a series of rapid events and makes sure the function runs **only once after the user stops triggering it** for a certain delay.
- **Use case**:
  - Search box auto-suggestions (wait until user stops typing).
  - Window resize event (execute only after resize is finished).
- **Analogy**: Think of ringing a doorbell—debouncing means you wait until the person stops pressing it, then ring **once**.

Throttling

- **What**: Ensures a function runs at **fixed intervals**, no matter how many times the event is triggered.
- **Use case**:

    - Scroll event (load more data only every 200ms).
    - Button spam prevention (submit only once per second).
- **Analogy**: Like a speed governor on a car—it allows acceleration, but only up to a fixed limit.

IIFE (Immediately Invoked Function Expression)**

- **What**: A function that runs **immediately after it is defined**.
- **Why**: Used to create a private scope (avoid polluting global scope).
- **How**: Wrap in parentheses and call it right away.
  👉 **Interview line**:
  An IIFE is a function in JavaScript that executes as soon as it's defined, mainly used to create private variables and avoid global scope pollution.

Generators

- **What**: Special functions that can **pause** and **resume** execution.
- **How**: Declared with `function*` and use `yield`.
- **Why**: Useful for lazy evaluation, async tasks, and managing state step by step.
  Interview line:
  Generators are functions in JS that can pause with `yield` and later resume, allowing controlled execution and better handling of sequences or async flows.

Currying

Currying is a technique where a function with multiple parameters is transformed into multiple functions that take one parameter each, improving reusability.

Ways to merge two objects in JavaScript:

1. **Spread operator (ES6+)**
   `const merged = { ...obj1, ...obj2 };`
   👉 Simple, clean, and commonly used. If both objects have the same key, `obj2` overwrites `obj1`.
2. **Object.assign()**
   `const merged = Object.assign({}, obj1, obj2);`
   👉 Works the same as spread. You pass an empty object `{}` as the target to avoid mutating `obj1`.
3. **Manual merge with loop** (rare, but shows understanding)

```
const merged = {};
 for (let key in obj1) merged[key] = obj1[key];
 for (let key in obj2) merged[key] = obj2[key];
```

How to Clone an Object in JS

Cloning means creating a copy of an object so that changes in the new object don't affect the original.
Ways to clone (shallow copy):

1. Spread operator: `{ ...obj }`
2. Object.assign: `Object.assign({}, obj)`
   Deep copy (for nested objects):

- `JSON.parse(JSON.stringify(obj))`
- Or using structured cloning (`structuredClone(obj)` in modern JS)

Difference between map, forEach, filter, and reduce

Method	Purpose / Returns	Mutates Original?	When to Use
forEach	Loops through array, executes a callback, returns undefined	×	When you want to perform side effects, like logging or updating external state
map	Loops through array, returns a new array with transformed values	×	When you want a new array based on old array values
filter	Loops through array, returns new array with elements that pass a condition	×	When you want to select elements that meet certain criteria
reduce	Loops through array, returns a single value (accumulator)	×	When you want to calculate a sum, product, or transform array to single value/object

`forEach` is for side effects, `map` creates a new array with transformed values, `filter` creates a new array with selected elements, and `reduce` reduces the array to a single value.

Using `Set` (modern & simplest)

- A `Set` automatically removes duplicates.

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

2. Using `filter`

- Keep only the first occurrence of each element.

```
const arr = [1, 2, 2, 3, 4, 4]; const unique = arr.filter((item, index) =>
arr.indexOf(item) === index); console.log(unique); // [1, 2, 3, 4]
```

To remove duplicates from an array, the simplest way is using a Set. Other ways include using `filter` with `indexOf` or using `reduce` to accumulate unique values.

1. Flatten a Nested Array

Definition: Flattening means converting a nested array (array of arrays) into a single-level array.
Ways to do it:

1. **`Array.prototype.flat()`** (ES2019+)

```
const arr = [1, [2, 3], [4, [5]]]; const flatArr = arr.flat(2); // [1, 2,
3, 4, 5]
```

2. `Destructuring in JS`
   Definition: Destructuring allows you to **extract values from arrays or objects into variables** in a concise way.
   ```
   const arr = [1, 2, 3];
   const [a, b] = arr; // a=1, b=2
   ```

Promises

Definition:
A **Promise** in JavaScript represents the **future result of an asynchronous operation**.

- It can be in **three states**:
  1. **Pending** → operation not finished
  2. **Fulfilled** → operation succeeded, has a value
  3. **Rejected** → operation failed, has a reason/error

Interview line:
A promise is an object that represents the eventual completion or failure of an async operation, allowing us to handle results or errors with `.then()` and `.catch()`.

Difference between Promise.all, Promise.any, Promise.allSettled, and Promise.race

Method	Waits for	Returns / Resolves To	Rejects When?
Promise.all	All promises	Array of results	Any promise rejects

Method	Waits for	Returns / Resolves To	Rejects When?
Promise.any	First fulfilled promise	Value of the first fulfilled promise	All promises reject
Promise.allSettled	All promises	Array of objects `{status, value/reason}`	Never rejects
Promise.race	First settled promise	Value or reason of the first settled promise	First promise settles (either resolved or rejected)

Interview line:

- `Promise.all` *waits for all to succeed, fails fast if any reject.*
- `Promise.any` *waits for the first success, fails only if all reject.*
- `Promise.allSettled` *waits for all to settle, never fails.*
- `Promise.race` *resolves/rejects as soon as the first promise settles.*

1. Async/Await

- **Async/Await** is syntactic sugar over **Promises** that makes asynchronous code **look and behave like synchronous code**.
- `async` → marks a function as asynchronous and makes it **return a promise**.
- `await` → pauses the execution inside the async function until the promise resolves.
  Interview line:
  Async/await lets us write asynchronous code in a readable, synchronous style. The `async` *keyword returns a promise, and* `await` *pauses execution until the promise resolves.*

What happens if you don't use `await` inside an async function?

- The promise **will still execute**, but the function will **not wait** for it to resolve.
- You will get a **pending promise**, and subsequent code may run **before the async task completes**.
  Interview line:
  If you don't use `await`*, the async function continues executing without waiting for the promise, potentially causing race conditions or unexpected behavior.*