

JS

Frontend · most asked

JavaScript interview questions

25 questions · 12 easy · 11 medium · 2 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What is the difference between `var`, `let` and `const`?

Easy



#scope #hoisting

Scope: `var` is function-scoped; `let` and `const` are block-scoped (inside `{}`).

Hoisting: all three are hoisted, but `var` is initialised to `undefined`, while `let` / `const` stay in the `temporal dead zone` until their line runs, so using them earlier throws a `ReferenceError`.

Re-assignment: `const` cannot be re-assigned, but the object it points to can still be mutated.

- Default to `const`, use `let` when the value changes, avoid `var`.

Example

```
const user = { name: 'Asha' }
user.name = 'Ben'           // allowed: mutating, not re-assigning
console.log(user.name)      // Ben

try { console.log(x) } catch (e) { console.log(e.name) } //
ReferenceError (TDZ)
let x = 1
console.log(typeof y)       // undefined: var y is hoisted
var y = 2
```

Likely follow-up: Why does `for (var i...) setTimeout(() => log(i))` print 3,3,3?

2

What is a closure? Give a real use.

Easy



#closures #scope

A closure is a function bundled with the variables of the scope where it was **created**. It can still read and update them after that outer function has returned.

Real uses:

- private state (counters, caches) without classes
- function factories such as `makeMultiplier(2)`
- `once`, `memoize`, `debounce`, which all keep state between calls
- event handlers that remember data from when they were attached

Example

```
function counter() {  
  let count = 0 // private: only the returned  
  functions see it  
  return { inc: () => ++count, get: () => count }  
}  
const c = counter()  
c.inc(); c.inc()  
console.log(c.get()) // 2  
console.log(c.count) // undefined
```

Likely follow-up: Can closures cause memory leaks?

3

Explain the event loop. What prints first: a `setTimeout(..., 0)` or a resolved promise?

Medium



#event loop #async

JavaScript runs on one thread with a **call stack**. Async work (timers, I/O, fetch) is handed to the browser or Node, and its callbacks wait in queues.

The loop: run all synchronous code → empty the **microtask** queue (promise callbacks, `queueMicrotask`, `await` continuations) → take **one** macrotask (timer, I/O, event) → microtasks again → render (in the browser) → repeat.

So the promise callback runs before `setTimeout(..., 0)`, even though both are "async".

Example

```
console.log('1 sync')
setTimeout(() => console.log('4 timeout (macrotask)'), 0)
Promise.resolve().then(() => console.log('3 promise (microtask)'))
console.log('2 sync')
```

? Likely follow-up: What happens if a microtask keeps queueing microtasks?

4

How is `this` decided in JavaScript?

Medium 🔥🔥 #this #functions

`this` depends on **how a function is called**, not where it is written:

- `obj.method()`: `this` is `obj`
- a plain call `fn()`: `undefined` in strict mode (the global object otherwise)
- `new Fn()`: the newly created object
- `fn.call(x)` / `apply` / `bind`: whatever you pass
- **arrow functions** have no own `this`; they use the one from the surrounding scope

A classic bug is passing `obj.method` as a callback, which loses `this`. Fix it with `bind` or an arrow function.

Example

```
'use strict'
const user = {
  name: 'Asha',
  hi() { return 'Hi ' + (this ? this.name : 'undefined this') },
}
const hi = user.hi
console.log(user.hi())           // Hi Asha
console.log(hi())                // lost: plain call
console.log(hi.call({ name: 'Ben' }))
console.log(user.hi.bind(user)())
```

5

Difference between `call`, `apply` and `bind`?

Easy



#this #functions

All three set `this` explicitly:

- `fn.call(thisArg, a, b)`: calls **now**, arguments one by one
- `fn.apply(thisArg, [a, b])`: calls **now**, arguments as an array
- `fn.bind(thisArg, a)`: does **not** call; returns a new function with `this` (and optionally the first arguments) fixed

`bind` is what you use for callbacks and event handlers.

Example

```
function greet(greeting, mark) { return `${greeting},  
${this.name}${mark}` }  
const asha = { name: 'Asha' }  
console.log(greet.call(asha, 'Hi', '!'))  
console.log(greet.apply(asha, ['Hello', '?']))  
const later = greet.bind(asha, 'Hey')  
console.log(later('.'))
```

? Likely follow-up: Implement your own `bind`.

6

`==` vs `===`?

Easy



#types #coercion

`===` (strict equality) compares value **and** type with no conversion. `==` converts types first, with surprising rules (`'' == 0` and `null == undefined` are true).

Always use `===`. The one common exception is `x == null`, which checks for both `null` and `undefined`.

Objects are compared **by reference** with both operators: two different objects with the same contents are not equal.

Example

```
console.log(1 == '1', 1 === '1') // true false
console.log('' == 0, null == undefined) // true true
console.log(NaN === NaN, Object.is(NaN, NaN)) // false true
console.log({} === {}) // false: different references
```

7

What is the difference between `null` and `undefined`?

Easy



#types

`undefined` means a value was **never assigned**: an uninitialised variable, a missing property or argument, or a function with no `return`.

`null` is an **intentional "no value"** that you assign yourself.

`typeof null` is `'object'`, a historic bug. `??` treats both as missing.

8

What is hoisting?

Easy



#hoisting #scope

Before code runs, the engine sets up each scope. Declarations are "moved to the top":

- **function declarations** are hoisted completely, so you can call them before their line
- `var` is hoisted and initialised to `undefined`
- `let`, `const` and `class` are hoisted but stay uninitialised (temporal dead zone)
- **function expressions and arrow functions** follow the rules of the variable that holds them

Example

```
console.log(add(2, 3)) // 5: declaration fully hoisted
function add(a, b) { return a + b }

try { mul(2, 3) } catch (e) { console.log(e.name) } // ReferenceError
(const in TDZ)
const mul = (a, b) => a * b
```

9

How does prototypal inheritance work?

Medium



#prototypes #oop

Every object has a hidden link, `[[Prototype]]`, to another object. When a property is not found on the object, JavaScript looks it up along this `prototype chain` until it reaches `null`.

`class` is syntax on top of this: methods live on `Class.prototype`, and `extends` links one prototype to another.

Example

```
const animal = { speak() { return this.name + ' makes a sound' } }
const dog = Object.create(animal) // dog -> animal -> Object.prototype
                                   -> null
dog.name = 'Rex'
console.log(dog.speak())
console.log(Object.getPrototypeOf(dog) === animal)
console.log(dog.hasOwnProperty('speak')) // false: found on the chain
```

10

What are promises, and how does `async` / `await` relate to them?

Easy



#async #promises

A **promise** stands for a value that will be ready later. It is `pending`, then either fulfilled or rejected, and you react with `.then` / `.catch` / `.finally`.

`async` functions always return a promise. `await` pauses the function (not the thread) until the promise settles, so async code reads top to bottom. Errors are handled with `try/catch`.

Example

```
const wait = (ms, v) => new Promise((resolve) => setTimeout(() =>
  resolve(v), ms))

async function main() {
  try {
    const a = await wait(50, 'first')
    console.log(a)
    await Promise.reject(new Error('boom'))
  } catch (e) {
    console.log('caught', e.message)
  }
}
main()
```

🔗 Likely follow-up: How do you run three requests in parallel and wait for all of them?

11

Promise.all vs **allSettled** vs **race** vs **any**?

Medium 🔥🔥 #async #promises

- **Promise.all**: resolves when **all** succeed and rejects on the **first** failure (fail-fast)
- **Promise.allSettled**: waits for all of them and never rejects; each result is **{status, value|reason}**
- **Promise.race**: settles with whichever settles **first**, success or failure (useful for timeouts)
- **Promise.any**: resolves with the first **success**; rejects only if all fail

Example

```
const ok = (v, ms) => new Promise((r) => setTimeout(() => r(v), ms))
const bad = (m, ms) => new Promise((_, j) => setTimeout(() => j(new
Error(m)), ms))

async function main() {
  console.log(await Promise.all([ok('a', 20), ok('b', 10)]))
  console.log((await Promise.allSettled([ok('a', 5), bad('x',
5)]))?.map((r) => r.status))
  console.log(await Promise.race([ok('slow', 50), ok('fast', 5)]))
  console.log(await Promise.any([bad('x', 5), ok('first ok', 20)]))
}
main()
```

12

Debounce vs throttle. Implement debounce.

Medium 🔥🔥 #closures #performance #coding

- **Debounce**: run only after the calls have **stopped** for **ms** (search box, window resize end)
- **Throttle**: run at most **once every** **ms** while the calls keep coming (scroll, mousemove)

Both are closures that keep a timer or a timestamp between calls.

Example

```
function debounce(fn, ms) {
  let timer
  return (...args) => {
    clearTimeout(timer)
    timer = setTimeout(() => fn(...args), ms)
  }
}

function throttle(fn, ms) {
  let last = 0
  return (...args) => {
    const now = Date.now()
    if (now - last >= ms) { last = now; fn(...args) }
  }
}

const search = debounce((q) => console.log('search for', q), 50)
search('r'); search('re'); search('rea'); search('react') // only "react"
runs
```

13

Shallow copy vs deep copy? How do you deep-copy an object?

Easy



#objects

A **shallow copy** (`{...obj}`, `Object.assign`, `arr.slice()`) copies only the top level. Nested objects are still **shared**.

A **deep copy** duplicates every level. Use `structuredClone(obj)`, which handles Dates, Maps, Sets and cycles but not functions or DOM nodes.

`JSON.parse(JSON.stringify(x))` loses Dates, `undefined`, functions and Maps.

Example

```
const user = { name: 'Asha', address: { city: 'Pune' } }
const shallow = { ...user }
const deep = structuredClone(user)
user.address.city = 'Delhi'
console.log(shallow.address.city) // Delhi: shared
console.log(deep.address.city)    // Pune
```

14

Which array methods mutate the array, and `map` vs `forEach`?

Easy



#arrays

Mutating: `push`, `pop`, `shift`, `unshift`, `splice`, `sort`, `reverse`, `fill`.**Non-mutating:** `map`, `filter`, `reduce`, `slice`, `concat`, plus `toSorted`, `toReversed` and `with` (ES2023).`map` returns a new array of the results. `forEach` returns `undefined` and is used only for side effects. Neither can be stopped early; use `for...of` with `break`, or `some` / `find`.**Example**

```
const nums = [3, 1, 2]
const sorted = nums.toSorted((a, b) => a - b)
console.log(nums, sorted)           // original kept
console.log(nums.map(n => n * 2))
console.log(nums.reduce((sum, n) => sum + n, 0))
console.log([10, 1, 2].sort())      // [1, 10, 2]: default sort
                                     compares strings!
```

15

What is event delegation? What are bubbling and capturing?

Medium



#dom #events

DOM events go **down** from `window` to the target (capturing), then back **up** (bubbling). Listeners run in the bubbling phase unless you pass `{capture: true}`.**Event delegation:** attach one listener to a parent and check `event.target` (often `target.closest('li')`). You need fewer listeners, and it also works for children added later.`stopPropagation()` stops the event moving further; `preventDefault()` cancels the browser's default action, such as following a link.

16

Spread vs rest, and destructuring?

Easy



#syntax #es6

The same `...` syntax does opposite things:

- **spread** expands: `[...a, ...b]`, `{...obj, x: 1}`, `fn(...args)`
- **rest** collects: `function f(first, ...others)`, `const {id, ...rest} = obj`

Destructuring unpacks values: `const {name, age = 18} = user`, `const [first, , third] = arr`. You can also rename: `{name: userName}`.

Example

```
const user = { id: 7, name: 'Asha', role: 'admin' }
const { id, ...rest } = user
console.log(id, rest)
const [first, ...others] = [1, 2, 3]
console.log(first, others)
console.log(Math.max(...[4, 9, 2]))
```

17

How are arrow functions different from regular functions?

Easy



#functions #this

- no own `this`: they use `this` from the surrounding scope, which is great for callbacks
- no `arguments` object; use rest parameters
- cannot be called with `new` and have no `prototype`
- shorter syntax with an implicit return: `x => x * 2`

Don't use them as **object methods** that need `this`, or as constructors.

18

What is currying? Implement `curry(fn)`.

Medium



#functions #closures #coding

Currying turns `f(a, b, c)` into `f(a)(b)(c)`: each call takes some arguments and returns a function waiting for the rest. It is useful for building specialised functions through partial application.

A general `curry` keeps collecting arguments until it has `fn.length` of them.

Example

```
function curry(fn) {
  return function curried(...args) {
    return args.length >= fn.length ? fn(...args) : (...more) =>
    curried(...args, ...more)
  }
}

const volume = curry((l, w, h) => l * w * h)
console.log(volume(2)(3)(4), volume(2, 3)(4), volume(2, 3, 4))
```

19

Write a polyfill for `Array.prototype.map` (or `filter` / `reduce`).

Medium 🔥🔥 #coding #arrays #prototypes

Interviewers check that you know the callback receives `(element, index, array)`, that `this` is the array, and that a new array is returned.

For `reduce`, remember the case with **no initial value**: start from the first element. Calling it on an empty array without one must throw.

Example

```
Array.prototype.myMap = function (cb, thisArg) {
  const out = []
  for (let i = 0; i < this.length; i++) {
    if (i in this) out[i] = cb.call(thisArg, this[i], i, this) // skip
    holes like the real map
  }
  return out
}

Array.prototype.myReduce = function (cb, init) {
  let i = 0
  let acc = init
  if (arguments.length < 2) {
    if (this.length === 0) throw new TypeError('Reduce of empty array
    with no initial value')
    acc = this[i++]
  }
  for (; i < this.length; i++) acc = cb(acc, this[i], i, this)
  return acc
}

console.log([1, 2, 3].myMap((n, i) => n * i))
console.log([1, 2, 3].myReduce((a, b) => a + b))
```

20

What causes memory leaks in JavaScript apps?

Hard



#performance #memory

The garbage collector frees anything that can no longer be **reached**. A leak is memory you no longer need but still reference:

- event listeners and **setInterval** that are never removed (common in SPAs: clean them up on unmount)
- detached DOM nodes still held in variables
- caches, Maps or global arrays that grow forever (use **WeakMap** for per-object data)
- closures keeping large objects alive

Find them with DevTools → Memory → heap snapshots.

21

ES modules vs CommonJS?

Medium



#modules #node

ES modules (`import` / `export`) are the standard. They are loaded statically, so bundlers can remove unused code (`tree-shaking`). They are always in strict mode, can use top-level `await`, and their exports are live bindings.

CommonJS (`require` / `module.exports`) is Node's older system. It loads synchronously at runtime and exports a copy of the value.

In Node, `.mjs` or `"type": "module"` switches a file to ESM.

22

Map vs plain object, and Set? What is a WeakMap?

Medium



#data structures

A `Map` accepts **any** key type (objects too), keeps insertion order, has `.size`, and has no inherited keys like `toString`. It is faster for frequent adds and deletes. A `Set` stores unique values and is a quick way to de-duplicate: `[...new Set(arr)]`.

A `WeakMap` takes only objects as keys and holds them **weakly**: when the key object is garbage-collected, its entry disappears. Use it for metadata attached to objects without leaking memory.

Example

```
const visits = new Map()
const user = { id: 1 }
visits.set(user, 3)
console.log(visits.get(user), visits.size)
console.log([...new Set([1, 2, 2, 3, 3])])
```

23

What are generators and iterators?

Hard



#iterators

An **iterator** is any object with `next()` that returns `{value, done}`. Objects that have `[Symbol.iterator]` work with `for...of` and spread.

A **generator** (`function*`) is an easy way to write one: each `yield` pauses the function, and the next `next()` resumes it. Generators are lazy, so infinite sequences are fine.

Example

```
function* ids() {
  let id = 1
  while (true) yield id++
}

const gen = ids()
console.log(gen.next().value, gen.next().value, gen.next().value)

function* take(it, n) { for (const x of it) { if (n-- <= 0) return; yield x } }
console.log([...take(ids(), 5)])
```

24

What do `?.` and `??` do, and how is `??` different from `||`?

Easy



#syntax

`a?.b?.c` returns `undefined` instead of throwing when something in the chain is `null` or `undefined`. It also works for calls (`fn?.()`) and indexes (`arr?.[0]`).

`a ?? b` gives `b` only when `a` is `null` or `undefined`. `a || b` falls back on **any** falsy value, so `0 || 5` is `5` but `0 ?? 5` is `0`. Use `??` for defaults where `0` or `''` are valid values.

Example

```
const config = { retries: 0, user: null }
console.log(config.retries || 3, config.retries ?? 3) // 3 0
console.log(config.user?.name)                       // undefined, no
crash
console.log(config.user?.name ?? 'guest')
```

25

Output question: what does this print, and why?

Medium



#event loop #scope #quiz

This is a classic mix of hoisting, the event loop and loop scope:

1. `var` loop + `setTimeout` prints `3 3 3`: one shared `i`, and the timers run after the loop
2. `let` loop prints `0 1 2`: each iteration gets its own `j`
3. The synchronous log runs first, then microtasks, then timers

Interviewers want the **reasoning** (one variable vs one per iteration, and the queue order), not just the answer.

Example

```
for (var i = 0; i < 3; i++) setTimeout(() => console.log('var', i), 0)
for (let j = 0; j < 3; j++) setTimeout(() => console.log('let', j), 0)
Promise.resolve().then(() => console.log('microtask'))
console.log('sync')
```