

JavaScript

Scope, closures, this, prototypes, the event loop and async

1 Types, equality and copying

Definition

JavaScript is **dynamically typed** and single-threaded. 7 primitives: **string**, **number**, **bigint**, **boolean**, **undefined**, **null**, **symbol** – everything else is an **object** (arrays and functions too).

Equality, typeof and copies

```
console.log(1 == '1', 1 === '1') // true false: == converts types
console.log(typeof null)          // "object" (a historic bug)
console.log(typeof [], Array.isArray([]))
console.log(0.1 + 0.2 === 0.3)    // false: floating point
console.log(NaN === NaN, Number.isNaN(NaN))

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

- ✎ Primitives are copied by **value**; objects by **reference**.
- ✎ Falsy values: **false**, **0**, **-0**, **0n**, **""**, **null**, **undefined**, **NaN** – everything else is **truthy**.
- ✎ **??** falls back only on **null** / **undefined**; **||** on any falsy value (so **0 || 5** is 5 but **0 ?? 5** is 0).

2

var, let, const and hoisting

	Scope	Hoisted?	Re-assign?
var	function	yes, as undefined	yes
let	block	yes, but in the TDZ	yes
const	block	yes, but in the TDZ	no (objects stay mutable)

The classic loop bug

```
const a = [], b = []
for (var i = 0; i < 3; i++) a.push(() => i) // one shared i
for (let j = 0; j < 3; j++) b.push(() => j) // a new j each iteration
console.log(a.map((f) => f()), b.map((f) => f())) // [3,3,3] [0,1,2]
```

Common mistake

Using a `let` / `const` before its line throws a `ReferenceError` (temporal dead zone). Function declarations are fully hoisted; function expressions are not.

Definition

A **closure** is a function that remembers the variables of the scope it was created in, even after that scope has returned.

Private state, once() and memoize()

```
function counter() {
  let count = 0 // private
  return { inc: () => ++count, get: () => count }
}
const c = counter(); c.inc(); c.inc()
console.log(c.get()) // 2

function once(fn) {
  let done = false, result
  return (...args) => (done ? result : ((done = true), (result =
fn(...args))))
}
const init = once(() => { console.log('init runs'); return 42 })
console.log(init(), init())

function memoize(fn) {
  const cache = new Map()
  return (n) => cache.has(n) ? cache.get(n) : (cache.set(n, fn(n)),
cache.get(n))
}
const slowSquare = (n) => n * n
console.log(memoize(slowSquare)(12))
```

4

this, call, apply and bind

How the function is called	this is
<code>obj.method()</code>	<code>obj</code>
<code>fn()</code> (strict mode)	<code>undefined</code>
<code>new Fn()</code>	the new object
<code>fn.call(x)</code> / <code>apply(x)</code> / <code>bind(x)</code>	<code>x</code>
arrow function	inherited from the surrounding scope

Losing this – and fixing it

```
const user = {
  name: 'Ben',
  hi() { return 'Hi ' + this?.name },
  later() { return [1].map(() => this.name) }, // arrow keeps this
}
const hi = user.hi
console.log(hi()) // this is lost: undefined in strict mode,
// else the global object
console.log(hi.call({ name: 'Chen' }))
console.log(user.hi.bind(user)())
console.log(user.later())
```

5

Prototypes and classes

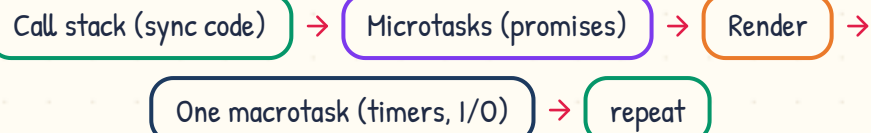
Every object has a hidden link (`[[Prototype]]`) to another object. Property lookup walks this **prototype chain**. `class` is syntax on top of prototypes.

class and the chain

```
class Animal {  
  constructor(name) { this.name = name }  
  speak() { return this.name + ' makes a sound' }  
}  
class Dog extends Animal {  
  speak() { return super.speak() + ' - woof' }  
}  
const d = new Dog('Rex')  
console.log(d.speak())  
console.log(Object.getPrototypeOf(d) === Dog.prototype) // true  
console.log(Object.getPrototypeOf(Dog.prototype) === Animal.prototype)
```

6

The event loop



After each task, ALL microtasks run before the next macrotask

Predict the order

```
console.log('1 sync')
setTimeout(() => console.log('5 timeout (macrotask)'), 0)
Promise.resolve().then(() => console.log('3 promise (microtask)'))
queueMicrotask(() => console.log('4 microtask'))
console.log('2 sync')
```

Interview answer: "sync first, then all microtasks, then one macrotask". `await` splits a function — the rest runs as a microtask.

Promises and async/await

Sequential vs parallel

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

async function main() {
  let t = Date.now()
  const a = await wait(100, 'a')
  const b = await wait(100, 'b') // waits for a first
  console.log('sequential', a, b, Date.now() - t, 'ms')

  t = Date.now()
  const [x, y] = await Promise.all([wait(100, 'x'), wait(100, 'y')])
  console.log('parallel', x, y, Date.now() - t, 'ms')

  const results = await Promise.allSettled([wait(10, 'ok'), Promise.reject(new
  Error('boom'))])
  console.log(results.map((r) => r.status))

  try {
    await Promise.reject(new Error('handled'))
  } catch (e) {
    console.log('caught:', e.message)
  }
}
await main()
```

Combinator	Resolves when
<code>Promise.all</code>	all succeed (rejects on the first failure)
<code>Promise.allSettled</code>	all finish, success or failure
<code>Promise.race</code>	the first one settles
<code>Promise.any</code>	the first one succeeds

Common mistake

`await` inside `forEach` does not wait — use `for...of` (sequential) or `Promise.all(arr.map(...))` (parallel).

Must-know utilities: debounce, throttle, array methods

debounce and throttle

```
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

const nums = [5, 1, 4, 2, 3]
console.log(nums.map((n) => n * 2), nums.filter((n) => n % 2), nums.reduce((a,
b) => a + b, 0))
console.log([...nums].sort((a, b) => a - b), nums.find((n) => n > 3),
nums.some((n) => n > 4))
```

 **Debounce:** run after calls stop (search box). **Throttle:** run at most every X ms (scroll, resize).

 `map` / `filter` / `slice` / `toSorted` return new arrays;

`sort` / `splice` / `push` / `reverse` mutate.

 ES modules: `import` / `export`, statically analysed (tree-shaking); CommonJS:

`require` / `module.exports`, loaded at runtime.

Revision

let/const + TDZ, closures, `this` depends on the call site, prototype chain, sync → microtasks → macrotask, Promise.all for parallel work, debounce vs throttle.

Q. What is a closure?

A. A function that keeps access to variables from the scope where it was created, even after that scope finished. Used for private state, `once()`, `memoize()` and factories.

Q. var vs let vs const?

A. `var` is function-scoped and hoisted as undefined; `let` and `const` are block-scoped and in the TDZ until declared; `const` can't be reassigned.

Q. Explain the event loop.

A. Run all sync code on the call stack, then drain every microtask (promise callbacks), then run one macrotask (timer, I/O) and repeat.

Q. `vs` =?

A. `=` compares value and type; converts types first. Use `===` by default.

Q. How is "this" decided?

A. By the call site: method call → the object; plain call → undefined in strict mode; new → the new object; call/apply/bind → the given value; arrow functions inherit it.

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

A. `all`: every promise must succeed; `allSettled`: wait for all results; `race`: first to settle; `any`: first to succeed.

Q. Debounce vs throttle?

A. Debounce waits until calls stop; throttle runs at most once per interval.

Q. Shallow vs deep copy?

A. Spread and `Object.assign` copy only the top level; `structuredClone` copies nested objects too.

Q. What is prototypal inheritance?

A. Objects link to a prototype object; property lookups walk up that chain. `class` is syntax over it.

Q. Why doesn't await work inside forEach?

A. `forEach` ignores the returned promises. Use `for...of` to run sequentially or `Promise.all` with `map` to run in parallel.

Q. null vs undefined?

A. undefined: declared but not assigned (or missing). null: deliberately "no value".

Q. What is event delegation?

A. One listener on a parent handles events from its children via event.target — fewer listeners, and it works for elements added later.