



JavaScript

Version ES2026 · verified 2026-07-27

A gradual, compact walk through modern JavaScript —
scope, coercion, functions, classes, async, and modules.

Contents

Mental model	2
Variables & scope	2
Values, coercion & equality	3
Destructuring, spread & optional chaining	3
Functions, closures & this	4
Objects, classes & private fields	5
Arrays & iteration	6
Promises, async/await & the event loop	7
Modules & the yearly editions	8
Further reading	9

Mental model

JavaScript is single-threaded: one call stack, and an event loop that runs queued callbacks only when that stack is empty — nothing ever interrupts running code. Variables hold references to objects, not objects themselves, so copying a variable never copies the data. Most historical confusion (coercion, `var`, `this`) has a modern replacement that sidesteps it; this sheet teaches the replacements and flags the legacy traps.

Variables & scope

Declaration	Behavior
<code>const x = 1</code>	block-scoped, no reassignment
<code>let x = 1</code>	block-scoped, reassignable
<code>var x = 1</code>	function-scoped — legacy, avoid

Default to `const`; use `let` only when you reassign. `let/const` exist from the top of their block but are unreadable until their declaration line — the temporal dead zone — so a too-early read throws instead of silently yielding `undefined` the way `var` does.

```
{  
  const user = { name: "Ada" };  
  user.name = "Grace"; // ok - mutation  
  // user = {};        // TypeError  
}
```

Gotcha: `const` prevents reassignment, not mutation. Freezing the object itself needs `Object.freeze(user)` — and even that is shallow.

Values, coercion & equality

Check	Result
<code>1 === "1"</code>	false — no coercion
<code>1 == "1"</code>	true — coerces, avoid
<code>typeof null</code>	"object" — historic bug
<code>x ?? fallback</code>	fallback only if nullish
<code>Object.is(NaN, NaN)</code>	true

Always compare with `===`. For defaults, `??` triggers only on `null` / `undefined`, while `||` also replaces valid falsy values like `0` and `""` — a real bug source in numeric settings.

```
const volume = 0;
console.log(volume || 50); // 50 - wrong
console.log(volume ?? 50); // 0 - right
```

Gotcha: `NaN === NaN` is false. Test with `Number.isNaN(x)` — the global `isNaN()` coerces first, so `isNaN("abc")` is also true.

Destructuring, spread & optional chaining

Syntax	Meaning
<code>const { a, b = 2 } = obj</code>	pick props, default

Syntax	Meaning
<code>const [x, ...rest] = arr</code>	first + the rest
<code>{ ...obj, a: 1 }</code>	shallow copy + override
<code>obj?.a?.b</code>	stop at null, no throw

Spread copies are shallow — nested objects are still shared references.

For a real deep copy of plain data, use `structuredClone(obj)`.

```
const cfg = { port: 80, tls: { on: true } };
const copy = { ...cfg, port: 443 };
copy.tls.on = false;
console.log(cfg.tls.on); // false - shared!

const deep = structuredClone(cfg);
```

Functions, closures & this

Form	this binding
<code>function f() {}</code>	dynamic — set by caller
<code>obj.method()</code>	<code>obj</code> , at the call site
<code>() => {}</code>	lexical — from enclosing scope
<code>f.bind(obj)</code>	fixed to <code>obj</code> forever

A closure is a function that keeps live access to the variables of the scope it was created in — the basis of callbacks, private state, and every hook-style API. Arrow functions don't have their own `this`, which makes

them the right choice for callbacks and the wrong choice for object methods.

```
function counter() {  
  let n = 0;  
  return () => ++n; // closes over n  
}  
  
const next = counter();  
console.log(next(), next()); // 1 2
```

Gotcha: extracting a method loses its `this`:

```
const f = obj.method; f() runs with this of undefined. Use  
obj.method.bind(obj) or an arrow.
```

Objects, classes & private fields

Syntax	Meaning
<code>class A extends B</code>	prototype-based inheritance
<code>#secret</code>	truly private field
<code>static create()</code>	on the class, not instances
<code>get x() / set x(v)</code>	computed property access

Classes are syntax over prototypes — there is still one shared method object per class, not a copy per instance. `#`-prefixed fields are enforced-private: accessing one from outside the class is a `SyntaxError` at parse time, not a runtime convention like `_name`.

```

class Counter {
  #n = 0;
  bump() { return ++this.#n; }
  static of(start) {
    const c = new Counter();
    c.#n = start;
    return c;
  }
}

console.log(Counter.of(5).bump()); // 6

```

Arrays & iteration

Method	Effect
<code>.map</code> / <code>.filter</code> / <code>.reduce</code>	transform, keep original
<code>.find</code> / <code>.findLast</code>	first/last match or undefined
<code>.at(-1)</code>	last element
<code>.toSorted()</code> / <code>.toReversed()</code>	sorted copy, no mutation
<code>Object.groupBy(items, fn)</code>	group into an object

Prefer the copying methods: `.toSorted()` returns a new array where `.sort()` mutates in place — passing a mutating sort's result around while the original silently changed is a classic aliasing bug.

```

const xs = [3, 1, 2];
const sorted = xs.toSorted((a, b) => a - b);

```

```
console.log(xs, sorted);  
// [ 3, 1, 2 ] [ 1, 2, 3 ]
```

Gotcha: `.sort()` without a comparator sorts *as strings*:

`[10, 9, 1].sort()` gives `[1, 10, 9]`. Always pass `(a, b) => a - b` for numbers.

Promises, async/await & the event loop

API	Use
<code>await p</code>	pause this function, free the thread
<code>Promise.all([...])</code>	parallel, fail-fast
<code>Promise.allSettled([...])</code>	parallel, never rejects
<code>Promise.try(fn)</code>	sync fn into a promise chain
<code>queueMicrotask(fn)</code>	run after current stack

`async` functions return promises; `await` suspends only the current function, never the whole thread. Sequential `await`s serialize — for independent work, start all the promises first, then `await` them together.

```
const slow = (ms, v) =>  
  new Promise(r => setTimeout(() => r(v), ms));  
  
const [a, b] = await Promise.all([  
  slow(100, "a"),  
  slow(100, "b"),  
]);
```

```
]); // ~100ms total, not 200  
console.log(a, b); // a b
```

Gotcha: `forEach(async ...)` does not await anything — the callbacks all start and `forEach` returns immediately. Use `for...of` with `await`, or `Promise.all(items.map(async ...))`.

Modules & the yearly editions

Syntax	Meaning
<code>export const x / export default</code>	named / default export
<code>import { x } from "./m.js"</code>	static import, hoisted
<code>await import(path)</code>	dynamic, lazy import
<code>import cfg from "./c.json" with { type: "json" }</code>	import attributes

ES modules are static — imports are resolved before any code runs, which is what enables tree-shaking and top-level `await`. The language now ships a small edition every June: ES2025 added iterator helpers, `Set` methods (`union`, `intersection`, `difference`), and `Promise.try`; ES2026 adds `Array.fromAsync`, `Error.isError`, `Math.sumPrecise`, and `Uint8Array.toBase64` — check runtime support before use.

```
const evens = [1, 2, 3, 4, 5, 6]  
  .values()  
  .filter(n => n % 2 === 0)
```

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

Note: Temporal — the replacement for Date — missed the ES2026 cutoff and is slated for a later edition. Don't build on it without a polyfill yet.

Further reading

- [MDN JavaScript reference](#)
- [ECMAScript 2026 specification](#)
- [TC39 finished proposals](#)
- [MDN — Event loop](#)