Day 02 — The Modern Language Surface You Should Already Reflexively Use

There is a set of language features that landed between roughly 2020 and 2024 that separate code that looks current from code that looks like it was written by someone who stopped paying attention. None of these is exotic; all of them are shipping everywhere and safe to use unconditionally in 2026. The goal today is not to learn them for the first time but to make them reflexes — to get to the point where you reach for `??` without thinking and where seeing a hand-rolled deep clone makes you wince. We will move quickly, cover more ground than yesterday, and end by refactoring a deliberately dated file into modern form so the difference is tactile rather than theoretical.

In this chapter, we will cover the following topics:

- Nullish and optional operators
- Copying data correctly with `structuredClone`
- Safer object and array introspection
- Modern strings and the `Intl` layer
- Numeric ergonomics: separators and `BigInt`
- Modern classes with real privacy
- Errors that carry their history
- Top-level `await`

Technical requirements

You will continue in the `reforged-sandbox` project from Day 1, running on **Node.js 24** with **TypeScript** and **Biome** installed. Every feature covered here ships in Node.js 24 and current browsers, so no extra packages are required. Code samples assume the strict `tsconfig.json` baseline you configured yesterday.

Understanding the modern language surface

Nullish and optional operators: the end of a whole bug class

The trio of optional chaining (`?.`), nullish coalescing (`??`), and logical assignment (`??=`, `||=`, `&&=`) exists to fix the ambiguity of truthiness. The classic bug is `const port = config.port || 3000`, which silently rewrites a legitimate `0` or `""` to the default because those are falsy. `??` only falls back on `null` or `undefined`, which is almost always what you actually meant: `const port = config.port ?? 3000`. Optional chaining short-circuits an entire access path the moment it hits `null`/`undefined`, returning `undefined` rather than throwing, and it works for property access (`a?.b`), indexing (`a?.[k]`), and calls (`fn?.()`). Logical assignment operators fuse the check

and the write: `options.timeout ??= 5000` sets the default only if it was nullish, in one expression. Internalize the rule: use `??` and `?.` by default, and reserve `||` for cases where you genuinely want all falsy values treated as absent.

A subtlety worth knowing: you cannot mix `??` with `||` or `&&` without parentheses, because the language refuses to guess your precedence. `a ?? b || c` is a syntax error; write `(a ?? b) || c`. This is a feature, not an annoyance — it prevents the exact ambiguity these operators were designed to eliminate. Another subtlety: optional chaining short-circuits the rest of the chain, so `a?.b.c` does not throw if `a` is nullish, but does throw if `a.b` is nullish and you access `.c` without a second `?.` — write `a?.b?.c` when both links may be absent. And optional call (`fn?.()`) only guards `fn` being nullish, not `fn` being a non-function, so it is a nullish guard, not a type guard. Knowing exactly what each operator does and does not protect against is what separates using them correctly from using them superstitiously.

Copying data correctly: structuredClone

For years, deep-cloning an object meant `JSON.parse(JSON.stringify(x))`, which silently destroys `Date` objects, `Map`, `Set`, `undefined`, `BigInt`, circular references, typed arrays, and more. The platform now ships `structuredClone(value)`, a built-in that deep-copies using the structured clone algorithm — the same one used to pass data to workers and store data in IndexedDB. It handles cycles, `Map`s, `Set`s, `Date`s, `RegExp`s, `ArrayBuffer`s, typed arrays, and more. It cannot clone functions, DOM nodes, or class prototypes (you get a plain object back for class instances, losing the prototype chain and methods), so it is for data, not behaviour. In 2026, hand-rolled recursive clone utilities and the JSON round-trip are both code smells; reach for `structuredClone` first and only fall back when you hit its documented limits.

Understand why the JSON round-trip is so dangerous, because the danger is invisible until it bites. It does not error on a `Date` — it silently converts it to an ISO string, so `clone.createdAt.getFullYear()` throws far from the clone site, and the bug looks like a date problem rather than a cloning problem. It turns a `Map` or `Set` into an empty `{}`, `undefined` values vanish, `NaN` and `Infinity` become `null`, and a circular reference throws outright. Every one of these is a class of production bug that `structuredClone` eliminates. The lesson generalizes: when a language ships a purpose-built primitive for a common operation, the ad-hoc idiom it replaces was almost certainly hiding subtle failures, and adopting the primitive is a correctness win, not just an ergonomic one.

Safer object and array introspection

`Object.hasOwn(obj, key)` replaces the awkward `Object.prototype.hasOwnProperty.call(obj, key)` and the buggy `obj.hasOwnProperty(key)` (which breaks on objects created with `Object.create(null)` or that shadow the method with their own `hasOwnProperty` property). Prefer it universally. On arrays, `Array.prototype.at(index)` supports

negative indices, so `arr.at(-1)` is the clean way to read the last element without `arr[arr.length - 1]`, and it works on strings and typed arrays too.

The immutable-by-default array methods — `toSorted`, `toReversed`, `toSpliced`, and `with` — return a new array instead of mutating in place, which matters enormously in reactive UI code where mutating a prop or a piece of state is a bug that breaks change detection. `arr.with(2, "x")` gives you a copy with index 2 replaced; `arr.toSorted((a,b) => a-b)` sorts a copy without disturbing the original. Their mutating counterparts (`sort`, `reverse`, `splice`) remain, but in 2026 the immutable versions are the default you reach for, and the mutating ones are the exception you use deliberately when you genuinely own the array and want in-place mutation for performance. `findLast`/`findLastIndex` search from the end, ending the `[...arr].reverse().find(...)` dance.

Two more introspection helpers worth having as reflexes: `Object.groupBy(items, fn)` and `Map.groupBy(items, fn)` group a collection into an object or `Map` keyed by the function's return value — `Object.groupBy(users, u => u.role)` gives `{ admin: [...], member: [...] }` in one call, replacing the reduce-into-an-object boilerplate you have written a hundred times. And `Array.fromAsync(asyncIterable)` collects an async iterable into an array, the async counterpart to `Array.from`, which pairs with the async iterators you will meet on Day 8.

Strings grew up: replaceAll, at, and Intl basics

Two small string additions remove long-standing friction. `String.prototype.replaceAll(a, b)` replaces every occurrence with a plain string argument, ending the `str.replace(/a/g, b)` regex dance you reached for whenever the target was a literal — and, crucially, ending the trap where `replace` with a string first argument silently changed only the first match, a bug that hid until the second occurrence appeared in real data. Strings also got `.at()`, so `str.at(-1)` reads the last character with the same negative-index clarity you get on arrays, no `str[str.length - 1]` or `str.slice(-1)` required.

The larger shift is **`Intl`**, the built-in internationalization layer that you should reach for instead of hand-rolling formatting. `new Intl.NumberFormat("en-US", { style: "currency", currency: "USD" }).format(1234.5)` yields `"$1,234.50"` correctly, handling thousands separators, decimal places, and currency symbols per locale — the kind of thing people still reimplement with `toFixed` and manual comma insertion, getting it subtly wrong for every locale that is not their own. `Intl.DateTimeFormat` formats dates and times without pulling in a date library for the common cases; `Intl.RelativeTimeFormat` produces "3 days ago" and "in 2 hours" strings that respect language and pluralization rules; and `Intl.Collator` gives you locale-aware string comparison for sorting, so `[...names].sort(new Intl.Collator("de").compare)` orders names the way a German speaker expects

rather than by raw code-point order, which is what the default sort gives you and which mangles accented characters. The senior habit here is recognizing that formatting and sorting human-facing text is a locale problem the platform already solved; reaching for `Intl` is both less code and more correct than anything you would write by hand, and it costs nothing because it ships in every runtime.

Numeric ergonomics: separators and BigInt

Numeric literals can now carry underscore separators purely for legibility: `1_000_000` and `0xFF_EC_DE_5E` and `1_000_000_000n` are the same values as their unseparated forms, but a reader can see the magnitude at a glance instead of counting zeros. The separators are ignored by the runtime entirely — they exist only for the humans reading the code — so use them freely in any literal large enough that miscounting is plausible, which is exactly where off-by-a-factor-of-ten bugs come from.

`BigInt` addresses a sharper correctness problem. JavaScript's `number` is an IEEE 754 double, which represents integers exactly only up to `Number.MAX_SAFE_INTEGER` (2^53 − 1, about 9 quadrillion); beyond that, integers silently lose precision, so `9007199254740993 === 9007199254740992` evaluates to `true`. `BigInt`, written with an `n` suffix (`9007199254740993n`) or via `BigInt(x)`, represents arbitrarily large integers exactly. Reach for it when you handle values that can exceed the safe range and must stay exact: Twitter/Discord-style snowflake IDs, database `bigint` primary keys, high-resolution timestamps in nanoseconds, cryptographic and hashing work, and precise currency arithmetic in the smallest unit. The trade-offs are real and worth stating: `BigInt` and `number` do not mix in arithmetic (`1n + 1` throws a `TypeError`), `BigInt` is slower than native doubles, `JSON.stringify` cannot serialize it without a custom replacer, and it is integer-only — there is no `BigInt` fraction. So it is not a general replacement for `number`; it is a precise tool for the specific case where exact large-integer semantics matter, and knowing that boundary is the point. A common real-world instance: an API returns a 64-bit ID as a JSON number, and by the time your `number`-typed code touches it the low bits are already wrong — the fix is to keep such IDs as strings across the wire.

Modern classes are real classes now

Class syntax quietly grew up. Private fields and methods use the `#` sigil and are genuinely private — enforced by the runtime, not by convention or by TypeScript's compile-time `private` (which is erased and offers no runtime protection). `#balance` cannot be read from outside the class at all, and attempting to access `obj.#balance` from outside is a syntax error, not a runtime one. You can test for brand membership with `#field in obj`, which safely checks whether an object is an instance created by your class without throwing — useful for writing methods that need to distinguish "my kind of object" from a look-alike. Static initialization blocks (`static { ... }`) let you run setup logic when the class is defined, useful for registering the class somewhere or computing derived static state that needs statements rather than a

single expression. Accessor pairs (`get`/`set`), static fields, `#`-private methods, and static private fields together mean you rarely need the old closure-based encapsulation tricks that hid state in a constructor closure.

When you write a class in 2026, use real privacy with `#` for anything that should not leak, and reserve TypeScript's `private` keyword for the specific cases where you want the field visible at runtime (for serialization, say, or interop with code that reflects over properties) but hidden from the type surface. The distinction — `#` for runtime-enforced privacy, `private` for type-only privacy — is a real design choice, and knowing which you want for a given field is a mark of understanding rather than cargo-culting one everywhere.

Errors that carry their history

The `Error` constructor now takes an options object with a `cause`: `throw new Error("failed to load user", { cause: originalError })`. This preserves the chain instead of flattening it, so a caught error can be rethrown with added context without losing the root cause, and a logger or error tracker (Day 57) can walk `err.cause` to reconstruct the full failure path. Combined with `AggregateError` (which bundles multiple errors, as produced by `Promise.any` when all promises reject — Day 8), you can build error-handling that reports why something failed all the way down rather than presenting a single, context-free message.

Get in the habit of attaching `cause` whenever you wrap and rethrow, and of reading `cause` when you handle errors. The anti-pattern this replaces is the `catch (e) { throw new Error("something failed") }` that discards `e` entirely, turning a diagnosable failure into a mystery. There is also a small but useful companion, `Error.isError(value)`, a reliable way to check whether something is a genuine `Error` (more robust than `instanceof` across realms/iframes/worker boundaries where prototype identity can differ). Errors that carry structured context are the difference between an on-call engineer diagnosing an incident in minutes and grepping logs for an hour.

Top-level await and the death of the IIFE

In modules, `await` works at the top level — no more wrapping your entry point in `(async () => { ... })()`. This makes module initialization that depends on async work (loading config, connecting to a resource, dynamically importing based on runtime conditions) clean and linear. It does have a real consequence to understand: a module with top-level `await` becomes an async dependency for anything importing it, which affects load ordering — importers wait for the awaited work to complete before their own code runs. Use it freely in application entry points where that ordering is exactly what you want; use it thoughtfully in widely-imported library modules, where making every consumer wait on your initialization may be undesirable. Paired with dynamic `import()` (which returns a promise), top-level

`await` lets you write conditional, async-aware module loading that reads top to bottom instead of nesting in callbacks.

Practical implementation

We will take a file written in a deliberately 2018 style and modernize it, feeling each improvement as a concrete before-and-after rather than an abstract recommendation.

Step 1 — Create the dated file. In your sandbox, make `src/legacy.ts`:

```
function lastActive(users) {
  var sorted = users.slice().sort(function (a, b) {
    return a.lastSeen - b.lastSeen;
  });
  return sorted[sorted.length - 1];
}

function getPort(config) {
  return config.port || 3000;
}

function clone(obj) {
  return JSON.parse(JSON.stringify(obj));
}

function hasKey(obj, key) {
  return obj.hasOwnProperty(key);
}

function byRole(users) {
  return users.reduce(function (acc, u) {
    (acc[u.role] = acc[u.role] || []).push(u);
    return acc;
  }, {});
}
```

Step 2 — Modernize `lastActive`. Replace the copy-then-sort-then-index with immutable methods and `at`:

```
type User = { id: string; lastSeen: number; joined: Date; role: string
};

const lastActive = (users: readonly User[]): User | undefined =>
  users.toSorted((a, b) => a.lastSeen - b.lastSeen).at(-1);
```

Notice three wins at once: `toSorted` does not mutate the caller's array, `at(-1)` reads the last element clearly, and the return type honestly admits the array might be empty (`| undefined`), which `noUncheckedIndexedAccess` would have forced anyway.

Step 3 — Fix the falsy-default bug. Change `getPort` to use `??`:

```
const getPort = (config: { port?: number }): number => config.port ??
3000;
```

Prove the bug to yourself: with the old `||` version, `getPort({ port: 0 })` returns `3000`, discarding a valid port. With `??`, it correctly returns `0`. Run both and watch the difference — feeling the bug is more convincing than reading about it.

Step 4 — Replace the clone. Swap the JSON round-trip for the built-in and demonstrate what it fixes:

```
const original = { joined: new Date("2026-01-01"), tags: new Set(["a",
"b"]), score: undefined };
const copy = structuredClone(original);
console.log(copy.joined instanceof Date); // true — the JSON version
would give a string
console.log(copy.tags instanceof Set);    // true — the JSON version
would give {}
console.log("score" in copy);             // true — the JSON version
would drop it
```

Step 5 — Correct the membership check. Replace `hasOwnProperty` with `Object.hasOwn`, and prove why it matters:

```
const bare = Object.create(null) as Record<string, unknown>;
bare.token = "x";
// bare.hasOwnProperty("token") would THROW — bare has no prototype,
so no hasOwnProperty
console.log(Object.hasOwn(bare, "token")); // true, safely
```

Step 6 — Replace the reduce with `Object.groupBy`.

```
const byRole = (users: readonly User[]) => Object.groupBy(users, (u)
=> u.role);
// { admin: [...], member: [...] } — no accumulator boilerplate
```

Step 7 — Add a private, self-describing error and a brand check.

```
class Store {
  #data = new Map<string, User>();
  static #instances = 0;
  static { Store.#instances = 0; }               // static init block
  get(id: string): User {
    const u = this.#data.get(id);
    if (!u) throw new Error(`user ${id} not found`, { cause: { id,
size: this.#data.size } });
    return u;
  }
  static isStore(x: unknown): x is Store {
    return typeof x === "object" && x !== null && #data in (x as
Store); // brand check
  }
}
```

Step 8 — Format human-facing output with `Intl` and reach for `BigInt` where exactness matters. Add a display helper and an ID handler that show why the platform primitives beat the hand-rolled versions:

```
const price = (cents: number) =>
  new Intl.NumberFormat("en-US", { style: "currency", currency: "USD"
}).format(cents / 100);
console.log(price(123_456)); // "$1,234.56" — separators for
readability, Intl for correctness

// A 64-bit snowflake ID arrives as a string precisely because a JSON
number would corrupt it:
const raw = "1305818302943657984";
const id = BigInt(raw);
console.log(id + 1n); // exact; Number(raw) + 1 would already be wrong
in the low bits

const names = ["Zöe", "adam", "Ärger", "Bob"];
console.log([...names].sort(new Intl.Collator("de").compare)); //
locale-aware, not code-point order
```

Run `biome check --write .` over the finished file and note how little it complains — modern code and the linter agree, which is itself a signal that you are writing idiomatic 2026 JavaScript.

Exercises

Take a real file from a project you own and count how many of today's features it should be using but is not; refactor one function per feature and observe how the code shrinks and clarifies. Write a short note to yourself explaining the `0/""` truthiness bug in a way you could give to a junior — teaching it is how you make it permanent. Find one place in your codebase where a `JSON.parse(JSON.stringify(...))` clone is silently corrupting `Date`s, `Map`s, or `undefined` values, and fix it with `structuredClone`, then check whether the corruption was causing a bug you had not connected to cloning. Replace one hand-rolled reduce-into-an-object with `Object.groupBy`. Find one place where you format a number, date, or currency by hand — with `toFixed`, manual comma insertion, or a template string — and replace it with the appropriate `Intl` formatter, then test it against a non-English locale to see what your hand-rolled version was getting wrong. Search your codebase for any 64-bit ID or large counter that flows through a `number`, and reason about whether it can exceed `Number.MAX_SAFE_INTEGER` and silently lose precision; if it can, decide whether it should be carried as a string or a `BigInt`. Finally, audit a class in your codebase and decide, field by field, whether each should use runtime-private `#` or type-only `private`, and justify each choice.

Summary

The modern language surface is not optional polish; it eliminates entire bug classes, from truthiness defaults to silent clone corruption to hand-rolled locale formatting.

Making these features reflexes — not merely recognizing them — is the difference between code that looks current and code that reveals you stopped paying attention. In the next chapter, we will move from the settled recent past to the leading edge that is already usable: the features finalized in ES2025 and the most important proposals arriving in engines during the 2025–2026 window.

Key takeaways from today:

- Default to `??` and `?.` and reserve `||` for true falsy-means-absent cases, knowing precisely what each operator guards (nullish, not type) and how short-circuiting propagates.
- Reach for `structuredClone` instead of the JSON round-trip, which silently corrupts `Dates`, `Maps`, `Sets`, and `undefined` — a correctness win, not just ergonomics.
- Use `Object.hasOwn`, `at`, the immutable `toSorted`/`toReversed`/`with` family, and `Object.groupBy`/`Map.groupBy` reflexively, treating the mutating array methods as the deliberate exception.
- Let `Intl` format and sort human-facing numbers, dates, currency, and text; use underscore separators in large literals for legibility; and reach for `BigInt` precisely when integers can exceed the safe range and must stay exact, knowing it does not mix with `number`.
- Write real privacy with `#` fields (runtime-enforced, distinct from TypeScript's type-only `private`), use brand checks (`#field in obj`) and static blocks where they fit, and attach `cause` whenever you wrap an error so failures carry their own diagnosable history.