Day 03 — ES2025 and the ES2026 Pipeline

Yesterday's features were the settled recent past. Today is 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. These are the things that make you look current in a code review in 2026 — iterator helpers, the new `Set` algebra, `Promise.try`, `RegExp.escape`, import attributes, the Temporal date/time API, decorators, and explicit resource management with `using`. Several of these change how you structure code, not just how you spell it, which is why they are worth real study rather than a skim. We will go deeper than a feature list, because understanding *why* each was added tells you *when* to reach for it.

In this chapter, we will cover the following topics:

- Iterator helpers and lazy pipelines
- `Array.fromAsync` for draining async iterables
- `Set` algebra as a built-in
- `Promise.try` for unifying sync and async failure
- `RegExp.escape` and safer patterns
- Import attributes and JSON modules
- `Float16Array` and low-precision numerics
- Temporal, the replacement for `Date`
- Decorators
- Explicit resource management with `using` and `await using`

Technical requirements

You will work in the `reforged-sandbox` project on **Node.js 24**. The ES2025 features covered here (iterator helpers, `Set` methods, `Promise.try`, `Array.fromAsync`, `RegExp.escape`, import attributes) are available in Node.js 24 and current browsers. **Temporal** and standard **decorators** are in the pipeline and arriving in engines; the Temporal example installs the `temporal-polyfill` package so you can run it everywhere today. Explicit resource management (`using`) requires a runtime and TypeScript version that support `Symbol.dispose`.

Understanding ES2025 and the ES2026 pipeline

Iterator helpers: lazy pipelines without arrays

For a decade, transforming a sequence meant materializing arrays: `array.map(...).filter(...).slice(0, 10)` builds a full intermediate array at each step, so a three-stage pipeline over a million items allocates three million-element arrays even if you only wanted ten results. ES2025 adds helper methods directly on

iterators — `map`, `filter`, `take`, `drop`, `flatMap`, `reduce`, `some`, `every`, `find`, and `toArray` — and they are lazy. Calling `.map()` on an iterator does not do the work; it returns a new iterator that does the work one element at a time as you pull. This means you can compose transformations over infinite or expensive sequences and only pay for what you consume. `numbers.values().map(x => x * x).filter(x => x % 2 === 0).take(5).toArray()` computes exactly five results and stops, even if `numbers` is enormous.

The structural consequence is that generators become first-class pipeline sources: any `function*` you write now composes with the full helper suite, so you can model a paginated API, a file read, or an event stream as a generator and transform it lazily with the same operations you use on arrays. The mental shift is from "collections you process eagerly" to "streams you pull from lazily," and it is the biggest ergonomic change to everyday JavaScript in years. It also changes performance characteristics: laziness turns "load everything then filter" into "filter while loading," which is the difference between an operation that fits in memory and one that does not. Reach for iterator helpers whenever a pipeline is long, the source is large or infinite, or you only need a prefix of the results — and reach for array methods when you genuinely want the whole materialized array (which you often do, so this is a judgment, not a blanket replacement).

Array.fromAsync: collecting an async iterable

Iterator helpers cover the synchronous side; `Array.fromAsync` is the async counterpart, and it fills a gap that has quietly annoyed everyone who works with streams. `Array.from` has always turned any sync iterable into an array, but there was no built-in way to drain an async iterable — a paginated fetch, a Node.js stream, a database cursor, an async generator — into an array without hand-writing a `for await...of` loop that pushes into an accumulator. `Array.fromAsync(asyncIterable)` does exactly that, awaiting each value in sequence and resolving to a real array: `const rows = await Array.fromAsync(cursor)`. It is the `for await...of`-to-array idiom compressed into one honest expression, and like `Array.from` it accepts an optional mapping function, so `await Array.fromAsync(stream, chunk => decode(chunk))` collects and transforms in one pass. Crucially it also accepts a sync iterable of promises and awaits each element, which makes it the natural way to settle a sequence lazily rather than all-at-once the way `Promise.all` does. The distinction from `Promise.all` matters: `Promise.all` fires everything concurrently and waits for the lot, whereas `Array.fromAsync` pulls sequentially, respecting backpressure and ordering — the right tool when each step depends on the previous one, or when hammering a source with full concurrency would overwhelm it. Reach for it whenever you have "some async source of items and I just want the array"; reach for `Promise.all` when you have a fixed set of independent promises you want in flight at once.

Set algebra as a built-in

`Set` finally behaves like a mathematical set. ES2025 adds `union`, `intersection`, `difference`, `symmetricDifference`, `isSubsetOf`, `isSupersetOf`, and `isDisjointFrom`. Code that used to be a tangle of `filter` and `has` calls collapses to `a.intersection(b)` or `admins.isSubsetOf(users)`. These operate on any set-like object with `size`, `has`, and `keys`, and they return real `Set` instances (for the ones that produce sets) or booleans (for the predicates). Reach for them whenever you are reasoning about membership across two collections — permissions (which required permissions is the user missing? `required.difference(granted)`), tags, feature flags, diffing two lists of ids, or deduplicating a union. The readability gain is real: `required.isSubsetOf(granted)` states the intent directly, whereas the equivalent `[...required].every(r => granted.has(r))` makes the reader reconstruct it. When your logic is fundamentally about set relationships, using set operations makes the code say what it means.

Promise.try: unifying sync and async failure

A recurring papercut: you call a function that might throw synchronously or might return a rejected promise, and you want both handled by the same `.catch`. Without help, a synchronous throw escapes the promise chain entirely — `doThing().catch(handle)` does not catch a synchronous throw inside `doThing`'s call, because the throw happens before a promise exists. `Promise.try(fn)` runs `fn` and wraps the result — whether it returns a value, throws synchronously, or returns a promise — into a single promise, so a `.catch` on it handles all three uniformly. It replaces the awkward `new Promise(resolve => resolve(fn()))` idiom and the `Promise.resolve().then(fn)` trick, and it makes "run this possibly-sync, possibly-async thing and handle all failures the same way" a one-liner. It is small, but it removes a real class of "the error escaped my `.catch` because it threw synchronously" bugs, which are especially common when wrapping user-supplied callbacks or plugin functions whose sync-versus-async nature you do not control.

RegExp.escape and safer patterns

Building a regular expression from user input was, for years, a security and correctness hazard: forget to escape a `.` or `(` and your pattern misbehaves, matches too much, or — with a pathological pattern — becomes a denial-of-service vector through catastrophic backtracking. ES2025's `RegExp.escape(string)` returns a version of the string safe to embed literally in a pattern, escaping every character that has special meaning. Combined with the inline pattern modifiers — `(?i:...)` to make just part of a pattern case-insensitive rather than the whole regex — regex construction is finally both safe and precise. Any time you interpolate untrusted or dynamic text into a `RegExp`, `RegExp.escape` is now the correct first move, and forgetting it should feel like forgetting to parameterize a SQL query (Day 30) — the same category of "untrusted input reinterpreted as code" mistake.

Import attributes and JSON modules

The `with` syntax standardizes importing non-JavaScript resources with an explicit type assertion: `import config from "./config.json" with { type: "json" }`. This replaces the older, now-removed `assert` keyword (TypeScript 7 makes the old `assert` form a hard error, as you will see tomorrow). Import attributes let the host safely load JSON — and, increasingly, other module types — without bundler-specific magic, and they make the intent explicit at the import site, which matters because loading a `.json` as a module has different security and caching implications than loading JavaScript. Dynamic imports take the attributes too: `await import("./data.json", { with: { type: "json" } })`. The broader significance is that import attributes are the standardized extension point for the module system to load new kinds of resources safely, so as the platform adds module types, they arrive through this one consistent syntax rather than a proliferation of loaders.

Float16Array and low-precision numerics

Most JavaScript developers never think about float width because the language only ever gave them one — the 64-bit double behind every `number`. ES2025 adds `Float16Array`, a typed array of IEEE half-precision (16-bit) floats, alongside `Math.f16round` (which rounds a number to the nearest representable half-precision value) and `DataView.prototype.getFloat16`/`setFloat16` for reading and writing them in binary buffers. Half precision buys you a fraction of the range and about three or four significant decimal digits, which sounds like a downgrade until you remember why you would want it: it halves memory and, more importantly, halves bandwidth. That trade is exactly the one the rest of the stack has already made. GPU pipelines and WebGPU consume f16 vertex and texture data natively; machine-learning models are routinely quantized to 16-bit (and lower) weights because the precision is enough for inference and the memory savings are enormous; and any time you ship large numeric arrays over the wire — telemetry, audio, point clouds, embeddings — sending them as f16 rather than f32 or f64 cuts the payload without a manual bit-packing scheme. Before `Float16Array`, feeding half-precision data to WebGPU or a WASM ML runtime meant converting through a `Uint16Array` and juggling the bit layout by hand; now the typed array is the interchange format, and `Math.f16round` lets you simulate the precision loss deterministically so your JavaScript matches what the GPU or model will actually compute. It is a specialist tool — you will not reach for it in ordinary business logic — but for graphics, ML, and bandwidth-sensitive numeric transport it removes a genuinely fiddly layer of manual conversion.

Temporal: the replacement for Date

The single most consequential arrival of the 2025–2026 window is **Temporal**, the new date and time API that replaces the famously broken `Date`. `Date`'s problems are legendary: it conflates instants and calendar dates, is mutable (so passing one around risks spooky action at a distance), uses zero-based months (December is

month 11), parses inconsistently across engines, and mishandles time zones in ways that have caused countless production bugs. Temporal fixes all of it with a family of immutable, explicit types, each modelling exactly one concept. `Temporal.Instant` is an exact moment on the universal timeline. `Temporal.ZonedDateTime` is a moment in a specific named time zone, carrying the zone so arithmetic respects daylight-saving transitions. `Temporal.PlainDate`, `PlainTime`, and `PlainDateTime` are calendar values without a zone — a birthday, a store's opening time — which are genuinely different from instants and were a constant source of `Date` bugs when conflated. `Temporal.Duration` represents a span, and `Temporal.PlainYearMonth`/`PlainMonthDay` cover partial dates.

Everything is immutable, so arithmetic returns new values (`date.add({ days: 30 })` does not mutate `date`), and time-zone and calendar handling is first-class and correct — adding a month handles varying month lengths, adding across a DST boundary does the right thing, and converting between zones is explicit. `Temporal.Now.zonedDateTimeISO()` gives the current moment in the local zone. Temporal is shipping in engines through this window and is polyfillable everywhere else, so you can adopt it now. The guidance is unambiguous: write new date logic against Temporal, treat `Date` as legacy interop (convert at the boundary), and enjoy the disappearance of an entire genre of date-and-timezone bugs. The one adjustment is conceptual — you must decide, for each value, which Temporal type models it (is this an instant, a zoned datetime, or a plain date?), and that decision is exactly the clarity `Date` lacked and the reason it was so bug-prone.

Decorators: metadata and behaviour on classes

Decorators — the `@decorator` syntax on classes, methods, fields, and accessors — reached Stage 3, shipped in TypeScript, and are arriving in engines. A decorator is a function that receives the thing it decorates and can observe or replace it, enabling declarative patterns: logging or timing a method by annotating it, registering a class, validating a field, or wiring dependency injection. Frameworks (Angular, and many backend libraries) lean on them for expressive, declarative APIs. The 2026 standard decorators differ from the older experimental TypeScript decorators, so when you read older code or configure a project, be aware of which flavour is in play. Use decorators where they genuinely clarify intent — a `@cache` or `@authorized` annotation that reads as documentation — and avoid over-using them to hide too much behaviour, since heavy decorator magic can obscure control flow. They are a tool for declarative cross-cutting concerns, best in moderation.

Explicit resource management: using and await using

Borrowed in spirit from C#'s `using` and Python's `with`, JavaScript now has deterministic cleanup. A resource that implements `Symbol.dispose` (or `Symbol.asyncDispose`) can be declared with `using` (or `await using`), and its disposal runs automatically when the enclosing block exits — even on early return or throw. This ends the `try`/`finally` pyramid for files, database connections, locks,

subscriptions, and tracing spans: `using conn = await pool.acquire();` guarantees `conn` is released at block end, no matter how the block exits. It composes: multiple `using` declarations dispose in reverse order of acquisition, like a stack, so cleanup mirrors setup correctly even for interdependent resources. Library authors are adding `Symbol.dispose`/`Symbol.asyncDispose` to anything that holds a resource, and you should too when you write such a type — a class that opens something should implement dispose so consumers get deterministic cleanup for free. This is the feature that most changes how you write resource-handling code: the moment you find yourself writing `try { ... } finally { cleanup() }`, ask whether the resource should implement dispose and be used with `using` instead, which is both cleaner and harder to get wrong (you cannot forget the `finally`).

Practical implementation

Step 1 — Build a lazy pipeline. In your sandbox, create `src/pipeline.ts` and prove laziness with an infinite generator:

```
function* naturals(): Generator<number> {
  let n = 1;
  while (true) yield n++;
}

const firstFiveEvenSquares = naturals()
  .map((x) => x * x)
  .filter((x) => x % 2 === 0)
  .take(5)
  .toArray();

console.log(firstFiveEvenSquares); // [4, 16, 36, 64, 100]
```

If iterator helpers were eager, this would loop forever. It does not, because `take(5)` only pulls five values through the chain. Add a `console.log` inside the `map` to see that it runs exactly the number of times needed, not infinitely.

Step 2 — Replace membership logic with `Set` algebra.

```
const required = new Set(["read", "write", "deploy"]);
const granted = new Set(["read", "write"]);

const missing = required.difference(granted);   // Set { "deploy" }
const canDeploy = required.isSubsetOf(granted);  // false
console.log([...missing], canDeploy);
```

Rewrite the same logic with `filter`/`has` and compare the two for readability — the set version states the relationship directly.

Step 3 — Unify error handling with `Promise.try`.

```
function maybeThrows(x: number): number {
  if (x < 0) throw new RangeError("negative");
  return x * 2;
}

Promise.try(() => maybeThrows(-1))
  .then((v) => console.log("ok", v))
  .catch((e) => console.log("caught uniformly:", (e as
Error).message));
```

Confirm that a bare `maybeThrows(-1)` inside a `.then` chain would have thrown synchronously and escaped the `.catch`, while `Promise.try` captures it.

Step 4 — Safely build a regex from input.

```
function highlight(text: string, term: string): string {
  const pattern = new RegExp(RegExp.escape(term), "gi");
  return text.replace(pattern, (m) => `[${m}]`);
}
console.log(highlight("a.b.c and a x b", "a.b")); // only the literal
"a.b" matches
```

Without `RegExp.escape`, the `.` would match any character and corrupt the result — try it both ways to see the difference.

Step 5 — Do date math with Temporal. Use the polyfill if your runtime does not yet expose it globally. First install the package:

```
pnpm add temporal-polyfill
```

Then write the date logic against it:

```
import { Temporal } from "temporal-polyfill";

const now = Temporal.Now.zonedDateTimeISO("Europe/London");
const inThirtyDays = now.add({ days: 30 });
const daysUntil = now.until(inThirtyDays, { largestUnit: "day" });
console.log(now.toString());
console.log(inThirtyDays.toString());
console.log(daysUntil.toString()); // P30D

// distinguish a plain date (a birthday) from an instant:
const birthday = Temporal.PlainDate.from("1990-05-15");
console.log(birthday.add({ years: 36 }).toString()); // 2026-05-15
```

Note that `add` returned a new value (immutability), the time zone rode along correctly, and `PlainDate` modeled a calendar date with no spurious time-of-day or zone.

Step 6 — Guarantee cleanup with `using`. Implement a disposable and watch it clean up on throw:

```
class Span {
  constructor(private label: string) { console.log(`start ${label}`);
}
  [Symbol.dispose]() { console.log(`end ${this.label}`); }
}

function work() {
  using span = new Span("request");
  using inner = new Span("db-query");   // disposed before span
(reverse order)
  throw new Error("boom");              // both disposes still run
before the throw propagates
}

try { work(); } catch { /* handled */ }
```

The `end db-query` then `end request` lines prove disposal ran despite the exception, in reverse order, with no `finally` in sight.

Step 7 — Drain an async source with `Array.fromAsync`. Model a paginated API as an async generator and collect it in one expression:

```
async function* pages(): AsyncGenerator<number> {
  for (let page = 1; page <= 3; page++) {
    // pretend each yield is an awaited fetch of one page
    await Promise.resolve();
    yield page * 10;
  }
}

const all = await Array.fromAsync(pages(), (n) => n + 1);
console.log(all); // [11, 21, 31]
```

Contrast this with `Promise.all`: `fromAsync` pulled the pages sequentially, respecting order and backpressure, whereas `Promise.all` would have required all the promises to exist up front and fired them concurrently.

Step 8 — Add an async disposable. Implement `Symbol.asyncDispose` on a mock connection and use `await using` to confirm asynchronous cleanup runs deterministically at block exit.

Exercises

Rewrite a real array-processing chain in your codebase as a lazy iterator pipeline and identify one place where laziness lets you avoid building a large intermediate array or process an infinite/streaming source. Replace a permissions or tag-diffing routine with `Set` algebra and note the readability gain. Convert one module's date handling from `Date` to Temporal, and for each value decide explicitly which Temporal type models it — noting every place the old code was silently wrong about months, time zones, or mutation. Wrap a user-supplied callback with `Promise.try` to unify its failure handling. Take a place where you currently hand-write a `for await...of` loop that pushes into an array — draining a stream, cursor, or paginated

fetch — and replace it with `Array.fromAsync`, then articulate why sequential draining is correct there rather than `Promise.all`'s full concurrency. If you have any numeric-transport or graphics code, identify one array of floats that would tolerate half precision and reason about the bandwidth you would save encoding it as a `Float16Array`. Finally, find a `try/finally` that releases a resource and convert it to `using`, and add `Symbol.dispose` to one of your own resource-holding classes so its consumers get deterministic cleanup for free.

Summary

ES2025 and the 2025–2026 pipeline are usable now and change structure, not just spelling: iterator helpers turn collections into lazy streams, Temporal replaces the broken `Date`, `Set` algebra makes membership reasoning declarative, and `using` retires the `try/finally` cleanup pyramid. Absorbing these puts you on the leading edge of what a 2026 code review expects. In the next chapter, we will turn to the tool that makes checking all of this cheap: TypeScript 7 and its Go-native compiler, and how its order-of-magnitude speedup changes your daily loop.

Key takeaways from today:

- Iterator helpers turn collections into lazy, composable streams that only compute what you consume, while array methods remain right when you want the whole materialized result.
- `Array.fromAsync` is the async twin of `Array.from`, draining any async iterable into an array with correct ordering and backpressure — the right tool when steps depend on each other, where `Promise.all` is right for independent promises you want in flight at once.
- `Set` algebra makes membership reasoning declarative, `Promise.try` unifies synchronous and asynchronous failure so nothing escapes your `.catch`, and `RegExp.escape` makes regex construction safe against the same untrusted-input-as-code hazard as SQL injection.
- Import attributes standardize loading JSON and future module types, and `Float16Array` with `Math.f16round` brings half-precision numerics into the language for graphics, ML, and bandwidth-sensitive work.
- Temporal replaces the broken, mutable, timezone-hostile `Date` with immutable, explicit types where you deliberately choose which concept each value models; decorators enable declarative cross-cutting concerns in moderation; and explicit resource management with `using` replaces error-prone cleanup with deterministic, composable, reverse-order disposal you cannot forget.