What's up in ES2026?

A talk by Christophe Porteneuve at Confoo Montréal 2024

whoami

```
const christophe = {
  family: { wife: "👩 Élodie", sons: ["👦 Maxence", "👦 Elliott"] },
  city: "Paris, FR",
  company: {
    name: "Doctolib",
    hiring: true,
    superCool: true
  },
  webDevSince: 1995,
  mightBeKnownFor: [
    "Prototype.js",
    "Prototype and Script.aculo.us (“The Bungie Book”)",
    "dotJS",
    "Paris Web",
    "NodeSchool Paris",
  ],
};
```

A word about Doctolib...

Doctolib is *the leading healthcare technology company in Europe*

We provide a *top-notch* "operating system" of services and tools to *care teams and patients* in multiple countries

Two core missions:

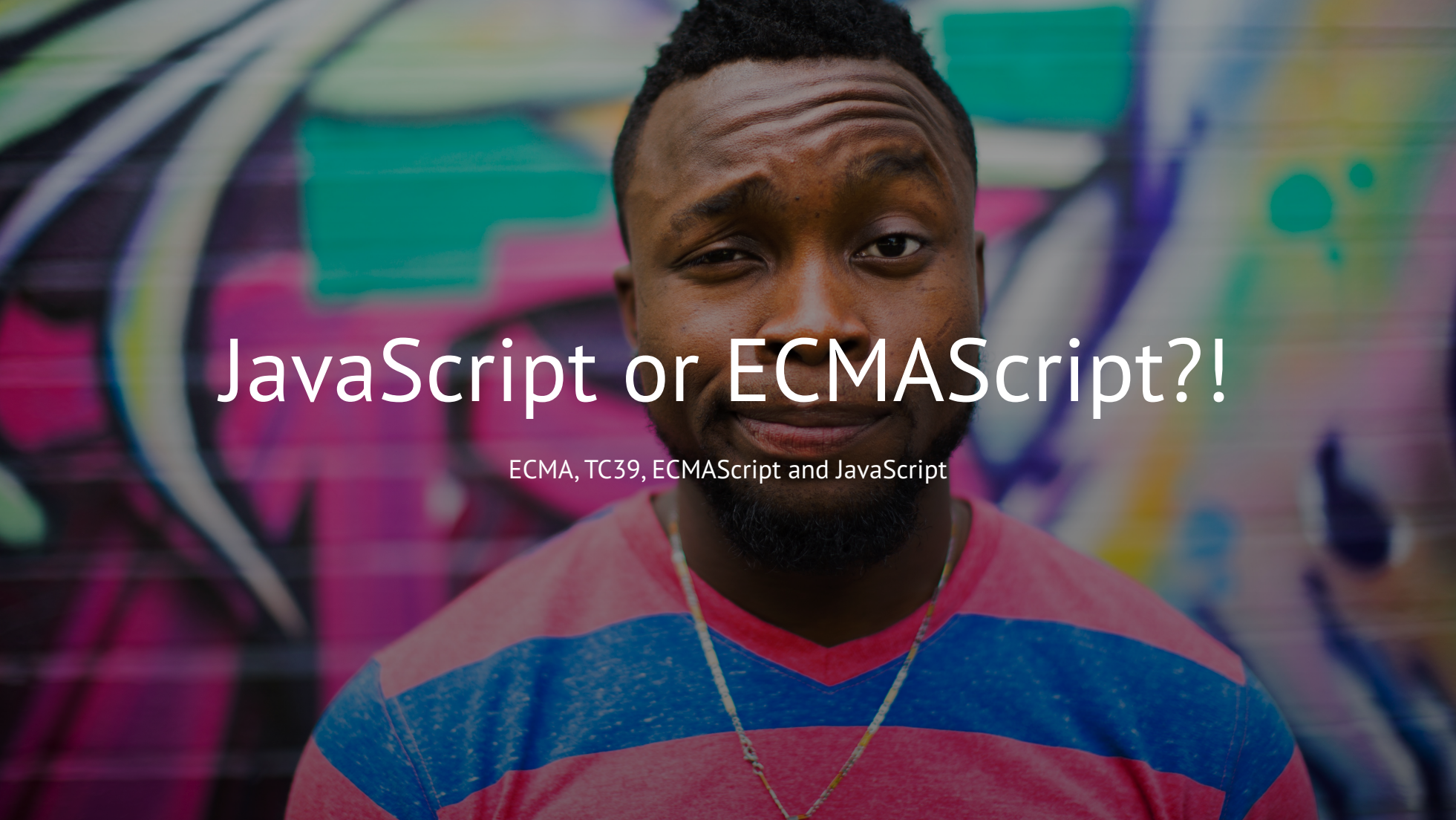
- *Improve the daily lives of care teams*
- *Improve everyone's health and healthcare access*

700+ developers and tech staff among 2,800+ Doctolibers. Great culture.

Tech stack: Ruby/Rails, React/TypeScript, PostgreSQL, MongoDB, AWS.

Offices in Paris (HQ), Nantes, Niort, Berlin and Milan, nice remote policies.

We're always hiring talented tech staff! Check out our careers page!


JavaScript or ECMAScript?!
ECMA, TC39, ECMAScript and JavaScript

ECMA and TC39

ECMA is an international standards body (much like ISO, IETF, W3C or the WHATWG, for instance)

ES = ECMAScript. The official standard for JavaScript*

TC39 = Technical Committee 39. Caretaker of several standards: ECMAScript (ECMA-262), Intl (ECMA-402), JSON (ECMA-404), etc.

Which happens to be, for the U.S., a registered trademark of Oracle Corp. I know. 🥒

How TC39 moves JavaScript forward

Meets every two months (remote, in-room, hybrid)

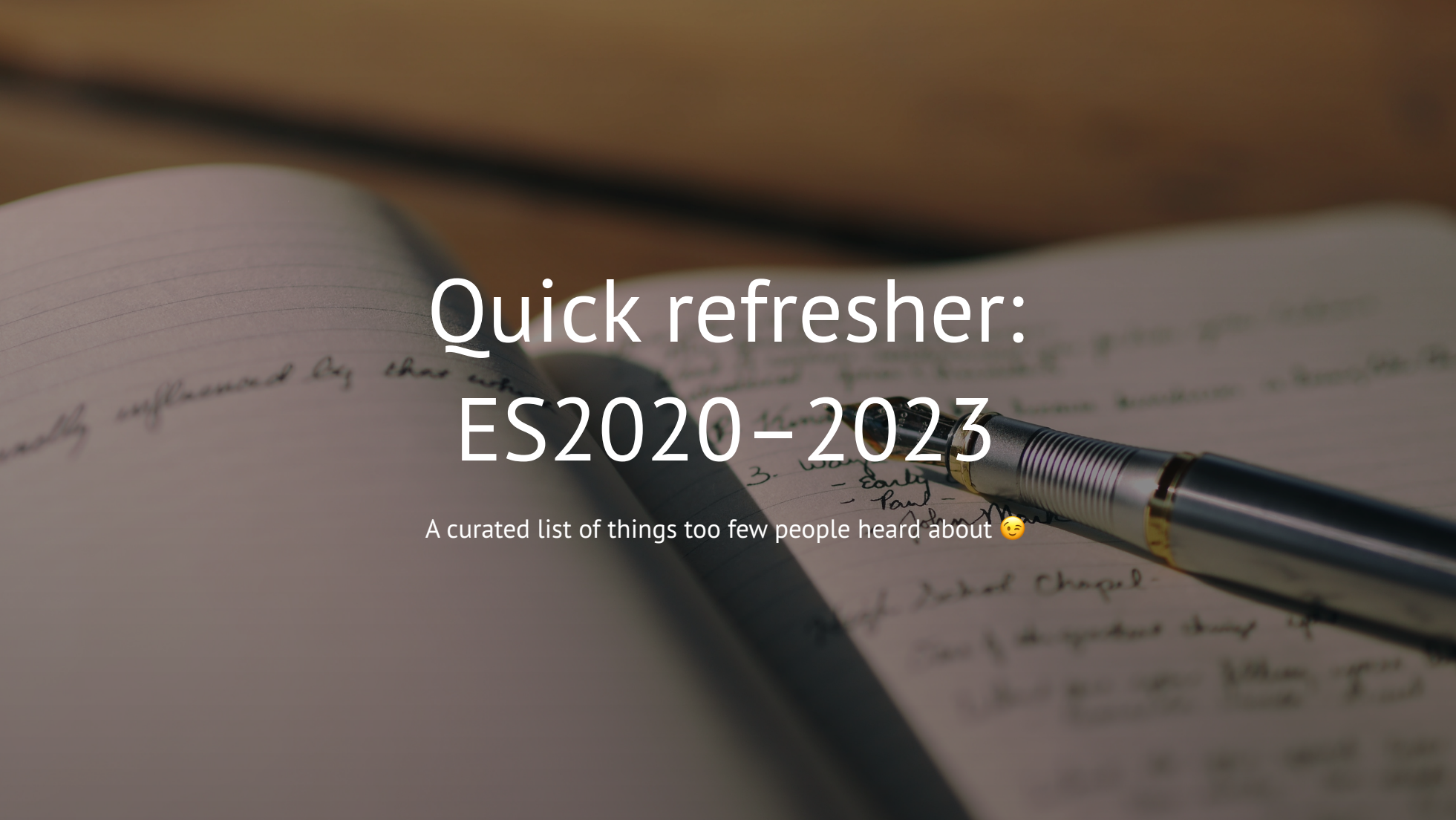
Yearly release cycle: feature-freezing in January or March, official release in June.

“ES6” = ES2015, “ES7” = ES2016, and we now say ES2024, etc.

This is all transparent and public.

The *5 stages* of ~~grief~~ the TC39 process

Stage	Description
0 Strawman	“Say, wouldn't a unicorn (🦄) operator be awesome for…”
1 Proposal	A TC39 member becomes the proposal's “champion.” General API is defined and most cross-cutting concerns are handled.
2 Draft	Initial *Spec Text* is done, which covers all critical aspects and tech semantics.
3 Candidate	Spec is finalized, duly reviewed and approved. API is finalized, all edge cases are handled.
4 Finished	Full Test262 coverage, 2+ native implementations (often v8 and Spidermonkey), significant real-world feedback, and *Spec Editor* sign-off. Will ship in the next feature freeze (January/March), and then in the follow-up official release.



Quick refresher: ES2020–2023

A curated list of things too few people heard about 😉

ES2020: `String#matchAll`

Captures ***all groups*** for a sticky or ***global*** regex.

```
const text = 'Get in touch at tel:0983450176 or sms:478-555-1234'

text.match(/(?<protocol>[a-z]{3}):(?<number>[\d-]+)/g)
// => ['tel:0983450176', 'sms:478-555-1234'] -- 😔 DUDE, WHERE ARE MY GROUPS?!
```

```
Array.from(text.matchAll(/([a-z]{3}):([\d-]+)/g)).map(
  ([, protocol, number]) => ({ protocol, number })
)
// => [{ number: '0983450176', protocol: 'tel' }, { number: '478-555-1234', protocol: 'sms' }]

Array.from(text.matchAll(/(?<protocol>[a-z]{3}):(?<number>[\d-]+)/g)).map((mr) => mr.groups)
// => [{ number: '0983450176', protocol: 'tel' }, { number: '478-555-1234', protocol: 'sms' }]
```

ES2020 / ES2021: `Promise.allSettled` / `any`

The two missing combinators: `any` short-circuits on the *first fulfillment*, whilst `allSettled` doesn't short-circuit at all: you get all settlements for analysis.

Together with `all` (short-circuits on first rejection) and `race` (short-circuits on first settlement) from ES2015, we now cover all scenarios.

```
// May the fastest strategy win!
const data = await Promise.any([fetchFromDB(), fetchFromCache(), fetchFromHighSpeedLAN()])

// Run all tests in parallel, no short-circuit!
await Promise.allSettled(tests)
// => [
//   { status: 'fulfilled', value: Response… },
//   { status: 'fulfilled', value: undefined },
//   { status: 'rejected', reason: Error: snapshot… }
// ]
```


ES2022: `at()` on position-based native iterables 🤩

You know how `Array` and `String` let you use negative indices with `slice`, `splice`, etc. but not with `[…]`? This novelty lets you grab last elements without a cringe.

From now on, ***all position-based native iterables*** offer `.at(…)` that understands negative indices!

```
1  const roomSeries = ['St-Laurent', 'Westmount', 'Outremount']
2  roomSeries.at(-1) // => 'Outremount'
3  roomSeries.at(-2) // => 'Westmount'
```

ES2023: Find From Last 😙

`Array`s have had `find` and `findIndex` for quite a while (ES2015), but what about searching ***from the end***?

After all, we've had `reduceRight` and `lastIndexOf` since forever, right?

Until recently you had to roll your own loops 😔 or bring out the big guns and do a (mutative!) `reverse()` first, but not anymore!

```
1  const upcomingTalks = [
2    { time: '10:00', title: 'Chopping the Monolith', tags: ['Architecture'] },
3    { time: '11:00', title: 'A Look at the Future of Software Development', tags: ['Architecture',
4    { time: '13:00', title: '4 Reliability Anti-Patterns', tags: ['Architecture'] },
5    { time: '14:00', title: 'Building DS with Web Components', tags: ['HTML', 'CSS', 'JS'] },
6    { time: '15:00', title: 'Le monolithe est mort, vive le monolithe !', tags: ['Architecture', 'PH
7  ]
8
9  const lastMorningTalk = upcomingTalks.findLast(({ time }) => time <= '12:00')
10 // => { title: 'A Look at the Future of Software Development' ... }
11 const latestArchTalkIndex = upcomingTalks.findLastIndex(({ tags }) => tags.includes('Architecture
12 // => 4
```

ES2023: Change Array by Copy

A series of cool utilities that let you derive arrays (yay immutability). `Array`'s API so far exposed 8 derivative methods (producing new arrays) and 9 mutative methods (modifying arrays in place), including `reverse()` and `sort()`, which many folks didn't realize were mutative!

```
const trackSpeakers = ['Nicolas', 'Hugh', 'Teiva', 'Simon', 'Sébastien']

trackSpeakers.toReversed()
// => ['Sébastien', 'Simon', 'Teiva', 'Hugh', 'Nicolas']
trackSpeakers.toSorted((s1, s2) => s1.localeCompare(s2))
// => ['Hugh', 'Nicolas', 'Sébastien', 'Simon', 'Teiva']
trackSpeakers.toSpliced(-2, 2)
// => ['Nicolas', 'Hugh', 'Teiva']
trackSpeakers.with(-2, 'Yann')
// => ['Nicolas', 'Hugh', 'Teiva', 'Yann', 'Sébastien']

trackSpeakers // => ['Nicolas', 'Hugh', 'Teiva', 'Simon', 'Sébastien']
```

ES2024

Quite a few things are nearly done, so who knows...

ES2024: Array grouping 🎉

One more nail in Lodash's coffin.

```
const thisSlot = [
  { title: 'Vertical Slice Architecture', mainTag: 'Architecture' },
  { title: 'On Inheriting Legacy Codebases', mainTag: 'Architecture' },
  { title: 'Introduction to OpenTelemetry…', maintag: 'DevOps' },
  { title: 'Writing Effective JUnit Tests', mainTag: 'Tests' },
  // …
]
schedule.group(({ mainTag }) => mainTag)
// {
//   Architecture: [{ title: 'Vertical Slice…'… }, { title: 'On Inheriting…'… }],
//   DevOps: [{ title: 'Introduction to OpenTelemetry…'… }, { title: 'Accélérez vos API…'… }],
//   Tests: [{ title: 'Writing Effective…'… }],
//   ...
// }

schedule.groupToMap(({ mainTag }) => mainTag)
// => Same thing, **as a Map** (so any grouping key type!)
```

ES2024: `v` flag for regexes

Allows for ***nested classes*** (classes represent possibilities for a single character match), which in turn allows ***difference*** and ***intersection*** of classes. Wicked cool.

Use the `v` flag instead of ES2015's Unicode flag (`u`) when you need that feature.

```
// All of Unicode's decimal digits, except ASCII ones:
text.match(/[\p{Decimal_Number}--[0-9]]/gv)

// Equivalently:
text.match(/[\p{Decimal_Number}--\p{ASCII}]/gv)

// All Khmer letters (= Khmer Script + Letter Property)
text.match(/[\p{Script=Khmer}&&\p{Letter}]/gv)
```

ES2024 `Promise.withResolvers()`

stage 3

A common pattern rolled by hand anytime we need access to promise outcome methods outside of the `Promise` constructor callback. Pretty neat to avoid scope juggling and when working with event-based underlying APIs for our async processing.

```
1   const { promise, resolve, reject } = Promise.withResolvers()
```

⚠ ***Don't expose the outcome methods to your consumers!*** It's only there to simplify your implementation code!

E2024? `Array.fromAsync(…)`

stage 3

We've had `Array.from(…)` since ES2015, that consumes any ***synchronous iterable*** to turn it into an actual array.

We'll likely get `Array.fromAsync(…)`, that does the same thing with ***async iterables***.

```
1  // Reads all STDIN (readable stream) lines into an array
2  process.stdin.setEncoding('utf-8')
3  const inputLines = await Array.fromAsync(process.stdin)
```

```
1  // Let's remove trailing whitespace / LF / CR, while we're at it
2  process.stdin.setEncoding('utf-8')
3  const inputLines = await Array.fromAsync(process.stdin, (line) => line.trimEnd())
```

E2024? Collection / iterator utilities

stage 3

We're not going to stop processing data collections (and iterables in general) anytime soon, so we might as well have more tools in our standard toolbelt for this…

We're about to get many ***new*** `Set` ***methods*** (intersection, union, difference, disjunction, super/subset, etc.) and a ton of ***iterator helpers*** (instead of having to roll our own generative functions for `take`, `filter` or `map`, for instance).

```
function* fibonacci() { /* … */ }

const firstTens = new Set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
const fibs = new Set(fibonacci().take(10))
const earlyFibs = firstTens.intersection(fibonacci) // => Set { 1, 2, 3, 5, 8 }
const earlyNonFibs = firstTens.difference(fibonacci) // => Set { 4, 6, 7, 9, 10 }
const evenFibs = earlyFibs.values().filter((n) => n % 2 === 0)
```

Asynchronous versions are in the pipeline too, at stage 2 right now (February 2024).

ES2024? Guaranteed resource cleanup

stage 3

Finally a mechanism to guarantee resource disposal!

Quite like C#'s `using`, Python's `with` or Java's try-with-resources: disposes of the resource in a guaranteed way when the scope or closure is discarded.

Exists in synchronous and asynchronous variants. Based on two new well-known symbols (`Symbol.dispose` et `Symbol.asyncDispose`), supported out-of-the-box by timers and streams.

```
1  async function copy4K(s1, s2) {
2    using f1 = await fs.promises.open(s1, constants.O_RDONLY),
3          f2 = await fs.promises.open(s2, constants.O_WRONLY)
4
5    const buffer = Buffer.alloc(4096)
6    const { bytesRead } = await f1.read(buffer)
7    await f2.write(buffer, 0, bytesRead)
8  } // 'f2' is disposed first, then 'f1' is disposed second
```

Proposal's name: Explicit Resource Management. *TypeScript 5.2+ and Babel 7.22+ support it. Early implementations in Node.js (e.g. Timers).*

ES2024? Import / export attributes

stage 3

Provides free-form metadata on imports, with an inline syntax.

The dominating use case, long discussed, is extra module types with matching type expectations for security reasons (a bit like HTTP's `X-Content-Type-Options: nosniff` response header). We then use the `type` metadata, leveraged by engines.

```
// Static imports
import config from '../config/config.json' with { type: 'json' }

// Dynamic imports
const { default: config } = await import('../config/config.json', { with: { type: 'json' } })
```

The spec suggests matching upgrades for Web Worker instantiation and HTML's `script` tag.

This proposal supersedes the same-stage JSON Modules *proposal, that used a more specific* `assert` *syntax.*

ES2024? More flexible named capture groups

stage 3

Named capture groups are a major readability / maintainability boost for regexes, but an oversight in their initial spec prevented using the same group in multiple parts of an alternative.

It should have been ready for ES2023 but lacked some tests and a second native implementation. Tests are done now and we're waiting for either v8 or Spidermonkey to jump the gun: this will very likely be part of ES2024.

```
1  const year = dateText.match(/(?<year>[0-9]{4})-[0-9]{2}|[0-9]{2}-(?<year>[0-9]{4})/)?.groups.year
```

ES2024? Decorators

stage 3

This takes ***forever***… Went through a few false-starts, then we had to wrap the test suite, and now we're waiting for native implementations. The spec is done, anyway, and TypeScript aligns with it. This is a great way of doing AOP *(as are ES proxies, by the way)*. The language provides the plumbing, and the community provides the actual decorators.

```
class SuperWidget extends Component {
  @deprecate
  deauth() { … }

  @memoize('1m')
  userFullName() { … }

  @autobind
  logOut() {
    this.#oauthToken = null
  }

  @override
  render() { … }
}
```

ES2024? Shadow Realms

stage "2.7"

This provides the building blocks for having full control of ***sandboxed JS evaluation*** (among other things, you can customize available globals and standard library elements).

This is a ***godsend*** for web-based IDEs, DOM virtualisation, test frameworks, server-side rendering, secure end-user scripts, and more!

```
1  const realm = new ShadowRealm()
2
3  const process = await realm.importValue('./utils/processor.js', 'process')
4  const processedData = process(data)
5
6  // True isolation!
7  globalThis.userLocation = 'Freiburg'
8  realm.evaluate('globalThis.userLocation = "Paris"')
9  globalThis.userLocation // => 'Freiburg'
```

Check out this explainer for full details.


ES2025 and beyond

Temporal 🥳

stage 3

This will (advantageously) replace Moment, Luxon, date-fns, etc. We already have `Intl` for formatting, but we're upping our game here. Immutable-style API, nanosecond precision, all TZ supported, distinguishes absolute and local time, duration vs. interval, etc. Just awesome! Check out the docs, cookbook and Maggie's talk at dotJS 2019!

```
const meeting1 = Temporal.Date.from('2020-01-01')
const meeting2 = Temporal.Date.from('2020-04-01')
const time = Temporal.Time.from('10:00:00')
const timeZone = new Temporal.TimeZone('America/Montreal')
timeZone.getAbsoluteFor(meeting1.withTime(time)) // => 2020-01-01T15:00:00.000Z
timeZone.getAbsoluteFor(meeting2.withTime(time)) // => 2020-01-01T14:00:00.000Z
```

```
const departure = Temporal.ZonedDateTime.from('2020-03-08T11:55:00+08:00[Asia/Hong_Kong]');
const arrival = Temporal.ZonedDateTime.from('2020-03-08T09:50:00-07:00[America/Los_Angeles]');
departure.until(arrival).toString() // => 'PT12H55M'

const flightTime = Temporal.Duration.from({ hours: 14, minutes: 10 }); // { minutes: 850 } would
const parisArrival = departure.add(flightTime).withTimeZone('Europe/Paris');
parisArrival.toString() // => '2020-03-08T19:05:00+01:00[Europe/Paris]')
```

Collection normalization and `Map#emplace()`

stage 2

Lets you intercept incoming data for `Map`s so you can normalize / cleanup or even constraint / deny them.

```
const headers = new Map(undefined, {
  coerceKey: (name) => name.toLowerCase()
})
headers.set('X-Requested-With', 'politeness')
headers // => Map { 'x-requested-with': 'politeness' }
```

As for `emplace()`, it provides a sort of automatic *upsert*, with variable behavior.

```
function addVisit(path) {
  visitCounts.emplace(path, {
    insert: () => 0,                    // Arguments: key, map
    update: (existing) => existing + 1  // Arguments: existing, key, map
  })
}
```

`Iterator.range` 🤩

stage 2

Finally an arithmetic sequence generator! Coupled with iterator helpers, it's just too good…

```
Iterator.range(0, 5).toArray()
// => [0, 1, 2, 3, 4]

Iterator.range(1, 10, 2).toArray()
// => [1, 3, 5, 7, 9]

Iterator.range(1, 7, { step: 3, inclusive: true })
  .map((n) => '*'.repeat(n))
  .toArray()
// => ['*', '****', '*******']
```

Go have fun in the playground!

Records & Tuples: Immutability FTW 💖

stage 2

Deep, native immutable objects (records) and arrays (tuples). We get all the benefits of immutability (e.g. referential equality), and it helps promote functional programming in JS.

All the usual operators and APIs work (`in`, `Object.keys()`, `Object.is()`, `===`, etc.), and this plays nicely with the standard library. You can easily convert from mutable versions using factories. Cherry-on-top: `JSON.parseImmutable()`!

```
1  // Records
2  const grace1 = #{ given: 'Grace', family: 'Hopper' }
3  const grace2 = #{ given: 'Grace', family: 'Kelly' }
4  const grace3 = #{ ...grace2, family: 'Hopper' }
5  grace1 === grace3 // => true!
6  Object.keys(grace1) // => ['family', 'given'] -- sorted!
7
8  // Tuples
9  #[1, 2, 3] === #[1, 2, 3] // => true!
```

Have fun with the tutorial, sweet playground and amazing cookbook!

`Object.pick()` / `omit()` 🥹

stage 1

I so want to get rid of Lodash for this… This is kinda recent (July 2022) and doesn't seem to be high-priority, but hey. Accepts key sets or a predicate (with an optional `this` specifier).

```
const conference = { name: 'Smashing Conference Freiburg', year: 2023, city: 'Freiburg', speake
Object.pick(conference, ['name', 'year'])
// => { name: 'Smashing Conference Freiburg', year: 2023 }

Object.pick(conference, (value) => typeof value === 'number')
// => { year: 2023, speakers: 13 }

Object.omit(conference, (value) => typeof value === 'number')
// => { name: 'Smashing Conference Freiburg', city: 'Freiburg' }
```

We *might* even get syntactic sugar for picking!

```
conference.{name, year} // => { name: 'Smashing Conference Freiburg', year: 2023 }

const keys = ['name', 'city']
conference.[...keys] // => { name: 'Smashing Conference Freiburg', city: 'Freiburg' }
```

`Promise.try()` 🤏

stage 2

A faster alternative to the usual `Promise.resolve().then(f)` or `new Promise((resolve) => resolve(f()))` shenanigans for allowing promise-based consumer semantics over a function that may be sync or async.

Ensures same-tick execution when synchronous whilst being a lot more ergonomic!

```
1  // `init` is a value-returning function that may be sync or promise-based async
2  async function runProcess({ init... }) {
3    const initial = await Promise.try(init)
4    // ...
5  }
```

The pipeline operator 🪄

stage 2

Massive cleanup of processing chains based on nested calls, interpolation, arithmetic operators, etc.

```
// BEFORE 🤮
console.log(
  chalk.dim(
    `$ ${Object.keys(envars)
      .map(envar =>
        `${envar}=${envars[envar]}`)
      .join(' ')
    }`,
    'node',
    args.join(' ')))

const result = Array.from(
  take(3,
    map((v) => v + 1,
      filter((v) => v % 2 === 0, numbers))))
```

```
// AFTER 🤩
Object.keys(envars)
  .map(envar => `${envar}=${envars[envar]}`)
  .join(' ')
  |> `$ ${%}`
  |> chalk.dim(%, 'node', args.join(' '))
  |> console.log(%)


const result = numbers
  |> filter(%, (v) => v % 2 === 0)
  |> map(%, (v) => v + 1)
  |> take(%, 3)
  |> Array.from
```

Note that the substitution syntax (`%`) is nowhere near settled.

Pattern matching 🤯

stage 1

A `match` expression that provides sort of a shape-based `switch`. Has equivalents in Rust, Python, F#, Elixir/Erlang, etc. This is just a ***tiny peak*** at what it envisions:

```
match (res) {
  when ({ status: 200, body, ...rest }): handleData(body, rest)
  when ({ status, destination: url }) if (300 <= status && status < 400):
    handleRedirect(url)
  when ({ status: 500 }) if (!this.hasRetried): do {
    retry(req)
    this.hasRetried = true
  }
  default: throwSomething()
}

const commandResult = match (command) {
  when ([ 'go', dir and ('north' or 'east' or 'south' or 'west')]): go(dir);
  when ([ 'take', item and /[a-z]+ ball/ and { weight }]): take(item);
  default: lookAround()
}
```

Finally *truly* legible regexes! 🎉

stage 1

Perl, C#, Ruby have it… JS might finally get fully extended regex syntax. This ignores whitespace (including carriage returns) and comments. Yummy!

```
const TAG_REGEX = new RegExp(String.raw`
  <
  # Tag name
  (?<tag>[\w-]+)
  \s+
  # Attributes
  (?<attrs>.+?)
  >
  # Contents
  (?<content>.+?)
  # Closing tag, matching the opening one
  </\k<tag>>
`, 'x')
```

🙏 Thank you! 😊

These slides are at `bit.ly/confoo-es2026`.

Christophe: `@porteneuve` / `@porteneuve@piaille.fr` / LinkedIn

Photo credits: Cloudy blue sky by mosi knife, Confused by Ayo Ogunsende, Summary by Aaron Burden, Gift unwrapping by Kira auf der Heide and Open door by Jan Tinneberg, all from Unsplash.