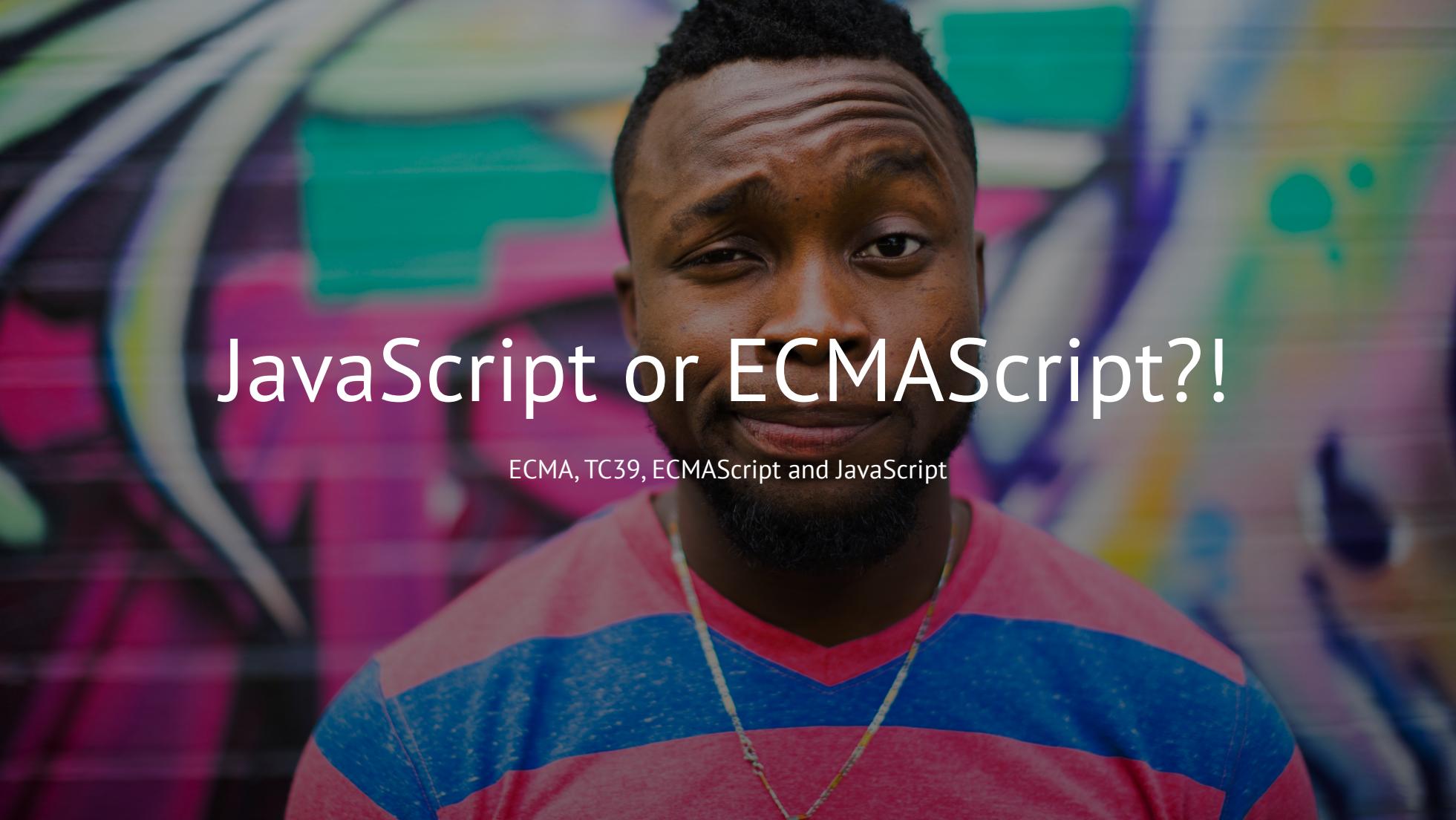


# Things You Don't Know about JavaScript in 2023

A talk by [Christophe Porteneuve](#) at [Smashing Conference Freiburg 2023](#)

# whoami

```
1 const christophe = {
2   family: { wife: '👩 Élodie', sons: ['👦 Maxence', '👦 Elliott'] },
3   city: 'Paris, FR',
4   company: 'Delicious Insights',
5   trainings: [
6     '360° ES', 'TypeScript', 'Advanced TypeScript',
7     'React PWA', 'Advanced React', 'Node.js',
8     '360° Git',
9   ],
10  webDevSince: 1995,
11  mightBeKnownFor: [
12    'Prototype.js',
13    'Prototype and Script.aculo.us ("The Bungie Book")',
14    'dotJS',
15    'Paris Web',
16    'NodeSchool Paris',
17  ],
18 }
```

A close-up portrait of a young Black man with short hair, looking directly at the camera with a gentle smile. He is wearing a red and blue horizontally striped V-neck t-shirt and a thin chain necklace. The background is a vibrant, abstract mural featuring various colors like teal, pink, yellow, and purple.

# 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 ES2023, etc.

This is all [transparent and public](#).

# The 5 stages of grief the TC39 process

Stage	Description
<b>0 Strawman</b>	“Say, wouldn't a unicorn (  ) operator be awesome for...”
<b>1 Proposal</b>	A TC39 member becomes the proposal's “champion.” General API is defined and most cross-cutting concerns are handled.
<b>2 Draft</b>	Initial <i>Spec Text</i> is done, which covers all critical aspects and tech semantics.
<b>3 Candidate</b>	Spec is finalized, duly reviewed and approved. API is finalized, all edge cases are handled.
<b>4 Finished</b>	Full Test262 coverage, 2+ native implementations (often v8 and Spidermonkey), significant real-world feedback, and <i>Spec Editor</i> 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.

```
1 const text = 'Get in touch at tel:0983450176 or sms:478-555-1234'  
2  
3 text.match(/(?:<protocol>[a-z]{3}):(?:<number>[\d-]+)/g)  
4 // => ['tel:0983450176', 'sms:478-555-1234'] -- 🤔 DUDE, WHERE ARE MY GROUPS?!  
  
1 Array.from(text.matchAll(/(?:[a-z]{3}):(?:[\d-]+)/g)).map(  
2   ([, protocol, number]) => ({ protocol, number })  
3 )  
4 // => [{ number: '0983450176', protocol: 'tel' }, { number: '478-555-1234', protocol: 'sms' }]  
5  
6 Array.from(text.matchAll(/(?:<protocol>[a-z]{3}):(?:<number>[\d-]+)/g)).map((mr) => mr.groups)  
7 // => [{ 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.

```
1 // May the fastest strategy win!
2 const data = await Promise.any([fetchFromDB(), fetchFromCache(), fetchFromHighSpeedLAN()])
3
4 // Run all tests in parallel, no short-circuit!
5 await Promise.allSettled(tests)
6 // => [
7 //   { status: 'fulfilled', value: Response... },
8 //   { status: 'fulfilled', value: undefined },
9 //   { status: 'rejected', reason: Error: snapshot... }
10 // ]
```

# 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 cities = ['Freiburg', 'Antwerp', 'San Francisco', 'New York']
2 cities.at(-1) // => 'New York'
3 cities.at(-2) // => 'San Francisco'
```

# 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 brightOnes = [
2   { name: 'Proxima Centauri', lightYears: 4.24, massKg: 2.428e29 },
3   { name: 'Sirius A', lightYears: 8.70, massKg: 3.978e30 },
4   { name: 'Tau Ceti', lightYears: 11.75, massKg: 1.557e30 },
5   { name: 'Altair', lightYears: 16.73, massKg: 3.56e30 },
6 ]
7
8 const kindaFarthest = brightOnes.findLast(({ lightYears }) => lightYears <= 15)
9 // => { name: 'Tau Ceti', lightYears: 11.75 }
10 const farthestMassive = brightOnes.findLastIndex(({ massKg }) => massKg >= 3e30)
11 // => 3
```

# 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!

```
1 const todaysOtherKnownSpeakers = ['Christine', 'Elliot', 'Miriam', 'Harrison']
2
3 todaysOtherKnownSpeakers.toReversed() // => ['Harrison', 'Miriam', 'Elliot', 'Christine']
4 todaysOtherKnownSpeakers.toSorted() // => ['Miriam', 'Harrison', 'Elliot', 'Christine']
5 todaysOtherKnownSpeakers.toSpliced(-2, 2) // => ['Miriam', 'Harrison']
6 todaysOtherKnownSpeakers.with(-2, 'Suzanne') // => ['Christine', 'Elliot', 'Suzanne', 'Harrison']
7
8 todaysOtherKnownSpeakers // => ['Christine', 'Elliot', 'Miriam', 'Harrison']
```

# ES2024

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

# 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).

```
1 function* fibonacci() { /* ... */ }

2

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

Asynchronous versions are in the pipeline too, at stage 2 right now (September 2023).

# Array grouping

stage 2

One more nail in Lodash's coffin.

```
1  const schedule = [
2    { label: 'Morning run', time: '07:00', type: 'break' },
3    { label: 'Welcome back!', time: '09:40', type: 'keynote' },
4    { label: 'Life is a Mystery', time: '10:00', type: 'talk' },
5    { label: 'Morning Break', time: '10:50', type: 'break' },
6    { label: 'Design Beyond Breakpoints', time: '11:05', type: 'talk' },
7    // ...
8  ]
9  schedule.group(({ type }) => type)
10 // {
11 //   break: [{ Label: 'Morning Run'... }, { Label: 'Morning Break'... }],
12 //   keynote: [{ Label: 'Welcome Back!'... }],
13 //   talk: [{ Label: 'Life is a Mystery'... }, { Label: 'Design Beyond Breakpoints'... }],
14 // }
15
16 schedule.groupToMap(({ type }) => type)
17 // => Same thing, **as a Map** (so any grouping key type!)
```

# 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
```

---

*The proposal is titled Explicit Resource Management. TypeScript 5.2 and Babel 7.22 support it, by the way.*

# 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.

```
1 // Static imports
2 import config from '../config/config.json' with { type: 'json' }
3
4 // Dynamic imports
5 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: 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.

```
1 // ALL of Unicode's decimal digits, except ASCII ones:  
2 text.match(/[\p{Decimal_Number}--[0-9]]/gv)  
3  
4 // Equivalently:  
5 text.match(/[\p{Decimal_Number}--\p{ASCII}]|gv)  
6  
7 // ALL Khmer Letters (= Khmer Script + Letter Property)  
8 text.match(/[\p{Script=Khmer}&&\p{Letter}]/gv)
```

# 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? `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!

# 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.

```
1  class SuperWidget extends Component {
2    @deprecate
3    deauth() { ... }
4
5    @memoize('1m')
6    userFullName() { ... }
7
8    @autobind
9    logOut() {
10      this.#oauthToken = null
11    }
12
13    @override
14    render() { ... }
15 }
```

# ES2024? Shadow Realms

stage 3

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.

A close-up photograph of a teal-colored wooden door. The door has a brass handle and a small brass lock plate. The door is slightly ajar, revealing a bright, blurred interior space.

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!](#)

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

```
1 const departure = Temporal.ZonedDateTime.from('2020-03-08T11:55:00+08:00[Asia/Hong_Kong]');
2 const arrival = Temporal.ZonedDateTime.from('2020-03-08T09:50:00-07:00[America/Los_Angeles]');
3 departure.until(arrival).toString() // => 'PT12H55M'
4
5 const flightTime = Temporal.Duration.from({ hours: 14, minutes: 10 }); // { minutes: 850 } would work too
6 const parisArrival = departure.add(flightTime).withTimeZone('Europe/Paris');
7 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.

```
1 const headers = new Map(undefined, {
2   coerceKey: (name) => name.toLowerCase()
3 })
4 headers.set('X-Requested-With', 'politeness')
5 headers // => Map { 'x-requested-with': 'politeness' }
```

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

```
1 function addVisit(path) {
2   visitCounts.emplace(path, {
3     insert: () => 0,           // Arguments: key, map
4     update: (existing) => existing + 1 // Arguments: existing, key, map
5   })
6 }
```

# Iterator.range



stage 2

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

```
1  Iterator.range(0, 5).toArray()  
2  // => [0, 1, 2, 3, 4]  
3  
4  Iterator.range(1, 10, 2).toArray()  
5  // => [1, 3, 5, 7, 9]  
6  
7  Iterator.range(1, 7, { step: 3, inclusive: true })  
8    .map((n) => '*' .repeat(n))  
9    .toArray()  
10 // => ['*', '****', '*****']
```

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).

```
1  const conference = { name: 'Smashing Conference Freiburg', year: 2023, city: 'Freiburg', speakers: 13 }
2  Object.pick(conference, ['name', 'year'])
3  // => { name: 'Smashing Conference Freiburg', year: 2023 }
4
5  Object.pick(conference, (value) => typeof value === 'number')
6  // => { year: 2023, speakers: 13 }
7
8  Object.omit(conference, (value) => typeof value === 'number')
9  // => { name: 'Smashing Conference Freiburg', city: 'Freiburg' }
```

We *might* even get syntactic sugar for picking!

```
1  conference.{name, year} // => { name: 'Smashing Conference Freiburg', year: 2023 }
2
3  const keys = ['name', 'city']
4  conference.[...keys] // => { name: 'Smashing Conference Freiburg', city: 'Freiburg' }
```

# The pipeline operator

stage 2

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

```
1 // BEFORE 😞
2 console.log(
3   chalk.dim(
4     `$ ${Object.keys(envars)
5       .map(envar =>
6         `${envar}=${envars[envar]}`)
7       .join(' '))
8   `,
9   'node',
10  args.join(' ')))
11
12 const result = Array.from(
13  take(3,
14    map((v) => v + 1,
15      filter((v) => v % 2 === 0, numbers))))
```

```
1 // AFTER 😍
2 Object.keys(envars)
3   .map(envar => `${envar}=${envars[envar]}`)
4   .join(' ')
5 |> `${%}`
6 |> chalk.dim(% , 'node' , args.join(' '))
7 |> console.log(%)
8
9
10 const result = numbers
11 |> filter(% , (v) => v % 2 === 0)
12 |> map(% , (v) => v + 1)
13 |> take(% , 3)
14 |> Array.from
```

Note that the substitution syntax (%) is [nowhere near settled](#).

Things You Don't Know about JavaScript in 2023 · A talk by Christophe Porteneuve at Smashing Conference Freiburg 2023 · © 2023–présent Delicious Insights

# 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:

```
1  match (res) {
2    when ({ status: 200, body, ...rest }): handleData(body, rest)
3    when ({ status, destination: url }) if (300 <= status && status < 400):
4      handleRedirect(url)
5    when ({ status: 500 }) if (!this.hasRetried): do {
6      retry(req)
7      this.hasRetried = true
8    }
9    default: throwSomething()
10 }
11
12 const commandResult = match (command) {
13   when ([ 'go', dir and ('north' or 'east' or 'south' or 'west')]): go(dir);
14   when ([ 'take', item and /[a-z]+ ball/ and { weight }]): take(item);
15   default: lookAround()
16 }
```

# 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!

```
1  const TAG_REGEX = new RegExp(String.raw`  
2    <  
3      # Tag name  
4      (?<tag>[\w-]+)  
5      \s+  
6      # Attributes  
7      (?<attrs>.+?)  
8      >  
9      # Contents  
10     (?<content>.+?)  
11     # Closing tag, matching the opening one  
12     </\k<tag>>  
13   ` , 'x')
```



# Thank you!



These slides are at [bit.ly/freiburg-javascript](https://bit.ly/freiburg-javascript).

Christophe: [@porteneuve](https://twitter.com/porteneuve) / [@porteneuve@piaille.fr](mailto:@porteneuve@piaille.fr) / [LinkedIn](#)

Delicious Insights: [LinkedIn](#) / [Trainings](#)

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](#).