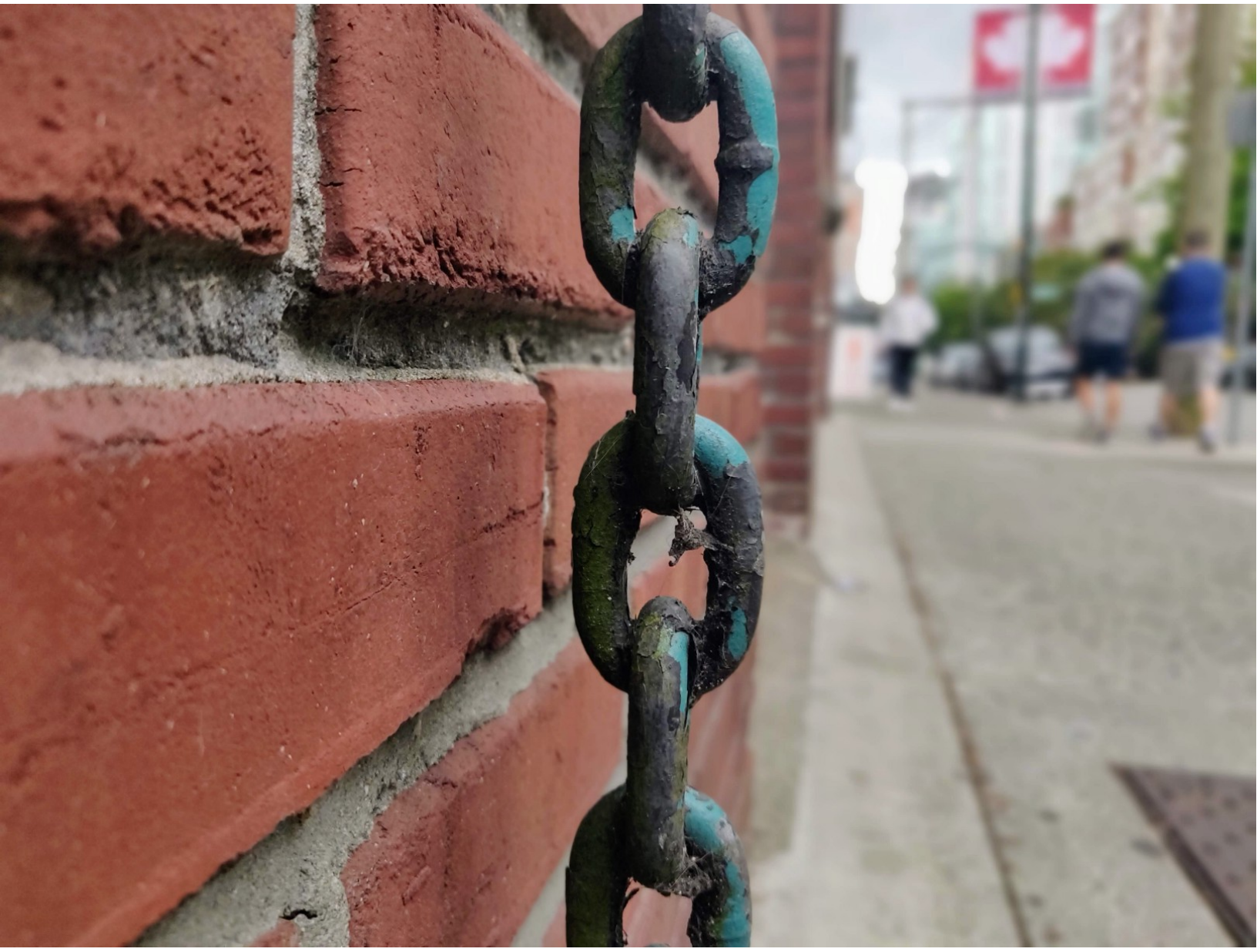
Dave Lunny

974 Followers · About Follow



Source: My own photo library 📸

Why I'm so jazzed about the Optional Chaining proposal in JavaScript 🎷

> *Edit: Since the time of writing, Optional Chaining has actually moved to Stage-3!*

Recently, the TC39 committee has finally moved the Optional Chaining proposal to `stage-2` 🎉

While most of these proposals are boring or too crazy or are way above my head, I am actually pretty jazzed about the optional chaining proposal because it's something that could actually see myself using on a regular basis.

In this article, I'm going to give a brief overview of the proposal process (for those who aren't yet familiar), then I'll show you what optional chaining is, and finally I'll get into some of the problems that it attempts to solve.

Proposal Process

The process by which the JavaScript language evolves starts with "champions" (folks who want to add/change something in the language) drafting a proposal document for what they want to add/change. Sometimes they make nice slides to help explain things.

Generally these proposals are somewhat boring and are written in a very technical lingo. Sadly they do not *nearly* contain enough emojis or gifs, at least in *my opinion*. ✌️👌💯

These proposals will make their way through these 5 stages:

- **Stage 0 (Strawperson)** - Just an idea; looking for initial feedback.
- **Stage 1 (Proposal)** - More fleshed out; has an API; has a "champion(s)" who wants to move it forward.
- **Stage 2 (Draft)** - More formalized spec (like this); likely has a Babel plugin to test out usage.
- **Stage 3 (Candidate)** - Basically done/finalizing; getting feedback from real users in the field.
- **Stage 4 (Finished)** - Done and is ready for acceptance into the ECMAScript standard.

> Source: *this boring (and CSS-lacking) TC39 document*.

Despite not being part of the language, you can usually start using these new features by transpiling your code with a tool such as Babel. Babel takes your "pre-Stage 4" code and

Some folks/teams prefer to play it safe by only using proposals which are probably going to actually make it into the language (Stage 3 or 4). This keeps source code closer to "real" JavaScript, and helps avoid the need for refactors later if the spec changes, as well as to protect against unstable APIs which haven't yet been tested in the wild.

Generally speaking though, once a proposal makes it to Stage 2, it's probably in pretty good shape, and will likely eventually be included in the language. Optional Chaining is at Stage 2 now, so I think it's time to discuss it. 🗣

Optional Chaining

> ⚠️ *Keep in mind that being a Stage 2 proposal, the API is subject to change slightly, and it is not standardized yet. Don't use without transpiling (see below).*

What is optional chaining? Let's dive right in with a simple example:

```
if (user.address?.street) {
  console.log('User has a street address!');
}
```

One of the things I like the most about optional chaining is readability. I bet without me even needing to elaborate any further, you can probably already guess what this is doing. But elaborate I shall…

Problem: Accessing deep nodes

The core problem that optional chaining is trying to solve is accessing deeply nested properties from within objects. Consider the following structure:

```
const foo = {
  bar: {
    baz: {
      isDeep: true
    }
  }
};
```

If you needed to check `isDeep`, intuition says you might try this:

However, if any of those intermediate nodes don't exist for some reason, then we're going to get a nasty TypeError like this:

```
TypeError: Cannot read property 'baz' of undefined
```

This forces us to do a bunch of defensive programming to ensure that we don't encounter these issues at runtime. Currently in JavaScript, to safely check for `isDeep`, you'd need do something like this:

```
if (foo && foo.bar && foo.bar.baz && foo.bar.baz.isDeep) {
  return "woah, so deep dude...";
}
```

Oof.

This works by checking each node, and if it's falsey or "nullish", exit early instead of trying to access those `undefined` properties.

This is a bit of a burden, both for the person writing it and for the person reading it. Generally, I'm not creating deeply nested structures like the one above, however you will inevitably still run into this issue when writing JavaScript. Especially if you need to deal with 3rd-party APIs, so let's look at a more realistic example:

```
fetch('https://www.reddit.com/r/aww.json')
  .then(res => res.json())
  .then(result => {
    //  Wow, this is annoying:
    if (result && result.data && result.data.children) {
      // Do something with result.data.children...
    }
  });
```

How does optional chaining solve this?

```
}
```

Wow, that's nice! 👌

This tells JavaScript "check if `result` exists, then check if `result.data` exists, then finally give me `result.data.children`." If any intermediate nodes do not exist, it will return `undefined`, thereby ending the property digging early.

You can also access dynamic properties, but only if the underlying property actually exists:

```
// Without optional chaining
foo && foo[bar] && foo[bar][baz];

// With optional chaining
foo?.[bar]?.[baz];
```

Problem: Method calls

What about calling methods that may or may not exist?

This example is a bit contrived, but imagine that you are using a library which is in beta, where the API is subject to change frequently. Maybe you want to check that a method exists before you call it, so as to not get an `undefined is not a function` error. Right now in JavaScript you'd do something like this:

```
betaLib.someMethod && betaLib.someMethod();
```

Or if you wanted to be *really* safe:

```
if (betaLib.someMethod && typeof betaLib.someMethod === 'function') {
  betaLib.someMethod();
}
```

Look how much nicer it is with optional chaining:

Okay okay, I'll admit that it *does* look a little weird when you're not used to it, but hey, arrow functions did looked a little weird at first too, and now look at how widespread they are.

Basically, all you need to remember here is to include the `.` before you invoke the function, on the right-hand side of the `?`.

How about a more practical example? Have you ever tried to call `.focus` a text `input` element which doesn't exist? (Of course you have, that was a rhetorical question…)

```
const input = document.querySelector("#my-input");
// ...
input.focus();
```

If that `input` DOM element doesn't exist, you're going to get an error. To do this safely, you'll need to wrap the `.focus` call in an `if` statement to see if it's a real element, then call focus.

With optional chaining, this looks so much nicer:

```
// Check if `input` exists, then call focus
input?.focus();

// OR if `input` exists, check if it has a
// method called "focus" and call if it does
input?.focus?.();
```

What optional chaining is not

The optional chaining proposal has specifically outlined some things which (at least at time of writing) will not be implemented. This is largely due to the fact that there isn't really a broad set of foreseeable use cases for them.

One of these is optional constructors, and another is optional template literals, and any mixing thereof, such as:

```
// Warning: these are not in the spec/do not use/will not work!
```

Another one which is not supported (but might be in the future) is optional property assignment:

```
// Warning: Again, not in the spec/won't work/don't use
foo?.bar = 'o p t i o n a l';

// Equivalent in current JS
if (foo) {
    foo.bar = 'o p t i o n a l'
}
```

Personally I actually kind of like this, and hope that they do include it in the final spec. While I do agree that the spec is fine without it, I still think it would round things out and make it feel more "complete".

Note that there are also further concerns, such as optional destructuring, which have been declared out of scope for this spec and will not likely be included. They could be included in a future spec, however.

Prior Art

Where did this idea come from? Well, like many aspects of computer science, optional chaining is not a new concept.

For example, C# has pretty much the exact same thing with what it calls "null-conditional operators":

```
foo?.bar?.baz
```

Many other languages have the same/similar features as well, including Ruby, Swift and Dart. However perhaps the biggest influence is taken from CoffeeScript (remember CoffeeScript?), which has a feature known as an "existential operator":

```
foo.bar?.baz?()
```

Can I use?

Remember to keep in mind that this is a proposal and is not actually valid in most environments, so technically it's not "real" JavaScript, and is subject to change before being moved to Stage 4.

You can start using optional chaining today by adding this Babel plugin:

```
yarn add -D @babel/plugin-syntax-optional-chaining

npm install -D @babel/plugin-syntax-optional-chaining
```

Update your Babel config file like so:

```
{
   "plugins": ["@babel/plugin-syntax-optional-chaining"]
}
```

…and boom, you're good to go. 💥💪

If you just want to mess around with optional chaining, you can use the Babel REPL and see what it gets transpiled out to (which should give you a better understanding of how it works).

Champions

- Claude Pache
- Gabriel Isenberg
- Dustin Savery

Kudos to the champions of this proposal, as well as all the TC39 folks for pushing this forward. I think it will make a great addition to the language and will be closely following this proposal in the coming months. In the meantime I would like to spread knowledge about it so that folks are less confounded if they come across it in the wild. I sure hope I start seeing it more!