

You Don't Know JS: Up & Going

Chapter 3: Into YDKJS

What is this series all about? Put simply, it's about taking seriously the task of learning *all parts of JavaScript*, not just some subset of the language that someone called "the good parts," and not just whatever minimal amount you need to get your job done at work.

Serious developers in other languages expect to put in the effort to learn most or all of the language(s) they primarily write in, but JS developers seem to stand out from the crowd in the sense of typically not learning very much of the language. This is not a good thing, and it's not something we should continue to allow to be the norm.

The *You Don't Know JS* (YDKJS) series stands in stark contrast to the typical approaches to learning JS, and is unlike almost any other JS books you will read. It challenges you to go beyond your comfort zone and to ask the deeper "why" questions for every single behavior you encounter. Are you up for that challenge?

I'm going to use this final chapter to briefly summarize what to expect from the rest of the books in the series, and how to most effectively go about building a foundation of JS learning on top of YDKJS.

Scope & Closures

Perhaps one of the most fundamental things you'll need to quickly come to terms with is how scoping of variables really works in JavaScript. It's not enough to have anecdotal fuzzy *beliefs* about scope.

The *Scope & Closures* title starts by debunking the common misconception that JS is an "interpreted language" and therefore not compiled. Nope.

The JS engine compiles your code right before (and sometimes during!) execution. So we use some deeper understanding of the compiler's approach to our code to understand how it finds and deals with variable and function declarations. Along the way, we see the typical metaphor for JS variable scope management, "Hoisting."

This critical understanding of "lexical scope" is what we then base our exploration of closure on for the last chapter of the book. Closure is perhaps the single most important concept in all of JS, but if you haven't first grasped firmly how scope works, closure will likely remain beyond your grasp.

One important application of closure is the module pattern, as we briefly introduced in this book in Chapter 2. The module pattern is perhaps the most prevalent code organization pattern in all of JavaScript; deep understanding of it should be one of your highest priorities.

this & Object Prototypes

Perhaps one of the most widespread and persistent mistruths about JavaScript is that the `this` keyword refers to the function it appears in. Terribly mistaken.

The `this` keyword is dynamically bound based on how the function in question is executed, and it turns out there are four simple rules to understand and fully determine `this` binding.

Closely related to the `this` keyword is the object prototype mechanism, which is a look-up chain for properties, similar to how lexical scope variables are found. But wrapped up in the prototypes is the other huge miscue about JS: the idea of emulating (fake) classes and (so-called "prototypal") inheritance.

Unfortunately, the desire to bring class and inheritance design pattern thinking to JavaScript is just about the worst thing you could try to do, because while the syntax may trick you into thinking there's something like classes present, in fact the prototype mechanism is fundamentally opposite in its behavior.

What's at issue is whether it's better to ignore the mismatch and pretend that what you're implementing is "inheritance," or whether it's more appropriate to learn and embrace how the object prototype system actually works. The latter is more appropriately named "behavior delegation."

This is more than syntactic preference. Delegation is an entirely different, and more powerful, design pattern, one that replaces the need to design with classes and inheritance. But these assertions will absolutely fly in the face of nearly every other blog post, book, and conference talk on the subject for the entirety of JavaScript's lifetime.

The claims I make regarding delegation versus inheritance come not from a dislike of the language and its syntax, but from the desire to see the true capability of the language properly leveraged and the endless confusion and frustration wiped away.

But the case I make regarding prototypes and delegation is a much more involved one than what I will indulge here. If you're ready to reconsider everything you think you know about JavaScript "classes" and "inheritance," I offer you the chance to "take the red pill" (*Matrix* 1999) and check out Chapters 4-6 of the *this & Object Prototypes* title of this series.

Types & Grammar

The third title in this series primarily focuses on tackling yet another highly controversial topic: type coercion. Perhaps no topic causes more frustration with JS developers than when you talk about the confusions surrounding implicit coercion.

By far, the conventional wisdom is that implicit coercion is a "bad part" of the language and should be avoided at all costs. In fact, some have gone so far as to call it a "flaw" in the design of the language. Indeed, there are tools whose entire job is to do nothing but scan your code and complain if you're doing anything even remotely like coercion.

But is coercion really so confusing, so bad, so treacherous, that your code is doomed from the start if you use it?

I say no. After having built up an understanding of how types and values really work in Chapters 1-3, Chapter 4 takes on this debate and fully explains how coercion works, in all its nooks and crevices. We see just what parts of coercion really are surprising and what parts actually make complete sense if given the time to learn.

But I'm not merely suggesting that coercion is sensible and learnable, I'm asserting that coercion is an incredibly useful and totally underestimated tool that *you should be using in your code*. I'm saying that coercion, when used properly, not only works, but makes your code better. All the naysayers and doubters will surely scoff at such a position, but I believe it's one of the main keys to upping your JS game.

Do you want to just keep following what the crowd says, or are you willing to set all the assumptions aside and look at coercion with a fresh perspective? The *Types & Grammar* title of this series will coerce your thinking.

Async & Performance

The first three titles of this series focus on the core mechanics of the language, but the fourth title branches out slightly to cover patterns on top of the language mechanics for managing asynchronous programming. Asynchrony is not only critical to the performance of our applications, it's increasingly becoming *the* critical factor in writability and maintainability.

The book starts first by clearing up a lot of terminology and concept confusion around things like "async," "parallel," and "concurrent," and explains in depth how such things do and do not apply to JS.

Then we move into examining callbacks as the primary method of enabling asynchrony. But it's here that we quickly see that the callback alone is hopelessly insufficient for the modern demands of asynchronous programming. We identify two major deficiencies of callbacks-only coding: *Inversion of Control* (IoC) trust loss and lack of linear reason-ability.

To address these two major deficiencies, ES6 introduces two new mechanisms (and indeed, patterns): promises and generators.

Promises are a time-independent wrapper around a "future value," which lets you reason about and compose them regardless of if the value is ready or not yet. Moreover, they effectively solve the IoC trust issues by routing callbacks through a trustable and composable promise mechanism.

Generators introduce a new mode of execution for JS functions, whereby the generator can be paused at `yield` points and be resumed asynchronously later. The pause-and-resume capability enables synchronous, sequential looking code in the generator to be processed asynchronously behind the scenes. By doing so, we address the non-linear, non-local-jump confusions of callbacks and thereby make our asynchronous code sync-looking so as to be more reason-able.

But it's the combination of promises and generators that "yields" our most effective asynchronous coding pattern to date in JavaScript. In fact, much of the future sophistication of asynchrony coming in ES7 and later will certainly be built on this foundation. To be serious about programming effectively in an async world, you're going to need to get really comfortable with combining promises and generators.

If promises and generators are about expressing patterns that let our programs run more concurrently and thus get more processing accomplished in a shorter period, JS has many other facets of performance optimization worth exploring.

Chapter 5 delves into topics like program parallelism with Web Workers and data parallelism with SIMD, as well as low-level optimization techniques like ASM.js. Chapter 6 takes a look at performance optimization from the perspective of proper benchmarking techniques, including what kinds of performance to worry about and what to ignore.

Writing JavaScript effectively means writing code that can break the constraint barriers of being run dynamically in a wide range of browsers and other environments. It requires a lot of intricate and detailed planning and effort on our parts to take a program from "it works" to "it works well."

The *Async & Performance* title is designed to give you all the tools and skills you need to write reasonable and performant JavaScript code.

ES6 & Beyond

No matter how much you feel you've mastered JavaScript to this point, the truth is that JavaScript is never going to stop evolving, and moreover, the rate of evolution is increasing rapidly. This fact is almost a metaphor for the spirit of this series, to embrace that we'll never fully *know* every part of JS, because as soon as you master it all, there's going to be new stuff coming down the line that you'll need to learn.

This title is dedicated to both the short- and mid-term visions of where the language is headed, not just the *known* stuff like ES6 but the *likely* stuff beyond.

While all the titles of this series embrace the state of JavaScript at the time of this writing, which is mid-way through ES6 adoption, the primary focus in the series has been more on ES5. Now, we want to turn our attention to ES6, ES7, and ...

Since ES6 is nearly complete at the time of this writing, *ES6 & Beyond* starts by dividing up the concrete stuff from the ES6 landscape into several key categories, including new syntax, new data structures (collections), and new processing capabilities and APIs. We cover each of these new ES6 features, in varying levels of detail, including reviewing details that are touched on in other books of this series.

Some exciting ES6 things to look forward to reading about: destructuring, default parameter values, symbols, concise methods, computed properties, arrow functions, block scoping, promises, generators, iterators, modules, proxies, weakmaps, and much, much more! Phew, ES6 packs quite a punch!

The first part of the book is a roadmap for all the stuff you need to learn to get ready for the new and improved JavaScript you'll be writing and exploring over the next couple of years.

The latter part of the book turns attention to briefly glance at things that we can likely expect to see in the near future of JavaScript. The most important realization here is that post-ES6, JS is likely going to evolve feature by feature rather than version by version, which means we can expect to see these near-future things coming much sooner than you might imagine.

The future for JavaScript is bright. Isn't it time we start learning it!?

Review

The *YDKJS* series is dedicated to the proposition that all JS developers can and should learn all of the parts of this great language. No person's opinion, no framework's assumptions, and no project's deadline should be the excuse for why you never learn and deeply understand JavaScript.

We take each important area of focus in the language and dedicate a short but very dense book to fully explore all the parts of it that you perhaps thought you knew but probably didn't fully.

"You Don't Know JS" isn't a criticism or an insult. It's a realization that all of us, myself included, must come to terms with. Learning JavaScript isn't an end goal but a process. We don't know JavaScript, yet. But we will!