You Don't Know JS Yet: Get Started - 2nd Edition

Chapter 4: The Bigger Picture

This book surveys what you need to be aware of as you *get started* with JS. The goal is to fill in gaps that readers newer to JS might have tripped over in their early encounters with the language. I also hope that we've hinted at enough deeper detail throughout to pique your curiosity to want to dig more into the language.

The rest of the books in this series are where we will unpack all of the rest of the language, in far greater detail than we could have done in a few brief chapters here.

Remember to take your time, though. Rather than rushing onto the next book in an attempt to churn through all the books expediently, spend some time going back over the material in this book. Spend some more time looking through code in your current projects, and comparing what you see to what's been discussed so far.

When you're ready, this final chapter divides the organization of the JS language into three main pillars, then offers a brief roadmap of what to expect from the rest of the book series, and how I suggest you proceed. Also, don't skip the appendices, especially Appendix B, "Practice, Practice, Practice!".

Pillar 1: Scope and Closure

The organization of variables into units of scope (functions, blocks) is one of the most foundational characteristics of any language; perhaps no other characteristic has a greater impact on how programs behave.

Scopes are like buckets, and variables are like marbles you put into those buckets. The scope model of a language is like the rules that help you determine which color marbles go in which matching-color buckets.

Scopes nest inside each other, and for any given expression or statement, only variables at that level of scope nesting, or in higher/outer scopes, are accessible; variables from lower/inner scopes are hidden and inaccessible.

This is how scopes behave in most languages, which is called lexical scope. The scope unit boundaries, and how variables are organized in them, is determined at the time the program is parsed (compiled). In other words, it's an author-time decision: where you locate a function/scope in the program determines what the scope structure of that part of the program will be.

JS is lexically scoped, though many claim it isn't, because of two particular characteristics of its model that are not present in other lexically scoped languages.

The first is commonly called *hoisting*: when all variables declared anywhere in a scope are treated as if they're declared at the beginning of the scope. The other is that `var` declared variables are function scoped, even if they appear inside a block.

Neither hoisting nor function-scoped `var` are sufficient to back the claim that JS is not lexically scoped. `let` / `const` declarations have a peculiar error behavior called the "Temporal Dead Zone" (TDZ) which results in observable but unusable variables. Though TDZ can be strange to encounter, it's *also* not an invalidation of lexical scoping. All of these are just unique parts of the language that should be learned and understood by all JS developers.

Closure is a natural result of lexical scope when the language has functions as first-class values, as JS does. When a function makes reference to variables from an outer scope, and that function is passed around as a value and executed in other scopes, it maintains access to its original scope variables; this is closure.

Across all of programming, but especially in JS, closure drives many of the most important programming patterns, including modules. As I see it, modules are as *with the grain* as you can get, when it comes to code organization in JS.

To dig further into scope, closures, and how modules work, read Book 2, *Scope & Closures*.

Pillar 2: Prototypes

The second pillar of the language is the prototypes system. We covered this topic in-depth in Chapter 3 ("Prototypes"), but I just want to make a few more comments about its importance.

JS is one of very few languages where you have the option to create objects directly and explicitly, without first defining their structure in a class.

For many years, people implemented the class design pattern on top of prototypes – so called, "prototypal inheritance" (see Appendix A) – and then with the advent of ES6's `class` keyword, the language doubled-down on its inclination towards OO/class style programming.

But I think that focus has obscured the beauty and power of the prototype system: the ability for two objects to simply connect with each other and cooperate dynamically (during function/method execution) through sharing a `this` context.

Classes are just one pattern you can build on top of such power. But another approach, in a very different direction, is to simply embrace objects as objects, forget classes altogether, and let objects cooperate through the prototype chain.

This is called *behavior delegation*. I think delegation is more powerful than class inheritance, as a means for organizing behavior and data in our programs.

But class inheritance gets almost all the attention. And the rest goes to functional programming (FP), as the sort of "anti-class" way of designing programs. This saddens me, because it snuffs out any chance for exploration of delegation as a viable alternative.

I encourage you to spend plenty of time deep in Book 3, *Objects & Classes*, to see how object delegation holds far more potential than we've perhaps realized. This isn't an anti-`class` message, but it is intentionally a "classes aren't the only way to use objects" message that I want more JS developers to consider.

Object delegation is, I would argue, far more *with the grain* of JS, than classes (more on *grains* in a bit).

Pillar 3: Types and Coercion

The third pillar of JS is by far the most overlooked part of JS's nature.

The vast majority of developers have strong misconceptions about how *types* work in programming languages, and especially how they work in JS. A tidal wave of interest in the broader JS community has begun to shift to "static typing" approaches, using type-aware tooling like TypeScript or Flow.

I agree that JS developers should learn more about types, and should learn more about how JS manages type conversions. I also agree that type-aware tooling can help developers, assuming they have gained and used this knowledge in the first place!

But I don't agree at all that the inevitable conclusion of this is to decide JS's type mechanism is bad and that we need to cover up JS's types with solutions outside the language. We don't have to follow the "static typing" way to be smart and solid with types in our programs. There are other options, if you're just willing to go *against the grain* of the crowd, and *with the grain* of JS (again, more on that below).

Arguably, this pillar is more important than the other two, in the sense that no JS program will do anything useful if it doesn't properly leverage JS's value types, as well as the conversion (coercion) of values between types.

Even if you love TypeScript / Flow, you are not going to get the most out of those tools or coding approaches if you aren't deeply familiar with how the language itself manages value types.

To learn more about JS types and coercion, check out Book 4, *Types & Grammar*. But please don't skip over this topic just because you've always heard that we should use `===` and forget about the rest.

Without learning this pillar, your foundation in JS is shaky and incomplete at best.

With The Grain

I have some advice to share on continuing your learning journey with JS, and your path through the rest of this book series: be aware of the *grain* – recall various references to *grain* earlier in this chapter.

First, consider the *grain* (as in, wood) of how most people approach and use JS. You've probably already noticed that these books cut against that *grain* in many respects. In YDKJSY, I respect you the reader enough to explain all the parts of JS, not only some select popular parts. I believe you're both capable and deserving of that knowledge.

But that is not what you'll find from a lot of other material out there. It also means that the more you follow and adhere to the guidance from these books – that you think carefully and analyze for yourself what's best in your code – the more you will stand out. That can be a good and bad thing. If you ever want to break out from the crowd, you're going to have to break from how the crowd does it!

But I've also had many people tell me that they quoted some topic/explanation from these books during a job interview, and the interviewer told the candidate they were wrong; indeed, people have reportedly lost out on job offers as a result.

As much as possible, I endeavor in these books to provide completely accurate information about JS, informed generally from the specification itself. But I also dose out quite a bit of my opinions on how you can interpret and use JS to the best benefit in your programs. I don't present opinion as fact, or vice versa. You'll always know which is which in these books.

Facts about JS are not really up for debate. Either the specification says something, or it doesn't. If you don't like what the specification says, or my relaying of it, take that up with TC39! If you're in an interview and they claim you're wrong on the facts, ask them right then and there if you can look it up in the specification. If the interviewer won't re-consider, then you shouldn't want to work there anyway.

But if you choose to align with my opinions, you have to be prepared to back up those choices with *why* you feel that way. Don't just parrot what I say. Own your opinions. Defend them. And if someone you were hoping to work with disagrees, walk away with your head still held high. It's a big JS, and there's plenty of room for lots of different ways.

In other words, don't be afraid to go against the *grain*, as I have done with these books and all my teachings. Nobody can tell you how you will best make use of JS; that's for you to decide. I'm merely trying to empower you in coming to your own conclusions, no matter what they are.

On the other hand, there's a *grain* you really should pay attention to and follow: the *grain* of how JS works, at the language level. There are things that work

well and naturally in JS, given the right practice and approach, and there are things you really shouldn't try to do in the language.

Can you make your JS program look like a Java, C#, or Perl program? What about Python or Ruby, or even PHP? To varying degrees, sure you can. But should you?

No, I don't think you should. I think you should learn and embrace the JS way, and make your JS programs as JS'y as is practical. Some will think that means sloppy and informal programming, but I don't mean that at all. I just mean that JS has a lot of patterns and idioms that are recognizably "JS", and going with that *grain* is the general path to best success.

Finally, maybe the most important *grain* to recognize is how the existing program(s) you're working on, and developers you're working with, do stuff. Don't read these books and then try to change *all that grain* in your existing projects over night. That approach will always fail.

You'll have to shift these things little by little, over time. Work on building consensus with your fellow developers on why it's important to re-visit and re-consider an approach. But do so with just one small topic at a time, and let before-and-after code comparisons do most of the talking. Bring everyone on the team together to discuss, and push for decisions that are based on analysis and evidence from the code rather than the inertia of, "our senior devs have always done it this way".

That's the most important advice I can impart to help you learn JS. Always keep looking for better ways to use what JS gives us to author more readable code. Everyone who works on your code, including your future self, will thank you!

In Order

So now you've got a broader perspective on what's left to explore in JS, and the right attitude to approach the rest of your journey.

But one of the most common practical questions I get at this point is, "What order should I read the books?" There is a straightforward answer... but it also depends.

My suggestion for most readers is to proceed through this series like in this order:

1. Get started with a solid foundation of JS from *Get Started* (Book 1) – good news, you've already almost finished this book!
2. In *Scope & Closures* (Book 2), dig into the first pillar of JS: lexical scope, how that supports closure, and how the module pattern organizes code.

3. In *Objects & Classes* (Book 3), focus on the second pillar of JS: how JS's `this` works, how object prototypes support delegation, and how prototypes enable the `class` mechanism for OO-style code organization.

4. In *Types & Grammar* (Book 4), tackle the third and final pillar of JS: types and type coercion, as well as how JS's syntax and grammar define how we write our code.

5. With the **three pillars** solidly in place, *Sync & Async* (Book 5) then explores how we use flow control to model state change in our programs, both synchronously (right away) and asynchronously (over time).

6. The series concludes with *ES.Next & Beyond* (Book 6), a forward look at the near- and mid-term future of JS, including a variety of features likely coming to your JS programs before too long.

That's the intended order to read this book series.

However, books 2, 3, and 4 can generally be read in any order, depending on which topic you feel most curious about and comfortable exploring first. But I don't recommend you skip any of these three books – not even *Types & Grammar*, as some of you will be tempted to do! – even if you think you already have that topic down.

Book 5 (*Sync & Async*) is crucial for deeply understanding JS, but if you start digging in and find it's too intimidating, this book can be deferred until you're more experienced with the language. The more JS you've written (and struggled with!), the more you'll come to appreciate this book. So don't be afraid to come back to it at a later time.

The final book in the series, *ES.Next & Beyond*, in some respects stands alone. It can be read at the end, as I suggest, or right after *Getting Started* if you're looking for a shortcut to broaden your radar of what JS is all about. This book will also be more likely to receive updates in the future, so you'll probably want to re-visit it occasionally.

However you choose to proceed with YDKJSY, check out the appendices of this book first, especially practicing the snippets in Appendix B, "Practice, Practice, Practice!" Did I mention you should go practice!? There's no better way to learn code than to write it.