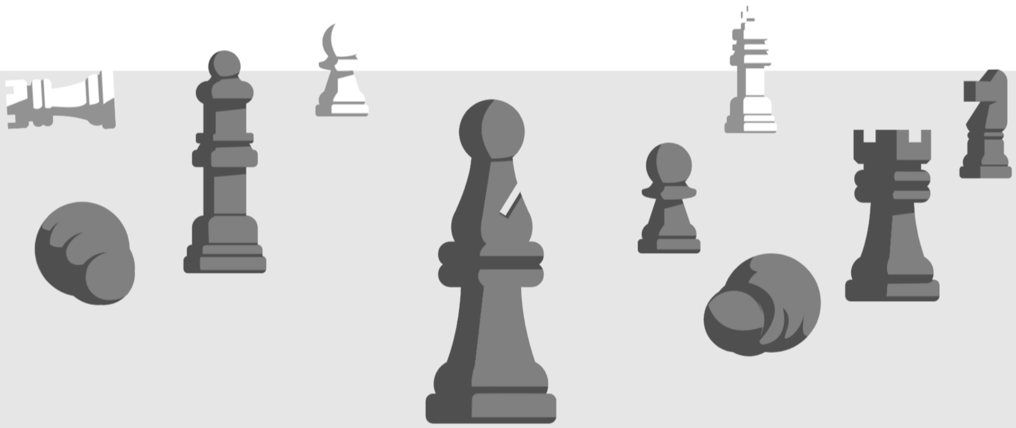
SECOND EDITION

YOU DON'T KNOW JS YET

SCOPE & CLOSURES



Frontend Masters

Kyle Simpson

You Don’t Know JS Yet: Scope & Closures

Kyle Simpson

This book is for sale at
http://leanpub.com/ydkjsy-scope-closures

This version was published on 2020-04-27

ISBN 978-1-64786-221-3

Published by GetiPub (http://getipub.com), a division of Getify Solutions, Inc., and produced by Leanpub (https://leanpub.com/fljs).

Tweet This Book!

Please help Kyle Simpson by spreading the word about this book on Twitter!

The suggested tweet for this book is:

#YDKJSYet Still getting to know JS more deeply… Now reading ”Scope & Closures”, the second book in @YDKJS 2nd Edition series!
https://leanpub.com/ydkjsy-scope-closures +@YDKJSY

The suggested hashtag for this book is #YDKJSYet.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#YDKJSYet

Published by GetiPub (http://getipub.com), a division of Getify Solutions, Inc., and produced by Leanpub (https://leanpub.com/fljs).

Editor: Simon St.Laurent Copy Editor: Jasmine Kwityn

Cover Art: David Neal (@reverentgeek)

March 2020: Second Edition

Revision History for the Second Edition

2020-03-03: First Release

I must first thank my wife and kids, whose constant support is what allows me to keep going. I also want to thank the 500 original backers of the Kickstarter for "You Don't Know JS" (1st ed), as well as the hundreds of thousands of folks who bought and read those books since. Without your financial support, this second edition wouldn't be happening. Thanks also to the interviewer at a certain avian social media company who said I didn't "know enough about JS"... you helped me name the series.

Next, I owe much of my current career path to Marc Grabanski and Frontend Masters. Marc took a chance on me and gave me my first shot at teaching years ago, and I wouldn't have then become a writer had it not been for that! Frontend Masters is the Premier Sponsor of YDKJSY 2nd Edition. Thank you, Frontend Masters (and Marc).

Lastly, my editor, Simon St.Laurent, who helped me conceive the original YDKJS and was my first book editor. Simon's support and guidance have profoundly impacted me and been an integral part of shaping me into the writer I am today. From those drinks we enjoyed at the Driskill all those years back, where YDKJS was born, through today, thank you so much Simon for shepherding and improving these books!

Contents

Preface

Welcome to the 2nd edition of the widely acclaimed *You Don't Know JS* (**YDKJS**) book series: *You Don't Know JS **Yet*** (**YDKJSY**).

If you've read any of the 1st edition books, you can expect a refreshed approach in these new ones, with plenty of updated coverage of what's changed in JS over the last five years. But what I hope and believe you'll still *get* is the same commitment to respecting JS and digging into what really makes it tick.

If this is your first time reading these books, I'm glad you're here. Prepare for a deep and extensive journey into all the corners of JavaScript.

If you are new to programming or JS, be aware that these books are not intended as a gentle "intro to JavaScript." This material is, at times, complex and challenging, and goes much deeper than is typical for a first-time learner. You're welcome here no matter what your background is, but these books are written assuming you're already comfortable with JS and have at least 6–9 months experience with it.

The Parts

These books approach JavaScript intentionally opposite of how *The Good Parts* treats the language. No, that doesn't mean we're looking at *the bad parts*, but rather, exploring **all the parts**.

You may have been told, or felt yourself, that JS is a deeply flawed language that was poorly designed and inconsistently implemented. Many have asserted that it's the worst most popular language in the world; that nobody writes JS because they want to, only because they have to given its place at the center of the web. That's a ridiculous, unhealthy, and wholly condescending claim.

Millions of developers write JavaScript every day, and many of them appreciate and respect the language.

Like any great language, it has its brilliant parts as well as its scars. Even the creator of JavaScript himself, Brendan Eich, laments some of those parts as mistakes. But he's wrong: they weren't mistakes at all. JS is what it is today—the world's most ubiquitous and thus most influential programming language—precisely because of *all those parts*.

Don't buy the lie that you should only learn and use a small collection of *good parts* while avoiding all the bad stuff. Don't buy the "X is the new Y" snake oil, that some new feature of the language instantly relegates all usage of a previous feature as obsolete and ignorant. Don't listen when someone says your code isn't "modern" because it isn't yet using a stage-0 feature that was only proposed a few weeks ago!

Every part of JS is useful. Some parts are more useful than others. Some parts require you to be more careful and intentional.

I find it absurd to try to be a truly effective JavaScript developer while only using a small sliver of what the language has to offer. Can you imagine a construction worker with a toolbox full of tools, who only uses their hammer and scoffs at the screwdriver or tape measure as inferior? That's just silly.

My unreserved claim is that you should go about learning all parts of JavaScript, and where appropriate, use them! And if I may be so bold as to suggest: it's time to discard any JS books that tell you otherwise.

The Title?

So what's the title of the series all about?

I'm not trying to insult you with criticism about your current lack of knowledge or understanding of JavaScript. I'm not suggesting you can't or won't be able to learn JavaScript. I'm not boasting about secret advanced insider wisdom that I and only a select few possess.

Seriously, all those were real reactions to the original series title before folks even read the books. And they're baseless.

The primary point of the title "You Don't Know JS Yet" is to point out that most JS developers don't take the time to really understand how the code that they write works. They know *that* it works—that it produces a desired outcome. But they either don't understand exactly *how*, or worse, they have an inaccurate mental model for the *how* that falters on closer scrutiny.

I'm presenting a gentle but earnest challenge to you the reader, to set aside the assumptions you have about JS, and approach it with fresh eyes and an invigorated curiosity that leads you to ask *why* for every line of code you write. Why does it do what it does? Why is one way better or more appropriate than the other half-dozen ways you could have accomplished it? Why do all the “popular kids” say to do X with your code, but it turns out that Y might be a better choice?

I added “Yet” to the title, not only because it’s the second edition, but because ultimately I want these books to challenge you in a hopeful rather than discouraging way.

But let me be clear: I don’t think it’s possible to ever fully *know* JS. That’s not an achievement to be obtained, but a goal to strive after. You don’t finish knowing everything about JS, you just keep learning more and more as you spend more time with the language. And the deeper you go, the more you revisit what you *knew* before, and you re-learn it from that more experienced perspective.

I encourage you to adopt a mindset around JavaScript, and indeed all of software development, that you will never fully have mastered it, but that you can and should keep working to get closer to that end, a journey that will stretch for the entirety of your software development career, and beyond.

You can always know JS better than you currently do. That’s what I hope these YDKJSY books represent.

The Mission

The case doesn’t really need to be made for why developers should take JS seriously—I think it’s already more than proven worthy of first-class status among the world’s programming languages.

But a different, more important case still needs to be made, and these books rise to that challenge.

I've taught more than 5,000 developers from teams and companies all over the world, in more than 25 countries on six continents. And what I've seen is that far too often, what *counts* is generally just the result of the program, not how the program is written or how/why it works.

My experience not only as a developer but in teaching many other developers tells me: you will always be more effective in your development work if you more completely understand how your code works than you are solely *just* getting it to produce a desired outcome.

In other words, *good enough to work* is not, and should not be, *good enough*.

All developers regularly struggle with some piece of code not working correctly, and they can't figure out why. But far too often, JS developers will blame this on the language rather than admitting it's their own understanding that is falling short. These books serve as both the question and answer: why did it do *this*, and here's how to get it to do *that* instead.

My mission with YDKJSY is to empower every single JS developer to fully own the code they write, to understand it and to write with intention and clarity.

The Path

Some of you have started reading this book with the goal of completing all six books, back to back.

I would like to caution you to consider changing that plan.

It is not my intention that YDKJSY be read straight through. The material in these books is dense, because JavaScript is powerful, sophisticated, and in parts rather complex. Nobody can really hope to *download* all this information to their brains in a single pass and retain any significant amount of it. That's unreasonable, and it's foolish to try.

My suggestion is you take your time going through YDKJSY. Take one chapter, read it completely through start to finish, and then go back and re-read it section by section. Stop in between each section, and practice the code or ideas from that section. For larger concepts, it probably is a good idea to expect to spend several days digesting, re-reading, practicing, then digesting some more.

You could spend a week or two on each chapter, and a month or two on each book, and a year or more on the whole series, and you would still not be squeezing every ounce of YDKJSY out.

Don't binge these books; be patient and spread out your reading. Interleave reading with lots of practice on real code in your job or on projects you participate in. Wrestle with the opinions I've presented along the way, debate with others, and most of all, disagree with me! Run a study group or book club. Teach mini-workshops at your office. Write blog posts on what you've learned. Speak about these topics at local JS meetups.

It's never my goal to convince you to agree with my opinion, but to encourage you to own and be able to defend your opinions. You can't get *there* with an expedient read-through of these books. That's something that takes a long while to emerge, little by little, as you study and ponder and re-visit.

These books are meant to be a field-guide on your wanderings through JavaScript, from wherever you currently are with the language, to a place of deeper understanding. And the deeper you understand JS, the more questions you will ask and the more you will have to explore! That's what I find so exciting!

I'm so glad you're embarking on this journey, and I am so honored you would consider and consult these books along the way. It's time to start *getting to know JS*.

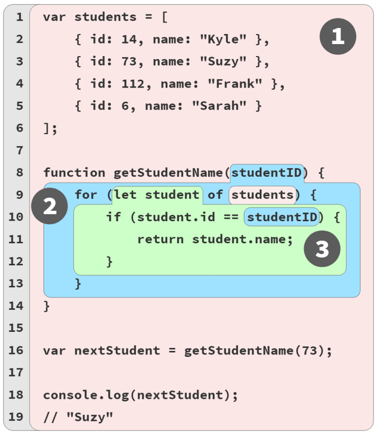
Chapter 3: The Scope Chain

Chapters 1 and 2 laid down a concrete definition of *lexical scope* (and its parts) and illustrated helpful metaphors for its conceptual foundation. Before proceeding with this chapter, find someone else to explain (written or aloud), in your own words, what lexical scope is and why it's useful to understand.

That seems like a step you might skip, but I've found it really does help to take the time to reformulate these ideas as explanations to others. That helps our brains digest what we're learning!

Now it's time to dig into the nuts and bolts, so expect that things will get a lot more detailed from here forward. Stick with it, though, because these discussions really hammer home just how much we all *don't know* about scope, yet. Make sure to take your time with the text and all the code snippets provided.

To refresh the context of our running example, let's recall the color-coded illustration of the nested scope bubbles, from Chapter 2, Figure 2:



```js
var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" }
];

function getStudentName(studentID) {
    for (let student of students) {
        if (student.id == studentID) {
            return student.name;
        }
    }
}

var nextStudent = getStudentName(73);

console.log(nextStudent);
// "Suzy"
```

Fig. 2 (Ch. 2): Colored Scope Bubbles

The connections between scopes that are nested within other scopes is called the scope chain, which determines the path along which variables can be accessed. The chain is directed, meaning the lookup moves upward/outward only.

"Lookup" Is (Mostly) Conceptual

In Figure 2, notice the color of the `students` variable reference in the `for`-loop. How exactly did we determine that it's a RED(1) marble?

In Chapter 2, we described the runtime access of a variable as a "lookup," where the *Engine* has to start by asking the current scope's *Scope Manager* if it knows about an identifier/variable, and proceeding upward/outward back through the chain of nested scopes (toward the global scope) until found, if ever. The lookup stops as soon as the first matching named declaration in a scope bucket is found.

The lookup process thus determined that `students` is a RED(1) marble, because we had not yet found a matching variable name as we traversed the scope chain, until we arrived at the final RED(1) global scope.

Similarly, `studentID` in the `if`-statement is determined to be a BLUE(2) marble.

This suggestion of a runtime lookup process works well for conceptual understanding, but it's not actually how things usually work in practice.

The color of a marble's bucket (aka, meta information of what scope a variable originates from) is *usually determined* during the initial compilation processing. Because lexical scope is pretty much finalized at that point, a marble's color will not change based on anything that can happen later during runtime.

Since the marble's color is known from compilation, and it's immutable, this information would likely be stored with (or at least accessible from) each variable's entry in the AST; that information is then used explicitly by the executable instructions that constitute the program's runtime.

In other words, *Engine* (from Chapter 2) doesn't need to lookup through a bunch of scopes to figure out which scope bucket a variable comes from. That information is already known! Avoiding the need for a runtime lookup is a key optimization benefit of lexical scope. The runtime operates more performantly without spending time on all these lookups.

But I said "…usually determined…" just a moment ago, with respect to figuring out a marble's color during compilation. So in what case would it ever *not* be known during compilation?

Consider a reference to a variable that isn't declared in any lexically available scopes in the current file—see *Get Started*, Chapter 1, which asserts that each file is its own separate program from the perspective of JS compilation. If no declaration is found, that's not *necessarily* an error. Another file (program) in the runtime may indeed declare that variable in the shared global scope.

So the ultimate determination of whether the variable was ever appropriately declared in some accessible bucket may need to be deferred to the runtime.

Any reference to a variable that's initially *undeclared* is left as an uncolored marble during that file's compilation; this color cannot be determined until other relevant file(s) have been compiled and the application runtime commences. That deferred lookup will eventually resolve the color to whichever scope the variable is found in (likely the global scope).

However, this lookup would only be needed once per variable at most, since nothing else during runtime could later change that marble's color.

The "Lookup Failures" section in Chapter 2 covers what happens if a marble is ultimately still uncolored at the moment its reference is runtime executed.

Shadowing

"Shadowing" might sound mysterious and a little bit sketchy. But don't worry, it's completely legit!

Our running example for these chapters uses different variable names across the scope boundaries. Since they all have unique names, in a way it wouldn't matter if all of them were just stored in one bucket (like RED(1)).

Where having different lexical scope buckets starts to matter more is when you have two or more variables, each in different scopes, with the same lexical names. A single scope cannot have two or more variables with the same name; such multiple references would be assumed as just one variable.

So if you need to maintain two or more variables of the same name, you must use separate (often nested) scopes. And in that case, it's very relevant how the different scope buckets are laid out.

Consider:

```
var studentName = "Suzy";

function printStudent(studentName) {
    studentName = studentName.toUpperCase();
    console.log(studentName);
}

printStudent("Frank");
// FRANK

printStudent(studentName);
// SUZY

console.log(studentName);
// Suzy
```

Tip

Before you move on, take some time to analyze this code using the various techniques/metaphors we've covered in the book. In particular, make sure to identify the marble/bubble colors in this snippet. It's good practice!

The `studentName` variable on line 1 (the `var studentName = ..` statement) creates a RED(1) marble. The same named variable is declared as a BLUE(2) marble on line 3, the parameter in the `printStudent(..)` function definition.

What color marble will `studentName` be in the `studentName = studentName.toUpperCase()` assignment statement and the `console.log(studentName)` statement? All three `studentName` references will be BLUE(2).

With the conceptual notion of the "lookup," we asserted that it starts with the current scope and works its way outward/upward, stopping as soon as a matching variable is found. The BLUE(2) `studentName` is found right away. The RED(1) `studentName` is never even considered.

This is a key aspect of lexical scope behavior, called *shadowing*. The BLUE(2) `studentName` variable (parameter) shadows the RED(1) `studentName`. So, the parameter is shadowing the (shadowed) global variable. Repeat that sentence to yourself a few times to make sure you have the terminology straight!

That's why the re-assignment of `studentName` affects only the inner (parameter) variable: the BLUE(2) `studentName`, not the global RED(1) `studentName`.

When you choose to shadow a variable from an outer scope, one direct impact is that from that scope inward/downward (through any nested scopes) it's now impossible for any marble to be colored as the shadowed variable—(RED(1), in this case). In other words, any `studentName` identifier reference will correspond to that parameter variable, never the global `studentName` variable. It's lexically impossible to reference the global `studentName` anywhere inside of the `printStudent(..)` function (or from any nested scopes).

Global Unshadowing Trick

Please beware: leveraging the technique I'm about to describe is not very good practice, as it's limited in utility, confusing for readers of your code, and likely to invite bugs to your program. I'm covering it only because you may run across this behavior in existing programs, and understanding what's happening is critical to not getting tripped up.

It *is* possible to access a global variable from a scope where that variable has been shadowed, but not through a typical lexical identifier reference.

In the global scope (RED(1)), `var` declarations and `function` declarations also expose themselves as properties (of the same name as the identifier) on the *global object*—essentially an object representation of the global scope. If you've written JS for a browser environment, you probably recognize the global object as `window`. That's not *entirely* accurate, but it's good enough for our discussion. In the next chapter, we'll explore the global scope/object topic more.

Consider this program, specifically executed as a standalone .js file in a browser environment:

```
var studentName = "Suzy";

function printStudent(studentName) {
    console.log(studentName);
    console.log(window.studentName);
}

printStudent("Frank");
// "Frank"
// "Suzy"
```

Notice the `window.studentName` reference? This expression is accessing the global variable `studentName` as a property on `window` (which we're pretending for now is synonymous with the global object). That's the only way to access a shadowed variable from inside a scope where the shadowing variable is present.

The `window.studentName` is a mirror of the global `studentName` variable, not a separate snapshot copy. Changes to one are still seen from the other, in either direction. You can think of `window.studentName` as a getter/setter that accesses the actual `studentName` variable. As a matter of fact, you can even *add* a variable to the global scope by creating/setting a property on the global object.

Warning

Remember: just because you *can* doesn't mean you *should*. Don't shadow a global variable that you need to access, and conversely, avoid using this trick to access a global variable that you've shadowed. And definitely don't confuse readers of your code by creating global variables as `window` properties instead of with formal declarations!

This little "trick" only works for accessing a global scope variable (not a shadowed variable from a nested scope), and even then, only one that was declared with `var` or `function`.

Other forms of global scope declarations do not create mirrored global object properties:

```
var one = 1;
let notOne = 2;
const notTwo = 3;
class notThree {}

console.log(window.one);       // 1
console.log(window.notOne);    // undefined
console.log(window.notTwo);    // undefined
console.log(window.notThree);  // undefined
```

Variables (no matter how they're declared!) that exist in any other scope than the global scope are completely inaccessible from a scope where they've been shadowed:

```
var special = 42;

function lookingFor(special) {
    // The identifier `special` (parameter) in this
    // scope is shadowed inside keepLooking(), and
    // is thus inaccessible from that scope.

    function keepLooking() {
        var special = 3.141592;
        console.log(special);
        console.log(window.special);
    }

    keepLooking();
}

lookingFor(112358132134);
// 3.141592
// 42
```

The global RED(1) `special` is shadowed by the BLUE(2) `special` (parameter), and the BLUE(2) `special` is itself shadowed by the GREEN(3) `special` inside `keepLooking()`. We can still access the RED(1) `special` using the indirect reference `window.special`. But there's no way for `keepLooking()` to access the BLUE(2) `special` that holds the number `112358132134`.

Copying Is Not Accessing

I've been asked the following "But what about…?" question dozens of times. Consider:

```
var special = 42;

function lookingFor(special) {
    var another = {
        special: special
    };

    function keepLooking() {
        var special = 3.141592;
        console.log(special);
        console.log(another.special);  // Ooo, tricky!
        console.log(window.special);
    }

    keepLooking();
}

lookingFor(112358132134);
// 3.141592
// 112358132134
// 42
```

Oh! So does this `another` object technique disprove my claim that the `special` parameter is "completely inaccessible" from inside `keepLooking()`? No, the claim is still correct.

`special: special` is copying the value of the `special` parameter variable into another container (a property of the same name). Of course, if you put a value in another container, shadowing no longer applies (unless `another` was shadowed, too!). But that doesn't mean we're accessing the parameter `special`; it means we're accessing the copy of the value it had at that moment, by way of *another* container (object property). We cannot reassign the BLUE(2) `special` parameter to a different value from inside `keepLooking()`.

Another "But…!?" you may be about to raise: what if I'd used objects or arrays as the values instead of the numbers (`112358132134`, etc.)? Would us having references to objects instead of copies of primitive values "fix" the inaccessibility?

No. Mutating the contents of the object value via a reference copy is **not** the same thing as lexically accessing the variable itself. We still can't reassign the BLUE(2) `special` parameter.

Illegal Shadowing

Not all combinations of declaration shadowing are allowed. `let` can shadow `var`, but `var` cannot shadow `let`:

```
function something() {
    var special = "JavaScript";

    {
        let special = 42;   // totally fine shadowing

        // ..
    }
}
```

```
function another() {
    // ..

    {
        let special = "JavaScript";

        {
            var special = "JavaScript";
            // ^^^ Syntax Error

            // ..
        }
    }
}
```

Notice in the `another()` function, the inner `var special` declaration is attempting to declare a function-wide `special`, which in and of itself is fine (as shown by the `something()` function).

The syntax error description in this case indicates that `special` has already been defined, but that error message is a little misleading—again, no such error happens in `something()`, as shadowing is generally allowed just fine.

The real reason it's raised as a `SyntaxError` is because the `var` is basically trying to "cross the boundary" of (or hop over) the `let` declaration of the same name, which is not allowed.

That boundary-crossing prohibition effectively stops at each function boundary, so this variant raises no exception:

```
function another() {
    // ..

    {
        let special = "JavaScript";

        ajax("https://some.url",function callback(){
            // totally fine shadowing
            var special = "JavaScript";

            // ..
        });
    }
}
```

Summary: `let` (in an inner scope) can always shadow an outer scope's `var`. `var` (in an inner scope) can only shadow an outer scope's `let` if there is a function boundary in between.

Function Name Scope

As you've seen by now, a `function` declaration looks like this:

```
function askQuestion() {
    // ..
}
```

And as discussed in Chapters 1 and 2, such a `function` declaration will create an identifier in the enclosing scope (in this case, the global scope) named `askQuestion`.

What about this program?

```
var askQuestion = function(){
    // ..
};
```

The same is true for the variable `askQuestion` being created. But since it's a `function` expression—a function definition used as value instead of a standalone declaration—the function itself will not "hoist" (see Chapter 5).

One major difference between `function` declarations and `function` expressions is what happens to the name identifier of the function. Consider a named `function` expression:

```
var askQuestion = function ofTheTeacher(){
    // ..
};
```

We know `askQuestion` ends up in the outer scope. But what about the `ofTheTeacher` identifier? For formal `function` declarations, the name identifier ends up in the outer/enclosing scope, so it may be reasonable to assume that's true here. But `ofTheTeacher` is declared as an identifier **inside the function itself**:

```
var askQuestion = function ofTheTeacher() {
    console.log(ofTheTeacher);
};

askQuestion();
// function ofTheTeacher()...

console.log(ofTheTeacher);
// ReferenceError: ofTheTeacher is not defined
```

Note

Actually, `ofTheTeacher` is not exactly *in the scope of the function*. Appendix A, "Implied Scopes" will explain further.

Not only is `ofTheTeacher` declared inside the function rather than outside, but it's also defined as read-only:

```
var askQuestion = function ofTheTeacher() {
    "use strict";
    ofTheTeacher = 42;   // TypeError

    //..
};

askQuestion();
// TypeError
```

Because we used strict-mode, the assignment failure is reported as a `TypeError`; in non-strict-mode, such an assignment fails silently with no exception.

What about when a `function` expression has no name identifier?

```
var askQuestion = function(){
   // ..
};
```

A `function` expression with a name identifier is referred to as a "named function expression," but one without a name identifier is referred to as an "anonymous function expression." Anonymous function expressions clearly have no name identifier that affects either scope.

Note

We'll discuss named vs. anonymous `function` expressions in much more detail, including what factors affect the decision to use one or the other, in Appendix A.

Arrow Functions

ES6 added an additional `function` expression form to the language, called "arrow functions":

```
var askQuestion = () => {
    // ..
};
```

The `=>` arrow function doesn't require the word `function` to define it. Also, the `( .. )` around the parameter list is optional in some simple cases. Likewise, the `{ .. }` around the function body is optional in some cases. And when the `{ .. }` are omitted, a return value is sent out without using a `return` keyword.

Note

The attractiveness of `=>` arrow functions is often sold as "shorter syntax," and that's claimed to equate to objectively more readable code. This claim is dubious at best, and I believe outright misguided. We'll dig into the "readability" of various function forms in Appendix A.

Arrow functions are lexically anonymous, meaning they have no directly related identifier that references the function. The assignment to `askQuestion` creates an inferred name of "askQuestion", but that's **not the same thing as being non-anonymous**:

```
var askQuestion = () => {
    // ..
};

askQuestion.name;   // askQuestion
```

Arrow functions achieve their syntactic brevity at the expense of having to mentally juggle a bunch of variations for different forms/conditions. Just a few, for example:

```
() => 42;

id => id.toUpperCase();

(id,name) => ({ id, name });

(...args) => {
    return args[args.length - 1];
};
```

The real reason I bring up arrow functions is because of the common but incorrect claim that arrow functions somehow behave differently with respect to lexical scope from standard `function` functions.

This is incorrect.

Other than being anonymous (and having no declarative form), => arrow functions have the same lexical scope rules as `function` functions do. An arrow function, with or without `{ .. }` around its body, still creates a separate, inner nested bucket of scope. Variable declarations inside this nested scope bucket behave the same as in a `function` scope.

Backing Out

When a function (declaration or expression) is defined, a new scope is created. The positioning of scopes nested inside one another creates a natural scope hierarchy throughout the program, called the scope chain. The scope chain controls variable access, directionally oriented upward and outward.

Each new scope offers a clean slate, a space to hold its own set of variables. When a variable name is repeated at different levels of the scope chain, shadowing occurs, which prevents access to the outer variable from that point inward.

As we step back out from these finer details, the next chapter shifts focus to the primary scope all JS programs include: the global scope.

Chapter 4: Around the Global Scope

Chapter 3 mentioned the "global scope" several times, but you may still be wondering why a program's outermost scope is all that important in modern JS. The vast majority of work is now done inside of functions and modules rather than globally.

Is it good enough to just assert, "Avoid using the global scope," and be done with it?

The global scope of a JS program is a rich topic, with much more utility and nuance than you would likely assume. This chapter first explores how the global scope is (still) useful and relevant to writing JS programs today, then looks at differences in where and *how to access* the global scope in different JS environments.

Fully understanding the global scope is critical in your mastery of using lexical scope to structure your programs.

Why Global Scope?

It's likely no surprise to readers that most applications are composed of multiple (sometimes many!) individual JS files. So how exactly do all those separate files get stitched together in a single runtime context by the JS engine?

With respect to browser-executed applications, there are three main ways.

First, if you're directly using ES modules (not transpiling them into some other module-bundle format), these files are loaded individually by the JS environment. Each module then `import`s references to whichever other modules it needs to access. The separate module files cooperate with each other exclusively through these shared imports, without needing any shared outer scope.

Second, if you're using a bundler in your build process, all the files are typically concatenated together before delivery to the browser and JS engine, which then only processes one big file. Even with all the pieces of the application co-located in a single file, some mechanism is necessary for each piece to register a *name* to be referred to by other pieces, as well as some facility for that access to occur.

In some build setups, the entire contents of the file are wrapped in a single enclosing scope, such as a wrapper function, universal module (UMD—see Appendix A), etc. Each piece can register itself for access from other pieces by way of local variables in that shared scope. For example:

```
(function wrappingOuterScope(){
    var moduleOne = (function one(){
        // ..
    })();

    var moduleTwo = (function two(){
        // ..

        function callModuleOne() {
            moduleOne.someMethod();
        }
        // ..
    })();
})();
```

As shown, the `moduleOne` and `moduleTwo` local variables inside the `wrappingOuterScope()` function scope are declared so that these modules can access each other for their cooperation.

While the scope of `wrappingOuterScope()` is a function and not the full environment global scope, it does act as a sort of "application-wide scope," a bucket where all the top-level identifiers can be stored, though not in the real global scope. It's kind of like a stand-in for the global scope in that respect.

And finally, the third way: whether a bundler tool is used for an application, or whether the (non-ES module) files are simply loaded in the browser individually (via `<script>` tags or other dynamic JS resource loading), if there is no single surrounding scope encompassing all these pieces, the **global scope** is the only way for them to cooperate with each other:

A bundled file of this sort often looks something like this:

```
var moduleOne = (function one(){
    // ..
})();
var moduleTwo = (function two(){
    // ..

    function callModuleOne() {
        moduleOne.someMethod();
    }

    // ..
})();
```

Here, since there is no surrounding function scope, these `moduleOne` and `moduleTwo` declarations are simply dropped into the global scope. This is effectively the same as if the files hadn't been concatenated, but loaded separately:

module1.js:

```
var moduleOne = (function one(){
    // ..
})();
```

module2.js:

```
var moduleTwo = (function two(){
    // ..

    function callModuleOne() {
        moduleOne.someMethod();
    }

    // ..
})();
```

If these files are loaded separately as normal standalone .js files in a browser environment, each top-level variable declaration will end up as a global variable, since the global scope is the only shared resource between these two separate files—they're independent programs, from the perspective of the JS engine.

In addition to (potentially) accounting for where an application's code resides during runtime, and how each piece is able to access the other pieces to cooperate, the global scope is also where:

- JS exposes its built-ins:
    - primitives: `undefined`, `null`, `Infinity`, `NaN`
    - natives: `Date()`, `Object()`, `String()`, etc.
    - global functions: `eval()`, `parseInt()`, etc.
    - namespaces: `Math`, `Atomics`, `JSON`
    - friends of JS: `Intl`, `WebAssembly`
- The environment hosting the JS engine exposes its own built-ins:
    - `console` (and its methods)
    - the DOM (`window`, `document`, etc)
    - timers (`setTimeout(..)`, etc)
    - web platform APIs: `navigator`, `history`, geolocation, WebRTC, etc.

These are just some of the many *globals* your programs will interact with.

Note

Node also exposes several elements "globally," but they're technically not in the `global` scope: `require()`, `__dirname`, `module`, `URL`, and so on.

Most developers agree that the global scope shouldn't just be a dumping ground for every variable in your application. That's a mess of bugs just waiting to happen. But it's also undeniable that the global scope is an important *glue* for practically every JS application.

Where Exactly is this Global Scope?

It might seem obvious that the global scope is located in the outermost portion of a file; that is, not inside any function or other block. But it's not quite as simple as that.

Different JS environments handle the scopes of your programs, especially the global scope, differently. It's quite common for JS developers to harbor misconceptions without even realizing it.

Browser "Window"

With respect to treatment of the global scope, the most *pure* environment JS can be run in is as a standalone .js file loaded in a web page environment in a browser. I don't mean "pure" as in nothing automatically added—lots may be added!—but rather in terms of minimal intrusion on the code or interference with its expected global scope behavior.

Consider this .js file:

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!
```

This code may be loaded in a web page environment using an inline `<script>` tag, a `<script src=..>` script tag in the markup, or even a dynamically created `<script>` DOM element. In all three cases, the `studentName` and `hello` identifiers are declared in the global scope.

That means if you access the global object (commonly, `window` in the browser), you'll find properties of those same names there:

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ window.studentName }!`);
}

window.hello();
// Hello, Kyle!
```

That's the default behavior one would expect from a reading of the JS specification: the outer scope *is* the global scope and `studentName` is legitimately created as global variable.

That's what I mean by *pure*. But unfortunately, that won't always be true of all JS environments you encounter, and that's often surprising to JS developers.

Globals Shadowing Globals

Recall the discussion of shadowing (and global unshadowing) from Chapter 3, where one variable declaration can override and prevent access to a declaration of the same name from an outer scope.

An unusual consequence of the difference between a global variable and a global property of the same name is that, within just the global scope itself, a global object property can be shadowed by a global variable:

```js
window.something = 42;

let something = "Kyle";

console.log(something);
// Kyle

console.log(window.something);
// 42
```

The `let` declaration adds a `something` global variable but not a global object property (see Chapter 3). The effect then is that the `something` lexical identifier shadows the `something` global object property.

It's almost certainly a bad idea to create a divergence between the global object and the global scope. Readers of your code will almost certainly be tripped up.

A simple way to avoid this gotcha with global declarations: always use `var` for globals. Reserve `let` and `const` for block scopes (see "Scoping with Blocks" in Chapter 6).

DOM Globals

I asserted that a browser-hosted JS environment has the most *pure* global scope behavior we'll see. However, it's not entirely *pure*.

One surprising behavior in the global scope you may encounter with browser-based JS applications: a DOM element with an `id` attribute automatically creates a global variable that references it.

Consider this markup:

```
<ul id="my-todo-list">
   <li id="first">Write a book</li>
   ..
</ul>
```

And the JS for that page could include:

```
first;
// <li id="first">..</li>

window["my-todo-list"];
// <ul id="my-todo-list">..</ul>
```

If the `id` value is a valid lexical name (like `first`), the lexical variable is created. If not, the only way to access that global is through the global object (`window[..]`).

The auto-registration of all `id`-bearing DOM elements as global variables is an old legacy browser behavior that nevertheless must remain because so many old sites still rely on it. My advice is never to use these global variables, even though they will always be silently created.

What's in a (Window) Name?

Another global scope oddity in browser-based JS:

```
var name = 42;

console.log(name, typeof name);
// "42" string
```

`window.name` is a pre-defined "global" in a browser context; it's a property on the global object, so it seems like a normal global variable (yet it's anything but "normal").

We used `var` for our declaration, which **does not** shadow the pre-defined `name` global property. That means, effectively, the `var` declaration is ignored, since there's already a global scope object property of that name. As we discussed earlier, had we used `let name`, we would have shadowed `window.name` with a separate global `name` variable.

But the truly surprising behavior is that even though we assigned the number `42` to `name` (and thus `window.name`), when we then retrieve its value, it's a string `"42"`! In this case, the weirdness is because `name` is actually a pre-defined getter/setter on the `window` object, which insists on its value being a string value. Yikes!

With the exception of some rare corner cases like DOM element ID's and `window.name`, JS running as a standalone file in a browser page has some of the most *pure* global scope behavior we will encounter.

Web Workers

Web Workers are a web platform extension on top of browser-JS behavior, which allows a JS file to run in a completely separate thread (operating system wise) from the thread that's running the main JS program.

Since these Web Worker programs run on a separate thread, they're restricted in their communications with the main application thread, to avoid/limit race conditions and other complications. Web Worker code does not have access to the DOM, for example. Some web APIs are, however, made available to the worker, such as `navigator`.

Since a Web Worker is treated as a wholly separate program, it does not share the global scope with the main JS program.

However, the browser's JS engine is still running the code, so we can expect similar *purity* of its global scope behavior. Since there is no DOM access, the `window` alias for the global scope doesn't exist.

In a Web Worker, the global object reference is typically made using `self`:

```
var studentName = "Kyle";
let studentID = 42;

function hello() {
    console.log(`Hello, ${ self.studentName }!`);
}

self.hello();
// Hello, Kyle!

self.studentID;
// undefined
```

Just as with main JS programs, `var` and `function` declarations create mirrored properties on the global object (aka, `self`), where other declarations (`let`, etc) do not.

So again, the global scope behavior we're seeing here is about as *pure* as it gets for running JS programs; perhaps it's even more *pure* since there's no DOM to muck things up!

Developer Tools Console/REPL

Recall from Chapter 1 in *Get Started* that Developer Tools don't create a completely adherent JS environment. They do process JS code, but they also lean in favor of the UX interaction being most friendly to developers (aka, developer experience, or DX).

In some cases, favoring DX when typing in short JS snippets, over the normal strict steps expected for processing a full JS program, produces observable differences in code behavior between programs and tools. For example, certain error conditions applicable to a JS program may be relaxed and not displayed when the code is entered into a developer tool.

With respect to our discussions here about scope, such observable differences in behavior may include:

- The behavior of the global scope
- Hoisting (see Chapter 5)
- Block-scoping declarators (`let` / `const`, see Chapter 6) when used in the outermost scope

Although it might seem, while using the console/REPL, that statements entered in the outermost scope are being processed in the real global scope, that's not quite accurate. Such tools typically emulate the global scope position to an extent; it's emulation, not strict adherence. These tool environments prioritize developer convenience, which means that at times (such as with our current discussions regarding scope), observed behavior may deviate from the JS specification.

The take-away is that Developer Tools, while optimized to be convenient and useful for a variety of developer activities, are **not** suitable environments to determine or verify explicit and nuanced behaviors of an actual JS program context.

ES Modules (ESM)

ES6 introduced first-class support for the module pattern (covered in Chapter 8). One of the most obvious impacts of using ESM is how it changes the behavior of the observably top-level scope in a file.

Recall this code snippet from earlier (which we'll adjust to ESM format by using the `export` keyword):

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!

export hello;
```

If that code is in a file that's loaded as an ES module, it will still run exactly the same. However, the observable effects, from the overall application perspective, will be different.

Despite being declared at the top level of the (module) file, in the outermost obvious scope, `studentName` and `hello` are not global variables. Instead, they are module-wide, or if you prefer, "module-global."

However, in a module there's no implicit "module-wide scope object" for these top-level declarations to be added to as properties, as there is when declarations appear in the top-level of non-module JS files. This is not to say that global variables cannot exist or be accessed in such programs. It's just that global variables don't get *created* by declaring variables in the top-level scope of a module.

The module's top-level scope is descended from the global scope, almost as if the entire contents of the module were wrapped in a function. Thus, all variables that exist in the global scope (whether they're on the global object or not!) are available as lexical identifiers from inside the module's scope.

ESM encourages a minimization of reliance on the global scope, where you import whatever modules you may need for the current module to operate. As such, you less often see usage of the global scope or its global object.

However, as noted earlier, there are still plenty of JS and web globals that you will continue to access from the global scope, whether you realize it or not!

Node

One aspect of Node that often catches JS developers off-guard is that Node treats every single .js file that it loads, including the main one you start the Node process with, as a *module* (ES module or CommonJS module, see Chapter 8). The practical effect is that the top level of your Node programs **is never actually the global scope**, the way it is when loading a non-module file in the browser.

As of time of this writing, Node has recently added support for ES modules. But additionally, Node has from its beginning supported a module format referred to as "CommonJS", which looks like this:

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!

module.exports.hello = hello;
```

Before processing, Node effectively wraps such code in a function, so that the `var` and `function` declarations are contained in that wrapping function's scope, **not** treated as global variables.

Envision the preceding code as being seen by Node as this (illustrative, not actual):

```
function Module(module,require,__dirname,...) {
    var studentName = "Kyle";

    function hello() {
        console.log(`Hello, ${ studentName }!`);
    }

    hello();
    // Hello, Kyle!

    module.exports.hello = hello;
}
```

Node then essentially invokes the added `Module(..)` function to run your module. You can clearly see here why `studentName` and `hello` identifiers are not global, but rather declared in the module scope.

As noted earlier, Node defines a number of "globals" like `require()`, but they're not actually identifiers in the global scope (nor properties of the global object). They're injected in the scope of every module, essentially a bit like the parameters listed in the `Module(..)` function declaration.

So how do you define actual global variables in Node? The only way to do so is to add properties to another of Node's automatically provided "globals," which is ironically called `global`. `global` is a reference to the real global scope object, somewhat like using `window` in a browser JS environment.

Consider:

```
global.studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!

module.exports.hello = hello;
```

Here we add `studentName` as a property on the `global` object, and then in the `console.log(..)` statement we're able to access `studentName` as a normal global variable.

Remember, the identifier `global` is not defined by JS; it's specifically defined by Node.

Global This

Reviewing the JS environments we've looked at so far, a program may or may not:

- Declare a global variable in the top-level scope with `var` or `function` declarations—or `let`, `const`, and `class`.
- Also add global variables declarations as properties of the global scope object if `var` or `function` are used for the declaration.
- Refer to the global scope object (for adding or retrieving global variables, as properties) with `window`, `self`, or `global`.

I think it's fair to say that global scope access and behavior is more complicated than most developers assume, as the preceding sections have illustrated. But the complexity is never more obvious than in trying to nail down a universally applicable reference to the global scope object.

Yet another "trick" for obtaining a reference to the global scope object looks like:

```
const theGlobalScopeObject =
    (new Function("return this"))();
```

Note

A function can be dynamically constructed from code stored in a string value with the `Function()` constructor, similar to `eval(..)` (see "Cheating: Runtime Scope Modifications" in Chapter 1). Such a function will automatically be run in non-strict-mode (for legacy reasons) when invoked with the normal `()` function invocation as shown; its `this` will point at the global object. See the third book in the series, *Objects & Classes*, for more information on determining `this` bindings.

So, we have `window`, `self`, `global`, and this ugly `new Function(..)` trick. That's a lot of different ways to try to get at this global object. Each has its pros and cons.

Why not introduce yet another!?!?

As of ES2020, JS has finally defined a standardized reference to the global scope object, called `globalThis`. So, subject to the recency of the JS engines your code runs in, you can use `globalThis` in place of any of those other approaches.

We could even attempt to define a cross-environment polyfill that's safer across pre-`globalThis` JS environments, such as:

```
const theGlobalScopeObject =
    (typeof globalThis != "undefined") ? globalThis :
    (typeof global != "undefined") ? global :
    (typeof window != "undefined") ? window :
    (typeof self != "undefined") ? self :
    (new Function("return this"))();
```

Phew! That's certainly not ideal, but it works if you find yourself needing a reliable global scope reference.

(The proposed name `globalThis` was fairly controversial while the feature was being added to JS. Specifically, I and many others felt the "this" reference in its name was misleading, since the reason you reference this object is to access to the global scope, never to access some sort of global/default `this` binding. There were many other names considered, but for a variety of reasons ruled out. Unfortunately, the name chosen ended up as a last resort. If you plan to interact with the global scope object in your programs, to reduce confusion, I strongly recommend choosing a better name, such as (the laughably long but accurate!) `theGlobalScopeObject` used here.)

Globally Aware

The global scope is present and relevant in every JS program, even though modern patterns for organizing code into modules de-emphasizes much of the reliance on storing identifiers in that namespace.

Still, as our code proliferates more and more beyond the confines of the browser, it's especially important we have a solid grasp on the differences in how the global scope (and global scope object!) behave across different JS environments.

With the big picture of global scope now sharper in focus, the next chapter again descends into the deeper details of lexical scope, examining how and when variables can be used.