You Don't Know JS Yet: Scope & Closures - 2nd Edition

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:

Colored Scope Bubbles

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 the case 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.