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

Appendix A: Exploring Further

We will now explore a number of nuances and edges around many of the topics covered in the main text of this book. This appendix is optional, supporting material.

Some people find diving too deeply into the nuanced corner cases and varying opinions creates nothing but noise and distraction—supposedly, developers are better served by sticking to the commonly-tread paths. My approach has been criticized as being impractical and counterproductive. I understand and appreciate that perspective, even if I don't necessarily share it.

I believe it's better to be empowered by knowledge of how things work than to just gloss over details with assumptions and lack of curiosity. Ultimately, you will encounter situations where something bubbles up from a piece you hadn't explored. In other words, you won't get to spend all your time riding on the smooth *happy path*. Wouldn't you rather be prepared for the inevitable bumps of off-roading?

These discussions will also be more heavily influenced by my opinions than the main text was, so keep that in mind as you consume and consider what is presented. This appendix is a bit like a collection of mini-blog posts that elaborate on various book topics. It's long and deep in the weeds, so take your time and don't rush through everything here.

Implied Scopes

Scopes are sometimes created in non-obvious places. In practice, these implied scopes don't often impact your program behavior, but it's still useful to know they're happening. Keep an eye out for the following surprising scopes:

- Parameter scope
- Function name scope

Parameter Scope

The conversation metaphor in Chapter 2 implies that function parameters are basically the same as locally declared variables in the function scope. But that's not always true.

Consider:

```
// outer/global scope: RED(1)

function getStudentName(studentID) {
    // function scope: BLUE(2)

    // ..
}
```

Here, `studentID` is a considered a "simple" parameter, so it does behave as a member of the BLUE(2) function scope. But if we change it to be a non-simple parameter, that's no longer technically the case. Parameter forms considered non-simple include parameters with default values, rest parameters (using `...`), and destructured parameters.

Consider:

```
// outer/global scope: RED(1)

function getStudentName(/*BLUE(2)*/ studentID = 0) {
    // function scope: GREEN(3)

    // ..
}
```

Here, the parameter list essentially becomes its own scope, and the function's scope is then nested inside *that* scope.

Why? What difference does it make? The non-simple parameter forms introduce various corner cases, so the parameter list becomes its own scope to more effectively deal with them.

Consider:

```
function getStudentName(studentID = maxID, maxID) {
    // ..
}
```

Assuming left-to-right operations, the default `= maxID` for the `studentID` parameter requires a `maxID` to already exist (and to have been initialized). This code produces a TDZ error (Chapter 5). The reason is that `maxID` is declared in the parameter scope, but it's not yet been initialized because of the order of parameters. If the parameter order is flipped, no TDZ error occurs:

```
function getStudentName(maxID,studentID = maxID) {
    // ..
}
```

The complication gets even more *in the weeds* if we introduce a function expression into the default parameter position, which then can create its own closure (Chapter 7) over parameters in this implied parameter scope:

```
function whatsTheDealHere(id,defaultID = () => id) {
    id = 5;
    console.log( defaultID() );
}

whatsTheDealHere(3);
// 5
```

That snippet probably makes sense, because the `defaultID()` arrow function closes over the `id` parameter/variable, which we then re-assign to `5`. But now let's introduce a shadowing definition of `id` in the function scope:

```
function whatsTheDealHere(id,defaultID = () => id) {
    var id = 5;
    console.log( defaultID() );
}

whatsTheDealHere(3);
// 3
```

Uh oh! The `var id = 5` is shadowing the `id` parameter, but the closure of the `defaultID()` function is over the parameter, not the shadowing variable in the function body. This proves there's a scope bubble around the parameter list.

But it gets even crazier than that!

```
function whatsTheDealHere(id,defaultID = () => id) {
    var id;

    console.log(`local variable 'id': ${ id }`);
    console.log(
        `parameter 'id' (closure): ${ defaultID() }`
    );

    console.log("reassigning 'id' to 5");
    id = 5;

    console.log(`local variable 'id': ${ id }`);
    console.log(
        `parameter 'id' (closure): ${ defaultID() }`
    );
}

whatsTheDealHere(3);
// local variable 'id': 3   <--- Huh!? Weird!
// parameter 'id' (closure): 3
// reassigning 'id' to 5
// local variable 'id': 5
// parameter 'id' (closure): 3
```

The strange bit here is the first console message. At that moment, the shadowing `id` local variable has just been `var id` declared, which Chapter 5 asserts is typically auto-initialized to `undefined` at the top of its scope. Why doesn't it print `undefined`?

In this specific corner case (for legacy compat reasons), JS doesn't auto-initialize `id` to `undefined`, but rather to the value of the `id` parameter (`3`)!

Though the two `id`s look at that moment like they're one variable, they're actually still separate (and in separate scopes). The `id = 5` assignment makes the divergence observable, where the `id` parameter stays `3` and the local variable becomes `5`.

My advice to avoid getting bitten by these weird nuances:

- Never shadow parameters with local variables
- Avoid using a default parameter function that closes over any of the parameters

At least now you're aware and can be careful about the fact that the parameter list is its own scope if any of the parameters are non-simple.

Function Name Scope

In the "Function Name Scope" section in Chapter 3, I asserted that the name of a function expression is added to the function's own scope. Recall:

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

It's true that `ofTheTeacher` is not added to the enclosing scope (where `askQuestion` is declared), but it's also not *just* added to the scope of the function, the way you're likely assuming. It's another strange corner case of implied scope.

The name identifier of a function expression is in its own implied scope, nested between the outer enclosing scope and the main inner function scope.

If `ofTheTeacher` was in the function's scope, we'd expect an error here:

```
var askQuestion = function ofTheTeacher(){
    // why is this not a duplicate declaration error?
    let ofTheTeacher = "Confused, yet?";
};
```

The `let` declaration form does not allow re-declaration (see Chapter 5). But this is perfectly legal shadowing, not re-declaration, because the two `ofTheTeacher` identifiers are in separate scopes.

You'll rarely run into any case where the scope of a function's name identifier matters. But again, it's good to know how these mechanisms actually work. To avoid being bitten, never shadow function name identifiers.

Anonymous vs. Named Functions

As discussed in Chapter 3, functions can be expressed either in named or anonymous form. It's vastly more common to use the anonymous form, but is that a good idea?

As you contemplate naming your functions, consider:

- Name inference is incomplete
- Lexical names allow self-reference
- Names are useful descriptions
- Arrow functions have no lexical names
- IIFEs also need names

Explicit or Inferred Names?

Every function in your program has a purpose. If it doesn't have a purpose, take it out, because you're just wasting space. If it *does* have a purpose, there *is* a name for that purpose.

So far many readers likely agree with me. But does that mean we should always put that name into the code? Here's where I'll raise more than a few eyebrows. I say, unequivocally, yes!

First of all, "anonymous" showing up in stack traces is just not all that helpful to debugging:

```
btn.addEventListener("click",function(){
    setTimeout(function(){
        ["a",42].map(function(v){
            console.log(v.toUpperCase());
        });
    },100);
});
// Uncaught TypeError: v.toUpperCase is not a function
//     at myProgram.js:4
//     at Array.map (<anonymous>)
//     at myProgram.js:3
```

Ugh. Compare to what is reported if I give the functions names:

```
btn.addEventListener("click",function onClick(){
    setTimeout(function waitAMoment(){
        ["a",42].map(function allUpper(v){
            console.log(v.toUpperCase());
        });
    },100);
});
// Uncaught TypeError: v.toUpperCase is not a function
//     at allUpper (myProgram.js:4)
//     at Array.map (<anonymous>)
//     at waitAMoment (myProgram.js:3)
```

See how `waitAMoment` and `allUpper` names appear and give the stack trace more useful information/context for debugging? The program is more debuggable if we use reasonable names for all our functions.

NOTE:

The unfortunate "<anonymous>" that still shows up refers to the fact that the implementation of `Array.map(..)` isn't present in our program, but is built into the JS engine. It's not from any confusion our program introduces with readability shortcuts.

By the way, let's make sure we're on the same page about what a named function is:

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

ajax("some.url",function thisIsAlsoNamed(){
   // ..
});

var notNamed = function(){
    // ..
};

makeRequest({
    data: 42,
    cb /* also not a name */: function(){
        // ..
    }
});

var stillNotNamed = function butThisIs(){
    // ..
};
```

"But wait!", you say. Some of those *are* named, right!?

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

var config = {
    cb: function(){
        // ..
    }
};

notNamed.name;
// notNamed

config.cb.name;
// cb
```

These are referred to as *inferred* names. Inferred names are fine, but they don't really address the full concern I'm discussing.

Missing Names?

Yes, these inferred names might show up in stack traces, which is definitely better than "anonymous" showing up. But...

```
function ajax(url,cb) {
    console.log(cb.name);
}

ajax("some.url",function(){
    // ..
});
// ""
```

Oops. Anonymous `function` expressions passed as callbacks are incapable of receiving an inferred name, so `cb.name` holds just the empty string `""`. The vast majority of all `function` expressions, especially anonymous ones, are used as callback arguments; none of these get a name. So relying on name inference is incomplete, at best.

And it's not just callbacks that fall short with inference:

```
var config = {};

config.cb = function(){
    // ..
};

config.cb.name;
// ""

var [ noName ] = [ function(){} ];
noName.name
// ""
```

Any assignment of a `function` expression that's not a *simple assignment* will also fail name inferencing. So, in other words, unless you're careful and intentional about it, essentially almost all anonymous `function` expressions in your program will in fact have no name at all.

Name inference is just... not enough.

And even if a `function` expression *does* get an inferred name, that still doesn't count as being a full named function.

Who am I?

Without a lexical name identifier, the function has no internal way to refer to itself. Self-reference is important for things like recursion and event handling:

```
// broken
runOperation(function(num){
    if (num <= 1) return 1;
    return num * oopsNoNameToCall(num - 1);
});

// also broken
btn.addEventListener("click",function(){
   console.log("should only respond to one click!");
   btn.removeEventListener("click",oopsNoNameHere);
});
```

Leaving off the lexical name from your callback makes it harder to reliably self-reference the function. You *could* declare a variable in an enclosing scope that references the function, but this variable is *controlled* by that enclosing scope—it could be re-assigned, etc.—so it's not as reliable as the function having its own internal self-reference.

Names are Descriptors

Lastly, and I think most importantly of all, leaving off a name from a function makes it harder for the reader to tell what the function's purpose is, at a quick glance. They have to read more of the code, including the code inside the function, and the surrounding code outside the function, to figure it out.

Consider:

```
[ 1, 2, 3, 4, 5 ].filter(function(v){
    return v % 2 == 1;
});
// [ 1, 3, 5 ]

[ 1, 2, 3, 4, 5 ].filter(function keepOnlyOdds(v){
    return v % 2 == 1;
});
// [ 1, 3, 5 ]
```

There's just no reasonable argument to be made that **omitting** the name `keepOnlyOdds` from the first callback more effectively communicates to the reader the purpose of this callback. You saved 13 characters, but lost important readability information. The name `keepOnlyOdds` very clearly tells the reader, at a quick first glance, what's happening.

The JS engine doesn't care about the name. But human readers of your code absolutely do.

Can the reader look at `v % 2 == 1` and figure out what it's doing? Sure. But they have to infer the purpose (and name) by mentally executing the code. Even a brief pause to do so slows down reading of the code. A good descriptive name makes this process almost effortless and instant.

Think of it this way: how many times does the author of this code need to figure out the purpose of a function before adding the name to the code? About once. Maybe two or three times if they need to adjust the name. But how many times will readers of this code have to figure out the name/purpose? Every single time this line is ever read. Hundreds of times? Thousands? More?

No matter the length or complexity of the function, my assertion is, the author should figure out a good descriptive name and add it to the code. Even the one-liner functions in `map(..)` and `then(..)` statements should be named:

```
lookupTheRecords(someData)
.then(function extractSalesRecords(resp){
   return resp.allSales;
})
.then(storeRecords);
```

The name `extractSalesRecords` tells the reader the purpose of this `then(..)` handler *better* than just inferring that purpose from mentally executing `return resp.allSales`.

The only excuse for not including a name on a function is either laziness (don't want to type a few extra characters) or uncreativity (can't come up with a good name). If you can't figure out a good name, you likely don't understand the function and its purpose yet. The function is perhaps poorly designed, or it does too many things, and should be re-worked. Once you have a well-designed, single-purpose function, its proper name should become evident.

Here's a trick I use: while first writing a function, if I don't fully understand its purpose and can't think of a good name to use, I just use `TODO` as the name. That way, later when reviewing my code, I'm likely to find those name placeholders, and I'm more inclined (and more prepared!) to go back and figure out a better name, rather than just leave it as `TODO`.

All functions need names. Every single one. No exceptions. Any name you omit is making the program harder to read, harder to debug, harder to extend and maintain later.

Arrow Functions

Arrow functions are **always** anonymous, even if (rarely) they're used in a way that gives them an inferred name. I just spent several pages explaining why anonymous functions are a bad idea, so you can probably guess what I think about arrow functions.

Don't use them as a general replacement for regular functions. They're more concise, yes, but that brevity comes at the cost of omitting key visual delimiters that help our brains quickly parse out what we're reading. And, to the point of this discussion, they're anonymous, which makes them worse for readability from that angle as well.

Arrow functions have a purpose, but that purpose is not to save keystrokes. Arrow functions have *lexical this* behavior, which is somewhat beyond the bounds of our discussion in this book.

Briefly: arrow functions don't define a `this` identifier keyword at all. If you use a `this` inside an arrow function, it behaves exactly as any other variable reference, which is that the scope chain is consulted to find a function scope (non-arrow function) where it *is* defined, and to use that one.

In other words, arrow functions treat `this` like any other lexical variable.

If you're used to hacks like `var self = this`, or if you prefer to call `.bind(this)` on inner `function` expressions, just to force them to inherit a `this` from an outer function like it was a lexical variable, then `=>` arrow functions are absolutely the better option. They're designed specifically to fix that problem.

So, in the rare cases you need *lexical this*, use an arrow function. It's the best tool for that job. But just be aware that in doing so, you're accepting the downsides of an anonymous function. You should expend additional effort to mitigate the readability *cost*, such as more descriptive variable names and code comments.

IIFE Variations

All functions should have names. I said that a few times, right!? That includes IIFEs.

```
(function(){
    // don't do this!
})();

(function doThisInstead(){
    // ..
})();
```

How do we come up with a name for an IIFE? Identify what the IIFE is there for. Why do you need a scope in that spot? Are you hiding a cache variable for student records?

```
var getStudents = (function StoreStudentRecords(){
    var studentRecords = [];

    return function getStudents() {
        // ..
    }
})();
```

I named the IIFE `StoreStudentRecords` because that's what it's doing: storing student records. Every IIFE should have a name. No exceptions.

IIFEs are typically defined by placing `( .. )` around the `function` expression, as shown in those previous snippets. But that's not the only way to define an IIFE. Technically, the only reason we're using that first surrounding set of `( .. )` is just so the `function` keyword isn't in a position to qualify as a `function` declaration to the JS parser. But there are other syntactic ways to avoid being parsed as a declaration:

```
!function thisIsAnIIFE(){
    // ..
}();

+function soIsThisOne(){
    // ..
}();

~function andThisOneToo(){
    // ..
}();
```

The `!`, `+`, `~`, and several other unary operators (operators with one operand) can all be placed in front of `function` to turn it into an expression. Then the final `()` call is valid, which makes it an IIFE.

I actually kind of like using the `void` unary operator when defining a standalone IIFE:

```
void function yepItsAnIIFE() {
    // ..
}();
```

The benefit of `void` is, it clearly communicates at the beginning of the function that this IIFE won't be returning any value.

However you define your IIFEs, show them some love by giving them names.

Hoisting: Functions and Variables

Chapter 5 articulated both *function hoisting* and *variable hoisting*. Since hoisting is often cited as mistake in the design of JS, I wanted to briefly explore why both these forms of hoisting *can* be beneficial and should still be considered.

Give hoisting a deeper level of consideration by considering the merits of:

- Executable code first, function declarations last
- Semantic placement of variable declarations

Function Hoisting

To review, this program works because of *function hoisting*:

```
getStudents();

// ..

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

The `function` declaration is hoisted during compilation, which means that `getStudents` is an identifier declared for the entire scope. Additionally, the `getStudents` identifier is auto-initialized with the function reference, again at the beginning of the scope.

Why is this useful? The reason I prefer to take advantage of *function hoisting* is that it puts the *executable* code in any scope at the top, and any further declarations (functions) below. This means it's easier to find the code that will run in any given area, rather than having to scroll and scroll, hoping to find a trailing `}` marking the end of a scope/function somewhere.

I take advantage of this inverse positioning in all levels of scope:

```
getStudents();

// *************

function getStudents() {
    var whatever = doSomething();

    // other stuff

    return whatever;

    // *************

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

When I first open a file like that, the very first line is executable code that kicks off its behavior. That's very easy to spot! Then, if I ever need to go find and inspect `getStudents()`, I like that its first line is also executable code. Only if I need to see the details of `doSomething()` do I go and find its definition down below.

In other words, I think *function hoisting* makes code more readable through a flowing, progressive reading order, from top to bottom.

Variable Hoisting

What about *variable hoisting*?

Even though `let` and `const` hoist, you cannot use those variables in their TDZ (see Chapter 5). So, the following discussion only applies to `var` declarations. Before I continue, I'll admit: in almost all cases, I completely agree that *variable hoisting* is a bad idea:

```
pleaseDontDoThis = "bad idea";

// much later
var pleaseDontDoThis;
```

While that kind of inverted ordering was helpful for *function hoisting*, here I think it usually makes code harder to reason about.

But there's one exception that I've found, somewhat rarely, in my own coding. It has to do with where I place my `var` declarations inside a CommonJS module definition.

Here's how I typically structure my module definitions in Node:

```
// dependencies
var aModuleINeed = require("very-helpful");
var anotherModule = require("kinda-helpful");

// public API
var publicAPI = Object.assign(module.exports,{
    getStudents,
    addStudents,
    // ..
});

// ********************************
// private implementation

var cache = { };
var otherData = [ ];

function getStudents() {
    // ..
}

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

Notice how the `cache` and `otherData` variables are in the "private" section of the module layout? That's because I don't plan to expose them publicly. So I organize the module so they're located alongside the other hidden implementation details of the module.

But I've had a few rare cases where I needed the assignments of those values to happen *above*, before I declare the exported public API of the module. For instance:

```
// public API
var publicAPI = Object.assign(module.exports,{
    getStudents,
    addStudents,
    refreshData: refreshData.bind(null,cache)
});
```

I need the `cache` variable to have already been assigned a value, because that value is used in the initialization of the public API (the `.bind(..)` partial-application).

Should I just move the `var cache = { .. }` up to the top, above this public API initialization? Well, perhaps. But now it's less obvious that `var cache` is a *private* implementation detail. Here's the compromise I've (somewhat rarely) used:

```
cache = {};   // used here, but declared below

// public API
var publicAPI = Object.assign(module.exports,{
    getStudents,
    addStudents,
    refreshData: refreshData.bind(null,cache)
});

// ********************************
// private implementation

var cache /* = {}*/;
```

See the *variable hoisting*? I've declared the `cache` down where it belongs, logically, but in this rare case I've used it earlier up above, in the area where its initialization is needed. I even left a hint at the value that's assigned to `cache` in a code comment.

That's literally the only case I've ever found for leveraging *variable hoisting* to assign a variable earlier in a scope than its declaration. But I think it's a reasonable exception to employ with caution.

The Case for `var`

Speaking of *variable hoisting*, let's have some real talk for a bit about `var`, a favorite villain devs love to blame for many of the woes of JS development. In Chapter 5, we explored `let` / `const` and promised we'd revisit where `var` falls in the whole mix.

As I lay out the case, don't miss:

- `var` was never broken
- `let` is your friend
- `const` has limited utility
- The best of both worlds: `var` *and* `let`

Don't Throw Out `var`

`var` is fine, and works just fine. It's been around for 25 years, and it'll be around and useful and functional for another 25 years or more. Claims that `var` is broken, deprecated, outdated, dangerous, or ill-designed are bogus bandwagoning.

Does that mean `var` is the right declarator for every single declaration in your program? Certainly not. But it still has its place in your programs. Refusing to use it because someone on the team chose an aggressive linter opinion that chokes on `var` is cutting off your nose to spite your face.

OK, now that I've got you really riled up, let me try to explain my position.

For the record, I'm a fan of `let`, for block-scoped declarations. I really dislike TDZ and I think that was a mistake. But `let` itself is great. I use it often. In fact, I probably use it as much or more than I use `var`.

`const`-antly Confused

`const` on the other hand, I don't use as often. I'm not going to dig into all the reasons why, but it comes down to `const` not *carrying its own weight*. That is, while there's a tiny bit of benefit of `const` in some cases, that benefit is outweighed by the long history of troubles around `const` confusion in a variety of languages, long before it ever showed up in JS.

`const` pretends to create values that can't be mutated—a misconception that's extremely common in developer communities across many languages—whereas what it really does is prevent re-assignment.

```
 const studentIDs = [ 14, 73, 112 ];

// later

studentIDs.push(6);   // whoa, wait... what!?
```

Using a `const` with a mutable value (like an array or object) is asking for a future developer (or reader of your code) to fall into the trap you set, which was that they either didn't know, or sorta forgot, that *value immutability* isn't at all the same thing as *assignment immutability*.

I just don't think we should set those traps. The only time I ever use `const` is when I'm assigning an already-immutable value (like `42` or `"Hello, friends!"`), and when it's clearly a "constant" in the sense of being a named placeholder for a literal value, for semantic purposes. That's what `const` is best used for. That's pretty rare in my code, though.

If variable re-assignment were a big deal, then `const` would be more useful. But variable re-assignment just isn't that big of a deal in terms of causing bugs. There's a long list of things that lead to bugs in programs, but "accidental re-assignment" is way, way down that list.

Combine that with the fact that `const` (and `let`) are supposed to be used in blocks, and blocks are supposed to be short, and you have a really small area of your code where a `const` declaration is even applicable. A `const` on line 1 of your ten-line block only tells you something about the next nine lines. And the thing it tells you is already obvious by glancing down at those nine lines: the variable is never on the left-hand side of an `=`; it's not re-assigned.

That's it, that's all `const` really does. Other than that, it's not very useful. Stacked up against the significant confusion of value vs. assignment immutability, `const` loses a lot of its luster.

A `let` (or `var`!) that's never re-assigned is already behaviorally a "constant", even though it doesn't have the compiler guarantee. That's good enough in most cases.

`var` *and* `let`

In my mind, `const` is pretty rarely useful, so this is only two-horse race between `let` and `var`. But it's not really a race either, because there doesn't have to be just one winner. They can both win... different races.

The fact is, you should be using both `var` and `let` in your programs. They are not interchangeable: you shouldn't use `var` where a `let` is called for, but you also shouldn't use `let` where a `var` is most appropriate.

So where should we still use `var`? Under what circumstances is it a better choice than `let`?

For one, I always use `var` in the top-level scope of any function, regardless of whether that's at the beginning, middle, or end of the function. I also use `var` in the global scope, though I try to minimize usage of the global scope.

Why use `var` for function scoping? Because that's exactly what `var` does. There literally is no better tool for the job of function scoping a declaration than a declarator that has, for 25 years, done exactly that.

You *could* use `let` in this top-level scope, but it's not the best tool for that job. I also find that if you use `let` everywhere, then it's less obvious which declarations are designed to be localized and which ones are intended to be used throughout the function.

By contrast, I rarely use a `var` inside a block. That's what `let` is for. Use the best tool for the job. If you see a `let`, it tells you that you're dealing with a localized declaration. If you see `var`, it tells you that you're dealing with a function-wide declaration. Simple as that.

```
function getStudents(data) {
    var studentRecords = [];

    for (let record of data.records) {
        let id = `student-${ record.id }`;
        studentRecords.push({
            id,
            record.name
        });
    }

    return studentRecords;
}
```

The `studentRecords` variable is intended for use across the whole function. `var` is the best declarator to tell the reader that. By contrast, `record` and `id` are intended for use only in the narrower scope of the loop iteration, so `let` is the best tool for that job.

In addition to this *best tool* semantic argument, `var` has a few other characteristics that, in certain limited circumstances, make it more powerful.

One example is when a loop is exclusively using a variable, but its conditional clause cannot see block-scoped declarations inside the iteration:

```
function commitAction() {
    do {
        let result = commit();
        var done = result && result.code == 1;
    } while (!done);
}
```

Here, `result` is clearly only used inside the block, so we use `let`. But `done` is a bit different. It's only useful for the loop, but the `while` clause cannot see `let` declarations that appear inside the loop. So we compromise and use `var`, so that `done` is hoisted to the outer scope where it can be seen.

The alternative—declaring `done` outside the loop—separates it from where it's first used, and either necessitates picking a default value to assign, or worse, leaving it unassigned and thus looking ambiguous to the reader. I think `var` inside the loop is preferable here.

Another helpful characteristic of `var` is seen with declarations inside unintended blocks. Unintended blocks are blocks that are created because the syntax requires a block, but where the intent of the developer is not really to create a localized scope. The best illustration of unintended scope is the `try..catch` statement:

```
function getStudents() {
    try {
        // not really a block scope
        var records = fromCache("students");
    }
    catch (err) {
        // oops, fall back to a default
        var records = [];
    }
    // ..
}
```

There are other ways to structure this code, yes. But I think this is the *best* way, given various trade-offs.

I don't want to declare `records` (with `var` or `let`) outside of the `try` block, and then assign to it in one or both blocks. I prefer initial declarations to always be as close as possible (ideally, same line) to the first usage of the variable. In this simple example, that would only be a couple of lines distance, but in real code it can grow to many more lines. The bigger the gap, the harder it is to figure out what variable from what scope you're assigning to. `var` used at the actual assignment makes it less ambiguous.

Also notice I used `var` in both the `try` and `catch` blocks. That's because I want to signal to the reader that no matter which path is taken, `records` always gets declared. Technically, that works because `var` is hoisted once to the function scope. But it's still a nice semantic signal to remind the reader what either `var` ensures. If `var` were only used in one of the blocks, and you were only reading the other block, you wouldn't as easily discover where `records` was coming from.

This is, in my opinion, a little superpower of `var`. Not only can it escape the unintentional `try..catch` blocks, but it's allowed to appear multiple times in a function's scope. You can't do that with `let`. It's not bad, it's actually a little helpful feature. Think of `var` more like a declarative annotation that's reminding you, each usage, where the variable comes from. "Ah ha, right, it belongs to the whole function."

This repeated-annotation superpower is useful in other cases:

```
function getStudents() {
    var data = [];

    // do something with data
    // .. 50 more lines of code ..

    // purely an annotation to remind us
    var data;

    // use data again
    // ..
}
```

The second `var data` is not re-declaring `data`, it's just annotating for the readers' benefit that `data` is a function-wide declaration. That way, the reader doesn't need to scroll up 50+ lines of code to find the initial declaration.

I'm perfectly fine with re-using variables for multiple purposes throughout a function scope. I'm also perfectly fine with having two usages of a variable be separated by quite a few lines of code. In both cases, the ability to safely "re-declare" (annotate) with `var` helps make sure I can tell where my `data` is coming from, no matter where I am in the function.

Again, sadly, `let` cannot do this.

There are other nuances and scenarios when `var` turns out to offer some assistance, but I'm not going to belabor the point any further. The takeaway is that `var` can be useful in our programs alongside `let` (and the occasional `const`). Are you willing to creatively use the tools the JS language provides to tell a richer story to your readers?

Don't just throw away a useful tool like `var` because someone shamed you into thinking it wasn't cool anymore. Don't avoid `var` because you got confused once years ago. Learn these tools and use them each for what they're best at.

What's the Deal with TDZ?

The TDZ (temporal dead zone) was explained in Chapter 5. We illustrated how it occurs, but we skimmed over any explanation of *why* it was necessary to introduce in the first place. Let's look briefly at the motivations of TDZ.

Some breadcrumbs in the TDZ origin story:

- `const`s should never change
- It's all about time
- Should `let` behave more like `const` or `var`?

Where It All Started

TDZ comes from `const`, actually.

During early ES6 development work, TC39 had to decide whether `const` (and `let`) were going to hoist to the top of their blocks. They decided these declarations would hoist, similar to how `var` does. Had that not been the case, I think some of the fear was confusion with mid-scope shadowing, such as:

```
let greeting = "Hi!";

{
    // what should print here?
    console.log(greeting);

    // .. a bunch of lines of code ..

    // now shadowing the `greeting` variable
    let greeting = "Hello, friends!";

    // ..
}
```

What should we do with that `console.log(..)` statement? Would it make any sense to JS devs for it to print "Hi!"? Seems like that could be a gotcha, to have shadowing kick in only for the second half of the block, but not the first half. That's not very intuitive, JS-like behavior. So `let` and `const` have to hoist to the top of the block, visible throughout.

But if `let` and `const` hoist to the top of the block (like `var` hoists to the top of a function), why don't `let` and `const` auto-initialize (to `undefined`) the way `var` does? Here was the main concern:

```
{
    // what should print here?
    console.log(studentName);

    // later

    const studentName = "Frank";

    // ..
}
```

Let's imagine that `studentName` not only hoisted to the top of this block, but was also auto-initialized to `undefined`. For the first half of the block, `studentName` could be observed to have the `undefined` value, such as with our `console.log(..)` statement. Once the `const studentName = ..` statement is reached, now `studentName` is assigned `"Frank"`. From that point forward, `studentName` can't ever be re-assigned.

But, is it strange or surprising that a constant observably has two different values, first `undefined`, then `"Frank"`? That does seem to go against what we think a `const`ant means; it should only ever be observable with one value.

So... now we have a problem. We can't auto-initialize `studentName` to `undefined` (or any other value for that matter). But the variable has to exist throughout the whole scope. What do we do with the period of time from when it first exists (beginning of scope) and when it's assigned its value?

We call this period of time the "dead zone," as in the "temporal dead zone" (TDZ). To prevent confusion, it was determined that any sort of access of a variable while in its TDZ is illegal and must result in the TDZ error.

OK, that line of reasoning does make some sense, I must admit.

Who `let` the TDZ Out?

But that's just `const`. What about `let`?

Well, TC39 made the decision: since we need a TDZ for `const`, we might as well have a TDZ for `let` as well. *In fact, if we make let have a TDZ, then we discourage all that ugly variable hoisting people do.* So there was a consistency perspective and, perhaps, a bit of social engineering to shift developers' behavior.

My counter-argument would be: if you're favoring consistency, be consistent with `var` instead of `const`; `let` is definitely more like `var` than `const`. That's especially true since they had already chosen consistency with `var` for the whole hoisting-to-the-top-of-the-scope thing. Let `const` be its own unique deal with a TDZ, and let the answer to TDZ purely be: just avoid the TDZ by always declaring your constants at the top of the scope. I think this would have been more reasonable.

But alas, that's not how it landed. `let` has a TDZ because `const` needs a TDZ, because `let` and `const` mimic `var` in their hoisting to the top of the (block) scope. There ya go. Too circular? Read it again a few times.

Are Synchronous Callbacks Still Closures?

Chapter 7 presented two different models for tackling closure:

- Closure is a function instance remembering its outer variables even as that function is passed around and **invoked in** other scopes.
- Closure is a function instance and its scope environment being preserved in-place while any references to it are passed around and **invoked from** other scopes.

These models are not wildly divergent, but they do approach from a different perspective. And that different perspective changes what we identify as a closure.

Don't get lost following this rabbit trail through closures and callbacks:

- Calling back to what (or where)?
- Maybe "synchronous callback" isn't the best label
- ***IIF*** functions don't move around, why would they need closure?
- Deferring over time is key to closure

What is a Callback?

Before we revisit closure, let me spend a brief moment addressing the word "callback." It's a generally accepted norm that saying "callback" is synonymous with both *asynchronous callbacks* and *synchronous callbacks*. I don't think I agree that this is a good idea, so I want to explain why and propose we move away from that to another term.

Let's first consider an *asynchronous callback*, a function reference that will be invoked at some future *later* point. What does "callback" mean, in this case?

It means that the current code has finished or paused, suspended itself, and that when the function in question is invoked later, execution is entering back into the suspended program, resuming it. Specifically, the point of re-entry is the code that was wrapped in the function reference:

```
setTimeout(function waitForASecond(){
    // this is where JS should call back into
    // the program when the timer has elapsed
},1000);

// this is where the current program finishes
// or suspends
```

In this context, "calling back" makes a lot of sense. The JS engine is resuming our suspended program by *calling back in* at a specific location. OK, so a callback is asynchronous.

Synchronous Callback?

But what about *synchronous callbacks*? Consider:

```
function getLabels(studentIDs) {
    return studentIDs.map(
        function formatIDLabel(id){
            return `Student ID: ${
               String(id).padStart(6)
            }`;
        }
    );
}

getLabels([ 14, 73, 112, 6 ]);
// [
//    "Student ID: 000014",
//    "Student ID: 000073",
//    "Student ID: 000112",
//    "Student ID: 000006"
// ]
```

Should we refer to `formatIDLabel(..)` as a callback? Is the `map(..)` utility really *calling back* into our program by invoking the function we provided?

There's nothing to *call back into* per se, because the program hasn't paused or exited. We're passing a function (reference) from one part of the program to another part of the program, and then it's immediately invoked.

There's other established terms that might match what we're doing—passing in a function (reference) so that another part of the program can invoke it on our behalf. You might think of this as *Dependency Injection* (DI) or *Inversion of Control* (IoC).

DI can be summarized as passing in necessary part(s) of functionality to another part of the program so that it can invoke them to complete its work. That's a decent description for the `map(..)` call above, isn't it? The `map(..)` utility knows to iterate over the list's values, but it doesn't know what to *do* with those values. That's why we pass it the `formatIDLabel(..)` function. We pass in the dependency.

IoC is a pretty similar, related concept. Inversion of control means that instead of the current area of your program controlling what's happening, you hand control off to another part of the program. We wrapped the logic for computing a label string in the function `formatIDLabel(..)`, then handed invocation control to the `map(..)` utility.

Notably, Martin Fowler cites IoC as the difference between a framework and a library: with a library, you call its functions; with a framework, it calls your functions. [1]

In the context of our discussion, either DI or IoC could work as an alternative label for a *synchronous callback*.

But I have a different suggestion. Let's refer to (the functions formerly known as) *synchronous callbacks*, as *inter-invoked functions* (IIFs). Yes, exactly, I'm playing off IIFEs. These kinds of functions are *inter-invoked*, meaning: another entity invokes them, as opposed to IIFEs, which invoke themselves immediately.

What's the relationship between an *asynchronous callback* and an IIF? An *asynchronous callback* is an IIF that's invoked asynchronously instead of synchronously.

Synchronous Closure?

Now that we've re-labeled *synchronous callbacks* as IIFs, we can return to our main question: are IIFs an example of closure? Obviously, the IIF would have to reference variable(s) from an outer scope for it to have any chance of being a closure. The `formatIDLabel(..)` IIF from earlier does not reference any variables outside its own scope, so it's definitely not a closure.

What about an IIF that does have external references, is that closure?

```
function printLabels(labels) {
    var list = document.getElementByID("labelsList");

    labels.forEach(
        function renderLabel(label){
            var li = document.createELement("li");
            li.innerText = label;
            list.appendChild(li);
        }
    );
}
```

The inner `renderLabel(..)` IIF references `list` from the enclosing scope, so it's an IIF that *could* have closure. But here's where the definition/model we choose for closure matters:

- If `renderLabel(..)` is a **function that gets passed somewhere else**, and that function is then invoked, then yes, `renderLabel(..)` is exercising a closure, because closure is what preserved its access to its original scope chain.

- But if, as in the alternative conceptual model from Chapter 7, `renderLabel(..)` stays in place, and only a reference to it is passed to `forEach(..)`, is there any need for closure to preserve the scope chain of `renderLabel(..)`, while it executes synchronously right inside its own scope?

No. That's just normal lexical scope.

To understand why, consider this alternative form of `printLabels(..)`:

```
function printLabels(labels) {
    var list = document.getElementByID("labelsList");

    for (let label of labels) {
        // just a normal function call in its own
        // scope, right? That's not really closure!
        renderLabel(label);
    }

    // **************

    function renderLabel(label) {
        var li = document.createELement("li");
        li.innerText = label;
        list.appendChild(li);
    }
}
```

These two versions of `printLabels(..)` are essentially the same.

The latter one is definitely not an example of closure, at least not in any useful or observable sense. It's just lexical scope. The former version, with `forEach(..)` calling our function reference, is essentially the same thing. It's also not closure, but rather just a plain ol' lexical scope function call.

Defer to Closure

By the way, Chapter 7 briefly mentioned partial application and currying (which *do* rely on closure!). This is a interesting scenario where manual currying can be used:

```
function printLabels(labels) {
    var list = document.getElementByID("labelsList");
    var renderLabel = renderTo(list);

    // definitely closure this time!
    labels.forEach( renderLabel );

    // **************

    function renderTo(list) {
        return function createLabel(label){
            var li = document.createELement("li");
            li.innerText = label;
            list.appendChild(li);
        };
    }
}
```

The inner function `createLabel(..)`, which we assign to `renderLabel`, is closed over `list`, so closure is definitely being utilized.

Closure allows us to remember `list` for later, while we defer execution of the actual label-creation logic from the `renderTo(..)` call to the subsequent `forEach(..)` invocations of the `createLabel(..)` IIF. That may only be a brief moment here, but any amount of time could pass, as closure bridges from call to call.

Classic Module Variations

Chapter 8 explained the classic module pattern, which can look like this:

```
var StudentList = (function defineModule(Student){
    var elems = [];

    var publicAPI = {
        renderList() {
            // ..
        }
    };

    return publicAPI;

})(Student);
```

Notice that we're passing `Student` (another module instance) in as a dependency. But there's lots of useful variations on this module form you may encounter. Some hints for recognizing these variations:

- Does the module know about its own API?
- Even if we use a fancy module loader, it's just a classic module
- Some modules need to work universally

Where's My API?

First, most classic modules don't define and use a `publicAPI` the way I have shown in this code. Instead, they typically look like:

```
var StudentList = (function defineModule(Student){
    var elems = [];

    return {
        renderList() {
            // ..
        }
    };

})(Student);
```

The only difference here is directly returning the object that serves as the public API for the module, as opposed to first saving it to an inner `publicAPI` variable. This is by far how most classic modules are defined.

But I strongly prefer, and always use myself, the former `publicAPI` form. Two reasons:

- `publicAPI` is a semantic descriptor that aids readability by making it more obvious what the purpose of the object is.
- Storing an inner `publicAPI` variable that references the same external public API object returned, can be useful if you need to access or modify the API during the lifetime of the module.

  For example, you may want to call one of the publicly exposed functions, from inside the module. Or, you may want to add or remove methods depending on certain conditions, or update the value of an exposed property.

  Whatever the case may be, it just seems rather silly to me that we *wouldn't* maintain a reference to access our own API. Right?

Asynchronous Module Defintion (AMD)

Another variation on the classic module form is AMD-style modules (popular several years back), such as those supported by the RequireJS utility:

```
define([ "./Student" ],function StudentList(Student){
    var elems = [];

    return {
        renderList() {
            // ..
        }
    };
});
```

If you look closely at `StudentList(..)`, it's a classic module factory function. Inside the machinery of `define(..)` (provided by RequireJS), the `StudentList(..)` function is executed, passing to it any other module instances declared as dependencies. The return value is an object representing the public API for the module.

This is based on exactly the same principles (including how the closure works!) as we explored with classic modules.

Universal Modules (UMD)

The final variation we'll look at is UMD, which is less a specific, exact format and more a collection of very similar formats. It was designed to create better interop (without any build-tool conversion) for modules that may be loaded in browsers, by AMD-style loaders, or in Node. I personally still publish many of my utility libraries using a form of UMD.

Here's the typical structure of a UMD:

```
(function UMD(name,context,definition){
    // loaded by an AMD-style loader?
    if (
        typeof define === "function" &&
        define.amd
    ) {
        define(definition);
    }
    // in Node?
    else if (
        typeof module !== "undefined" &&
        module.exports
    ) {
        module.exports = definition(name,context);
    }
    // assume standalone browser script
    else {
        context[name] = definition(name,context);
    }
})("StudentList",this,function DEF(name,context){

    var elems = [];

    return {
        renderList() {
            // ..
        }
    };

});
```

Though it may look a bit unusual, UMD is really just an IIFE.

What's different is that the main `function` expression part (at the top) of the IIFE contains a series of `if..else if` statements to detect which of the three supported environments the module is being loaded in.

The final `()` that normally invokes an IIFE is being passed three arguments: `"StudentsList"`, `this`, and another `function` expression. If you match those arguments to their parameters, you'll see they are: `name`, `context`, and `definition`, respectively. `"StudentList"` (`name`) is the name label for the module, primarily in case it's defined as a global variable. `this` (`context`) is generally the `window` (aka, global object; see Chapter 4) for defining the module by its name.

`definition(..)` is invoked to actually retrieve the definition of the module, and you'll notice that, sure enough, that's just a classic module form!

There's no question that as of the time of this writing, ESM (ES Modules) are becoming popular and widespread rapidly. But with millions and millions of modules written over the last 20 years, all using some pre-ESM variation of classic modules, they're still very important to be able to read and understand when you come across them.

---

1. *Inversion of Control*, Martin Fowler, https://martinfowler.com/bliki/InversionOfControl.html, 26 June 2005. ↩