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

Chapter 4: Block Scope

NOTE: |
:— |
Work in progress |

If you made it this far, through that long Chapter 3, you're likely feeling a lot more aware of, and hopefully more comfortable with, the breadth and depth of scopes and their impact on your code.

Now, we want to focus on one specific form of scope: block scope. Just as we're narrowing our discussion focus in this chapter, so too is block scope about narrowing the focus of a larger scope down to a smaller subset of our program.

Least Exposure

It makes sense that functions define their own scopes. But why do we need blocks to create scopes as well?

Software engineering articulates a fundamental pattern, typically applied to software security, called "The Principle of Least Privilege" (POLP, https://en.wikipedia.org/wiki/Principle_of_least_privilege). And a variation of this principle that applies to our current discussion is typically labeled "Least Exposure".

POLP expresses a defensive posture to software architecture: components of the system should be designed to function with least privilege, least access, least exposure. If each piece is connected with minimum-necessary capabilities, the overall system is stronger from a security standpoint, because a compromise or failure of one piece has a minimized impact on the rest of the system.

If PLOP focuses on system-level component design, the *Exposure* variant (POLE) can be focused on a lower level; we'll apply it to our exploration of how scopes interact with each other.

In following POLE, what do we want to minimize the exposure of? The variables registered in each scope.

Think of it this way: why wouldn't you just place all the variables of your program out in the global scope? That probably immediately feels like a bad idea, but it's worth considering why that is. When variables used by one part of the program are exposed to another part of the program, via scope, there are 3 main hazards that can arise:

1. **Naming Collisions**: if you use a common and useful variable/function name in two different parts of the program, but the identifier comes from one shared scope (like the global scope), then name collision occurs, and it's very likely that bugs will occur as one part uses the variable/function in a way the other part doesn't expect. For example, imagine if all your loops used a single global `i` index variable, and then it happened that one loop in a function was run during an iteration of a loop from another function, and now the shared `i` variable has an unexpected value.

2. **Unexpected Behavior**: if you expose variables/functions whose usage is otherwise *private* to a piece of the program, it allows other developers to use them in ways you didn't intend, which can violate expected behavior and cause bugs. For example, if your part of the program assumes an array contains all numbers, but someone else's code accesses and modifies the array to include booleans or strings, your code may then misbehave in unexpected ways.

   Worse, exposure of *private* details invites those with mal-intent to try to work around limitations you have imposed, to do things with your part of the software that shouldn't be allowed.

3. **Unintended Dependency**: if you expose variables/functions unnecessarily, it invites other developers to use and depend on those otherwise *private* pieces. While that doesn't break your program today, it creates a refactoring hazard in the future, because now you cannot as easily refactor that variable or function without potentially breaking other parts of the software that you don't control. For example, If your code relies on an array of numbers, and you later decide it's better to use some other data structure instead of an array, you now must take on the liability of fixing other affected parts of the software.

POLE, as applied to variable/function scoping, essentially says, default to exposing the bare minimum necessary, keeping everything else as private as possible. Declare variables in as small and deeply nested of scopes as possible, rather than placing everything in the global (or even outer function) scope.

If you design your software accordingly, you have a much greater chance of avoiding (or at least minimizing) these 3 hazards.

Consider:

```
function diff(x,y) {
    if (x > y) {
        let tmp = x;
        x = y;
        y = tmp;
    }
```

```
    return y - x;
}

diff(3,7);      // 4
diff(7,5);      // 2
```

In this `diff(..)` function, we want to ensure that `y` is greater than or equal to `x`, so that when we subtract (`y - x`), the result is `0` or larger. If `x` is larger, we swap `x` and `y` using a `tmp` variable.

In this simple example, it doesn't seem to matter whether `tmp` is inside the `if` block or whether it belongs at the function level – but it certainly shouldn't be a global variable! However, following the POLE principle, `tmp` should be as hidden in scope as possible. So we block scope `tmp` (using `let`) to the `if` block.

Hiding In Plain (Function) Scope

It should now be clear why it's important to hide our variable and function declarations in the lowest (most deeply nested) scopes possible. But how do we do so?

You've already seen `let` (and `const`) declarations, which are block scoped declarators, and we'll come back to them in more detail shortly. But what about hiding `var` or `function` declarations in scopes? That can easily be done by wrapping a `function` scope around a declaration.

Let's consider an example where `function` scoping can be useful.

The mathematical operation "factorial" (notated as "6!") is the multiplication of a given integer against all successively lower integers down to `1` – actually, you can stop at `2` since multiplying `1` does nothing. In other words, "6!" is the same as "6 * 5!", which is the same as "6 * 5 * 4!", and so on. Because of the nature of the math involved, once any given integer's factorial (like "4!") has been calculated, we shouldn't need to do that work again, as it'll always be the same answer.

So if you naively calculate factorial for `6`, then later want to calculate factorial for `7`, you might unnecessarily re-calculate the factorials of all the integers from `1` up to `6`. If you're willing to trade memory for speed, you can solve that wasted computation by caching each integer's factorial as it's calculated:

```
var cache = {};

function factorial(x) {
    if (x < 2) return 1;
    if (!(x in cache)) {
        cache[x] = x * factorial(x - 1);
```

```
    }
    return cache[x];
}

factorial(6);
// 720

cache;
// {
//      "2": 2,
//      "3": 6,
//      "4": 24,
//      "5": 120,
//      "6": 720
// }

factorial(7);
// 5040
```

We're storing all the computed factorials in `cache` so that across multiple calls to `factorial(..)`, the answers remain. But the `cache` variable is pretty obviously a *private* detail of how `factorial(..)` works, not something that should be exposed in an outer scope – especially not the global scope.

NOTE: |
:— |
`factorial(..)` here is recursive – a call to itself is made from inside – but that's just for brevity of code sake; a non-recursive implementation would yield the same scoping analysis with respect to `cache`. |

However, fixing this over-exposure issue is not as simple as hiding the `cache` variable inside `factorial(..)`, as it might seem. Since we need `cache` to survive multiple calls, it must be located in a scope outside that function. So what can we do?

Define another middle scope (between the outer/global scope and the inside of `factorial(..)`) for `cache` to live in:

```
// outer/global scope

function hideTheCache() {
    // "middle scope", where we hide `cache`
    var cache = {};

    function factorial(x) {
        // inner scope
        if (x < 2) return 1;
```

```
        if (!(x in cache)) {
            cache[x] = x * factorial(x - 1);
        }
        return cache[x];
    }

    return factorial;
}

var factorial = hideTheCache();

factorial(6);
// 720

factorial(7);
// 5040
```

The `hideTheCache()` function serves no other purpose than to create a scope for `cache` to persist in across multiple calls to `factorial(..)`. But for `factorial(..)` to have access to `cache`, we have to define `factorial(..)` inside that same scope. Then we return the function reference, as a value from `hideTheCache()`, and store it back in an outer scope variable, also named `factorial`. Now we can call `factorial(..)` (multiple times!); its persistent `cache` stays hidden but accessible only to `factorial(..)`!

OK, but... it's going to be tedious to define (and name!) a `hideTheCache(..)` function scope each time such a need for variable/function hiding occurs, especially since we'll likely want to avoid name collisions with this function by giving each occurrence a unique name. Ugh.

NOTE: |
:— |
The illustrated technique – caching a function's computed output to optimize performance when repeated calls of the same inputs are expected – is quite common in the Functional Programming (FP) world, canonically referred to as "memoization". FP libraries will provide a utility for memoization of functions, which would take the place of `hideTheCache(..)` above. Memoization is beyond the *scope* (pun intended!) of our discussion. See my "Functional-Light JavaScript" book for more information. |

Rather than defining a new and uniquely named function each time one of those scope-only-for-the-purpose-of-hiding-a-variable situations occurs, a perhaps better solution is to use a function expression:

```
var factorial = (function hideTheCache() {
    var cache = {};
```

```
    function factorial(x) {
        if (x < 2) return 1;
        if (!(x in cache)) {
            cache[x] = x * factorial(x - 1);
        }
        return cache[x];
    }

    return factorial;
})();

factorial(6);
// 720

factorial(7);
// 5040
```

Wait! This is still using a function to create the scope for hiding `cache`, and in this case, the function is still named `hideTheCache`, so how does that solve anything?

Recall from "Function Name Scope" (Chapter 3), what happens to the name identifier from a function expression. Since `hideTheCache(..)` is defined as a `function` expression instead of a `function` declaration, its name is in its own scope – essentially the same scope as `cache` – rather than in the outer/global scope.

That means we could name every single occurrence of such a function expression the exact same name, and never have any collision. More appropriately, we could name each occurrence semantically based on whatever it is we're trying to hide, and not worry that whatever name we choose is going to collide with any other `function` expression scope in the program.

In fact, as we discussed in "Function Name Scope" (Chapter 3), we *could* just leave off the name entirely – thus defining an "anonymous `function` expression" instead. Appendix A will explore the importance of names for such functions.

Invoking Function Expressions Immediately

But there's another important bit in the above program that's easy to miss: the line at the end of the `function` expression that contains `})();`.

Notice that we surrounded the entire `function` expression in a set of (..), and then on the end, we added that second () parentheses set; that's actually calling the `function` expression we just defined. Moreover, in this case, the first set of surrounding (..) around the function expression is not strictly

necessary (more on that in a moment), but we used them for readability sake anyway.

So, in other words, we're defining a `function` expression that's then immediately invoked. This common pattern has a (very creative!) name: Immediately Invoked Function Expression (IIFE).

IIFEs are useful when we want to create a scope to hide variables/functions. Since they are expressions, they can be used in **any** place in a JS program where an expression is allowed. IIFEs can be named, as with `hideTheCache()`, or (much more commonly!) unnamed/anonymous. And they can be standalone or, as above, part of another statement – `hideTheCache()` returns the `factorial()` function reference which is then `=` assigned to a variable `factorial`.

For comparison, here's an example of a standalone (anonymous) IIFE:

```
// outer scope

(function(){

    // inner hidden scope

})();

// more outer scope
```

Unlike earlier with `hideTheCache()`, where the outer surrounding `(..)` were noted as being an optional stylistic choice, for standalone IIFEs, they're **required**, to distinguish the `function` as an expression and not a statement. So for consistency, it's best to always surround IIFE `function`s with (..).

NOTE: |
:— |
Technically, the surrounding (..) aren't the only syntactic way to ensure a `function` in an IIFE is treated by the JS parser as a function expression. We'll look at some other options in Appendix A. |

Function Boundaries

Be aware that using an IIFE to define a scope can have some unintended consequences, depending on the code around it. Because an IIFE is a full function, the function boundary alters the behavior of certain statements/constructs.

For example, a `return` statement in some piece of code would change its meaning if then wrapped in an IIFE, because now the `return` would refer to the IIFE's function. Non-arrow function IIFEs also change the binding of a `this` keyword.

And statements like `break` and `continue` won't operate across an IIFE function boundary to control an outer loop or block.

So, if the code you need to wrap a scope around has `return`, `this`, `break`, or `continue` in it, an IIFE is probably not the best approach. In that case, you might look to create the scope with a block instead of a function.

Scoping With Blocks

You should by this point feel fairly comfortable with creating scopes to prevent unnecessary identifier exposure.

And so far, we looked at doing this via `function` (i.e., IIFE) scope. But let's now consider using `let` declarations with nested blocks. In general, any `{ .. }` curly-brace pair which is a statement will act as a block, but **not necessarily** as a scope. A block only becomes a scope if it needs to, to contain any block-scoped declarations (i.e., `let` or `const`) present within it.

Consider:

```
{
    let thisIsNowAScope = true;

    for (let i = 0; i < 5; i++) {
        // this is also a scope, activated each iteration

        if (i % 2 == 0) {
            // this is just a block, not a scope
            console.log(i);
        }
    }
}
// 0 2 4
```

NOTE: |
:— |
Not all `{ .. }` curly-brace pairs create blocks (and thus are eligible to become scopes). Object literals use `{ .. }` curly-brace pairs to delimit their key-value lists, but such objects are **not** scopes. `class` uses `{ .. }` curly-braces around its body definition, but this is not a block or scope. A `function` uses `{ .. }` around its body, but this is not technically a block – it's a single statement for the function body – though it *is* a (function) scope. The `{ .. }` curly-brace pair around the `case` clauses of a `switch` statement does not define a block/scope. |

Blocks can be defined as part of a statement (like an `if` or `for`), or as bare standalone block statements, as shown in the outermost `{ .. }` curly brace

pair in the snippet above. An explicit block of this sort, without any declarations (and thus, not actually a scope) serves no operational purpose, though it can be useful as a stylistic signal.

Explicit blocks were always valid JS syntax, but since they couldn't be a scope, they were extremely uncommon prior to ES6's introduction of `let` / `const`. Now they're starting to catch on a little bit.

In most languages that support block scoping, an explicit block scope is a very common pattern for creating a narrow slice of scope for one or a few variables. So following the POLE principle, we should embrace this pattern more widespread in JS as well; use block scoping to narrow the exposure of identifiers to the minimum practical.

An explicit block scope can be useful even inside of another block (whether it's a scope or not).

For example:

```
if (somethingHappened) {
    // this is a block, but not a scope

    {
        // this is an explicit block scope
        let msg = somethingHappened.message();
        notifyOthers(msg);
    }

    recoverFromSomething();
}
```

Here, the `{ .. }` curly-brace pair inside the `if` statement is an even smaller inner explicit block scope for `msg`, since that variable is not needed for the entire `if` block. Most developers would just block-scope `msg` to the `if` block and move on. When there's only a few lines to consider, it's a toss-up judgement call. As code grows, these issues become more pronounced.

So does it matter enough to add the extra `{ .. }` pair and indentation level? I think you should follow POLE and always define the smallest block (within reason!) for each variable. So I would recommend using the extra explicit block scope.

WARNING: |
:— |
Recall the discussion of TDZ errors from "Uninitialized" (Chapter 3). My suggestion to minimize the risk of TDZ errors with `let` (or `const`) declarations is to always put the declarations at the top of any scope. If you find yourself putting a `let` declaration in the middle of a scope block, first think, "Oh, no!

TDZ alert!". Then recognize that if this `let` declaration isn't actually needed for the whole block, you could/should use an inner explicit block scope to further narrow its exposure! |

Another example making use of an explicit block scope:

```
function getNextMonthStart(dateStr) {
    var nextMonth, year;

    {
        let curMonth;
        [ , year, curMonth ] = dateStr.match(/(\d{4})-(\d{2})-\d{2}/) || [];
        nextMonth = (Number(curMonth) % 12) + 1;
    }

    // did we cross a year boundary?
    if (nextMonth == 1) {
        year++;
    }

    return `${ year }-${ String(nextMonth).padStart(2,"0") }-01`;
}

getNextMonthStart("2019-12-25");
// 2020-01-01
```

Let's first identify the scopes and their identifiers:

1. The outer/global scope has one identifier, the function `getNextMonthStart(..)`.

2. The function scope for `getNextMonthStart(..)` has three identifiers: `dateStr` (the parameter), `nextMonth`, and `year`.

3. The `{ .. }` curly-brace pair defines an inner block scope that includes one variable: `curMonth`.

So why did we put `curMonth` in an explicit block scope instead of just alongside `nextMonth` and `year` in the top-level function scope? Because `curMonth` is only needed for those first two statements. Exposing it at the function scope level is, in essence, over-exposing it.

This example is small, so the hazards are pretty minimal in over-exposing the scoping of `curMonth`. But the benefits of the POLE principle are best achieved when you adopt the mindset of minimizing scope exposure as a habit. If you follow the principle consistently even in the small cases, it will serve you more as your programs grow.

Let's now look at an even more substantial example:

```javascript
function sortNamesByLength(names) {
    var buckets = [];

    for (let firstName of names) {
        if (buckets[firstName.length] == null) {
            buckets[firstName.length] = [];
        }
        buckets[firstName.length].push(firstName);
    }

    // a block to narrow the scope
    {
        let sortedNames = [];

        for (let bucket of buckets) {
            if (bucket) {
                // sort each bucket alphanumerically
                bucket.sort();

                // append the sorted names to our running list
                sortedNames = [ ...sortedNames, ...bucket ];
            }
        }

        return sortedNames;
    }
}

sortNamesByLength([
    "Sally",
    "Suzy",
    "Frank",
    "John",
    "Jennifer",
    "Scott"
]);
// [ "John", "Suzy", "Frank", "Sally", "Scott", "Jennifer" ]
```

There are six identifiers declared across five different scopes. Could all of these variables have existed in the single outer/global scope? Technically, yes, since they're all uniquely named and thus have no name collisions. But this would be really poor code organization, and would likely lead to both confusion and future bugs.

We split them out into each inner nested scope as appropriate. Each variable is defined at the innermost scope possible for the program to operate as desired.

`sortedNames` could have been defined in the top-level function scope, but it's only needed for the second half of this function. To avoid over-exposing that variable in a higher level scope, we again follow POLE and block-scope it in the inner block scope.

`var` AND `let`

Next, let's talk about the declaration `var buckets`. That variable is used across the entire function (except the final `return` statement). Any variable that is needed across all (or even most) of a function should be declared so that such usage is obvious.

NOTE: |
:— |
The parameter `names` isn't used across the whole function, but there's no way limit the scope of a parameter, so it behaves as a function-wide declaration regardless. |

So why did we use `var` instead of `let` to declare the `buckets` variable? There's both stylistic and technical reasons to choose `var` here.

Stylistically, `var` has always, from the earliest days of JS, signaled "variable that belongs to a whole function". As we asserted in "Lexical Scope" (Chapter 1), `var` attaches to the nearest enclosing function scope, no matter where it appears. That's true even if `var` appears inside a block:

```
function diff(x,y) {
    if (x > y) {
        var tmp = x;    // 'tmp' is function-scoped
        x = y;
        y = tmp;
    }

    return y - x;
}
```

Even though `var` is inside a block, its declaration is function-scoped (to `diff(..)`), not block-scoped.

While you can declare `var` inside a block (and still have it be function-scoped), I would recommend you minimize this approach except in a few specific cases (discussed in Appendix A). Otherwise, `var` should be reserved for use in the top-level scope of a function.

Why not just use `let` in that same location? Because `var` is visually distinct from `let` and therefore signals clearly, "this variable is function-scoped". Using `let` in the top-level scope, especially if not in the first few lines of a function,

especially when all the other declarations in blocks use `let`, does not visually draw attention to the difference with the function-scoped declaration.

In other words, I feel `var` better communicates function-scoped than `let` does, and `let` both communicates (and achieves!) block-scoping where `var` is insufficient. As long as your programs are going to need both function-scoped and block-scoped variables, the most sensible and readable approach is to use both `var` AND `let` together, each for their own best purpose.

There are other stylistic and operational reasons to choose `var` or `let` in different scenarios. We'll explore the case for `var` alongside `let` in more detail in Appendix A.

WARNING: |
:— |
My advice to use `var` AND `let` here is controversial. It's far more common to hear assertions like, "var is broken, let fixes it" and, "never use var, let is the replacement". Those opinions are as valid as my opinions, but they're just opinions; `var` is not factually broken or deprecated. It has worked since early JS and it will continue to always work as long as JS is around. |

When To `let`?

My advice to reserve `var` for (mostly only) a top-level function scope means that all other declarations should use `let`. But you may still be wondering how to decide where each declaration in your program belongs?

POLE already guides you on those decisions, but let's make sure we explicitly state it. The way to decide is not based on which keyword you want to use. The way to decide is to ask, "What is the most minimal scope exposure that's sufficient for this variable?" Once that is answered, you'll know if a variable belongs in a block scope or the function scope. If you decide initially that a variable should be block-scoped, and later realize it needs to be elevated to be function-scoped, then that dictates a change not only in the location of that variable's declaration, but also the keyword used. The decision making process really should proceed like that

If a declaration belongs in a block scope, use `let`. If it belongs in the function scope, use `var` (again, my opinion).

But another way to sort of visualize this decision making is to consider the pre-ES6 version of a program. For example, let's recall `diff(..)` from earlier:

```
function diff(x,y) {
    var tmp;

    if (x > y) {
        tmp = x;
```

```
        x = y;
        y = tmp;
    }

    return y - x;
}
```

In this version of `diff(..)`, `tmp` is clearly declared in the function scope. Is that appropriate for `tmp`? I would argue, no. `tmp` is only needed for those few statements. It's not needed once we get to the `return` statement. It should therefore be block-scoped.

Prior to ES6, we didn't have `let` so we couldn't *actually* block-scope it. But we could do the next-best thing:

```
function diff(x,y) {
    if (x > y) {
        // 'tmp' is still function-scoped, but
        // it signals block-scoping stylistically
        var tmp = x;
        x = y;
        y = tmp;
    }

    return y - x;
}
```

Placing the `var` declaration for `tmp` inside the `if` statement signals to the reader of the code that `tmp` belongs to that block. Even though JS doesn't enforce that scoping, the stylistic signal still has benefit for the reader of your code.

Now, you can just locate any `var` that's inside a block of this sort and switch it to `let` to enforce the signal already being sent stylistically.

Another example that used to be commonly based on `var` but which should pretty much always be `let` is a `for` loop:

```
for (var i = 0; i < 5; i++) {
    // do something
}
```

No matter where such a loop is defined, the `i` should basically always be used only inside the loop, in which case POLE tells us it should be declared with `let` instead of `var`:

```
for (let i = 0; i < 5; i++) {
    // do something
}
```

Basically the only time switching a `var` to a `let` in this way would "break" your code is if you were relying on accessing the loop's iterator (`i`) outside/after the loop, such as:

```
for (var i = 0; i < 5; i++) {
    if (checkValue(i)) {
        break;
    }
}

if (i < 5) {
    console.log("The loop stopped early!");
}
```

This usage pattern is not terribly uncommon, but most feel it signals poor code structure. A better approach is to introduce another outer-scoped variable:

```
var lastI;

for (let i = 0; i < 5; i++) {
    lastI = i;
    if (checkValue(i)) {
        break;
    }
}

if (lastI < 5) {
    console.log("The loop stopped early!");
}
```

`lastI` is needed across this whole scope (function or global), so it's declared with `var`. `i` is only needed in (each) loop iteration, so it's declared with `let`.

What's The Catch?

So far we've asserted that `var` and parameters are function-scoped, and `let` / `const` signal block-scoped declarations. There's one little exception to call out: the `catch` clause.

Since the introduction of `try..catch` way back in ES3, the `catch` clause has held an additional (little-known) block-scoping declaration capability:

```
try {
    doesntExist();
}
catch (err) {
    console.log(err);
    // ReferenceError: 'doesntExist' is not defined
    // ˆˆˆˆ this is printed from the caught exception

    let onlyHere = true;
    var outerVariable = true;
}

console.log(outerVariable);     // true

console.log(err);
// ReferenceError: 'err' is not defined
// ˆˆˆˆ this is another thrown (uncaught) exception
```

The `err` variable declared by the `catch` clause is block-scoped to that block. This clause block can have other block-scoped declarations via `let`. But a `var` declaration inside this block still attaches to the outer function/global scope.

ES2019 (recently, at the time of writing) changed `catch` clauses so their declaration is optional. If you need to catch *that an exception occurred* (so you can gracefully recover), but you don't care about the error value itself, you can omit that declaration:

```
try {
    doOptionOne();
}
catch {  // catch-declaration omitted
    doOptionTwoInstead();
}
```

NOTE: |
:— |
In this case, the `catch` block is **not** a scope unless a `let` declaration is added inside it. |

This is a small but delightful simplification of syntax for a fairly common use-case!

Functions Declarations In Blocks (FiB)

We've seen now that `let` / `const` declarations are block-scoped, and `var` declarations are function-scoped. So what about `function` declarations that appear directly inside blocks (as a feature, called "FiB")?

We typically think of `function` declarations like they're the equivalent of a `var` declaration. So are they function-scoped?

No, and yes. I know... that's confusing. Let's dig in.

Consider:

```
if (false) {
    function ask() {
        console.log("Does this run?");
    }
}

ask();
```

What do you expect for this program to do? Three reasonable outcomes:

1. The `ask()` call might fail with a Reference Error exception, because the `ask` identifier is block-scoped to the `if` block scope and thus isn't available in the outer/global scope.

2. The `ask()` call might fail with a Type Error exception, because the `ask` identifier exists, but it's `undefined` (since the `if` statement doesn't run) and thus not a callable function.

3. The `ask()` call might run correctly, printing out the "Does it run?" message.

Here's the confusing part: depending on which JS environment you try that code snippet in, you may get different results! This is one of those few crazy areas where existing legacy behavior interferes with getting a predictable outcome.

The JS specification says that `function` declarations inside of blocks are block-scoped, so the answer should be (1). However, most browser-based JS engines (including v8 which comes from Chrome but is also used in Node) will behave as (2), meaning the identifier is scoped outside the `if` block but the function value is not automatically initialized, so it remains `undefined`.

Why are browser JS engines allowed to behave contrary to the specification? Because these engines already had certain behaviors around FiB before ES6 introduced block scoping, and there was concern that changing to adhere to the specification might break some existing website JS code. As such, an exception was made in Appendix B of the specification, which allows certain deviations for browser JS engines (only!).

NOTE: |
:— |
You wouldn't typically categorize Node as being a browser JS environment, since

it usually runs on a server. But it's an interesting corner case since it shares the v8 engine with the Chrome (and Edge, now) browsers. Since v8 is first a browser JS engine, it follows this Appendix B exception, which then means that the browser exceptions are extended to Node. |

One of the most common use-cases for placing a `function` declaration in a block is to conditionally define a function one way or another (like with an `if..else` statement) depending on some environment state. For example:

```
if (typeof Array.isArray != "undefined") {
    function isArray(a) {
        return Array.isArray(a);
    }
}
else {
    function isArray(a) {
        return Object.prototype.toString.call(a) == "[object Array]";
    }
}
```

It's tempting to structure code this way for performance reasons, since the `typeof Array.isArray` check is only performed once, as opposed to defining just one `isArray(..)` and putting the `if` statement inside it, where the check then runs unnecessarily on every call.

WARNING: |
:— |
In addition to the risks of FiB deviations, one problem with the conditional-definition of functions is that it is harder to debug such a program. If you end up with a bug in the `isArray(..)` function, you first have to figure out *which* `isArray(..)` function definition is actually applied! Sometimes, the bug is that the wrong one got applied because the conditional check was incorrect! If you allow a program to define multiple versions of a function, that program is always harder to reason about and maintain. |

In addition to the situations in the above snippets, there are several other corner cases around FiB you can be bitten by; such behaviors in various browsers and non-browser JS environments (that is, JS engines that aren't primarily browser based) will likely vary.

For example:

```
if (true) {
    function ask() {
        console.log("Am I called?");
    }
}
```

18

```
if (true) {
    function ask() {
        console.log("Or what about me?");
    }
}

for (let i = 0; i < 5; i++) {
    function ask() {
        console.log("Or is it one of these?");
    }
}

ask();

function ask() {
    console.log("Maybe, it's this one?");
}
```

Recall that function hoisting as described in "When Can I Use A Variable?" (Chapter 3) might suggest that the final `ask()` in this snippet, with "Maybe. . . " as its message, would hoist above the call to `ask()`. Since it's the last function declaration of that name, it should "win", right? Unfortunately, no.

It's not my intention to document all these weird corner cases, nor to try to explain why each of them behaves a certain way. That information is, in my opinion, useless legacy trivia.

My real concern with FiB is, what advice can I give to ensure your code behaves predictably in all circumstances?

As far as I'm concerned, the only practical answer to avoiding the vagaries of FiB is to simply avoid FiB entirely. In other words, never place a `function` declaration directly inside any block. Always place `function` declarations at the top-level scope of a function (or in the global scope).

So for the earlier `if..else` example, my suggestion is to avoid conditionally defining functions if at all possible. Yes it may be slightly less performant, but this is a better overall option in my opinion:

```
function isArray(a) {
    if (typeof Array.isArray != "undefined") {
        return Array.isArray(a);
    }
    else {
        return Object.prototype.toString.call(a) == "[object Array]";
    }
}
```

If that performance hit becomes a critical path issue for your application, I suggest you consider this approach:

```
var isArray = function isArray(a) {
    return Array.isArray(a);
};

// override the definition, if you must
if (typeof Array.isArray == "undefined") {
    isArray = function isArray(a) {
        return Object.prototype.toString.call(a) == "[object Array]";
    };
}
```

It's important to notice that here I'm placing a `function` **expression**, not a declaration, inside the `if` statement. That's perfectly fine and valid, for `function` expressions to appear inside blocks. Our discussion about FiB is about avoiding `function` **declarations** in blocks.

Even if you test your program and it works correctly, the small benefit you may derive from using FiB style in your code is far outweighed by the potential risks in the future for confusion by other developers, or variances in how your code runs in other JS environments.

FiB is not worth it, and should be avoided.

Blocked Over

The whole point of lexical scoping rules in a programming language is so we can appropriately organize our program's variables, both for operational as well as stylistic code communication purposes.

And one of the most important organizational techniques we can learn to get better at is to ensure that no variable is over-exposed to unnecessary scopes (POLE). Hopefully you now appreciate block scoping much more deeply than before.

This whole discussion of lexical scoping so far in the book is intended to build up a solid foundation of understanding so we can tackle the topic of closure effectively. Before moving on to the next chapter, make sure you are fully comfortable with all the ins and outs of lexical scope.