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

Chapter 6: Limiting Scope Exposure

So far our focus has been explaining the mechanics of how scopes and variables work. With that foundation now firmly in place, our attention raises to a higher level of thinking: decisions and patterns we apply across the whole program.

To begin, we're going to look at how and why we should be using different levels of scope (functions and blocks) to organize our program's variables, specifically to reduce scope over-exposure.

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 discipline, typically applied to software security, called "The Principle of Least Privilege" (POLP).[1] And a variation of this principle that applies to our current discussion is typically labeled as "Least Exposure" (POLE).

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 POLP focuses on system-level component design, the POLE *Exposure* variant focuses on a lower level; we'll apply it to how scopes interact with each other.

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

Think of it this way: why shouldn'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 three main hazards that often arise:

- **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 happens that one loop in a function is running during an iteration of a loop from another function, and now the shared `i` variable gets an unexpected value.

- **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 and 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.

- **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 adjusting 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 three 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 initially larger (the result would be negative!), we swap `x` and `y` using a `tmp` variable, to keep the result positive.

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—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?

We've already seen the `let` and `const` keywords, which are block scoped declarators; we'll come back to them in more detail shortly. But first, 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 2 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 previous computations 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 be located:

```
// outer/global scope

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

    return factorial;

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

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

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 in an outer scope variable, also named `factorial`. Now as we call `factorial(..)` (multiple times!), its persistent `cache` stays hidden yet 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"; this caching relies on closure (see Chapter 7). Also, there are memory usage concerns (addressed in "A Word About Memory" in Appendix B). FP libraries will usually provide an optimized and vetted utility for memoization of functions, which would take the place of `hideTheCache(..)` here. Memoization is beyond the *scope* (pun intended!) of our discussion, but 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" (in 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 can name every single occurrence of such a function expression the exact same name, and never have any collision. More appropriately, we can 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, we *could* just leave off the name entirely—thus defining an "anonymous `function` expression" instead. But Appendix A will discuss the importance of names even for such scope-only functions.

Invoking Function Expressions Immediately

There's another important bit in the previous factorial recursive 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).

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

For comparison, here's an example of a standalone 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 a standalone IIFE they're **required**; they distinguish the `function` as an expression, not a statement. For consistency, however, always surround an IIFE `function` with `( .. )`.

NOTE:

Technically, the surrounding `( .. )` aren't the only syntactic way to ensure the `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

Beware 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 an IIFE is wrapped around it, because now the `return` would refer to the IIFE's function. Non-arrow function IIFEs also change the binding of a `this` keyword—more on that in the *Objects & Classes* book. 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 the merits of creating scopes to limit identifier exposure.

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 necessary, to contain its block-scoped declarations (i.e., `let` or `const`). Consider:

```
{
    // not necessarily a scope (yet)

    // ..

    // now we know the block needs to be a scope
    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
```

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 object values 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. It *is*, however, a (function) scope.
- The `{ .. }` curly-brace pair on a `switch` statement (around the set of `case` clauses) does not define a block/scope.

Other than such non-block examples, a `{ .. }` curly-brace pair can define a block attached to a statement (like an `if` or `for`), or stand alone by itself—see the outermost `{ .. }` curly brace pair in the previous snippet. An explicit block of this sort—if it has no declarations, it's not actually a scope—serves no operational purpose, though it can still be useful as a semantic signal.

Explicit standalone `{ .. }` blocks have always been valid JS syntax, but since they couldn't be a scope prior to ES6's `let` / `const`, they are quite rare. However, post ES6, they're starting to catch on a little bit.

In most languages that support block scoping, an explicit block scope is an extremely 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 (explicit) 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 the outer block is a scope or not).

For example:

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

    {
        // this is both a block and an
        // explicit 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. And to be fair, when there's only a few lines to consider, it's a toss-up judgement call. But as code grows, these over-exposure 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 (within reason!) define the smallest block for each variable. So I recommend using the extra explicit block scope as shown.

Recall the discussion of TDZ errors from "Uninitialized Variables (TDZ)" (Chapter 5). My suggestion there was: to minimize the risk of TDZ errors with `let` / `const` declarations, always put those declarations at the top of their scope.

If you find yourself placing a `let` declaration in the middle of a scope, first think, "Oh, no! TDZ alert!" If this `let` declaration isn't needed in the first half of that block, you should use an inner explicit block scope to further narrow its exposure!

Another example with 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;
    }

    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: `dateStr` (parameter), `nextMonth`, and `year`.
3. The `{ .. }` curly-brace pair defines an inner block scope that includes one variable: `curMonth`.

So why 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; at the function scope level it's over-exposed.

This example is small, so the hazards of over-exposing `curMonth` are pretty limited. But the benefits of the POLE principle are best achieved when you adopt the mindset of minimizing scope exposure by default, 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:

```
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 explicit 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 semantic 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 against 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, and 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 semantic and operational reasons to choose `var` or `let` in different scenarios. We'll explore the case for `var` *and* `let` in more detail in Appendix A.

WARNING:

My recommendation to use both `var` *and* `let` is clearly controversial and contradicts the majority. 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 valid, but they're merely opinions, just like mine. `var` is not factually broken or deprecated; it has worked since early JS and it will continue to work as long as JS is around.

Where To `let`?

My advice to reserve `var` for (mostly) only a top-level function scope means that most 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 declarator 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, just 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 for 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 in signaling our intent:

```
function diff(x,y) {
    if (x > y) {
        // `tmp` is still function-scoped, but
        // the placement here semantically
        // signals block-scoping
        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 semantic signal still has benefit for the reader of your code.

Following this perspective, you can find any `var` that's inside a block of this sort and switch it to `let` to enforce the semantic signal already being sent. That's proper usage of `let` in my opinion.

Another example that was historically based on `var` but which should now pretty much always use `let` is the `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 dictates it should be declared with `let` instead of `var`:

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

Almost the only case where 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 smells like poor code structure. A preferable approach is to use another outer-scoped variable for that purpose:

```
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, 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` back in ES3 (in 1999), the `catch` clause has used an additional (little-known) block-scoping declaration capability:

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

    let onlyHere = true;
    var outerVariable = true;
}

console.log(outerVariable);     // true

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

The `err` variable declared by the `catch` clause is block-scoped to that block. This `catch` clause block can hold 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 the declaration is omitted, the `catch` block is no longer (by default) a scope; it's still a block, though!

So if you need to react to the condition *that an exception occurred* (so you can gracefully recover), but you don't care about the error value itself, you can omit the `catch` declaration:

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

This is a small but delightful simplification of syntax for a fairly common use case, and may also be slightly more performant in removing an unnecessary scope!

Function Declarations in Blocks (FiB)

We've seen now that declarations using `let` or `const` are block-scoped, and `var` declarations are function-scoped. So what about `function` declarations that appear directly inside blocks? As a feature, this is called "FiB."

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

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

```
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 `ReferenceError` 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 `TypeError` 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 betrays 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 JS specification, which allows certain deviations for browser JS engines (only!).

NOTE:

You wouldn't typically categorize Node as a browser JS environment, since it usually runs on a server. But Node's v8 engine is shared with Chrome (and Edge) browsers. Since v8 is first a browser JS engine, it adopts 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—the check would then run unnecessarily on every call.

WARNING:

In addition to the risks of FiB deviations, another problem with conditional-definition of functions is it's 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(..)` implementation is actually running! Sometimes, the bug is that the wrong one was applied because the conditional check was incorrect! If you define multiple versions of a function, that program is always harder to reason about and maintain.

In addition to the previous snippets, several other FiB corner cases are lurking; such behaviors in various browsers and non-browser JS environments (JS engines that aren't browser based) will likely vary. For example:

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

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("Wait, maybe, it's this one?");
}
```

Recall that function hoisting as described in "When Can I Use a Variable?" (in Chapter 5) might suggest that the final `ask()` in this snippet, with "Wait, 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, arcane 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 anywhere in 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 the better overall approach:

```
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 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 semantic code communication purposes.

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

Hopefully by now you feel like you're standing on much more solid ground with understanding lexical scope. From that base, the next chapter jumps into the weighty topic of closure.

---

1. *Principle of Least Privilege*, https://en.wikipedia.org/wiki/Principle_of_least_privilege, 3 March 2020. ↩