You Don't Know JS: Scope & Closures

Chapter 3: Function vs. Block Scope

As we explored in Chapter 2, scope consists of a series of "bubbles" that each act as a container or bucket, in which identifiers (variables, functions) are declared. These bubbles nest neatly inside each other, and this nesting is defined at author-time.

But what exactly makes a new bubble? Is it only the function? Can other structures in JavaScript create bubbles of scope?

Scope From Functions

The most common answer to those questions is that JavaScript has function-based scope. That is, each function you declare creates a bubble for itself, but no other structures create their own scope bubbles. As we'll see in just a little bit, this is not quite true.

But first, let's explore function scope and its implications.

Consider this code:

```
function foo(a) {
        var b = 2;

        // some code

        function bar() {
                // ...
        }

        // more code

        var c = 3;
}
```

In this snippet, the scope bubble for `foo(..)` includes identifiers `a`, `b`, `c` and `bar`. **It doesn't matter** *where* in the scope a declaration appears, the variable or function belongs to the containing scope bubble, regardless. We'll explore how exactly *that* works in the next chapter.

`bar(..)` has its own scope bubble. So does the global scope, which has just one identifier attached to it: `foo`.

Because `a`, `b`, `c`, and `bar` all belong to the scope bubble of `foo(..)`, they are not accessible outside of `foo(..)`. That is, the following code would all result in `ReferenceError` errors, as the identifiers are not available to the global scope:

```
bar(); // fails

console.log( a, b, c ); // all 3 fail
```

However, all these identifiers (`a`, `b`, `c`, `foo`, and `bar`) are accessible *inside* of `foo(..)`, and indeed also available inside of `bar(..)` (assuming there are no shadow identifier declarations inside `bar(..)`).

Function scope encourages the idea that all variables belong to the function, and can be used and reused throughout the entirety of the function (and indeed, accessible even to nested scopes). This design approach can be quite useful, and certainly can make full use of the "dynamic" nature of JavaScript variables to take on values of different types as needed.

On the other hand, if you don't take careful precautions, variables existing across the entirety of a scope can lead to some unexpected pitfalls.

Hiding In Plain Scope

The traditional way of thinking about functions is that you declare a function, and then add code inside it. But the inverse thinking is equally powerful and useful: take any arbitrary section of code you've written, and wrap a function declaration around it, which in effect "hides" the code.

The practical result is to create a scope bubble around the code in question, which means that any declarations (variable or function) in that code will now be tied to the scope of the new wrapping function, rather than the previously enclosing scope. In other words, you can "hide" variables and functions by enclosing them in the scope of a function.

Why would "hiding" variables and functions be a useful technique?

There's a variety of reasons motivating this scope-based hiding. They tend to arise from the software design principle "Principle of Least Privilege" [^note-leastprivilege], also sometimes called "Least Authority" or "Least Exposure". This principle states that in the design of software, such as the API for a module/object, you should expose only what is minimally necessary, and "hide" everything else.

This principle extends to the choice of which scope to contain variables and functions. If all variables and functions were in the global scope, they would of course be accessible to any nested scope. But this would violate the "Least..." principle in that you are (likely) exposing many variables or functions which you should otherwise keep private, as proper use of the code would discourage access to those variables/functions.

For example:

```
function doSomething(a) {
        b = a + doSomethingElse( a * 2 );

        console.log( b * 3 );
}

function doSomethingElse(a) {
        return a - 1;
}

var b;

doSomething( 2 ); // 15
```

In this snippet, the `b` variable and the `doSomethingElse(..)` function are likely "private" details of how `doSomething(..)` does its job. Giving the enclosing scope "access" to `b` and `doSomethingElse(..)` is not only unnecessary but also possibly "dangerous", in that they may be used in unexpected ways, intentionally or not, and this may violate pre-condition assumptions of `doSomething(..)`.

A more "proper" design would hide these private details inside the scope of `doSomething(..)`, such as:

```
function doSomething(a) {
        function doSomethingElse(a) {
                return a - 1;
        }

        var b;

        b = a + doSomethingElse( a * 2 );

        console.log( b * 3 );
}

doSomething( 2 ); // 15
```

Now, `b` and `doSomethingElse(..)` are not accessible to any outside influence, instead controlled only by `doSomething(..)`. The functionality and end-result has not been affected, but the design keeps private details private, which is usually considered better software.

Collision Avoidance

Another benefit of "hiding" variables and functions inside a scope is to avoid unintended collision between two different identifiers with the same name but different intended usages. Collision results often in unexpected overwriting of values.

For example:

```
function foo() {
        function bar(a) {
                i = 3; // changing the `i` in the enclosing scope's for-loop
                console.log( a + i );
        }

        for (var i=0; i<10; i++) {
                bar( i * 2 ); // oops, infinite loop ahead!
        }
}

foo();
```

The `i = 3` assignment inside of `bar(..)` overwrites, unexpectedly, the `i` that was declared in `foo(..)` at the for-loop. In this case, it will result in an infinite loop, because `i` is set to a fixed value of `3` and that will forever remain `< 10`.

The assignment inside `bar(..)` needs to declare a local variable to use, regardless of what identifier name is chosen. `var i = 3;` would fix the problem (and would create the previously mentioned "shadowed variable" declaration for `i`). An *additional*, not alternate, option is to pick another identifier name entirely, such as `var j = 3;`. But your software design may naturally call for the same identifier name, so utilizing scope to "hide" your inner declaration is your best/only option in that case.

Global "Namespaces"

A particularly strong example of (likely) variable collision occurs in the global scope. Multiple libraries loaded into your program can quite easily collide with each other if they don't properly hide their internal/private functions and variables.

Such libraries typically will create a single variable declaration, often an object, with a sufficiently unique name, in the global scope. This object is then used as a "namespace" for that library, where all specific exposures of functionality are made as properties of that object (namespace), rather than as top-level lexically scoped identifiers themselves.

For example:

```
var MyReallyCoolLibrary = {
        awesome: "stuff",
        doSomething: function() {
                // ...
        },
        doAnotherThing: function() {
                // ...
        }
};
```

Module Management

Another option for collision avoidance is the more modern "module" approach, using any of various dependency managers. Using these tools, no libraries ever add any identifiers to the global scope, but are instead required to have their identifier(s) be explicitly imported into another specific scope through usage of the dependency manager's various mechanisms.

It should be observed that these tools do not possess "magic" functionality that is exempt from lexical scoping rules. They simply use the rules of scoping as explained here to enforce that no identifiers are injected into any shared scope, and are instead kept in private, non-collision-susceptible scopes, which prevents any accidental scope collisions.

As such, you can code defensively and achieve the same results as the dependency managers do without actually needing to use them, if you so choose. See the Chapter 5 for more information about the module pattern.

Functions As Scopes

We've seen that we can take any snippet of code and wrap a function around it, and that effectively "hides" any enclosed variable or function declarations from the outside scope inside that function's inner scope.

For example:

```
var a = 2;

function foo() { // <-- insert this

        var a = 3;
        console.log( a ); // 3

} // <-- and this
foo(); // <-- and this

console.log( a ); // 2
```

While this technique "works", it is not necessarily very ideal. There are a few problems it introduces. The first is that we have to declare a named-function `foo()`, which means that the identifier name `foo` itself "pollutes" the enclosing scope (global, in this case). We also have to explicitly call the function by name (`foo()`) so that the wrapped code actually executes.

It would be more ideal if the function didn't need a name (or, rather, the name didn't pollute the enclosing scope), and if the function could automatically be executed.

Fortunately, JavaScript offers a solution to both problems.

```
var a = 2;

(function foo(){ // <-- insert this

        var a = 3;
        console.log( a ); // 3

})(); // <-- and this

console.log( a ); // 2
```

Let's break down what's happening here.

First, notice that the wrapping function statement starts with `(function...` as opposed to just `function...`. While this may seem like a minor detail, it's actually a major change. Instead of treating the function as a standard declaration, the function is treated as a function-expression.

Note: The easiest way to distinguish declaration vs. expression is the position of the word "function" in the statement (not just a line, but a distinct statement). If "function" is the very first thing in the statement, then it's a function declaration. Otherwise, it's a function expression.

The key difference we can observe here between a function declaration and a function expression relates to where its name is bound as an identifier.

Compare the previous two snippets. In the first snippet, the name `foo` is bound in the enclosing scope, and we call it directly with `foo()`. In the second snippet, the name `foo` is not bound in the enclosing scope, but instead is bound only inside of its own function.

In other words, `(function foo(){ .. })` as an expression means the identifier `foo` is found *only* in the scope where the `..` indicates, not in the outer scope. Hiding the name `foo` inside itself means it does not pollute the enclosing scope unnecessarily.

Anonymous vs. Named

You are probably most familiar with function expressions as callback parameters, such as:

```
setTimeout( function(){
        console.log("I waited 1 second!");
}, 1000 );
```

This is called an "anonymous function expression", because `function()...` has no name identifier on it. Function expressions can be anonymous, but function declarations cannot omit the name -- that would be illegal JS grammar.

Anonymous function expressions are quick and easy to type, and many libraries and tools tend to encourage this idiomatic style of code. However, they have several draw-backs to consider:

1. Anonymous functions have no useful name to display in stack traces, which can make debugging more difficult.
2. Without a name, if the function needs to refer to itself, for recursion, etc., the **deprecated** `arguments.callee` reference is unfortunately required. Another example of needing to self-reference is when an event handler function wants to unbind itself after it fires.
3. Anonymous functions omit a name that is often helpful in providing more readable/understandable code. A descriptive name helps self-document the code in question.

Inline function expressions are powerful and useful -- the question of anonymous vs. named doesn't detract from that. Providing a name for your function expression quite effectively addresses all these draw-backs, but has no tangible downsides. The best practice is to always name your function expressions:

```
setTimeout( function timeoutHandler(){ // <-- Look, I have a name!
        console.log( "I waited 1 second!" );
}, 1000 );
```

Invoking Function Expressions Immediately

```
var a = 2;

(function foo(){

        var a = 3;
        console.log( a ); // 3

})();

console.log( a ); // 2
```

Now that we have a function as an expression by virtue of wrapping it in a `( )` pair, we can execute that function by adding another `()` on the end, like `(function foo(){ .. })()`. The first enclosing `( )` pair makes the function an expression, and the second `()` executes the function.

This pattern is so common, a few years ago the community agreed on a term for it: **IIFE**, which stands for **I**mmediately **I**nvoked **F**unction **E**xpression.

Of course, IIFE's don't need names, necessarily -- the most common form of IIFE is to use an anonymous function expression. While certainly less common, naming an IIFE has all the aforementioned benefits over anonymous function expressions, so it's a good practice to adopt.

```
var a = 2;

(function IIFE(){

        var a = 3;
        console.log( a ); // 3

})();

console.log( a ); // 2
```

There's a slight variation on the traditional IIFE form, which some prefer: `(function(){ .. }())`. Look closely to see the difference. In the first form, the function expression is wrapped in `( )`, and then the invoking `()` pair is on the outside right after it. In the second form, the invoking `()` pair is moved to the inside of the outer `( )` wrapping pair.

These two forms are identical in functionality. **It's purely a stylistic choice which you prefer.**

Another variation on IIFE's which is quite common is to use the fact that they are, in fact, just function calls, and pass in argument(s).

For instance:

```
var a = 2;

(function IIFE( global ){

        var a = 3;
        console.log( a ); // 3
        console.log( global.a ); // 2

})( window );

console.log( a ); // 2
```

We pass in the `window` object reference, but we name the parameter `global`, so that we have a clear stylistic delineation for global vs. non-global references. Of course, you can pass in anything from an enclosing scope you want, and you can name the parameter(s) anything that suits you. This is mostly just stylistic choice.

Another application of this pattern addresses the (minor niche) concern that the default `undefined` identifier might have its value incorrectly overwritten, causing unexpected results. By naming a parameter `undefined`, but not passing any value for that argument, we can guarantee that the `undefined` identifier is in fact the undefined value in a block of code:

```
undefined = true; // setting a land-mine for other code! avoid!

(function IIFE( undefined ){

        var a;
        if (a === undefined) {
                console.log( "Undefined is safe here!" );
        }

})();
```

Still another variation of the IIFE inverts the order of things, where the function to execute is given second, *after* the invocation and parameters to pass to it. This pattern is used in the UMD (Universal Module Definition) project. Some people find it a little cleaner to understand, though it is slightly more verbose.

```
var a = 2;

(function IIFE( def ){
        def( window );
})(function def( global ){

        var a = 3;
        console.log( a ); // 3
        console.log( global.a ); // 2

});
```

The `def` function expression is defined in the second-half of the snippet, and then passed as a parameter (also called `def`) to the `IIFE` function defined in the first half of the snippet. Finally, the parameter `def` (the function) is invoked, passing `window` in as the `global` parameter.

Blocks As Scopes

While functions are the most common unit of scope, and certainly the most wide-spread of the design approaches in the majority of JS in circulation, other units of scope are possible, and the usage of these other scope units can lead to even better, cleaner to maintain code.

Many languages other than JavaScript support Block Scope, and so developers from those languages are accustomed to the mindset, whereas those who've primarily only worked in JavaScript may find the concept slightly foreign.

But even if you've never written a single line of code in block-scoped fashion, you are still probably familiar with this extremely common idiom in JavaScript:

```
for (var i=0; i<10; i++) {
        console.log( i );
}
```

We declare the variable `i` directly inside the for-loop head, most likely because our *intent* is to use `i` only within the context of that for-loop, and essentially ignore the fact that the variable actually scopes itself to the enclosing scope (function or global).

That's what block-scoping is all about. Declaring variables as close as possible, as local as possible, to where they will be used. Another example:

```
var foo = true;

if (foo) {
        var bar = foo * 2;
        bar = something( bar );
        console.log( bar );
}
```

We are using a `bar` variable only in the context of the if-statement, so it makes a kind of sense that we would declare it inside the if-block. However, where we declare variables is not relevant when using `var`, because they will always belong to the enclosing scope. This snippet is essentially "fake" block-scoping, for stylistic reasons, and relying on self-enforcement not to accidentally use `bar` in another place in that scope.

Block scope is a tool to extend the earlier "Principle of Least ~~Privilege~~ Exposure" [^note-leastprivilege] from hiding information in functions to hiding information in blocks of our code.

Consider the for-loop example again:

```
for (var i=0; i<10; i++) {
        console.log( i );
}
```

Why pollute the entire scope of a function with the `i` variable that is only going to be (or only *should be*, at least) used for the for-loop?

But more importantly, developers may prefer to *check* themselves against accidentally (re)using variables outside of their intended purpose, such as being issued an error about an unknown variable if you try to use it in the wrong place. Block-scoping (if it were possible) for the `i` variable would make `i` available only for the for-loop, causing an error if `i` is accessed elsewhere in the function. This helps ensure variables are not re-used in confusing or hard-to-maintain ways.

But, the sad reality is that, on the surface, JavaScript has no facility for block scope.

That is, until you dig a little further.

`with`

We learned about `with` in Chapter 2. While it is a frowned upon construct, it *is* an example of (a form of) block scope, in that the scope that is created from the object only exists for the lifetime of that `with` statement, and not in the enclosing scope.

`try/catch`

It's a *very* little known fact that JavaScript in ES3 specified the variable declaration in the `catch` clause of a `try/catch` to be block-scoped to the `catch` block.

For instance:

```
try {
        undefined(); // illegal operation to force an exception!
}
catch (err) {
        console.log( err ); // works!
}

console.log( err ); // ReferenceError: `err` not found
```

As you can see, `err` exists only in the `catch` clause, and throws an error when you try to reference it elsewhere.

Note: While this behavior has been specified and true of practically all standard JS environments (except perhaps old IE), many linters seem to still complain if you have two or more `catch` clauses in the same scope which each declare their error variable with the same identifier name. This is not actually a re-definition, since the variables are safely block-scoped, but the linters still seem to, annoyingly, complain about this fact.

To avoid these unnecessary warnings, some devs will name their `catch` variables `err1`, `err2`, etc. Other devs will simply turn off the linting check for duplicate variable names.

The block-scoping nature of `catch` may seem like a useless academic fact, but see Appendix B for more information on just how useful it might be.

`let`

Thus far, we've seen that JavaScript only has some strange niche behaviors which expose block scope functionality. If that were all we had, and *it was* for many, many years, then block scoping would not be terribly useful to the JavaScript developer.

Fortunately, ES6 changes that, and introduces a new keyword `let` which sits alongside `var` as another way to declare variables.

The `let` keyword attaches the variable declaration to the scope of whatever block (commonly a `{ .. }` pair) it's contained in. In other words, `let` implicitly hijacks any block's scope for its variable declaration.

```
var foo = true;

if (foo) {
        let bar = foo * 2;
        bar = something( bar );
        console.log( bar );
}

console.log( bar ); // ReferenceError
```

Using `let` to attach a variable to an existing block is somewhat implicit. It can confuse you if you're not paying close attention to which blocks have variables scoped to them, and are in the habit of moving blocks around, wrapping them in other blocks, etc., as you develop and evolve code.

Creating explicit blocks for block-scoping can address some of these concerns, making it more obvious where variables are attached and not. Usually, explicit code is preferable over implicit or subtle code. This explicit block-scoping style is easy to achieve, and fits more naturally with how block-scoping works in other languages:

```
var foo = true;

if (foo) {
        { // <-- explicit block
                let bar = foo * 2;
                bar = something( bar );
                console.log( bar );
        }
}
```

```
console.log( bar ); // ReferenceError
```

We can create an arbitrary block for `let` to bind to by simply including a `{ .. }` pair anywhere a statement is valid grammar. In this case, we've made an explicit block *inside* the if-statement, which may be easier as a whole block to move around later in refactoring, without affecting the position and semantics of the enclosing if-statement.

Note: For another way to express explicit block scopes, see Appendix B.

In Chapter 4, we will address hoisting, which talks about declarations being taken as existing for the entire scope in which they occur.

However, declarations made with `let` will *not* hoist to the entire scope of the block they appear in. Such declarations will not observably "exist" in the block until the declaration statement.

```
{
   console.log( bar ); // ReferenceError!
   let bar = 2;
}
```

Garbage Collection

Another reason block-scoping is useful relates to closures and garbage collection to reclaim memory. We'll briefly illustrate here, but the closure mechanism is explained in detail in Chapter 5.

Consider:

```
function process(data) {
        // do something interesting
}

var someReallyBigData = { .. };

process( someReallyBigData );

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
        console.log("button clicked");
}, /*capturingPhase=*/false );
```

The `click` function click handler callback doesn't *need* the `someReallyBigData` variable at all. That means, theoretically, after `process(..)` runs, the big memory-heavy data structure could be garbage collected. However, it's quite likely (though implementation dependent) that the JS engine will still have to keep the structure around, since the `click` function has a closure over the entire scope.

Block-scoping can address this concern, making it clearer to the engine that it does not need to keep `someReallyBigData` around:

```
function process(data) {
        // do something interesting
}

// anything declared inside this block can go away after!
{
        let someReallyBigData = { .. };

        process( someReallyBigData );
}

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
        console.log("button clicked");
}, /*capturingPhase=*/false );
```

Declaring explicit blocks for variables to locally bind to is a powerful tool that you can add to your code toolbox.

`let` Loops

A particular case where `let` shines is in the for-loop case as we discussed previously.

```
for (let i=0; i<10; i++) {
        console.log( i );
}

console.log( i ); // ReferenceError
```

Not only does `let` in the for-loop header bind the `i` to the for-loop body, but in fact, it **re-binds it** to each *iteration* of the loop, making sure to re-assign it the value from the end of the previous loop iteration.

Here's another way of illustrating the per-iteration binding behavior that occurs:

```
{
        let j;
        for (j=0; j<10; j++) {
                let i = j; // re-bound for each iteration!
                console.log( i );
        }
}
```

The reason why this per-iteration binding is interesting will become clear in Chapter 5 when we discuss closures.

Because `let` declarations attach to arbitrary blocks rather than to the enclosing function's scope (or global), there can be gotchas where existing code has a hidden reliance on function-scoped `var` declarations, and replacing the `var` with `let` may require additional care when refactoring code.

Consider:

```
var foo = true, baz = 10;

if (foo) {
        var bar = 3;

        if (baz > bar) {
                console.log( baz );
        }

        // ...
}
```

This code is fairly easily re-factored as:

```
var foo = true, baz = 10;

if (foo) {
        var bar = 3;

        // ...
}

if (baz > bar) {
        console.log( baz );
}
```

But, be careful of such changes when using block-scoped variables:

```
var foo = true, baz = 10;

if (foo) {
        let bar = 3;
```

```
        if (baz > bar) { // <-- don't forget `bar` when moving!
                console.log( baz );
        }
}
```

See Appendix B for an alternate (more explicit) style of block-scoping which may provide easier to maintain/refactor code that's more robust to these scenarios.

`const`

In addition to `let`, ES6 introduces `const`, which also creates a block-scoped variable, but whose value is fixed (constant). Any attempt to change that value at a later time results in an error.

```
var foo = true;

if (foo) {
        var a = 2;
        const b = 3; // block-scoped to the containing `if`

        a = 3; // just fine!
        b = 4; // error!
}

console.log( a ); // 3
console.log( b ); // ReferenceError!
```

Review (TL;DR)

Functions are the most common unit of scope in JavaScript. Variables and functions that are declared inside another function are essentially "hidden" from any of the enclosing "scopes", which is an intentional design principle of good software.

But functions are by no means the only unit of scope. Block-scope refers to the idea that variables and functions can belong to an arbitrary block (generally, any `{ .. }` pair) of code, rather than only to the enclosing function.

Starting with ES3, the `try/catch` structure has block-scope in the `catch` clause.

In ES6, the `let` keyword (a cousin to the `var` keyword) is introduced to allow declarations of variables in any arbitrary block of code. `if (..) { let a = 2; }` will declare a variable `a` that essentially hijacks the scope of the `if`'s `{ .. }` block and attaches itself there.

Though some seem to believe so, block scope should not be taken as an outright replacement of `var` function scope. Both functionalities co-exist, and developers can and should use both function-scope and block-scope techniques where respectively appropriate to produce better, more readable/maintainable code.

[^note-leastprivilege]: Principle of Least Privilege