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

Chapter 5: Closures

NOTE: |
:— |
Work in progress |

.

.

.

.

.

.

.

NOTE: |
:— |
Everything below here is previous text from 1st edition, and is only here for reference while 2nd edition work is underway. Please ignore. |

We arrive at this point with hopefully a very healthy, solid understanding of how scope works.

We turn our attention to an incredibly important, but persistently elusive, *almost mythological*, part of the language: **closure**. If you have followed our discussion of lexical scope thus far, the payoff is that closure is going to be, largely, anticlimactic, almost self-obvious. *There's a man behind the wizard's curtain, and we're about to see him.* No, his name is not Crockford!

If however you have nagging questions about lexical scope, now would be a good time to go back and review Chapter 2 before proceeding.

Enlightenment

For those who are somewhat experienced in JavaScript, but have perhaps never fully grasped the concept of closures, *understanding closure* can seem like a special nirvana that one must strive and sacrifice to attain.

I recall years back when I had a firm grasp on JavaScript, but had no idea what closure was. The hint that there was *this other side* to the language, one which promised even more capability than I already possessed, teased and taunted me. I remember reading through the source code of early frameworks trying to understand how it actually worked. I remember the first time something of the "module pattern" began to emerge in my mind. I remember the *a-ha!* moments quite vividly.

What I didn't know back then, what took me years to understand, and what I hope to impart to you presently, is this secret: **closure is all around you in JavaScript, you just have to recognize and embrace it.** Closures are not a special opt-in tool that you must learn new syntax and patterns for. No, closures are not even a weapon that you must learn to wield and master as Luke trained in The Force.

Closures happen as a result of writing code that relies on lexical scope. They just happen. You do not even really have to intentionally create closures to take advantage of them. Closures are created and used for you all over your code. What you are *missing* is the proper mental context to recognize, embrace, and leverage closures for your own will.

The enlightenment moment should be: **oh, closures are already occurring all over my code, I can finally *see* them now.** Understanding closures is like when Neo sees the Matrix for the first time.

Nitty Gritty

OK, enough hyperbole and shameless movie references.

Here's a down-n-dirty definition of what you need to know to understand and recognize closures:

> Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.

Let's jump into some code to illustrate that definition.

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

    function bar() {
        console.log( a ); // 2
    }

    bar();
}
```

```
foo();
```

This code should look familiar from our discussions of Nested Scope. Function `bar()` has *access* to the variable `a` in the outer enclosing scope because of lexical scope look-up rules (in this case, it's an RHS reference look-up).

Is this "closure"?

Well, technically... *perhaps*. But by our what-you-need-to-know definition above... *not exactly*. I think the most accurate way to explain `bar()` referencing `a` is via lexical scope look-up rules, and those rules are *only* (an important!) **part** of what closure is.

From a purely academic perspective, what is said of the above snippet is that the function `bar()` has a *closure* over the scope of `foo()` (and indeed, even over the rest of the scopes it has access to, such as the global scope in our case). Put slightly differently, it's said that `bar()` closes over the scope of `foo()`. Why? Because `bar()` appears nested inside of `foo()`. Plain and simple.

But, closure defined in this way is not directly *observable*, nor do we see closure *exercised* in that snippet. We clearly see lexical scope, but closure remains sort of a mysterious shifting shadow behind the code.

Let us then consider code which brings closure into full light:

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

    function bar() {
        console.log( a );
    }

    return bar;
}

var baz = foo();

baz(); // 2 -- Whoa, closure was just observed, man.
```

The function `bar()` has lexical scope access to the inner scope of `foo()`. But then, we take `bar()`, the function itself, and pass it *as* a value. In this case, we `return` the function object itself that `bar` references.

After we execute `foo()`, we assign the value it returned (our inner `bar()` function) to a variable called `baz`, and then we actually invoke `baz()`, which of course is invoking our inner function `bar()`, just by a different identifier reference.

`bar()` is executed, for sure. But in this case, it's executed *outside* of its declared lexical scope.

After `foo()` executed, normally we would expect that the entirety of the inner scope of `foo()` would go away, because we know that the *Engine* employs a *Garbage Collector* that comes along and frees up memory once it's no longer in use. Since it would appear that the contents of `foo()` are no longer in use, it would seem natural that they should be considered *gone*.

But the "magic" of closures does not let this happen. That inner scope is in fact *still* "in use", and thus does not go away. Who's using it? **The function `bar()` itself**.

By virtue of where it was declared, `bar()` has a lexical scope closure over that inner scope of `foo()`, which keeps that scope alive for `bar()` to reference at any later time.

`bar()` still has a reference to that scope, and that reference is called closure.

So, a few microseconds later, when the variable `baz` is invoked (invoking the inner function we initially labeled `bar`), it duly has *access* to author-time lexical scope, so it can access the variable `a` just as we'd expect.

The function is being invoked well outside of its author-time lexical scope. **Closure** lets the function continue to access the lexical scope it was defined in at author-time.

Of course, any of the various ways that functions can be *passed around* as values, and indeed invoked in other locations, are all examples of observing/exercising closure.

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

    function baz() {
        console.log( a ); // 2
    }

    bar( baz );
}

function bar(fn) {
    fn(); // look ma, I saw closure!
}
```

We pass the inner function `baz` over to `bar`, and call that inner function (labeled `fn` now), and when we do, its closure over the inner scope of `foo()` is observed, by accessing `a`.

These passings-around of functions can be indirect, too.

```
var fn;

function foo() {
    var a = 2;

    function baz() {
        console.log( a );
    }

    fn = baz; // assign ‘baz‘ to global variable
}

function bar() {
    fn(); // look ma, I saw closure!
}

foo();

bar(); // 2
```

Whatever facility we use to *transport* an inner function outside of its lexical scope, it will maintain a scope reference to where it was originally declared, and wherever we execute it, that closure will be exercised.

Now I Can See

The previous code snippets are somewhat academic and artificially constructed to illustrate *using closure*. But I promised you something more than just a cool new toy. I promised that closure was something all around you in your existing code. Let us now *see* that truth.

```
function wait(message) {

    setTimeout( function timer(){
        console.log( message );
    }, 1000 );

}

wait( "Hello, closure!" );
```

We take an inner function (named `timer`) and pass it to `setTimeout(..)`. But `timer` has a scope closure over the scope of `wait(..)`, indeed keeping and using a reference to the variable `message`.

A thousand milliseconds after we have executed `wait(..)`, and its inner scope should otherwise be long gone, that inner function `timer` still has closure over that scope.

Deep down in the guts of the *Engine*, the built-in utility `setTimeout(..)` has reference to some parameter, probably called `fn` or `func` or something like that. *Engine* goes to invoke that function, which is invoking our inner `timer` function, and the lexical scope reference is still intact.

Closure.

Or, if you're of the jQuery persuasion (or any JS framework, for that matter):

```
function setupBot(name,selector) {
    $( selector ).click( function activator(){
        console.log( "Activating: " + name );
    } );
}

setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

I am not sure what kind of code you write, but I regularly write code which is responsible for controlling an entire global drone army of closure bots, so this is totally realistic!

(Some) joking aside, essentially *whenever* and *wherever* you treat functions (which access their own respective lexical scopes) as first-class values and pass them around, you are likely to see those functions exercising closure. Be that timers, event handlers, Ajax requests, cross-window messaging, web workers, or any of the other asynchronous (or synchronous!) tasks, when you pass in a *callback function*, get ready to sling some closure around!

Note: Chapter 3 introduced the IIFE pattern. While it is often said that IIFE (alone) is an example of observed closure, I would somewhat disagree, by our definition above.

```
var a = 2;

(function IIFE(){
    console.log( a );
})();
```

This code "works", but it's not strictly an observation of closure. Why? Because the function (which we named "IIFE" here) is not executed outside its lexical

scope. It's still invoked right there in the same scope as it was declared (the enclosing/global scope that also holds `a`). `a` is found via normal lexical scope look-up, not really via closure.

While closure might technically be happening at declaration time, it is *not* strictly observable, and so, as they say, *it's a tree falling in the forest with no one around to hear it.*

Though an IIFE is not *itself* an example of closure, it absolutely creates scope, and it's one of the most common tools we use to create scope which can be closed over. So IIFEs are indeed heavily related to closure, even if not exercising closure themselves.

Put this book down right now, dear reader. I have a task for you. Go open up some of your recent JavaScript code. Look for your functions-as-values and identify where you are already using closure and maybe didn't even know it before.

I'll wait.

Now... you see!

Loops + Closure

The most common canonical example used to illustrate closure involves the humble for-loop.

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

Note: Linters often complain when you put functions inside of loops, because the mistakes of not understanding closure are **so common among developers**. We explain how to do so properly here, leveraging the full power of closure. But that subtlety is often lost on linters and they will complain regardless, assuming you don't *actually* know what you're doing.

The spirit of this code snippet is that we would normally *expect* for the behavior to be that the numbers "1", "2", .. "5" would be printed out, one at a time, one per second, respectively.

In fact, if you run this code, you get "6" printed out 5 times, at the one-second intervals.

Huh?

Firstly, let's explain where `6` comes from. The terminating condition of the loop is when `i` is *not* `<=5`. The first time that's the case is when `i` is 6. So, the output is reflecting the final value of the `i` after the loop terminates.

This actually seems obvious on second glance. The timeout function callbacks are all running well after the completion of the loop. In fact, as timers go, even if it was `setTimeout(.., 0)` on each iteration, all those function callbacks would still run strictly after the completion of the loop, and thus print `6` each time.

But there's a deeper question at play here. What's *missing* from our code to actually have it behave as we semantically have implied?

What's missing is that we are trying to *imply* that each iteration of the loop "captures" its own copy of `i`, at the time of the iteration. But, the way scope works, all 5 of those functions, though they are defined separately in each loop iteration, all **are closed over the same shared global scope**, which has, in fact, only one `i` in it.

Put that way, *of course* all functions share a reference to the same `i`. Something about the loop structure tends to confuse us into thinking there's something else more sophisticated at work. There is not. There's no difference than if each of the 5 timeout callbacks were just declared one right after the other, with no loop at all.

OK, so, back to our burning question. What's missing? We need more ~~cowbell~~ closured scope. Specifically, we need a new closured scope for each iteration of the loop.

We learned in Chapter 3 that the IIFE creates scope by declaring a function and immediately executing it.

Let's try:

```
for (var i=1; i<=5; i++) {
    (function(){
        setTimeout( function timer(){
            console.log( i );
        }, i*1000 );
    })();
}
```

Does that work? Try it. Again, I'll wait.

I'll end the suspense for you. **Nope.** But why? We now obviously have more lexical scope. Each timeout function callback is indeed closing over its own per-iteration scope created respectively by each IIFE.

It's not enough to have a scope to close over **if that scope is empty**. Look closely. Our IIFE is just an empty do-nothing scope. It needs *something* in it to be useful to us.

It needs its own variable, with a copy of the `i` value at each iteration.

```
for (var i=1; i<=5; i++) {
    (function(){
        var j = i;
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })();
}
```

Eureka! It works!

A slight variation some prefer is:

```
for (var i=1; i<=5; i++) {
    (function(j){
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })( i );
}
```

Of course, since these IIFEs are just functions, we can pass in `i`, and we can call it `j` if we prefer, or we can even call it `i` again. Either way, the code works now.

The use of an IIFE inside each iteration created a new scope for each iteration, which gave our timeout function callbacks the opportunity to close over a new scope for each iteration, one which had a variable with the right per-iteration value in it for us to access.

Problem solved!

Block Scoping Revisited

Look carefully at our analysis of the previous solution. We used an IIFE to create new scope per-iteration. In other words, we actually *needed* a per-iteration **block scope**. Chapter 3 showed us the `let` declaration, which hijacks a block and declares a variable right there in the block.

It essentially turns a block into a scope that we can close over. So, the following awesome code "just works":

```
for (var i=1; i<=5; i++) {
    let j = i; // yay, block-scope for closure!
    setTimeout( function timer(){
```

```
        console.log( j );
    }, j*1000 );
}
```

But, that's not all! (in my best Bob Barker voice). There's a special behavior defined for `let` declarations used in the head of a for-loop. This behavior says that the variable will be declared not just once for the loop, **but each iteration**. And, it will, helpfully, be initialized at each subsequent iteration with the value from the end of the previous iteration.

```
for (let i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

How cool is that? Block scoping and closure working hand-in-hand, solving all the world's problems. I don't know about you, but that makes me a happy JavaScripter.

Review (TL;DR)

Closure seems to the un-enlightened like a mystical world set apart inside of JavaScript which only the few bravest souls can reach. But it's actually just a standard and almost obvious fact of how we write code in a lexically scoped environment, where functions are values and can be passed around at will.

Closure is when a function can remember and access its lexical scope even when it's invoked outside its lexical scope.

Closures can trip us up, for instance with loops, if we're not careful to recognize them and how they work. But they are also an immensely powerful tool, enabling patterns like *modules* in their various forms.

Modules require two key characteristics: 1) an outer wrapping function being invoked, to create the enclosing scope 2) the return value of the wrapping function must include reference to at least one inner function that then has closure over the private inner scope of the wrapper.

Now we can see closures all around our existing code, and we have the ability to recognize and leverage them to our own benefit!