CHAPTER 3

Variable Scope and Closures

This chapter introduces variable scope, an important foundational topic not only to functional programming, but to JavaScript in general. The term "binding" refers to the act of assigning a value to a name in JavaScript via `var` assignment, function arguments, `this` passing, and property assignment. This chapter first touches on dynamic scoping, as displayed in JavaScript's `this` reference, and proceeds onto function-level scoping and how it works. All of this builds up to a discussion of closures, or functions that capture nearby variable bindings when they are created. The mechanics of closures will be covered, along with their general use cases and some examples.

The term "scope" has various meanings in common use among JavaScript programmers:

- The value of the `this` binding
- The execution context defined by the value of the `this` binding
- The "lifetime" of a variable
- The variable value resolution scheme, or the lexical bindings

For the purposes of this book, I'll use *scope* to refer to the generic idea of the variable value resolution scheme. I'll dig deeper into various types of resolution schemes to cover the full spectrum of scope provided by JavaScript, starting with the most straightforward: global scope.

Global Scope

The *extent* of a scope refers to the lifetime of a variable (i.e., how long a variable holds a certain value). I'll start with variables with the longest lifespan—that of the "life" of the program itself—globals.

In JavaScript, the following variable would have global scope:

```
aGlobalVariable = 'livin la vida global';
```

Any variable declared in JavaScript without the `var` keyword is created in the global scope, or the scope accessible to every function and method in our program. Observe:

```
_.map(_.range(2), function() { return aGlobalVariable });
//=> ["livin la vida global", "livin la vida global"]
```

As shown, the variable `aGlobalVariable` is accessible from the anonymous function (one created without a name) supplied to the `_.map` call. Global scope is simple to understand and is used often in JavaScript programs (and sometimes with great effect). In fact, Underscore creates a global named _ that contains all of its functions. Although this may not provide the greatest name-spacing technique yet invented, it's what JavaScript uses, and Underscore at least provides an escape hatch with the `_.noConflict` function.

The funny thing about variables in JavaScript is that they are *mutable* by default (i.e., you can change their property values right in place):

```
aGlobalVariable = 'i drink your milkshake';

aGlobalVariable;
//=> "i drink your milkshake"
```

The problem with global variables, and the reason that they are so reviled, is that any piece of code can change them for any reason at any time. This anarchic condition can make for severe pain and missed holidays. In any case, the idea of global scope and its dangers should be known to you by now. However, being defined at the top of a file or lacking a `var` is not all that it takes for a variable to have global scope. Any object is wide open for change (unless it's frozen, which I'll talk about in Chapter 7):

```
function makeEmptyObject() {
  return new Object();
}
```

The `makeEmptyObject` function does exactly what it says: it makes an empty object. I can attach all manner of properties to the objects returned from this function, but so too can any other piece of code that gets a reference to them. Any mutable object that you pass around effectively allows change at a global scale on its properties. Heck, if I wanted, I could change every function in the Underscore object to return the string `'nerf herder'`—no one can stop me. This presents somewhat of a challenge to functional programming in JavaScript. However, as I'll show throughout this book, there are ways to alleviate the problem of an implicit global scope.

Just because a global variable holds a certain value for the entire life of a program doesn't mean that when you refer to it you'll get the global value. Scope becomes more interesting when we talk about something called *lexical scope*, described next.

Lexical Scope

Lexical scope refers to the visibility of a variable and its value analogous to its textual representation. For example, observe the following code:

```
aVariable = "Outer";

function afun() {
  var aVariable = "Middle";

  return _.map([1,2,3], function(e) {
    var aVariable = "In";

    return [aVariable, e].join(' ');
  });
}
```

What is the value of a call to `afun`?

```
afun();
//=> ["In 1", "In 2", "In 3"]
```

The innermost variable value, `In`, takes precedence when used within the function passed to `_.map`. Lexical scope dictates that since the assignment `aVariable` to `In` occurs textually close to its innermost use, then that is its value at the time of use. Figure 3-1 shows this condition graphically.

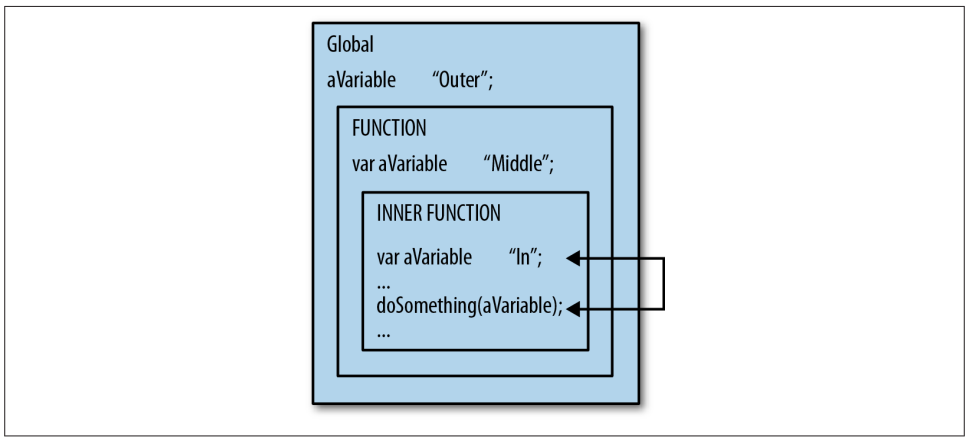

Figure 3-1. Variable lookup starts at the innermost scope and expands outward

In the simple case, variable lookup starts at the closest binding context and expands outward until it finds a binding.[1] Figure 3-1 describes *lexical scoping*, or the grouping of names with values according to the surrounding source code. I will cover the mechanics of different lookup schemes supported by JavaScript over the course of this chapter, starting with dynamic scope.

Dynamic Scope

One of the most under appreciated and over-abused concepts in programming is that of *dynamic scoping*. One reason for this is that very few languages use dynamic scope as their primary binding resolution scheme. Dynamic scoping, however, is a simplistic scheme used as the primary scoping mechanism in only a handful of modern programming languages, and has not seen widespread adoption outside of the earliest versions of Lisp.[2] Simulating a naive dynamic scoping mechanism requires very little code:

```
var globals = {};
```

First of all, dynamic scoping is built on the idea of a global table of named values.[3] At the heart of any JavaScript engine you will see—if not in implementation, then in spirit—one big honking lookup table:

```
function makeBindFun(resolver) {
  return function(k, v) {
    var stack = globals[k] || [];
    globals[k] = resolver(stack, v);
    return globals;
  };
}
```

With `globals` and `makeBindFun` in place, we can move onto the policies for adding bindings to the `globals` variable:

```
var stackBinder = makeBindFun(function(stack, v) {
  stack.push(v);
  return stack;
});

var stackUnbinder = makeBindFun(function(stack) {
  stack.pop();
```

1. There are other scoping modes that JavaScript provides that complicate lookup, including `this` resolution, function-level scope, and `with` blocks. I plan to cover all but the last of these.
2. To get a feel for what the early Lisps had to offer, read "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" by John McCarthy and the "LISP 1.5 Programmer's Manual" by McCarthy, Abrahams, Edwards, Hart, and Levin.
3. This is only one way to implement dynamic scoping, but it is likely the simplest.

```
  return stack;
});
```

The function `stackBinder` performs a very simple task (i.e., it takes a key and a value and pushes the value onto the global `bindings` map at the slot associated with the key). Maintaining a global map of stacks associated with binding names is the core of dynamic scoping, as shown in Figure 3-2.

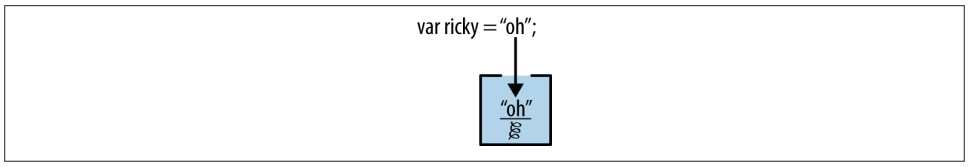

Figure 3-2. You can imagine that any time you declare a variable it comes with a little stack to hold its value; the current dynamic value is found at the top of the stack

The `stackUnbinder` function is the antithesis of `stackBinder` in that it pops the last value binding off of the stack associated with a name. Finally, we'll need a function to look up bound values:

```
var dynamicLookup = function(k) {
  var slot = globals[k] || [];
  return _.last(slot);
};
```

The `dynamicLookup` function provides a convenient way to look up the value at the top of a named value binding stack, and is used to simulate `this` reference resolution as you might visualize it in Figure 3-3.

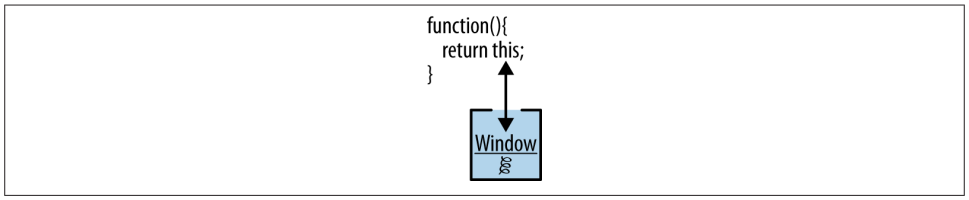

Figure 3-3. A lone function referencing "this" will deal with some global object (e.g., the window in the browser)

Now that our binding and lookup functions are defined, I can note the effects that various operations have on the simulated dynamic scope:

```
stackBinder('a', 1);
stackBinder('b', 100);

dynamicLookup('a');
//=> 1
```

```
globals;
//=> {'a': [1], 'b': [100]}
```

So far, everything looks as you might expect in the preceding code. Specifically, taking the keyed arrays in `globals` as stacks, you might see that since `a` and `b` have been bound only once each, the stacks would have only a single value inside. While `dynamicLookup` cannot easily simulate the `this` resolution in an object method, you can think of it as yet another push onto the stack, as shown in Figure 3-4.

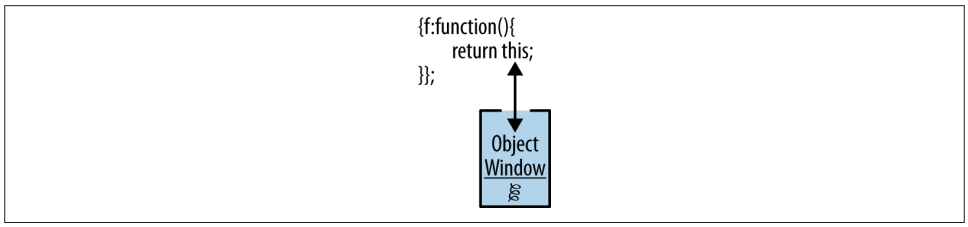

Figure 3-4. An object method referencing "this" will deal with the object itself

In a dynamic scoping scheme, the value at the top of a stack in a binding is the current value. Let's investigate what would happen if we bind again:

```
stackBinder('a', '*');

dynamicLookup('a');
//=> '*'

globals;
//=> {'a': [1, '*'], 'b': [100]}
```

As you'll notice, the new stack bound at `a` contains the stack `[1, '*']`, so any lookup occurring with that condition will result in `*`. To retrieve the previous binding is as simple as unbinding by popping the stack:

```
stackUnbinder('a');

dynamicLookup('a');
//=> 1
```

You may already imagine (or know) how a scheme like this (i.e., the manipulation of global named stacks) may cause trouble, but if not observe the following:

```
function f() { return dynamicLookup('a'); };
function g() { stackBinder('a', 'g'); return f(); };

f();

//=> 1
```

```
g();
//=> 'g'

globals;
// {a: [1, "g"], b: [100]}
```

Here we see that though `f` never manipulated the binding of `a`, the value that it saw was subject to the whim of its caller `g`! This is the poison of dynamic scoping: the value of any given binding cannot be known until the caller of any given function is known—which may be too late.

A point of note in the preceding code is that I had to explicitly "unbind" a dynamic binding, whereas in programming languages supporting dynamic binding this task is done automatically at the close of the dynamic binding's context.

JavaScript's Dynamic Scope

This section has not been an exercise in theory, but instead has set up the discussion for the one area where dynamic scoping rules apply to JavaScript, the `this` reference. In Chapter 2, I mentioned that the `this` reference can point to different values depending on the context in which it was first created, but it's actually much worse than that. Instead, the value of the `this` reference, like our binding of `a`, is also determined by the caller, as shown in the following:

```
function globalThis() { return this; }

globalThis();
//=> some global object, probably Window

globalThis.call('barnabas');
//=> 'barnabas'

globalThis.apply('orsulak', [])
//=> 'orsulak'
```

Yep, the value of the `this` reference is directly manipulable through the use of `apply` or `call`, as shown in Figure 3-5. That is, whatever object is passed into them as the first argument becomes the referenced object. Libraries like jQuery use this as a way to pass context objects and event targets into first-class functions, and as long as you keep your wits about you, it can prove to be a powerful technique. However, dynamic scope can confuse this.

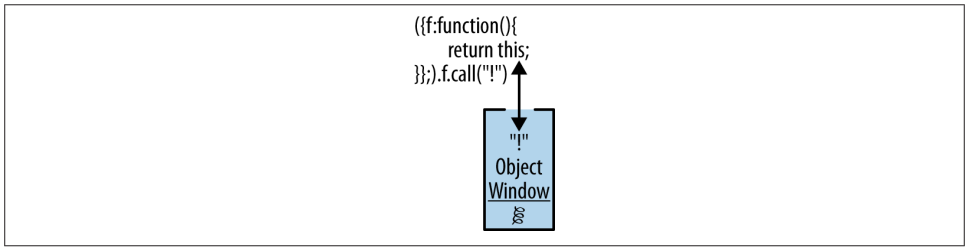

Figure 3-5. Using the Function#call method allows you to set the "this" reference to a known value

Thankfully, this problem does not arise if a `this` reference is never passed to `call` or `apply`, or if it is bound to `null`. Additionally, Underscore provides the function `_.bind` that allows you to lock the `this` reference from changing, like the following:

```
var nopeThis   = _.bind(globalThis, 'nope');

nopeThis.call('wat');
//=> 'nope';
```

Because the `this` reference is dynamically scoped, you may often find, especially in the case of event handling functions, that the `this` you get on something like a button click is not useful and may break your app. To tackle a problem like this, you can use the `_.bindAll` function to lock the `this` reference to a stable value for all of the named methods, as shown here:

```
var target = {name: 'the right value',
              aux: function() { return this.name; },
              act: function() { return this.aux(); }};

target.act.call('wat');
// TypeError: Object [object String] has no method 'aux'

_.bindAll(target, 'aux', 'act');

target.act.call('wat');
//=> 'the right value'
```

And thus, Underscore saves us from the perils of dynamic scoping. Now that I've covered dynamic scope in detail, it's high time to cover function scope.

Function Scope

In order to illustrate the difference between dynamic and function scoping, I'll need to modify the logic for binding and lookup. Instead of accessing bindings in a global hash map, the new model will instead require that all bindings be constrained to the smallest

area possible (namely, the function). This follows the scoping model adhered to by JavaScript.[4]

To simulate a function scoping scheme requires a bit of imagination. As you know, each JavaScript function can refer to a `this` reference. In the previous section, I talked about the dangers of the dynamic nature of `this`, but for the sake of illustration, I'll use it to prove a different point. First, observe how JavaScript acts by default:

```
function strangeIdentity(n) {
  // intentionally strange
  for(var i=0; i<n; i++);
  return i;
}

strangeIdentity(138);
//=> 138
```

In a language like Java, an attempt to access a variable like `i`, defined locally to a `for` block, would provoke an access error. However, in JavaScript, all `var` declarations in a function body are implicitly moved to the top of the function in which they occur. The action of JavaScript to rearrange variable declarations is called *hoisting*. In other words, the previously defined function would become something like:[5]

```
function strangeIdentity(n) {
  var i;
  for(i=0; i<n; i++);
  return i;
}
```

The implications of this are that any piece of code in the function can see all of the variables defined inside. Needless to say, this can cause problems at times, especially if you are not careful about how the variables are captured via closures (discussed in the following section).

In the meantime, I can show how function scope can be simulated quite easily by using the `this` reference:

```
function strangerIdentity(n) {
  // intentionally stranger still
  for(this['i'] = 0; this['i']<n; this['i']++);
  return this['i'];
```

4. The `ECMAScript.next` activity has defined numerous ways to define "lexical" variable scoping. Lexical scoping works similarly to function scoping except that it's "tighter." That is, it binds variables within JavaScript blocks and does not raise the declaration to the top of function bodies. I will not go into detail about lexical scoping here, but it's a topic well worth studying yourself.

5. The `ECMAScript.next` initiative is working through the specification of block-scope that would provide another level of scoping finer-grained than function scope. It's unclear when this feature will make it into JavaScript core. Its eventual inclusion should help justify the next edition of this book (crossing fingers).

```
}

strangerIdentity(108);
//=> 108
```

Of course, this is not a true simulation because in this circumstance I've actually modified the global object:

```
i;
//=> 108
```

Whoops! Instead, it would be more precise to supply a scratch space for the function to operate on, and thanks to the magic of the `this` reference, I can supply it on the call:

```
strangerIdentity.call({}, 10000);
//=> 10000

i;
//=> 108
```

Although our original global `i` persists, at least I've stopped modifying the global environment. I've again not provided a true simulator, because now I can only access locals inside of functions. However, there is no reason that I need to pass in an empty object as context. In fact, in this faked-out JavaScript, it would be more appropriate to pass in the global context, but not directly, or else I'd be back in the soup. Instead, a clone should do:

```
function f () {
  this['a'] = 200;
  return this['a'] + this['b'];
}

var globals = {'b': 2};

f.call(_.clone(globals));
//=> 202
```

And checking the global context proves clean:

```
globals;
//=> {'b': 2}
```

This model is a reasonable facsimile of how function scoping operates. For all intents and purposes, this is precisely what JavaScript does, except variable access is done implicitly within the function body instead of requiring an explicit lookup in `this`. Regardless of your thoughts about function-level scoping, at least JavaScript takes care of the underlying machinery for us—small victories and all that.

Closures

JavaScript closures are one of life's great mysteries. A recent survey on total Internet size places blog posts about JavaScript closures at around 23%.[6]

I kid. Closures, for whatever reason, remain a mystery to a substantial number of programmers for numerous reasons. In this section, I will take some time to go into detail on closures in JavaScript, but thankfully for you, they're quite simple. In fact, throughout this section, I'll build a small library that simulates scoping rules and closures. I'll then use this library to explore the details of this chapter, which include global scope, function scope, free variables, and closures.

To start, it's worth mentioning that closures go hand in hand with first-class functions. Languages without first-class functions can support closures, but they're often greatly stunted. Thankfully, JavaScript supports first-class functions, so its closures are a powerful way to pass around ad hoc encapsulated states.

For the remainder of this chapter and the next, I will capitalize all of the variables that are captured by a closure. This is in no way standard practice in the JavaScript you're likely to see in the wild, nor an endorsement of such activity, but only meant to teach. After these two chapters I will no longer use this convention.

Having said all of that, a closure is a function that "captures" values near where it was born. Figure 3-6 is a graphical representation of a closure.

6. If you factor in the commentary on Hacker News, it's closer to 36%. I have no citation for this because I just made it up for fun.

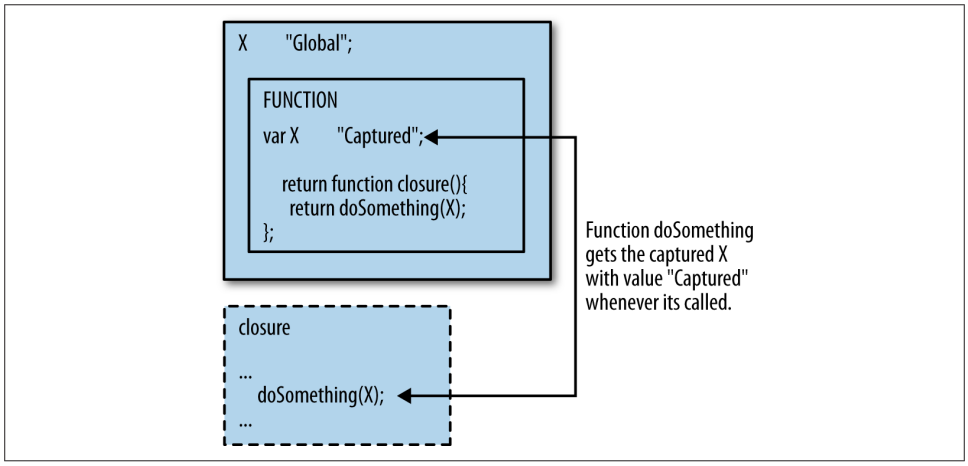

Figure 3-6. A closure is a function that "captures" values near where it was born

In the next few sections, I'll cover closures in depth, starting with a closure simulator.

Simulating Closures

It took only 30 years, but closures are finally becoming a key feature of mainstream programming languages. What is a closure? In a sentence, a closure is a function that captures the external bindings (i.e., not its own arguments) contained in the scope in which it was defined for later use (even after that scope has completed).

Before we go any further with how to simulate closures, let's take a look at how they behave by default. The simplest example of a closure is a first-class function that captures a local variable for later use:

```
function whatWasTheLocal() {
  var CAPTURED = "Oh hai";

  return function() {
    return "The local was: " + CAPTURED;
  };
}
```

Using the `whatWasTheLocal` function is as follows:

```
var reportLocal = whatWasTheLocal();
```

I've already talked about how function-local variables live only for the lifetime of a function's body, but when a closure captures a variable, it's able to live for an indeterminate extent:[7]

```
reportLocal();
//=> "The local was: Oh hai"
```

So it seems that the local variable `CAPTURED` was able to travel with the closure returned by `whatWasTheLocal`—indeed, this is effectively what happened. But local variables are not the only things that can be captured. As shown here, function arguments can be captured as well:

```
function createScaleFunction(FACTOR) {
  return function(v) {
    return _.map(v, function(n) {
      return (n * FACTOR);
    });
  };
}

var scale10 = createScaleFunction(10);

scale10([1,2,3]);
//=> [10, 20, 30]
```

The function `createScaleFunction` takes a scaling factor and returns a function that, given a collection of numbers, returns a list of its elements multiplied by the original scaling factor. As you may have noticed, the returned function refers to a variable `FACTOR` that seemingly falls out of scope once the `createScaleFunction` function exits. This observation is only partially true because in fact, the variable `FACTOR` is retained within the body of the return scaling function and is accessible anytime that function is called. This variable retention is precisely the definition of a closure.

So how would we simulate a closure using our function-scoped `this` scratchpad from the previous section? First of all, I'll need to devise a way for capturing closed variables while simultaneously maintaining access to non-closed variables normally. The most straightforward way to do that is to grab the variables defined in the outer function individually and bind them to the `this` of the returned function, as in the following:

```
function createWeirdScaleFunction(FACTOR) {
  return function(v) {
    this['FACTOR'] = FACTOR;
    var captures = this;

    return _.map(v, _.bind(function(n) {
```

7. Closures are the programming language equivalent of vampires—they capture minions and give them everlasting life until they themselves are destroyed. The only difference is that closures don't sparkle when exposed to sunlight.

```
      return (n * this['FACTOR']);
    }, captures));
  };
}

var scale10 = createWeirdScaleFunction(10);

scale10.call({}, [5,6,7]);
//=> [50, 60, 70];
```

Wow, keeping track of which variables are needed within the body of inner functions seems like a real pain. If you needed to keep track manually, like in this example, then JavaScript would be exceedingly difficult to write.[8] Thankfully for us, the machinery driving variable capture is automatic and straightforward to use.

Free variables

Free variables are related to closures in that it is the free variables that will be closed over in the creation of said closure. The basic principle behind closures is that if a function contains inner functions, then they can all see the variables declared therein; these variables are called "free" variables.[9] However, these variables can be grabbed and carried along for later use in inner functions that "escape" from a higher-level function via return.[10] The only caveat is that the capturing function *must be defined* within the outer function for the capture to occur. Variables used in the body of any function without prior local declaration (neither passed into, nor defined locally) within a function are then captured variables. Observe:

```
function makeAdder(CAPTURED) {
  return function(free) {
    return free + CAPTURED;
  };
}

var add10 = makeAdder(10);

add10(32);
//=> 42
```

The variable `CAPTURED` in the outer function is indeed captured in the returned function performing the addition because the inner never declares `CAPTURED`, but refers to it anyway. Thereafter, the function returned from `makeAdder` retains the variable `CAPTURED`

8. I could just refer directly to `captures` instead of dynamically passing it to the inner function passed to map via Underscore's `bind`, but then I would be using a closure to simulate a closure! That's cheating.
9. Not free as in beer, and not free as in freedom, but instead free as in theft.
10. Another name for this could be higher-order since the function returns another function. I go more in depth in Chapter 3.

that was captured when it was created and uses it in its calculation. Creating another adder will capture the same named variable `CAPTURED` but with a different value, because it will be created during a later invocation of `makeAdder`:

```
var add1024 = makeAdder(1024);
add1024(11);
//=> 1035

add10(98);
//=> 108
```

And finally, as shown in the preceding code, each new adder function retains its own unique instance of `CAPTURED`—the one captured when each was created. The value captured can be of any type, including another function. The following function, `averageDamp`, captures a function and returns a function that calculates the average of some value and the result of passing it to the captured function:[11]

```
function averageDamp(FUN) {
  return function(n) {
    return average([n, FUN(n)]);
  }
}

var averageSq = averageDamp(function(n) { return n * n });
averageSq(10);
//=> 55
```

Higher-order functions that capture other functions are a very powerful technique for building abstractions. I will perform this kind of act throughout the course of this book to great effect.

What happens if you create a function with a variable named the same as a variable in a higher scope? I'll talk briefly about this presently.

Shadowing

Variable shadowing happens in JavaScript when a variable of name `x` is declared within a certain scope and then another variable of the same name is declared later in a lower scope. Observe a simple example of shadowing:

```
var name = "Fogus";
var name = "Renamed";

name;
//=> "Renamed"
```

11. I defined `average` in Chapter 2.

That two consecutive declarations of variables with the same name assign the value of the second should be no surprise. However, shadowing via function parameters is where the complexity level rises:

```
var shadowed = 0;

function argShadow(shadowed) {
  return ["Value is", shadowed].join(' ');
}
```

What do you think is the value of the function call `argShadow(108)`? Observe:

```
argShadow(108)
//=> "Value is 108"

argShadow();
//=> "Value is "
```

The argument named `shadowed` in the function `argShadow` overrides the value assigned to the same named variable at the global scope. Even when no arguments are passed, the binding for `shadowed` is still set. In any case, the "closest" variable binding takes precedence. You can also see this in action via the use of `var`:

```
var shadowed = 0;

function varShadow(shadowed) {
  var shadowed = 4320000;
  return ["Value is", shadowed].join(' ');
}
```

If you guessed the value returned value of `varShadow(108)` is now `"Value is 4320000"` then you're absolutely correct. Shadowed variables are also carried along with closures, as shown in the following:

```
function captureShadow(SHADOWED) {
  return function(SHADOWED) {
    return SHADOWED + 1;
  };
}

var closureShadow = captureShadow(108);

closureShadow(2);
//=> 3 (it would stink if I were expecting 109 here)
```

I tend to avoid shadowed variables when writing JavaScript code, but I do so only via careful attention to naming. If you're not careful, then shadowing can cause confusion if you've not accounted for it. I'll now move on to some quick closure usage examples before wrapping up this chapter.

Using Closures

In this section, I'll touch briefly on the cases for using closures. Since the remainder of the book will use closures extensively, there's no need for me to belabor the point, but it's useful to show a few in action.

If you recall from "Other Examples of Applicative Programming" on page 36, the function `complement` took a predicate function and returned a new function that returned its opposite truthiness. While I glossed over the fact at the time, `complement` used a closure to great effect. Rewriting to illustrate the closure:

```
function complement(PRED) {
  return function() {
    return !PRED.apply(null, _.toArray(arguments));
  };
}
```

The `PRED` predicate is captured by the returned function. Take the case of a predicate that checks for evenness:

```
function isEven(n) { return (n%2) === 0 }
```

We can use `complement` to now define `isOdd`:

```
var isOdd = complement(isEven);

isOdd(2);
//=> false

isOdd(413);
//=> true
```

But what happens if the definition of `isEven` changes at some later time?

```
function isEven(n) { return false }

isEven(10);
//=> false
```

Will this change the behavior of `isOdd`? Observe:

```
isOdd(13);
//=> true;

isOdd(12);
//=> false
```

As you can see, the capture of a variable in a closure grabs the reference of the captured thing (in this case, the predicate `PRED`) at the time that the closure is created. Since I created a new reference for `isEven` by creating a fresh variable, the change was transparent to the closure `isOdd`. Let's run this to ground:

```
function showObject(OBJ) {
  return function() {
    return OBJ;
  };
}

var o = {a: 42};
var showO = showObject(o);

showO();
//=> {a: 42};
```

Everything is fine and good, no? Well, not exactly:

```
o.newField = 108;
showO();
//=> {a: 42, newField: 108};
```

Since the reference to `o` exists inside and outside of the closure, its changes can be communicated across seemingly private boundaries. This is potentially a recipe for confusion, so the typical use case minimizes the exposure of captured variables. A pattern you will see very often in JavaScript is to use captured variables as private data:

```
var pingpong = (function() {
  var PRIVATE = 0;

  return {
    inc: function(n) {
      return PRIVATE += n;
    },
    dec: function(n) {
      return PRIVATE -= n;
    }
  };
})();
```

The object `pingpong` is constructed within the anonymous function serving as a scope block, and contains two closures `inc` and `dec`. The interesting part is that the captured variable `PRIVATE` is private to the two closures and cannot be accessed through any means but by calling one of the two functions:

```
pingpong.inc(10);
//=> 10

pingpong.dec(7);
//=> 3
```

Even adding another function is safe:

```
pingpong.div = function(n) { return PRIVATE / n };

pingpong.div(3);
// ReferenceError: PRIVATE is not defined
```

The access protection provided by this closure pattern is a powerful technique available to JavaScript programmers for keeping sanity in the face of software complexity.

Closures as an Abstraction

While closures provide for private access in JavaScript, they are a wonderful way to offer abstraction (i.e., closures often allow you to create functions based solely on some "configuration" captured at creation time). The implementations of `makeAdder` and `complement` are good examples of this technique. Another example is a function named `plucker` that takes a key into an associative structure—such as an array or an object—and returns a function that, given a structure, returns the value at the key. The implementation is as follows:

```
function plucker(FIELD) {
  return function(obj) {
    return (obj && obj[FIELD]);
  };
}
```

Testing the implementation reveals its behavior:

```
var best = {title: "Infinite Jest", author: "DFW"};

var getTitle = plucker('title');

getTitle(best);
//=> "Infinite Jest"
```

As I mentioned, `plucker` also works with arrays:

```
var books = [{title: "Chthon"}, {stars: 5}, {title: "Botchan"}];

var third = plucker(2);

third(books);
//=> {title: "Botchan"}
```

`plucker` comes in handy in conjunction with `_.filter`, which is used to grab objects in an array with a certain field:

```
_.filter(books, getTitle);
//=> [{title: "Chthon"}, {title: "Botchan"}]
```

As this book proceeds, I will explore other uses and advantages of closures, but for now I think the groundwork for understanding them has been laid.

Summary

This chapter focused on two foundational topics not only in JavaScript, but also for functional programming in general: variable scope and closures.

The focus on variable scope started with global scope, the largest available to JavaScript, and worked its way inward through lexical scope and function scope. Additionally, I covered dynamic scoping, especially as it manifests in the use and behavior of the `this` reference, based on the use of the `call` and `apply` methods. While potentially confusing, the dynamic `this` could be fixed to a certain value using Underscore's `_.bind` and `_.bindAll` functions.

My coverage of closures focused on how you could simulate them in a language with all of JavaScript's features, including the dynamic `this`. After simulating closures, I showed how to not only use closures in your own functions, but also how they can be viewed as a way to "tweak" existing functions to derive new functional abstractions.

In the next chapter, I will expand on first-class functions and delve into higher-order functions, defined in terms of these points:

- Functions can be passed to other functions
- Functions can be returned from functions

If you're unclear about the content of this chapter, then you might want to go back and reread before proceeding on to the next chapter. Much of the power of higher-order functions is realized in concert with variable scoping, and especially closures.